NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ACCESSING HIERARCHICAL DATABASES VIA SQL TRANSACTIONS IN A MULTI-MODEL DATABASE SYSTEM

by

John A. Zawis

December 1987

Thesis Advisor: David K. Hsiao

NCLASSIFIED
SCURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1 REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2 SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
2 DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION aval Postgraduate School / 6b OFFICE SYMBOL (If applicable) Code 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
ADDRESS (City, State, and ZIP Code) onterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
NAME OF FUNDING/SPONSORING ORGANIZATION / 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

TITLE (Include Security Classification)
CESSING HIERARCHICAL DATABASES VIA SQL TRANSACTIONS IN A MULTI-MODEL TABASE SYSTEM (u)

PERSONAL AUTHOR(S)
awis, John A.

TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year, Month, Day)	15 PAGE COUNT
ster's Thesis	FROM ____ TO ____	1987 December	124

SUPPLEMENTARY NOTATION

COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Cross-access; Database design; Database implementation; Database management systems; Hierarchical Database; Multi-Lingual Database

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

There has been a tremendous growth in recent years in the use of data base management systems (DBMS) throughout the world. This has lead to efforts to increase the effectiveness and efficiency of systems designed to create and maintain large databases. The traditional approach has been to select a data model and its associated model-based data language and implement a database system based on that single model. The multi-lingual database system (MLDS) was designed to increase the functionality of data base systems by (Continued)

DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
NAME OF RESPONSIBLE INDIVIDUAL of. David K. Hsiao	22b TELEPHONE (Include Area Code) (408) 646-2253 / 22c OFFICE SYMBOL Code 52Hq

ORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED

18. SUBJECT TERMS (Continued)

System (MLDS); Multi-Backend Database System (MBDS); Multi-Model Database System (MMDS); Relational Database

19. ABSTRACT (Continued)

allowing the use of multiple data models and several model-based languages on a single system. With this approach, the system could support a heterogeneous collection of databases, each based on the data model most appropriate for the individual application requirements.

MLDS currently supports the use of relational, hierarchical, network, and functional databases. The goal of this thesis is to further increase the functionality of MLDS by permitting a user knowledgeable only in a relational-based data language (SQL) to access and manipulate information in a hierarchical database, while strictly maintaining the integrity of the hierarchical model. This extends the multi-lingual database system to a multi-model database system (MMDS). The emphasis in this thesis is two-fold. First, to provide the design analysis necessary to accomplish the translation. More specifically, to develop a process for transforming a hierarchical database schema into an equivalent relational schema and to analyze the SQL requests that are used to access a database and provide a methodology for equivalent access to a hierarchically- based database system. The second area of emphasis is in the implementation of the schema transformation process and language translation methodology within the current MLDS structure. The software engineering aspects of the implementation are detailed to provide a base for further expansion of similar systems.

S N 0102- LF- 014- 6601

Accessing Hierarchical Databases Via SQL Transactions in a Multi-Model Database System

by

John A. Zawis
Lieutenant, United States Navy
B.S., Hawaii Pacific College, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

December 1987

ABSTRACT

There has been a tremendous growth in recent years in the use of data base management systems (DBMS) throughout the world. This has lead to efforts to increase the effectiveness and efficiency of systems designed to create and maintain large databases. The traditional approach has been to select a data model and its associated model-based data language and implement a database system based on that single model. The multi-lingual database system (MLDS) was designed to increase the functionality of data base systems by allowing the use of multiple data models and several model-based languages on a single system. With this approach, the system could support a heterogeneous collection of databases, each based on the data model most appropriate for the individual application requirements.

MLDS currently supports the use of relational, hierarchical, network, and functional databases. The goal of this thesis is to further increase the functionality of MLDS by permitting a user knowledgeable only in a relational-based data language (SQL) to access and manipulate information in a hierarchical database, while strictly maintaining the integrity of the hierarchical model. This extends the multi-lingual database system to a multi-model database system (MMDS). The emphasis in this thesis is two-fold. First, to provide the design analysis necessary to accomplish the translation. More specifically, to develop a process for transforming a hierarchical database schema into an equivalent relational schema and to analyze the SQL requests that are used to access a database and

provide a methodology for equivalent access to a hierarchically- based database system. The second area of emphasis is in the implementation of the schema transformation process and language translation methodology within the current MLDS structure. The software engineering aspects of the implementation are detailed to provide a base for further expansion of similar systems.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

A. OVERVIEW

The traditional approach to designing and implementing a database system involves analyzing the needs and structure of the task and then choosing an appropriate data model. Possible models include the relational data model, the hierarchical data model, the network data model, or the entity-relationship model to name just a few. The next step in the process is to specify a data language based on the selected model. For example, SQL for the relational data model or DL/I for the hierarchical model.

A number of database system have been designed following this traditional approach. IBM's SQL/Data system supports the relational model and data language. Sperry Univac developed the DMS-1100 system which supports the network data model and although its data language is unnamed, it uses a CODASYL-based data manipulation language. A final example is IBM's Information Management System (IMS) which was developed around the hierarchical data model and the hierarchical model data language DL/I (Data Language I).

Each of these traditional database designs can be characterized as being *mono-lingual database systems.* That is, each is based on a single data model that acts as a high-level abstraction of the underlying data that makes up the database itself. The user interacts with the database by writing transactions in the model-specific data language designed to support that data model. The obvious limitation of this approach is that the user is restricted to a single data model and a specific model-based data language.

A much more flexible approach to database design was proposed by Demurjian and Hsiao [1]. Modern database systems should support and execute a large number of shared databases using a finite set of data models and associated data languages. Such a system is call a *multi-lingual database system (MLDS).*

There are a number of distinct advantages to such an approach. Since an MLDS design supports a number of different data models, an organization using this system could save on the cost of additional hardware and software when implementing a new database or database application. An analysis of the new requirements would lead to the selection of an appropriate model and subsequent implementation of the database on existing system components.

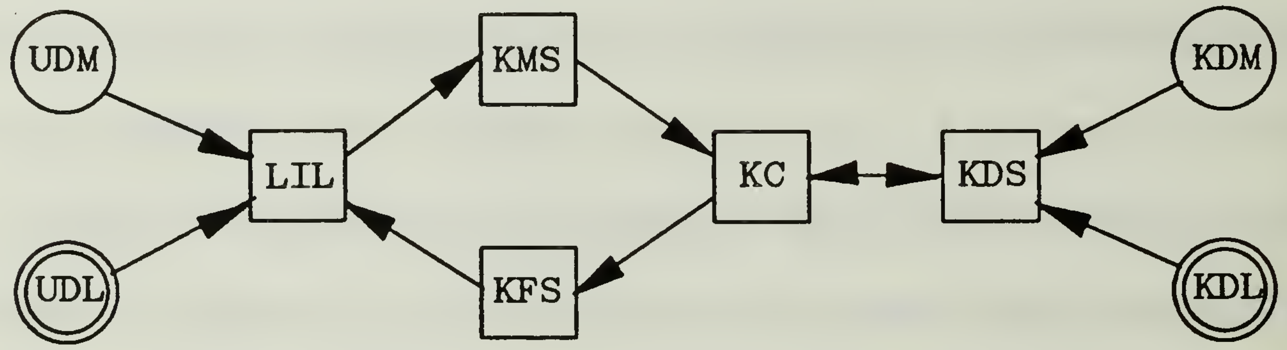
Another distinct advantage is in support and training. Since MLDS supports a variety of data models and languages on a single system, existing employee skills can be utilized on the multiple data models reducing overall training costs. Additionally, database resources are specialized to the particular mix of requirements within an organization rather than relying on a single, mono-lingual system that must attempt to be general enough to handle the diverse requirements. This specialization results in an increase in both performance and functionality.

A more subtle, but significant advantage of using an MLDS system involves the flexibility to explore the effectiveness of various data models for a given database application. The development of a new application might involve parallel implementation of a small number of databases utilizing different data models and languages that appear appropriate to the envisioned use. Further testing and analysis may indicate that the mix of transactions specific to that application are more efficiently and effectively handled by one of the selected models.

B. THE MULTI-LINGUAL DATABASE SYSTEM

A block diagram of the structure of a multi-lingual database system is shown in Figure 1. A user accesses and modifies the database by interaction with the *language interface layer (LIL)* through a specific *user data model (UDM)*. Transactions are written in a *user data language (UDL)* defined for the chosen model. Transactions are of two general types, database definition requests and database manipulation requests. The LIL identifies which of these types is currently being input by the user and routes the transaction sequences to the *kernel mapping system (KMS)* for processing.

The KMS handles the requests in two ways. If the transactions are database definition requests, the KMS transforms the UDM database definition to a *kernel data model (KDM)* database definition equivalent. The transformed definition is then forwarded to the *kernel controller (KC)* which, in turn, routes the requests to



UDM	:User Data Model
UDL	:User Data Language
LIL	:Language Interface Layer
KMS	:Kernel Mapping System
KC	:Kernel Controller
KFS	:Kernel Formatting System
KDM	:Kernel Data Model
KDL	:Kernel Data Language
KDS	:Kernel Database System

○	Data Model
◎	Data Language
□	System Module
→	Information Flow

Figure 1. The Multi-Lingual Database System

the *kernel database system (KDS)* for processing. When the KDS has finished with the database creation, it notifies KC of the completion, which in turn notifies the user through the LIL that the database definition request has been processed and further requests can be accepted.

If the transactions are database manipulation requests, the KMS transforms the UDL transactions to their KDL equivalents. These requests are then forwarded to the KDS, through the KC, for processing. The KDS returns the results of the transaction to KC. KC forwards these results to the *kernel formatting system (KFS)* where they are transformed from their KDM structure to a UDM equivalent. The results are then displayed to the user in a format consistent with the UDM.

The LIL, KMS, KFS, and KC define the language interface for a single user-defined data model. In a multi-lingual database system, a separate language interface is required for each model defined as shown in Figure 2. For example, in the current system, a unique language interface has been developed for the relational/SQL model, the hierarchical/DL/I model, the network/CODASYL-DML model, and the functional/Daplex model. In contrast, the KDS structure is a single, common component shared by all models. It is through the KDS that the physical database is accessed and manipulated by the various user-defined language interfaces.

The attribute-based data model and attribute-based data language (ABDL) have been implemented as the KDM and KDL, respectively, for MLDS. ABDL is a simple yet powerful language first described by Hsiao [2,3] and studied by Rollins [4]. Subsequent reports have been completed which show how relational [5], hierarchical [6], and network [7] constructs can be mapped to attribute-based equivalents and identified the background for the data-language

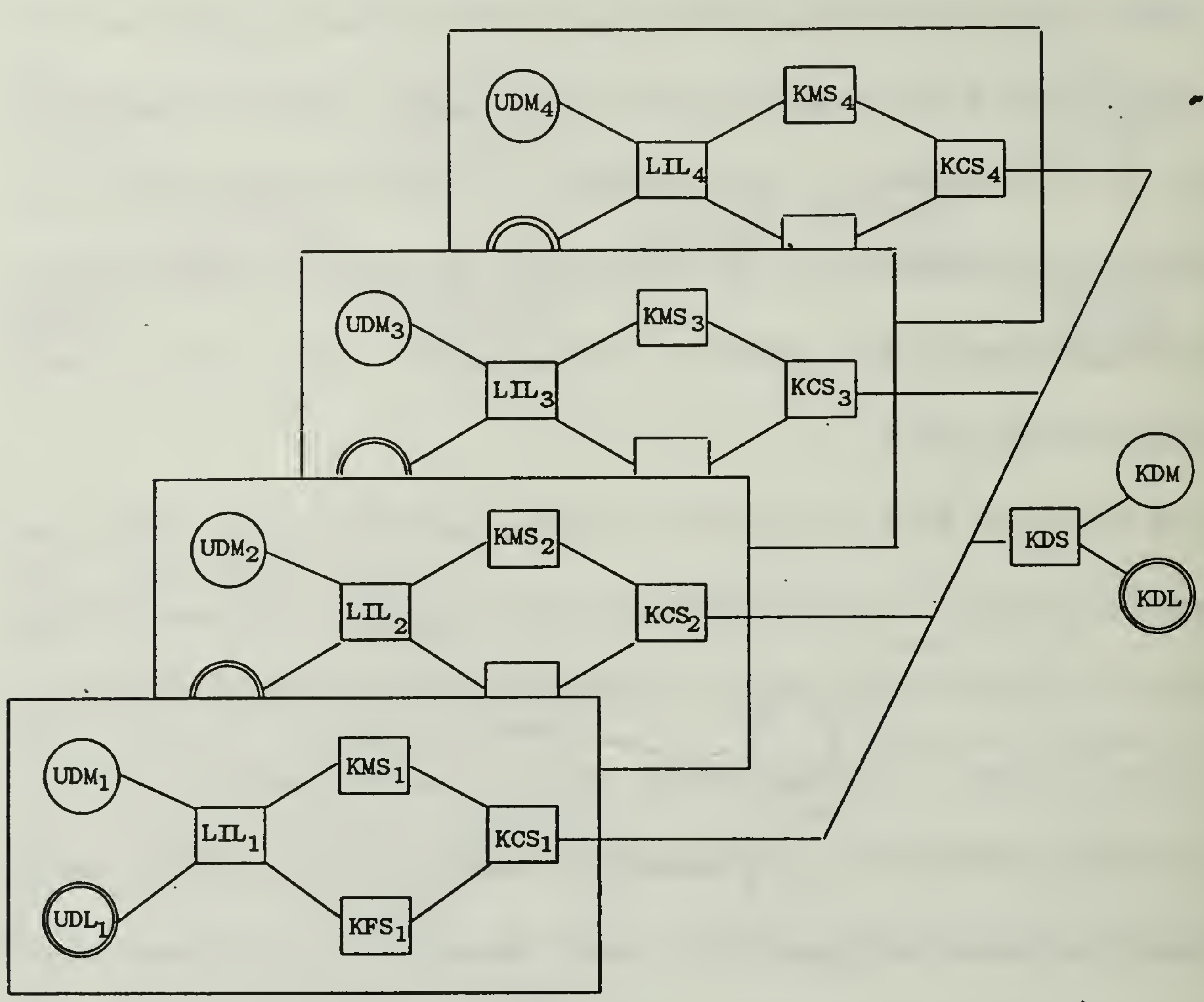


Figure 2. Multiple Language Interfaces

interfaces from SQL to ABDL [8,9], from DL/I to ABDL [10,11] and from CODASYL-DML to ABDL [12,13]. Additionally, the design for the Daplex to ABDL language interface [14] has been detailed, however, the implementation [15] has not been completed at the time this thesis was written.

C. THE MULTI-BACKEND DATABASE SYSTEM

The *multi-backend database system (MBDS)* has been designed to overcome the performance and replacement problems associated with a traditional mainframe-based approach to database system design. MBDS has solved these problems by moving the database functions to a separate system with its own dedicated hardware and software. As shown in Figure 3, the MBDS controller is a separate computer from the backends. It acts as an interface to a host computer or directly to users and performs the controlling functions of the database system. Transactions are passed to the controller and the results of database operations are routed back to the controller from the backends. The backends are the database engines of MBDS. Each is a separate mini- or micro- computer connected in parallel via a broadcast communications bus. Each backend maintains a portion of the database on one or more hard disk subsystems. This parallelism proves to be the key to the high-performance of the system. When a transaction is broadcast by the controller, each backend can execute the request on its portion of the database, independent of the other backends.

The benefits of the MBDS architecture lie in the capability to provide performance gains and to accommodate database growth. Performance gains can be realized by increasing the number of database backends. Assuming a constant size database, an MBDS system should produce a nearly proportional decrease in response times when the number of backends is increased. Additionally, a proportional increase in the number of backends in relation to an increase in the

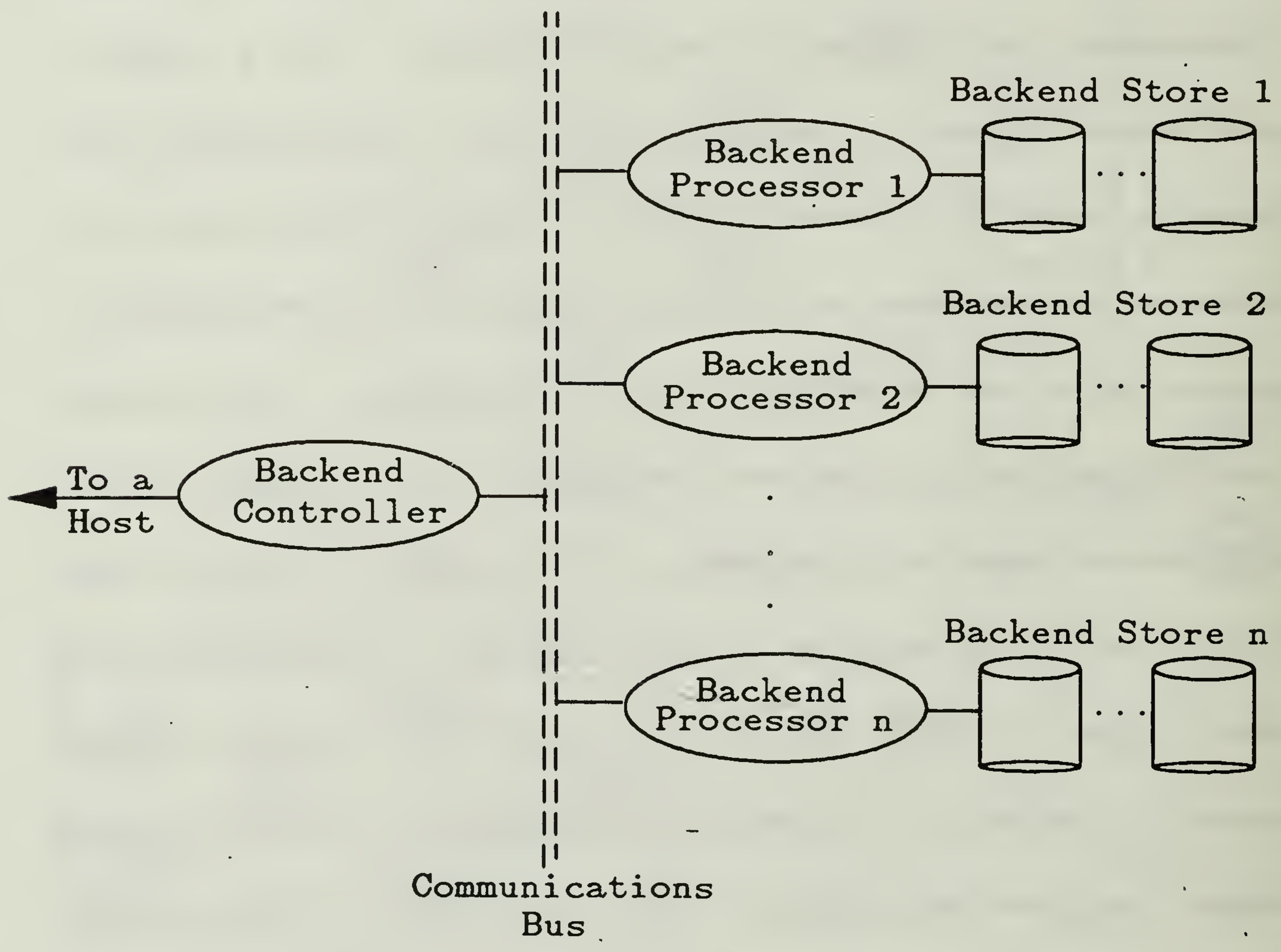


Figure 3. The Multi-Backend Database System

size of the database produces nearly invariant response times for a given set of transactions.

MBDS also provides a high degree of extensibility. The system can accommodate additional backends with no modification to existing software and no new programming. In addition, no modification to existing hardware is necessary and the disruption of system activity in minimal. The reader is referred to Hsiao and Menon [16,17] for a more detailed discussion on MBDS.

D. THESIS ORGANIZATION

The current implementation of MLDS is restricted in the complete utilization of the available databases. Specifically, the relational databases are accessible only through the SQL interface, the hierarchical databases are accessible only through DL/I, the network databases are accessible only through the CODASYL-DML interface, and the functional databases are accessible only through Daplex. This thesis is part of the effort to remove these restrictions, thereby allowing the databases based on a given models to be accessed by database languages associated with different data models. This extends the multi- lingual database system to a *multi-model database system (MMDS).*

We are concerned in this thesis with the design and implementation of a methodology which will allow a SQL user to access a hierarchical database. In Chapter II, we describe the attribute-based, relational, and hierarchical models and their associated languages in order to provide a base of understanding for the following discussion. In Chapter III we examine a number of strategies for implementing the cross-access of a hierarchical database via SQL transactions in the MMDS and select the most appropriate approach. Chapter IV details the implementation issues involved in transforming a hierarchical schema to a functionally equivalent relational schema and in Chapter V, we discuss the design and implementation issues involved in transforming SQL transactions into ABDL equivalents that will allow manipulation of data in a hierarchical database while maintaining the integrity of that database. Finally, in Chapter VI, we provide our conclusions concerning the actual design and implementation of the cross-language functionality.

Appendix A provides a schematic representation of the major data structures utilized in implementing the relational language interface, with emphasis on those structures modified for the MMDS implementation. Appendices B and C contain the specification details of the LIL and KMS, written in a System Specification Language for ease in understanding. These two modules were the most extensively modified during the implementation of MMDS. New or modified code has been italicized to more clearly identify changes in the MLDS design. The relational model implementation thesis by Kloepping and Mack [9], contains complete details on the data structures and module specification of the relational language interface in the Multi-Lingual Database System.

II. DATA MODELS

In this chapter, we briefly describe the various data models and model-based data languages necessary for a full understanding of the multi-model transformation. In section A, we discuss the attribute-based data model and its associated language ABDL. Section B outlines the relational data model and the SQL data language. Finally, in section C the hierarchical data model and the DL/I language are presented.

A. THE ATTRIBUTE-BASED DATA MODEL

As stated in Chapter I, the attribute-based data model and ABDL have been implemented as the KDM and KDL respectively in the multi-lingual database system. This model and its associated language, as originally developed by Hsiao [2,3], is a simple yet powerful construct for creating and manipulating databases.

1. Model Description

A *database* consists of a collection of files. Each *file* contains a group of related records. A *record* is made up of a collection of *attribute-value pairs.* An attribute-value pair is a Cartesian product consisting of an attribute name and an attribute value. For example, <GRADE, 'A'> is an attribute-value pair having GRADE as an attribute name and an associated attribute value of 'A'. A record may also contain an optional *record body,* containing textual information related to the record. An example of a record, without a record body, is shown below.

(<FILE, Student>, <NAME, 'Zawis'>, <SNUM, 0284>, <GRADE, 'A'>)

The first attribute-value pair in each record identifies the file name. In this case, the file name is 'Student'. There is at most one attribute value pair for each unique attribute defined in the database.

Access to the database is through a query of *keyword predicates.* A predicate is a three-tuple in the form <attribute, operator, value>, such as (SNUM <= 0284). A query on a database then, is a finite number of keyword predicates in disjunctive normal form. For example,

(((FILE = Student) and (SNAME = Zawis)) or
((FILE = Student) and (SNAME = Little)))

2. The Attribute-Based Data Language (ABDL)

Access and manipulation of a database are performed through five primary operations (insert, delete, update, retrieve and retrieve-common). These operations are formed by utilizing the queries as just described. A brief description of each operation follows.

The INSERT request in used to insert a new record into a specified file of an existing database and takes the form:

INSERT Record

An example of an INSERT operation which inserts a student record into a file named Student is:

INSERT (<FILE = Student>, <SNAME = Gorman>, <SNUM = 3462>)

The DELETE operation is used to remove one or more records from the database. A DELETE operation takes the form:

DELETE Query

An example of a DELETE which removes all students named 'Hayes' from the Student file is:

DELETE ((FILE = Student) and (SNAME = Hayes))

An UPDATE is used to modify records of the database. An UPDATE request consists of two parts. The syntax is:

UPDATE Query Modifier

An example of an UPDATE request which changes the grade of a student named Oliver to an 'A' is:

UPDATE ((FILE = Student) and (SNAME = Oliver) (GRADE = 'A'))

A RETRIEVE request is used to retrieve records from the database. The database is not altered by this operation. A RETRIEVE consists of three parts, a query, a target-list and an optional by-clause. The target list specifies the set of attributes to be output to the user. It may consist of an aggregate operation (avg, count, sum, min, max). The by-clause is used to group the output records. The syntax for a RETRIEVE request is:

```
RETRIEVE (Query) (Target-list) (by-clause)
```

For example:

```
RETRIEVE (FILE = Student) (SNAME) BY SNUM
```

would retrieve the names of all students, ordered by their student number.

The final operation is the RETRIEVE-COMMON request. It is used to merge two files by common attribute values. The syntax for a RETRIEVE-COMMON request is:

```
RETRIEVE (Query 1) (target-list 1)
COMMON  (attribute 1, attribute 2)
RETRIEVE (Query 2) (target-list 2)
```

An example of such a request is:

```
RETRIEVE (FILE = STUDENT) (SNAME)
COMMON (SNAME, TNAME)
RETRIEVE (FILE = TEACHER) (TNAME)
```

This request would display a list of students and teachers that share a common name. As with the retrieve command, the database is not modified by this operation.

B. THE RELATIONAL DATA MODEL

1. Model Description

The relational database is viewed by its users as a collection of *tables.* A table can be visualized as being organized in rows and columns. The rows, called *tuples,* are a sequence of related values. The column headings of the table are the *domain names* for the values listed below them. In terms of the attribute-based data model described earlier, a table is a file, while the tuples are records. Each tuple value equates to an attribute-value and the domain names are the attribute-names. The relational model was a major departure from earlier database models such as the hierarchical and network models in the sense that the user was no longer required to understand the underlying structure of the database in order to access and manipulate the data contained within it. Instead, the user is presented with a simple, tabular representation of the data and is allowed to manipulate the data directly without having to first navigate a set of logical connections leading to that data. Operations of the data are specified logically by relational algebra or calculus [18]. This allows maximum flexibility in the manipulation of the database. Any relationship that can be expressed in a logical query can be used to access the data.

2. The Data Manipulation Language (SQL)

A number of different relational data languages have been developed, but by far the most common is SQL (initially called SEQUEL) [19]. It is an English-like language that allows intuitive and simple access to and manipulation of relational databases. SQL is a powerful language which allows many variations of the basic commands. No effort is made in the following examples to provide an exhaustive description of these commands. Instead, the basic syntax is presented to provide a familiarity with the language constructs. A more detailed survey can be found in Date [20].

The SELECT command is used to retrieve information from the database. The basic syntax is:

```
SELECT attribute(s)   FROM relation   WHERE query
```

The attribute-values to be returned are listed in the SELECT clause. The relation or relations to search are identified in the FROM clause, and the conditions on the search are specified in the where clause. An example of a SELECT command is:

```
SELECT  SNUM
FROM    Student
WHERE   SNAME = Hayes
```

The SELECT command is an extremely flexible construct which provides numerous variations for accessing a database. One of the most useful is the NESTED SELECT in which the results of one SELECT request are used in the WHERE clause of a second SELECT to further refine the set of conditions used in accessing the database.

The INSERT command is used to to insert a new tuple (record) into an existing table (relation). Its syntactical structure is:

```
INSERT INTO relation :  (attribute-names)   < attribute-values >
```

For example:

```
INSERT INTO Student: (SNAME, SNUM, GRADE)
<'Brodhag', 9745, 'A'>
```

If all the attributes of the inserted tuple are specified in the INSERT command, the attribute names do not have to be explicitly listed.

The DELETE command is used to remove tuples from an existing database. The syntax is:

```
DELETE relation WHERE query
```

The set of tuples to be deleted is determined by the query of the WHERE clause. For example:

```
DELETE  Student
WHERE   GRADE = 'B'
```

will delete all tuples in the student relation which have a value of 'B' in the GRADE attribute.

The UPDATE command changes a value of a specific attribute within an existing tuple or set of tuples. The basic syntax is as follows:

```
UPDATE relation   SET modifier   WHERE query
```

An example of an UPDATE operation is:

```
UPDATE Teacher
SET    SALARY = SALARY + 10000
WHERE  DEPT = Computer Science
```

This transaction will update the tuples in the Teacher relation to reflect a 10,000 dollar pay increase for all instructors in the Computer Science department.

C. THE HIERARCHICAL DATA MODEL

1. Model Description

A hierarchical database is composed of an ordered set of trees [20]. A *tree* consists of a single, root record type with an ordered set of one or more dependent subtrees. Each subtree in turn consists of a single root record and set of zero or more dependent subtrees. Hierarchical structures are a very natural way to model real-world systems such as business organizations, military chains of command, university course offerings, etc. and thus are ideal structures for database organization. Each record type is composed of one or more attributes which uniquely define it. A record type is connected to dependent record types through directed arcs or links. These links provide implicit information about the relationships between the various record types that must be explicitly identified in a relational database. In a hierarchical model, the links define a one-to-many relationship from the parent to the child record type. At most, one link can exist between two record types. One of the key constraints in a hierarchical system is that *no occurrence of a child record type can exist without its parent.* This implies that many of the operations on a database must necessarily affect record occurrences other than those specifically identified. For example, if a record occurrence is deleted from the database, the entire subtree consisting of dependent child records to which it is linked must also be deleted. Similarly, a child record occurrence cannot be inserted into the database unless its parent currently exists.

2. <u>The Data Manipulation Language (DL/I)</u>

One of the first, and possibly still the most utilized, database system was introduced in 1968 by International Business Machines (IBM) under the product name Information Management System (IMS) [21]. An IMS database consists of a hierarchical arrangement of *segments* (records), each of which is composed of a collection of *fields*. The data manipulation language utilized by IMS is called Data Language/One (DL/I). Queries to a hierarchical database are designed to be made by issuing DL/I calls from within a host language such as COBOL or PL/I. Since MLDS is a stand-alone system, there is no need for such a host language. Therefore, a more descriptive syntax has been implemented following the general form outlined by Date [20]. Four basic operations (get, insert, delete and replace) have been implemented within MLDS.

The GET operations (GU, GHU, GNP, GHN, etc.) are used to set currency pointers within a hierarchical database and perform retrieval of segment occurrences. Various forms of this operation are also utilized to prepare the database for other manipulation commands. The following example is a Get Unique (GU) query which is used to retrieve the first student occurrence with a grade of B in course number C100 taught in July 1987.

```
GU course   (cnum = 'c100')
offering (date = '0787')
student  (grade = 'B')
```

In addition to the record retrieval, currency pointers have been set within the database and retrieval of additional records meeting the same criteria can be accomplished through looping constructs utilizing a Get Next (GN) operation. For example, the following loop transaction will retrieve the remaining records meeting the above constraints.

```
aa GN student
GOTO aa
```

An INSERT operation is accomplished by specifying a record occurrence and then identifying the hierarchical path to the desired insertion point. For example, the following query inserts a record in the offering segment for course C100, identifying the date, location, and format of that course.

```
BUILD (date, location, format) : ('0787', 'S123', 'lecture')
ISRT  course (cnum = 'c100')
offering
```

DELETE operations are performed by setting database currency pointers via a Get Hold Unique (GHU) operation and then issuing a delete command. The following query deletes a student named 'Sando' from course C100 taught in July 1987.

```
GHU course   (cnum = 'c100')
offering (date = '0787')
student (sname = 'Sando')
DLET
```

As previously mentioned, a DELETE operation automatically deletes all dependent occurrences in the hierarchical database.

The REPLACE operation is used to modify an occurrence within a database and can be accomplished by setting the database currency pointers via a Get Hold Unique (GHU) operation, identifying the field to change, and issuing a replace command. For example, to change the prerequisite for the advanced database course from AI to data structures, the following query can be input.

```
GHU course (ctitle = 'adv. database')
prereq (ptitle = 'ai')
CHANGE ptitle to 'data structures'
REPL
```

A constraint placed on the REPLACE operation in both IMS and MLDS is that a sequence (key) field cannot be updated. A desired change to a sequence field must be accomplished through the use of DELETE and INSERT operations.

III. MAPPING FROM THE HIERARCHICAL TO THE RELATIONAL MODEL

A. MAPPING METHODOLOGIES

As mentioned previously, MLDS is a single database system designed to support a number of different database models and their corresponding data languages. However, MLDS restricts a user to accessing a specific database through the data language implemented to support it. That is, a database user can access a relational database only via SQL transactions or a hierarchical database via DL/I transactions. MMDS is an extension of MLDS that is designed to allow cross-access of databases. For example, a relational user can access a hierarchical database via SQL transactions or a hierarchical user can access a network database using DL/I transactions.

Chapter I outlined the composition of the language interface needed to support each database model. Each interface is specific to the model it supports in terms of capturing the semantics of the data model. Specifically, the attribute-based database created in the data model transformation has the semantics of the corresponding user data model encoded within it. As a result, a given language interface can only access its associated attribute-based database. Notationally, the relational language interface can only access an AB(relational) database and the hierarchical language interface can only access an AB(hierarchical) database.

In view of these restrictions, we can see that the major challenge for MMDS is to develop a methodology that allows users of one data model to access databases created via the language interface of a different data model. More specifically, this thesis focuses on access to a hierarchical database by relational users via SQL transactions.

A number of different design strategies exist for implementing MMDS which can be characterized by the level at which the strategy is integrated into the already existing database system [22]. The basic strategies were first described by Rodeck [23] in terms of accessing functional databases using the CODASYL data language. The remainder of this chapter summarizes these design strategies and concludes with the selection of the strategy best suited for accessing hierarchical databases via SQL transactions.

1. The High-Level Preprocessing Method

The preprocessing strategy is considered to be a high-level process because it occurs on top of the language interface modules as shown in Figure 4. Modifications to the language interface involve three components, a schema transformer, a language translator, and a results reformatter. When a user selects a database which is not part of the local language interface (LI), all other LI's are searched in an attempt to find the database. If successful, the schema transformer uses the original database schema to create a parallel and equivalent schema in the local LI, based on the local database model. When the user executes a transaction against this transformed database schema using the local data

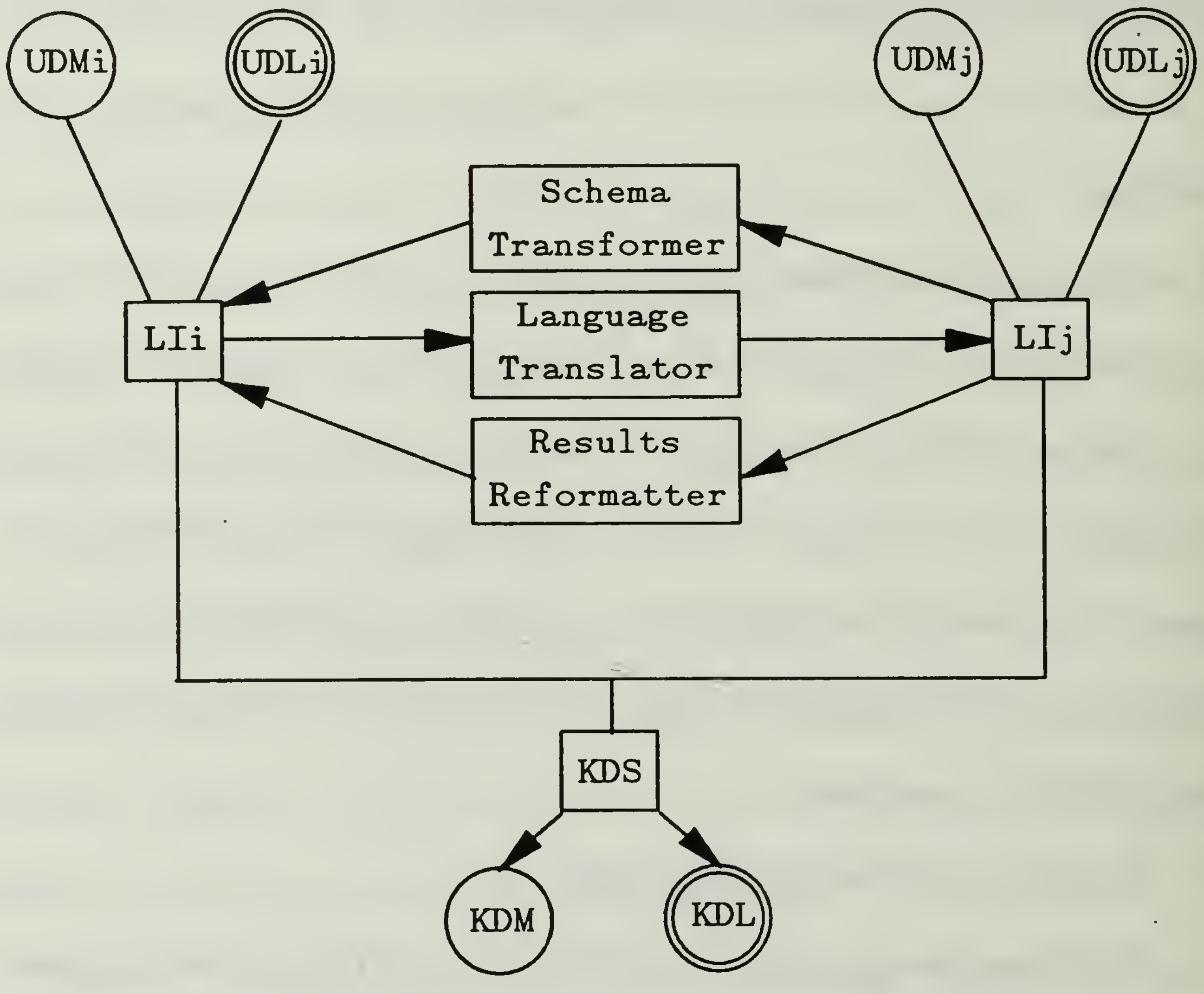


Figure 4. The High-Level Preprocessing Strategy

manipulation language, the language translator generates transactions in the original data manipulation language that can be used to access the database. The results reformatter formats the returned responses, if necessary, in the basic form of the local LI.

2. The Mixed-Processing Strategy

The mixed-processing strategy is a mid-level, direct method for the cross-access of databases as shown in Figure 5. Two components are involved, a schema transformer and a second language interface. As in the preprocessing strategy, when a user selects a database that is not in the local LI, all other LI's

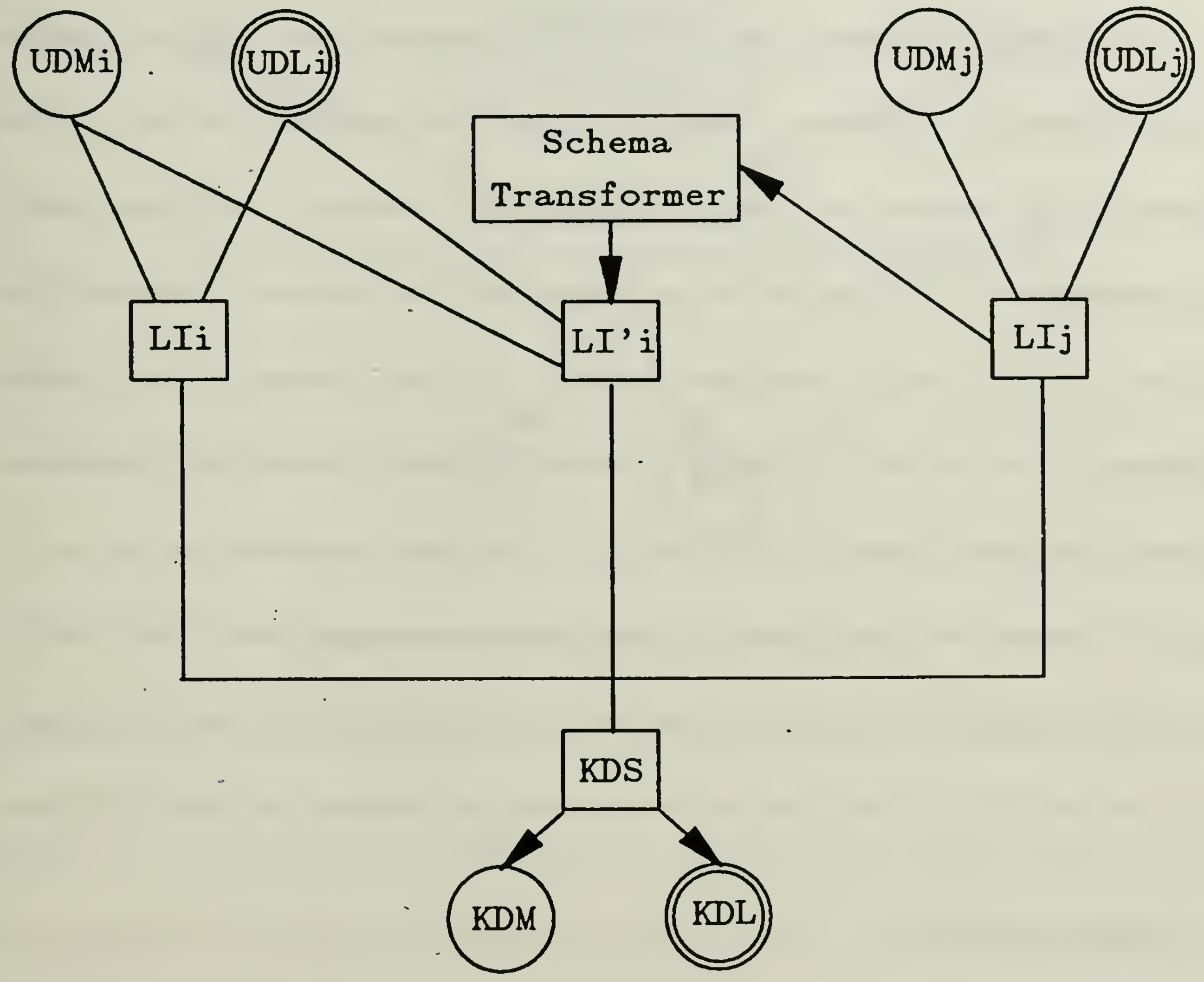


Figure 5. The Mixed-Processing Strategy

are searched for the desired database. When found, the original database schema is copied and transformed into an equivalent schema in the local LI. When a user executes a transaction in the local data manipulation language, the new language interface processes the request. The AB requests output from this language interface are in the form of the original database model which thereby eliminates the need for an extra language translation step.

3. The Postprocessing Strategy

As shown in Figure 6, the postprocessing strategy is a low-level method for cross-accessing databases. Similar to the preprocessing strategy, three components are involved, a schema transformer, a language translator, and a results reformatter. This method is considered low-level because it occurs below the LI's in the kernel database system. In this strategy, the schema transformation occurs from the kernel database schema of the original database to the kernel database schema of the local LI. Language translation occurs in the opposite direction from the kernel database language transactions of the local LI to the kernel database language transactions of the original database. The results reformatter would then translate the results into the format of the local LI's form.

B. DESIGN CHOICE

In selecting the most appropriate design strategy for accessing a hierarchical database via SQL transactions, we must weigh the advantages and disadvantages of each of the strategies developed. A major problem with the postprocessing

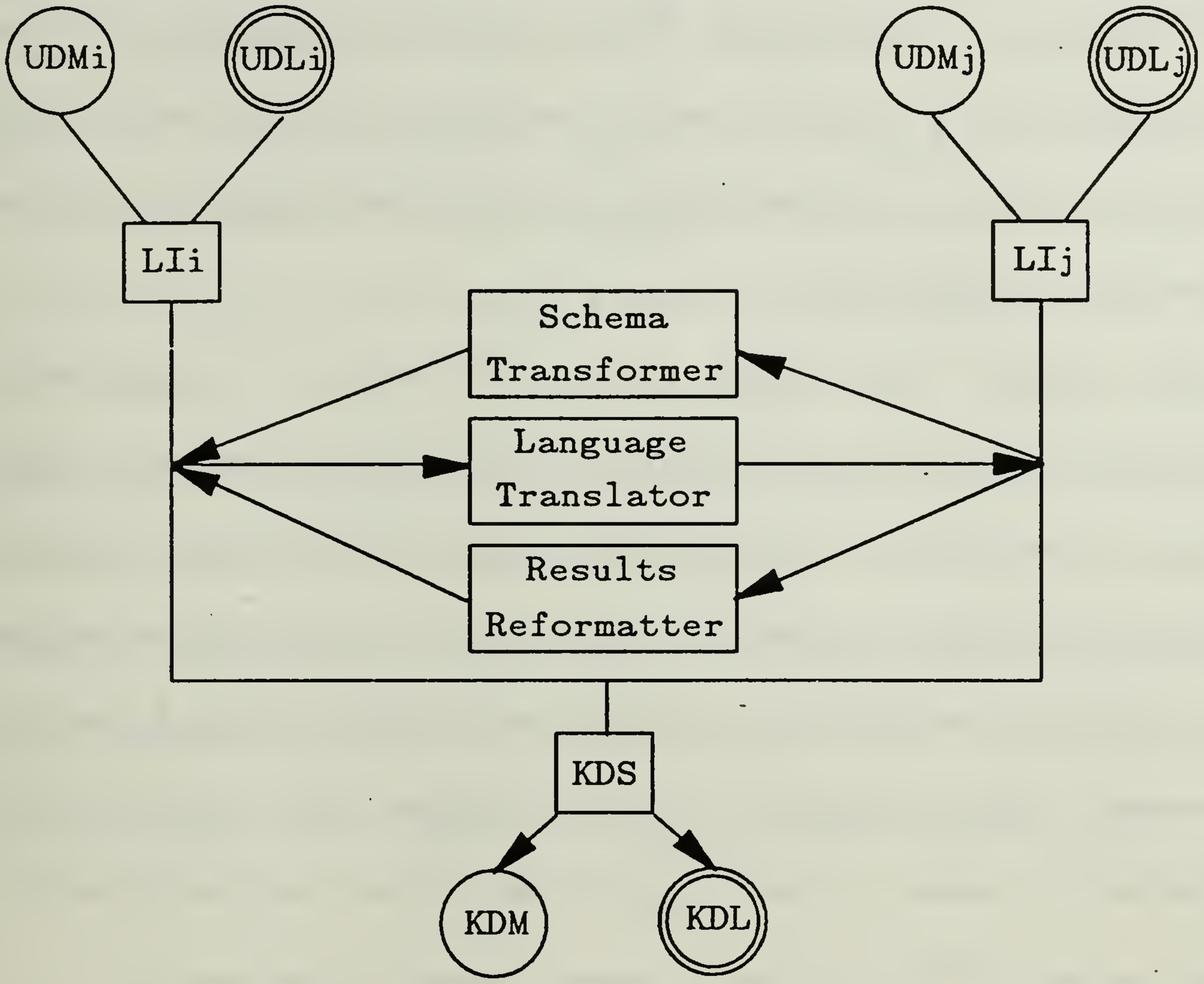


Figure 6. The Postprocessing Strategy

strategy is in the location of modifications. This approach deals with the kernel database system and as such, we can expect the focus of programming activity to be in this area. In the current implementation of MLDS, the kernel database schemas are not visible to the individual language interfaces and implementation of this strategy would force a major design change in the interaction between the kernel database system and the language interfaces. Additionally, the kernel database system was designed as an independent, stand-alone system upon which the MLDS language interfaces were added as a functional enhancement. We find it inadvisable, at this point, to attempt to combine the code of these two large projects by coding in an interdependency between specific language interfaces and the kernel database system.

The remaining two strategies, on the other hand, can be implemented completely at the language interface layer in MLDS. The preprocessing method appears, at least initially, to be conceptually easier to understand and implement by simply converting the transactions input in one data manipulation language into equivalent transactions of a second data manipulation language for access to a database. Upon investigation however, it becomes clear that the task of translating the syntax of a data manipulation language into the syntax of a second language while maintaining the semantic meaning is far from simple.

Additionally, we can expect the overall performance of the preprocessing method to be less than that of the mixed-processing method. In the preprocessing strategy, cross-access of databases requires a schema transformation, two language translations and two reformatting of results. On the other hand, cross-access via the mixed-processing strategy requires one schema translation, one language translation and one reformatting of results. It is clear that less processing activity is needed by the mixed-processing approach resulting in increased performance.

One final point in favor of the mixed-processing method deals with the amount of new code needed and modification to existing code. As outlined earlier in this chapter, the preprocessing strategy requires three components to be implemented, a schema transformer, language translator, and results reformatter. These are new software modules to be added to the language interface level. Modifications to the current language interfaces would be relatively minor. The mixed-processing strategy, on the other hand, would require extensive modification to the existing language interface to handle the cross accessing of databases, however, this code would be very similar to the code in the current LI making the implementation task much simpler. We would expect that the amount of code required to implement the mixed-processing strategy to be between one-half to two-thirds of that required to implement the preprocessing strategy.

IV. TRANSFORMING HIERARCHICAL SCHEMAS TO RELATIONAL

A. DESIGN

Having selected the mixed-processing strategy as the most appropriate for the MMDS design, the first step in implementing this approach, as outlined in the previous chapter, is to perform a schema transformation from a hierarchical database to a relational database. This process involves the translation of the relationships implicit in the hierarchical database to their functional equivalents in the relational model.

In order to describe this transformation, a sample hierarchical database will be used to illustrate the process. Figure 7 shows the DL/I definition of the sample database. The first line identifies the database name as 'schooldb'. There are five segments defined in this database (Course, Prereq, Offering, Teacher, and Student). The Course segment is the parent of both the Prereq and Offering segments as specified in the definitions of the Prereq and Offering segments in Figure 7. The Offering segment in turn is the parent to the Teacher and Student segments. Figure 8 depicts the hierarchical nature of the relationships.

The fields (attributes) of each segment follow the individual segment definitions in Figure 7. The first field of each segment is defined as the sequence field for that segment. This is a required field and must have an associated value

```
dbd    name= schooldb


segm   name= course
field  name= (cnum, seq), bytes = 4
field  name= ctitle, bytes = 10
field  name= descripn, bytes = 10


segm   name= prereq, parent = course
field  name= (pnum, seq), bytes = 4
field  name= ptitle, type= char, bytes = 10


segm   name= offering, parent= course
field  name= (date, seq), type = char, bytes = 4
field  name= location, bytes = 8
field  name= format, bytes = 6


segm   name= teacher, parent = offering
field  name= (tnum, seq), type = char, bytes = 4
field  name= tname, bytes = 10


segm   name= student, parent = offering
field  name= (snum, seq), bytes = 4
field  name= sname, type= char, bytes = 10
field  name= grade, bytes = 1
```

Figure 7. Hierarchical Database Definition

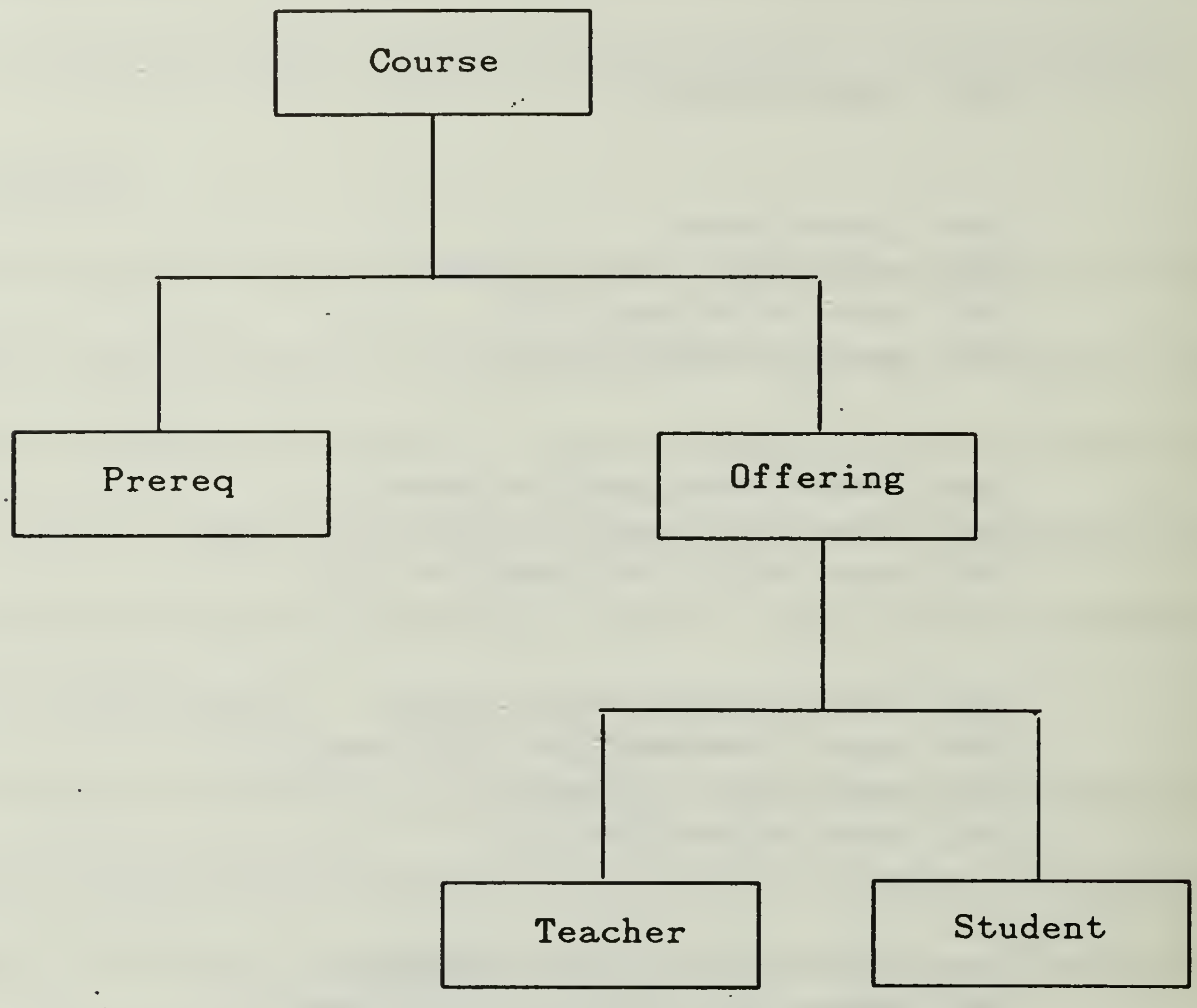


Figure 8. A Hierarchical Database Tree Structure

for each record entered into the database. Figure 9 shows the logical structure of the sample database with segment and field definitions.

A relational database model is often referred to as a 'flat' database because there are no structural relationships between tables as there are in a hierarchical or network database model. Each table is an independent data entity. Explicit,

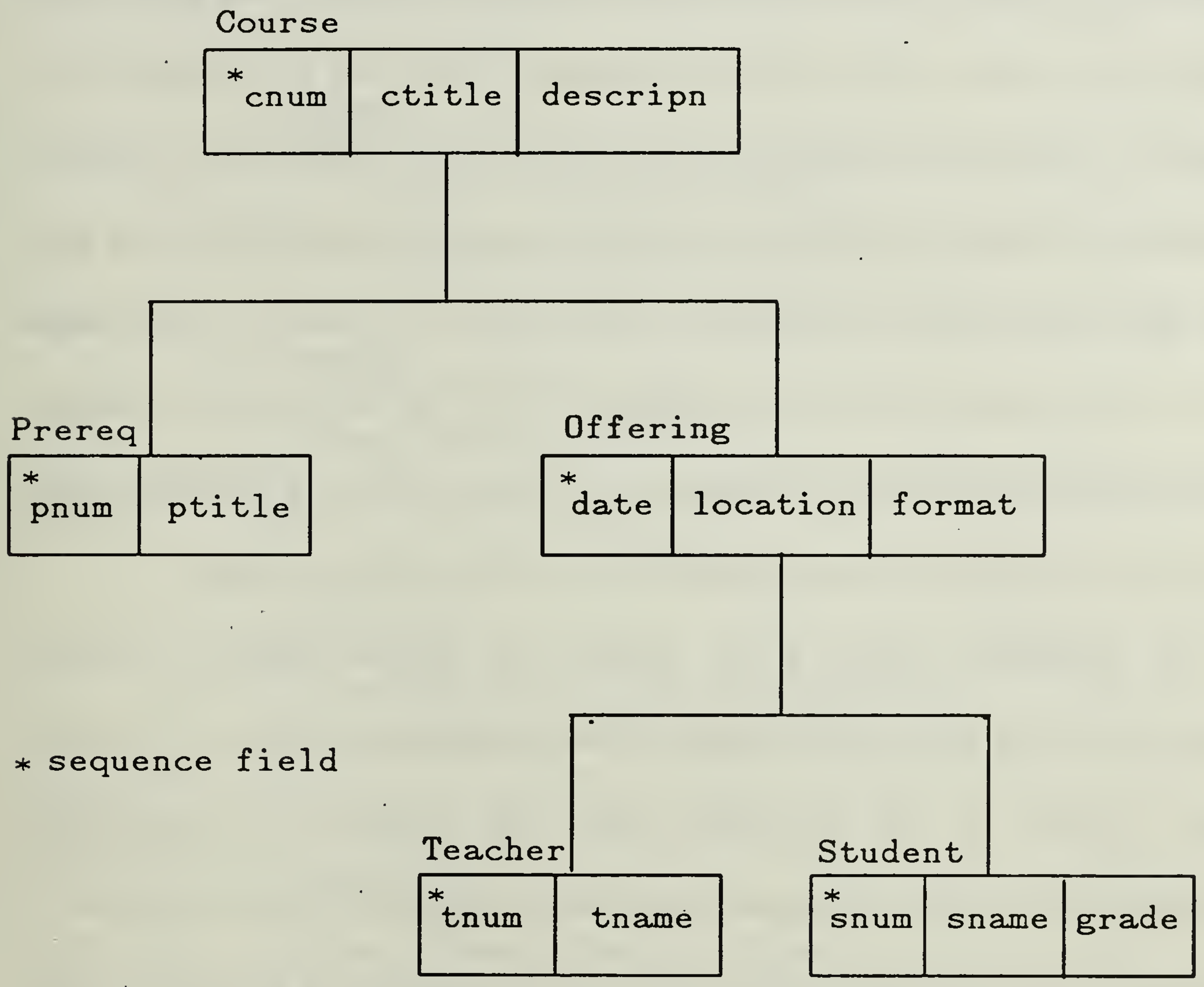


Figure 9. Logical Data Structure of the Hierarchical Database

logical relationships are formed through data manipulation language constructs such as JOIN and VIEW, but these are not part of the database schema itself.

The key issue in the schema transformation from a hierarchical database to a relational database then is the representation in the relational schema of the parent-child relationships between segments in the hierarchical database. There are two relatively direct methods of performing this transformation. The first method is to create a new table for each relationship desired. This table would contain the sequence fields of the two segments which are to be related. For example, in the sample database, we could create a new table called 'taught-by' to relate the Offering and Teacher tables and include the Date and Tnum from each table respectively. This method would provide the necessary relationships but at the expense of many additional tables in the database schema. Additionally, queries against the database would tend to be long and complicated for even the simplest databases, making this a rather unyieldy solution.

The alternative method is to 'cascade' the sequence fields of ancestor segments into all descendent segments when transforming them to the table format required in the relational model. For example, the parent-child relationship in the sample database from Course to Offering can be represented by including the sequence field, Cnum, from the Course segment in the newly created Offering table. Subsequently, the full relationship between Course, Offering, and Student can be represented by cascading the Cnum field from the Course segment and the Date field from the Offering segment into the newly formed Student table. Figure 10 depicts the relational database schema of the sample database following a transformation using this cascade method.

The obvious disadvantage of this technique is the additional space requirement necessary for the duplication of the sequence fields. The primary advantages, and the reasons that this method has been selected for

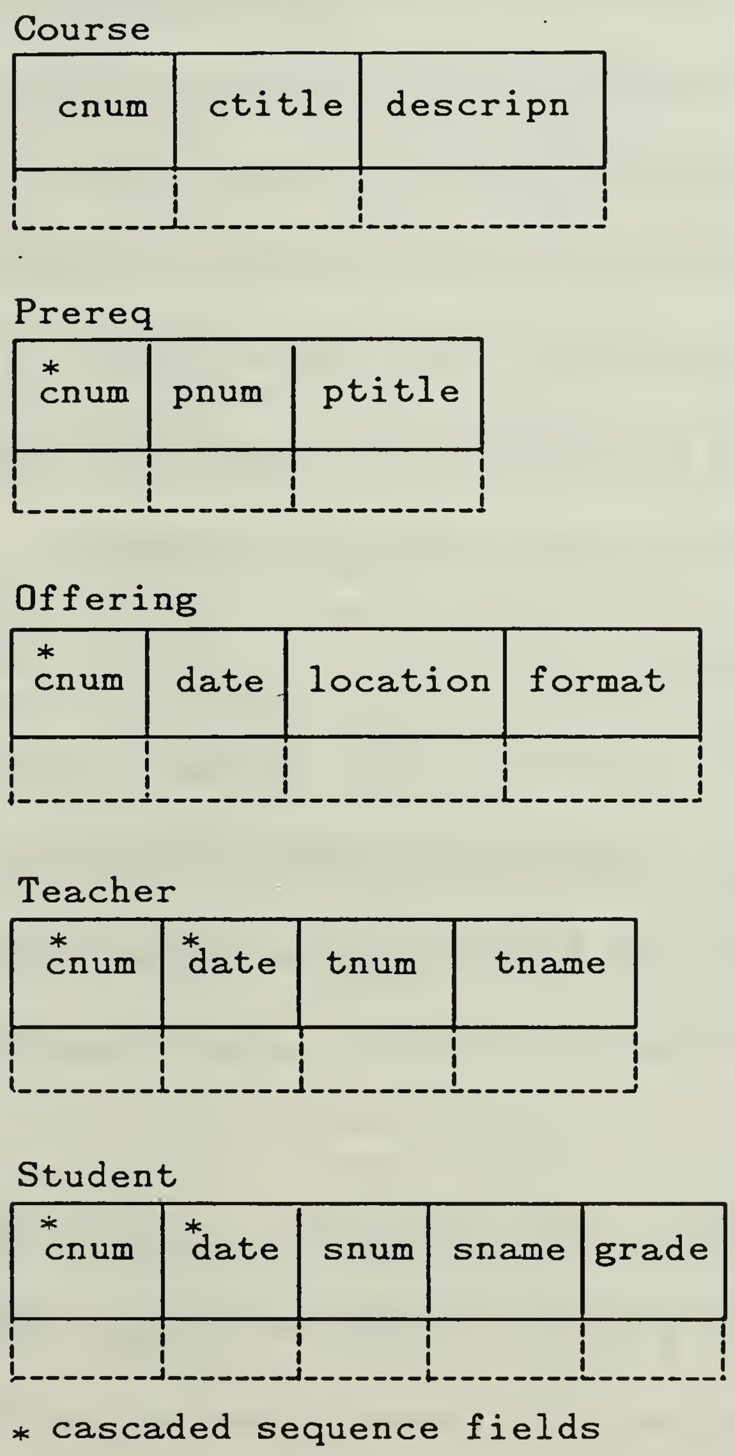


Figure 10. The Logical Structure of the Relational Schema

implementation, are that queries against the database are shorter, and less complicated because of the additional information within each table and that this method mirrors the transformation made in the AB(hierarchical) schema, making language translation algorithms much more efficient and straight-forward. Figure 11 details textually, the structure of the transformed schema. It should be noted that the cascaded sequence fields are represented as KEY attributes in each of the relational tables, indicating that a value must be specified for these attributes. This becomes essential in maintaining the integrity of the hierarchical database when data manipulation is performed using SQL transactions.

B. IMPLEMENTATION

The remainder of this chapter and the next chapter focus on the implementation of the mixed-processing strategy. The details of the schema transformation are presented in this chapter and the translations of SQL queries to AB(hierarchical) transactions in order to access a hierarchical database are detailed in Chapter V.

The implementation of the cross-access functionality involved extensive modification to selected portions of the relational language interface, specifically, the language interface layer (LIL), kernel mapping system (KMS), and kernel controller (KC). No attempt has been made to completely describe the language interface procedures and data structures. Instead, an overview of the major processes is presented for clarity and understanding, with emphasis on the areas of

```
database name = SCHOOLDB, number of relations = 5
database type = HIERARCHICAL

relation_name = COURSE, number of attributes = 3
        attr name = CNUM      , type = s, length = 4 , key = TRUE
        attr name = CTITLE    , type = s, length = 10, key = FALSE
        attr name = DESCRIPN  , type = s, length = 10, key = FALSE

relation_name = PREREQ, number of attributes = 3
        attr name = CNUM      , type = s, length = 4 , key = TRUE
        attr name = PNUM      , type = s, length = 4 , key = TRUE
        attr name = PTITLE    , type = s, length = 10, key = FALSE

relation_name = OFFERING, number of attributes = 4
        attr name = CNUM      , type = s, length = 4 , key = TRUE
        attr name = DATE      , type = s, length = 4 , key = TRUE
        attr name = LOCATION  , type = s, length = 8 , key = FALSE
        attr name = FORMAT    , type = s, length = 6 , key = FALSE

relation_name = TEACHER, number of attributes = 4
        attr name = CNUM      , type = s, length = 4 , key = TRUE
        attr name = DATE      , type = s, length = 4 , key = TRUE
        attr name = TNUM      , type = s, length = 4 , key = TRUE
        attr name = TNAME     , type = s, length = 10, key = FALSE

relation_name = STUDENT, number of attributes = 5
        attr name = CNUM      , type = s, length = 4 , key = TRUE
        attr name = DATE      , type = s, length = 4 , key = TRUE
        attr name = SNUM      , type = s, length = 4 , key = TRUE
        attr name = SNAME     , type = s, length = 10, key = FALSE
        attr name = GRADE     , type = s, length = 1 , key = FALSE
```

Figure 11. Textual Representation of the Relational Schema

significant modification. A logical rather than procedural view is provided in describing the flow of program control to eliminate unnecessary detail. A complete discussion of the relational language interface implementation can be found in Kloepping and Mack [9].

1. Language Interface Data Structures

When a relational user logs onto MMDS, a number of existing data structures are present that contain information relevant to that, and all other, users. The first of these is the dbid_node depicted in Figure 12. This structure points to the linked list of database schemas that have previously been defined in each of the language interfaces. It is through this data structure that a user has access to all of the database currently within the system.

The rel_dbid_node pointer identifies the first relational database schema. The central data structure for each schema is the rel_dbid_node as shown in

```
union dbid_node
 {
   struct   rel_dbid_node    *rel;
   struct   hie_dbid_node    *hie;
   struct   net_dbid_node    *net;
   struct   ent_dbid_node    *ent;
 }
```

Figure 12. The dbid_node Data Structure

Figure 13. This structure contains the database name, number of relations, pointers to the first and current relations, and a pointer to the next database schema. The rel_dbid_node structure has been modified to contain an additional field called dbtype that is used to identify the original database model in which the schema was created. For example, a schema transformation from a hierarchical model would include an HIE identifier in this field. Additional data structures pointed to by the rel_dbid_node structure completely specify the database schema.

A number of data structures are also created that are specific to the new relational user. The first of these is the user_info data structure shown in Figure 14. This structure uniquely identifies the new user in a multi-user environment and points to the data structures created for the exclusive use of that user. A

```
struct rel_dbid_node
 {
  char                          name[DBNLength + 1];
  int                           num_rel;
  struct   rel_node             *first_rel;
  struct   rel_node             *curr_rel;
  struct   rel_dbid_node *next_db;
  int      dbtype
 }
```

Figure 13. The rel_dbid_node Data Structure

```
struct user_info
 {
  char                          uid[UIDLength + 1];
  union    li_info              li_type;
  struct   user_info            *next_user;
 }
```

Figure 14. The user_info Data Structure

pointer links this information to the list of data structures associated with all other system users.

Figure 15 depicts the sql_info structure. This is the central data structure created for a relational user and contains much of the information or pointers to information used throughout the user session.

2. The Schema Transformation

The Language Interface Layer (LIL) is the primary control module from which all other modules are called. It has been designed to be menu-driven by inputs from the current user. It is through the LIL that a user can load new databases, select previously created databases for processing, and access databases by generating and selecting SQL transactions. Control always returns to the LIL following any of these operations. The user may end the current session and return to the operating system by making an appropriate choice from the top level menu.

```
struct sql_info
 {
   struct      curr_db_info    curr_db;
   struct      file_info       file;
   struct      ran_info        sql_tran;
   int                         operation;
   struct      ddl_info        *ddl_files;
   struct      tran_info       *abdl_tran;
   union       kms_info        kms_data;
   union       kfs_info        kfs_data;
   union       kc_info         kc_data;
   int                         error;
 }
```

Figure 15. The sql_info Data Structure

As previously mentioned, when a new user logs into the system, a number of user-specific data structures are created and initialized. These structures provide the temporary storage necessary for performing various database operations and holding returned results. The first menu presented to the relational user pertains to database selection:

```
Enter type of operation desired
    (l) - load a new database
    (p) - process old database
    (x) - return to the operating system

ACTION ----> _
```

At this point, the user may choose to load a new database schema, in which case he is prompted to enter the database name and set of creates, or process an already existing schema. If the user chooses to process an existing database schema, he is prompted for the database name. The program will attempt to locate the desired database schema by traversing the linked list of relational rel_dbid_node data structures described earlier. If found, the schema is loaded, and query processing may begin.

If the desired database schema was not found, the constraints of MLDS dictated that the user be presented with an error message and prompted to enter a different database name. Under MMDS however, processing does not stop. Instead, the program searches all other language interfaces for a matching database name and, if found, copies and transforms the located schema to a functional equivalent that can be used for access to the associated database via SQL transactions. It should be noted that this is the first thesis dealing with schema transformations to the relational model, therefore, only the hierarchical model transformation has been implemented to date.

If the user has selected a hierarchical database for processing, a new relational schema is created based on the desired hierarchical schema. The relational data structures previously discussed have functional equivalents in all other language interfaces. It is through these structures that the transformation is accomplished.

Initially, a new rel_dbid_node is created and attached to the end of the linked list of existing relational schemas. The hierarchical database name is then inserted, the number of relations is set equal to the number of segments in the hierarchical schema, and the schema is tagged as a hierarchical equivalent by setting the dbtype variable to 'HIE'. A new data structure, rel_node, shown in Figure 16, is created and attached to the schema. This structure describes each of the relations in a database and is initialized with information available from the equivalent segment data structures in the hierarchical schema. The relation name is set equal to the hierarchical segment name and pointers are set to the first attribute of the relation and to the next relation, if any, in the schema.

Each attribute in a relation is fully described by a rattr_node data structure as depicted in Figure 17. Initially, an attribute node is created for each field in a hierarchical segment. The attribute name, type, and length are

```
struct rel_node
 {
  char                       name[RNLength + 1];
  int                        num_attr;
  struct    rattr_node       *first_attr;
  struct    rattr_node       *curr_attr;
  struct    rel_node         *next_rel;
 }
```

Figure 16. The rel_node Data Structure

```
struct rattr_node
 {
  char                         name[ANLength + 1];
  char                         type;
  int                          length;
  int                          key_flag;
  struct    rattr_node         *next;
 }
```

Figure 17. The rattr_node Data Structure

transferred directly from the hierarchical field node. If the field is a sequence field, the attribute is tagged as a key attribute in the relational schema. Attributes in a relation are linked via the 'next' pointer.

At this point, the sequence fields are 'cascaded' into the relation. This is accomplished by traversing the hierarchical schema from the current segment to the root segment and creating an attribute node from the sequence field of each segment visited. This traversal is possible because each segment node contains, among others, a pointer to its parent segment.

Following this operation, the number of attributes is set equal to the number of fields in the associated segment plus the number of cascaded sequence fields. Processing continues with subsequent relations until the schema is completed. Control is then returned to the LIL for further access and manipulation of the database.

V. MAPPING SQL STATEMENTS TO AN AB(HIERACHICAL) DATABASE

The previous chapter detailed the schema transformation process necessary for the implementation of the mixed-processing strategy. The remaining major component required for the cross-access of a hierarchical database through SQL transactions is the new language interface (LI). One of the primary purposes of this component is to map the SQL queries input by the relational user to equivalent AB(hierarchical) transactions. This chapter describes the design and implementation of this component.

A. THE LI TRANSLATION PROCESS

As stated previously, each database schema created within a given model is transformed into an equivalent schema in the kernel attribute-based model. This AB schema has unique, embedded structures that ensure that the attribute-based database is the functional equivalent of the user defined model. For this reason, the LI must provide a language translation from transactions in the user data manipulation language to the attribute-based transactions specific to that language, e.g., from SQL to AB(relational). To provide a relational model user access to a hierarchical database, then, it is necessary to create a second language interface that will translate SQL transactions to their AB(hierarchical) equivalents.

This second language interface can be implemented in one of two ways. The first method is to create an entirely separate language interface (LIL, KMS, KC, and KFS), and branch to the appropriate version based on database selection by the user. The alternate approach is to modify the current language interface in such a way that program execution will branch to the appropriate translation and processing activities based on user input. In this manner, a new language interface can be *logically* created without duplication of a large amount of similar code. This reduction in code size was the primary factor in choosing to modify the existing code in the implementation of the cross-access capability.

1. Query Processing in the LIL

After the user has loaded a new database or selected an existing database for processing, he is prompted for the mode of query input as follows:

```
Enter mode of input desired
     (f) - read in a group of transactions from a file
     (t) - read in transactions from the terminal
     (x) - return to the previous menu

ACTION ----> _
```

The user now has the option of reading in a group of queries from a prepared file or directly entering the queries from the terminal. Regardless of the input method selected, processing continues in an identical manner. The list of transactions are displayed on the terminal, each preceded by an identifying number. The user is then presented with the following execution menu:

```
Pick the number or letter of the action desired
     (num) - execute one of the preceding queries
     (d)   - redisplay the list of queries
     (x)   - return to the previous menu

ACTION ----> _
```

At this point, the user can select a query for processing. Since each query is an independent entity, the order of processing is not important. Following each selection, the query is sent to the kernel mapping system (KMS) for translation, and then to the kernel controller (KC) for execution. Results, if any, are displayed to the user and the execution menu is re-displayed for further commands.

2. Query Processing in the KMS

SQL transactions are sent to the KMS from the LIL. The function of the KMS is two-fold, to parse the SQL query and verify its syntax, and to translate the query into an equivalent ABDL transaction. If the SQL query is determined to be valid, the ABDL is passed to the kernel controller for processing and execution by the MBDS.

The primary component within the KMS is the transaction parser. It has been implemented within MLDS by use of the UNIX utility Yet-Another-Compiler Compiler (YACC). YACC is a program generator that performs syntactic processing on an input stream of tokens. The compiler utilizes a set of grammar rules input by the programmer to generate a program that will parse a token stream and perform operations based on the recognition of the patterns within the input stream. The YACC-produced parser is a finite- state automata that performs a top-down parse. Parsing begins through the upper-levels of the grammar hierarchy and proceeds through the lower levels in a search for matches to the input tokens. As tokens and token strings are recognized, portions of the output code are executed. Processing may traverse up and down the grammar hierarchy as the parser attempts to recognize the input string by satisfying the grammatical rules. If the entire token string has been processed and associated with grammar rules, parser execution will terminate normally, otherwise, a syntax error is reported, the parser will abort, and control will return to the calling procedure.

In addition to the information provided through the data structures from the LIL, The KMS uses, primarily, five data structures during the parsing operation. These are outlined briefly for completeness. Figure 18 shows the rel_kms_info structure. This structure holds information for delayed use in the parsing process. The target list holds attribute names used in Select and Insert operations, the template records stores the names of the relations being accessed, and the insert list maintains attribute values used in Insert requests. The next two fields are character strings. The temp_str is used to store intermediate translation results and the join_str holds the translation of the second retrieve request of a join operation. The next_nest field is utilized only during the parse of a nested Select transaction, and is a pointer to the next rel_kms_info structure in the list. The final field, alt_tgt, has been added during the current implementation to hold information relating the translation to AB(hierarchical) statements.

```
struct rel_kms_info
 {
 struct          target_list_info    *first_tgt;
 struct          templates_info      templates;
 struct          insert_list_info    *first_val;
 char                                *temp-str;
 char                                *join_str;
 struct          rel_kms_info        *next_nest;
 struct          alt_list_info       *alt_tgt;
 }
```

Figure 18. The rel_kms_info Data Structure

Figure 19 depicts the four data structures pointed to by the rel_kms_info structure. They are used to represent the target list of attribute names, the names of the relations (templates) being accessed, a list of attribute values for the Insert request, and a list of attribute names, values, and operations used in the AB(hierarchical) translations, respectively. Further details on the KMS and its data structures can be found in Kloepping and Mack [9].

As described, the KMS parser is the central component in the translation of SQL transactions to ABDL statements, hence the modifications for the cross-access implementation deal primarily with that construct. The code changes principally involve branching subroutines that alter the code generation process when a grammar rule is recognized within the parser. The remaining part of this chapter describes the implementation details involved in mapping the four

```
target_list_info
 {
 char                                    name[ANLength + 1];
 char                                    tgt_rel[RNLength + 1];
 struct          target_list_info        *next_attr;
 }

templates_info
 {
 char                                    name1[RNLength + 1];
 char                                    name2[RNLength + 1];
 }

insert_list_info
 {
 char                                    *value;
 struct          insert_list_info        *next_val;
 }

alt_list_info
 {
 char                                    name[ANLength + 1];
 char                                    op[RNLength + 1];
 char                                    *value;
 struct          alt_list_info           *next_attr;
 }
```

Figure 19. KMS Parser Data Structures

primary SQL transactions (Select, Insert, Delete, and Update) to their AB(hierarchical) equivalents.

B. THE SELECT STATEMENT

The SQL Select statement is used to retrieve information from a database. Since this statement does not alter the database in any way, it can be used, without modification, to access an AB(hierarchical) database directly. This is possible because the original hierarchical schema has been mapped to an attribute-based schema in essentially the same manner in which the schema transformer from hierarchical to relational has been implemented for the cross-access capability. Figure 20 is an example of a Select transaction issued against the sample database, and the equivalent ABDL transaction issued against the hierarchical database. As seen in this example, the cascaded sequence fields are visible to the relational user and can be used in composing queries against the hierarchical database. The desired functionality is complete. A relational user can directly access the segments of a hierarchical database as if they were a set of relational tables. The various versions of the Select statement, such as nested selects, are fully supported.

```
SELECT tnum, tname
FROM Teacher
WHERE cnum = 'C100'

RETRIEVE ((TEMP = Teacher) and (CNUM = C100)) (TNUM, TNAME)
```

Figure 20. A SQL Select Transaction

C. THE INSERT STATEMENT

The purpose of the SQL Insert statement is to add information to an existing database. This statement modifies the database so steps must be taken to ensure that the integrity of the hierarchical database is maintained when this operation is invoked. Although the relational user is viewing the hierarchical database as a collection of independent tables, The parent-child relationships within the hierarchy must be preserved.

1. Design

One of the primary constraints on a hierarchical database, as stated in Chapter II, is that no occurrence of a child record type can exist without its parent. Since the relational user is not constrained by relationships between tables, it is the responsibility of the new language interface to ensure that these relationships are maintained.

To be more specific, whenever a record is inserted into a hierarchical database, related records must already exist in all of the ancestor segments associated with the segment receiving the new record. Using the sample database as an example, suppose a relational user wanted to execute the following Insert transaction:

```
INSERT INTO Student (cnum, date, snum, sname, grade)
<'C100', '0787', '0284', 'Miller', 'A'>
```

In order to remain within the constraints of the hierarchical model, the ancestor record occurrences shown in Figure 21 must already exist in a hierarchical tree of the database.

2. Implementation

There are two basic methods of assuring that these ancestor records exist prior to executing the Insert request. The first approach is to program the

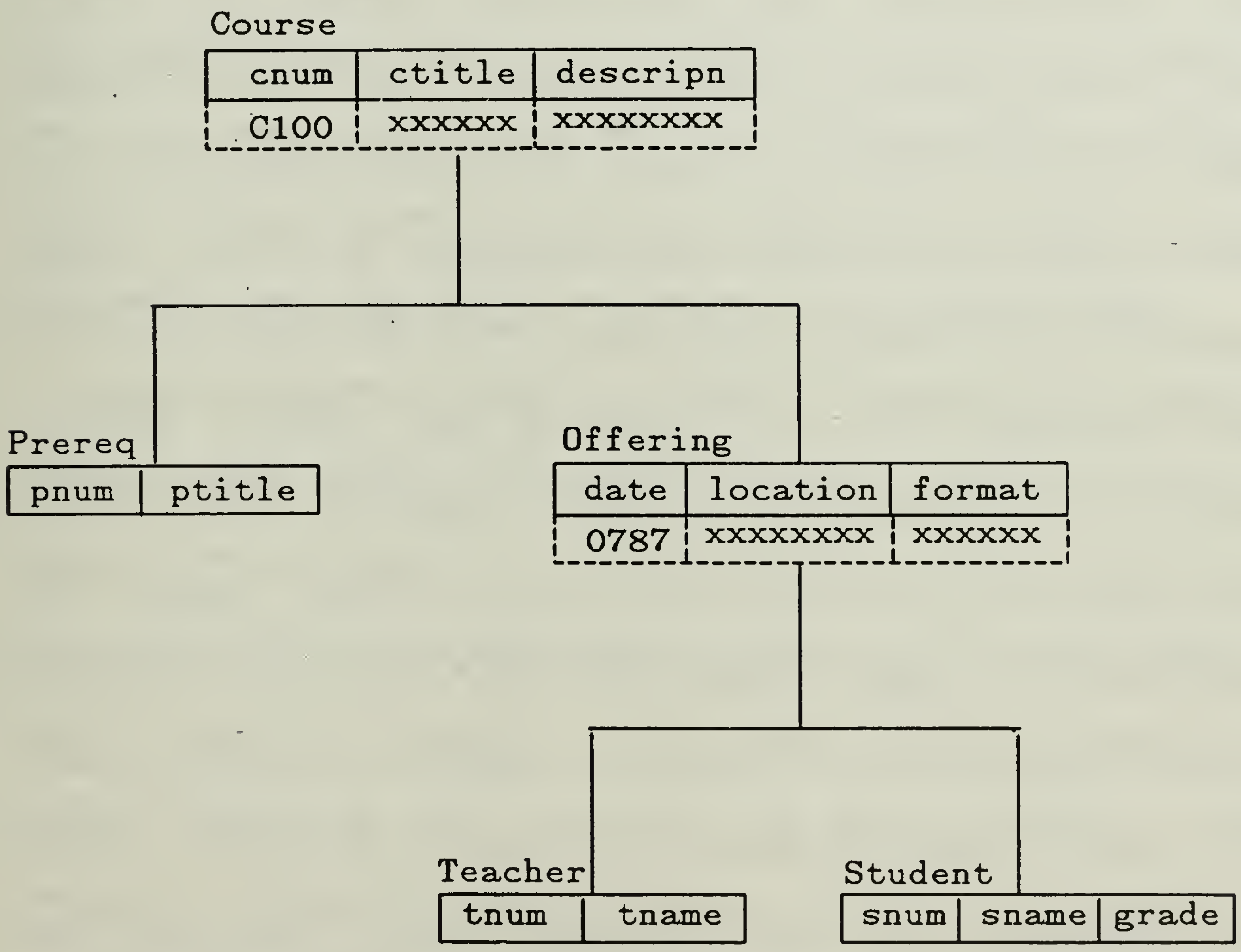


Figure 21. Hierarchical Database Prior to a SQL Insert

language interface to automatically insert the ancestor occurrences, if they do not already exist in the database, using information generated in the parse of the insert statement. This method would be entirely transparent to the relational user. That is, the insert would always be performed because the constraints on the database are being managed by the language interface. The overriding disadvantage of this approach is that the parsed Insert statement does not contain enough attribute information to fully specify the ancestor occurrences and it would be necessary to add dummy values into the unspecified fields. The user would then need to perform an Update transaction on each of the fields holding dummy values so that they can be replaced with the proper values. The second method, although less transparent, reduces significantly the number of required transactions. In this approach, the language interface determines if the proper ancestor occurrences exist prior to passing the Insert request to MBDS for execution. If the occurrences exist, the request is transmitted to MBDS, however, if the ancestor occurrences are not in the database, a message is displayed to the user stating that the proper hierarchical relationships do not exist, and informing the user of the hierarchical tree that must be completed. In terms of the example, if the necessary ancestor occurrences did not exist, the user would be informed that corresponding records need to be inserted into the Course and Offering Tables prior to executing the current Insert transaction, as shown in Figure 22.

As described above, it is necessary for the entire ancestor tree to be complete before an Insert operation can be executed, however, the implementation

```
INSERT NOT ALLOWED - in order to maintain the integrity of the
Hierarchical model, all ancestor segments/relations must contain
key-fields having the same values as the insert just attempted.
The ancestor relations and key-fields, from parent to root, are:
                OFFERING    DATE
                COURSE      CNUM
Inserts should be performed from the root down.
```

Figure 22. Response to Improper Hierarchical Insert Request

of this database integrity check can be accomplished by checking the immediate parent of the segment receiving the new record. This single check is sufficient because the remaining ancestor occurrences must already have existed at the time the immediate parent occurrence was inserted into the hierarchical database.

The implementation of this feature within the language interface required modification of the relational KMS and KC components. Within the KMS, the changes involved branching within the token stream parser. More specifically, the parse continues within the original language interface until the token stream is recognized as an Insert transaction. At this point, the KMS has generated the complete AB(relational) Insert request and the program branches to the subroutine that performs the logical translation to an AB(hierarchical) equivalent.

The first step to be performed within this subroutine is to search the original hierarchical schema for the segment receiving the new record. This segment data structure contains a pointer to the parent segment within the schema which will be used to check for the proper ancestor tree by creating a Retrieve request. Information contained in the Insert transaction and the parent segment will be used to build the request. The Retrieve request will be generated in all cases, except in the situation in which the segment receiving the new record is the root segment of the hierarchical database. In this case, the record can be inserted directly without further processing.

The Retrieve request is made against the parent segment, using the cascaded sequence fields and values obtained from the ABDL insert statement. Figure 23 provides an example using the sample database. The user wishes to insert a new record in the Student relation. The Retrieve that is generated will access the Offering relation, which is the immediate parent of the Student segment within the hierarchical schema.

Following the AB(hierarchical) translation, the parser completes its operations and control is returned to the LIL. The linked list of ABDL requests is

```
INSERT INTO Student (cnum, date, snum, sname, grade)
<'C100', '0787', '0284', 'Miller', 'A'>
------------------------------------------------------------
 [ RETRIEVE ((TEMP = Offering) and (CNUM = C100)
           and (DATE = 0787)) (CNUM) ]

[ INSERT (<TEMP, Student>, <CNUM, C100>, <DATE, 0787>,
        <SNUM, 0284>, <SNAME, Miller>, <GRADE, A>) ]
```

Figure 23. A SQL To AB(hierarchical) Insert Transaction

then passed to the kernel controller for execution. The KC acts as an interface to the MBDS and provides a temporary buffer for returned results. In this case, the KC recognizes that a hierarchical database is being accessed and branches to a routine that handles the request. The Retrieve statement is passed to MBDS and results are returned to a buffer. If one or more records were returned, the proper ancestor tree is in existence and the Insert transaction is transmitted to MBDS for processing. If, however, the return buffer is empty, the necessary ancestor occurrences do not exist and the Insert transaction is not sent to MBDS. The final step is to display the explanatory message (Figure 22) to the user and return control to the LIL for further processing.

D. THE DELETE STATEMENT

The SQL Delete statement modifies a relational database by removing one or more records from a single relation. As with the Insert statement, a database modification, when performed on a hierarchical database must ensure that the hierarchical model integrity is maintained. The primary task, then, in translating a SQL Delete to a AB(hierarchical) Delete transaction is to provide this integrity guarantee.

1. Design

The central difference between a Delete transaction on a relational database and a Delete transaction on a hierarchical database is that the relational operation affects only a single relation whereas the hierarchical operation may cause changes in multiple segments. The reason for this is that no occurrence is allowed to exist without a parent occurrence. For example, suppose a user performs the following Delete transaction on the database shown in Figure 24:

```
DELETE Course
WHERE Ctitle = 'Pascal'
```

If the single Course occurrence is deleted, the related occurrences in the Prereq and Offering segment, and in turn, the associated Teacher and Student occurrences would not be attached to a fully specified hierarchical tree. Therefore, in addition to deleting the specified Course record, it is necessary to delete all associated occurrences in the Prereq, Offering, Teacher, and Student segments as well.

2. Implementation

In order to accomplish this sequence of multiple deletes, a rather complex system of data structures is required in the KMS, and multiple buffering of intermediate results is necessary in the KC. These structures must handle traversal of the hierarchical path between segments and recursion of the ABDL transaction processing. None of these structures or capabilities currently exist within the relational language interface, so it would be necessary to program these components and integrate them into the existing interface. It is estimated that the code required to accomplish this would double the size of the current language interface. Additionally, the added processing required to initialize and update these components may adversely affect operating performance.

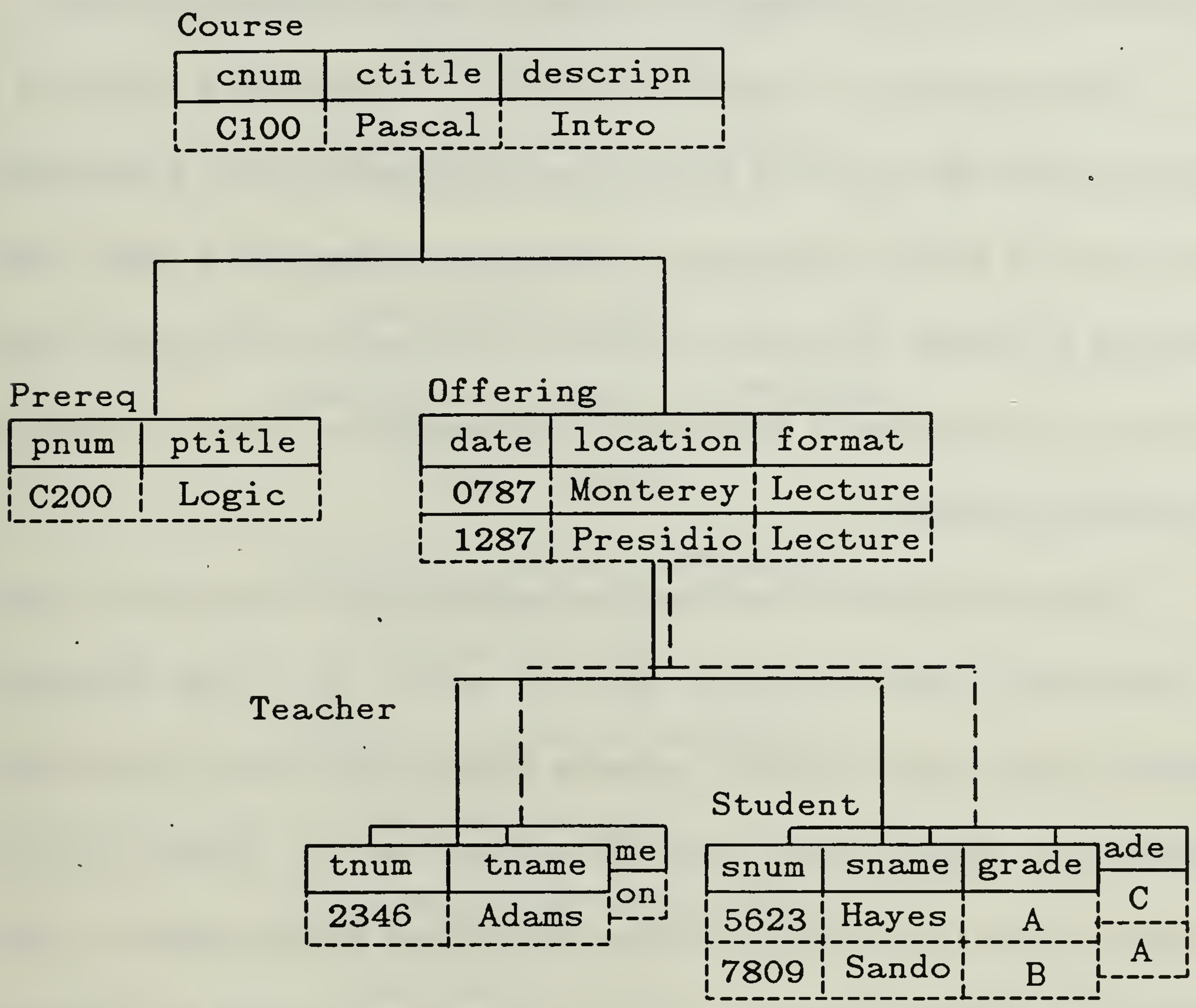


Figure 24. A Sample Hierarchical Database Prior to a Delete Operation

The hierarchical language interface does, however, contain the necessary data structures and functionality as a natural part of its hierarchy processing capability. As such, it is desirable to extend the concept of the new, logical language interface of the mixed-processing strategy to encompass portions of both the relational and hierarchical language interfaces. In this manner, it becomes possible to utilize the processing functions in the hierarchical interface to accomplish the desired operations without duplication and integration of code.

The composition of transactions necessary to accomplish a deletion is a function of where the occurrence is located in the hierarchical tree. If the deleted record is at the end of tree, i.e., in a leaf segment, then only a single delete transaction is required. If however, the deleted record occurs in any other portion of the tree, a combination of Retrieve and Delete operations may be necessary to accomplish the deletion.

Retrieve transactions are required as part of the Delete operations because the user-supplied Delete statement does not contain all of the information necessary to fully specify the entire sequence of delete transactions. The Retrieve statements are used to gather this information from the database and the returned values are then used to complete the required Delete statements. More specifically, a Retrieve is required at each level of the hierarchy and whenever processing switches to a different branch of the tree. Figure 25 depicts the sequence of transactions required to perform the previously mentioned SQL Delete statement on the sample hierarchical database. If Figure 24 represents the current composition of the database, then the first retrieve will return the Course occurrence <C100, Pascal, Intro>. The returned Cnum value of C100 will be used to complete the Delete statements on the Course and Prereq segments. At this point, processing in the branch containing the Prereq segment is complete and a

```
[ RETRIEVE ((TEMP = COURSE)
        and (CTITLE = Pascal))
        (CNUM) BY  CNUM ]

[ DELETE ((TEMP = COURSE)
        and (CNUM = **)) ]

[ DELETE ((TEMP = PREREQ)
    and (CNUM = **)) ]

[ RETRIEVE  ((TEMP = OFFERING)
        and (CNUM = **))
       (DATE) BY DATE ]

[ DELETE ((TEMP = OFFERING)
    and (CNUM = **)
    and (DATE = **)) ]

[ DELETE ((TEMP = TEACHER)
    and (CNUM = **)
    and (DATE = **)) ]

[ DELETE ((TEMP = STUDENT)
    and (CNUM = **)
    and (DATE = **)) ]
```

Figure 25. A Sample AB(hierarchical) Delete Transaction

switch is made to its sibling segment, Offering. Since information from an additional sequence field is required, a Retrieve operation is completed on the Offering segment, utilizing the C100 value from the previous Retrieve. The returned occurrences in this case are <0787, Monterey, Lecture> and <1287, Presidio, Lecture>. The Cnum value of C100 and the first returned Date value of

0787 are now used to complete the three remaining Delete transactions on the Offering, Teacher, and Student segments. Following execution of these Delete operations, the processing returns recursively to the Offering buffer and completes three more Delete transactions using the same C100 value for Cnum and the second value of 1287 for Date. These Deletes are subsequently sent to the MBDS for execution. Since there are no further occurrences in the Offering buffer, the current branch has been completely processed, and there are no other branches in the tree, the transaction is complete and processing terminates.

Implementation of this modification centered on the parser within the KMS of the relational language interface. During the parse on the user-specified transaction, a new data structure, alt_info, described earlier was used to accumulate a list of attribute names, operations, and attribute values recognized during the parse. When the parser identified the transaction as a Delete on a hierarchical database, program execution branched to a subroutine designed to perform the AB(hierarchical) translation. The subroutine discards the AB(relational) transaction and completely specifies the AB(hierarchical) transaction using information stored during the parse.

In order to build and execute a AB(hierarchical) Delete transaction, the data structures and functions of the hierarchical language interface are needed and it is at this point that these structures are created and initialized. The initial Retrieve and Delete statements are generated using the information in the alt_info data structures and the relational and hierarchical schemas. The remaining partially-specified Retrieves and Deletes are then built utilizing information from the hierarchical database schema. At this point, the transaction is complete and the KC is called to execute the sequence of operations.

The previous discussion has focused on a simple Delete to keep the details of the translation to an absolute minimum. The subroutine has been designed, however, to process Deletes with more complicated structures. For example, a Delete containing one or more OR'd predicates requires additional processing. Basically, each group of OR'd predicates is used to form a separate set of Delete transactions. After the first set has been executed, the individual statements are used as a template to form the next set of transactions. This continues until each group has been processed.

Following execution of the Delete transactions, the hierarchical data structures are released and the allocated memory returned to the operating system. The KMS then resumes processing and the relational data structures are re-initialized. Finally, control is returned to the LIL for the next user input.

E. THE UPDATE STATEMENT

The SQL Update statement is used to change attribute values within a relational database. Only a single value can be changed during each Update transaction. If more than one value is to be changed, a sequence of Update transactions must be sent to the MBDS. The hierarchical equivalent to the SQL Update statement is the REPLACE transaction.

1. Design

One of the major problems with the Update transaction is that a change to a sequence field in a hierarchical database may cause a loss of integrity in the database. That is, if a sequence field value is changed in a segment and, as a result, there is an incomplete ancestor tree relating to the new value, then one of the major constraints on the hierarchical database has been violated. This problem is enough of a concern that most hierarchical models, including IBM's IMS and MLDS, place a constraint on the Replace statement to the effect that changes can only be made to non-sequence fields. In order to remain consistent with the current model, this implementation remains within that constraint.

2. Implementation

Since attribute value restrictions are restricted to non-sequence fields, the Update translation from SQL to AB(hierarchical) can be handled within the relational language interface exclusively. The LIL passes the transaction to the KMS for translation to an ABDL statement. The parse continues within KMS until the token stream is recognized as an Update transaction on a hierarchical database. At that point, a subroutine is called that searches the database schema for the attribute being updated. If the attribute is determined to be a Key-attribute, the following error message is presented to the user:

```
UPDATE not allowed. The current implementation
of DL/I allows updates on NON-KEY fields only.
```

The remainder of the parse is aborted and control returns to the LIL for additional processing. If, however, the attribute is found to be a non-key field, then processing continues normally, and the AB(hierarchical) update transaction is passed to the KC for execution by the MBDS.

3. Additional Comments

Although most hierarchical model implementations do not allow changes to sequence fields, it may be possible to add this functionality to the MLDS language interface. It appears that the Update transaction can be handled in much the same manner as a Delete transaction. That is, when an Update transaction is issued against segment key field, a combination of Retrieve and Update statements can be generated that will modify the cascaded sequence field values of all descendent segments in the hierarchical tree. In addition to these statements, it would be necessary to perform an initial Retrieve on the immediate parent of the segment receiving the Update, using the new attribute value as a selection criteria. If one or more records are returned by this initial Retrieve, then the complete ancestor tree exists for the new value and the remaining Retrieve and update statements can be executed against the hierarchical database.

In order to achieve this capability, some relatively large modifications must be made in the hierarchical KMS parser and KC modules. Additionally, the current translation from the SQL Update statement to AB(hierarchical) transaction would need to modified accordingly. The major portion of this modification, involving the initial Update and cascaded Retrieves and Updates, has been written as part of the coding effort for this thesis and will be available if future thesis work involves additions to the functionality of the various language interfaces.

VI. CONCLUSIONS

The predominant approach to database design has been to implement a system based on a single database model and associated data manipulation language. This has proved to be an adequate, short-term solution, however, the lack of flexibility, capacity for expansion, and extensibility indicate the need for research into alternative approaches. One such approach has been designed and implemented at the Laboratory for Database Systems Research, Naval Postgraduate School, Monterey, California. The Multi-Lingual Database System (MLDS), as shown in Figure 26, allows a single database system to support multiple language models. Specifically supported models include relational/SQL, hierarchical/DL/I, network/CODASYL-DML, functional/Daplex, and attribute-based/ABDL. The system can easily be expanded to handle other database models and data manipulation languages.

Although MLDS allows access and manipulation of databases via five separate data models and languages, individual databases can be accessed only through the model within which it was created. For example, a relational database can only be accessed via the relational data model and the SQL data manipulation language. The extension of MLDS to support cross- access of all databases through any of the supported models is the current focus of research

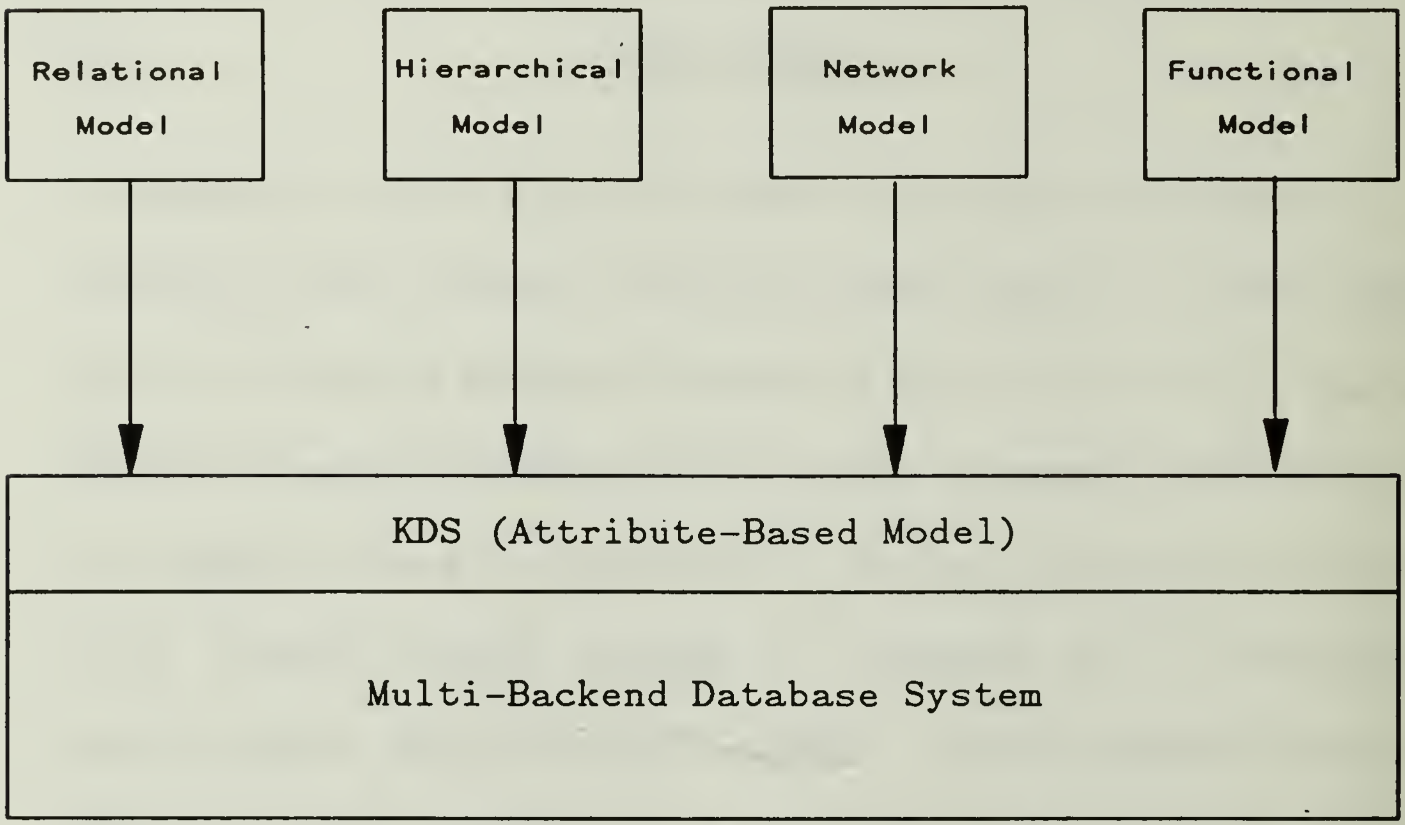


Figure 26. The Multi-Lingual Database System Concept

and design analysis. The design and implementation of one of the interfaces within the Multi-Model Database System (MMDS) has been the central topic of this thesis.

A. A REVIEW OF THE RESEARCH

The goal of the research documented in this thesis has been to increase the functionality of MLDS by allowing a database user knowledgeable only in the relational model to access and manipulate a hierarchical database through the use of SQL transactions. We presented and analyzed a number of design strategies for implementing this extension into MLDS, including the high-level preprocessing, mixed-processing, and postprocessing methods before selecting the mixed-processing strategy as the most feasible methodology.

In order to implement the mixed-processing strategy, two components, a schema transformer and a new language interface, had to be designed. We first discussed the design of a schema transformation algorithm from a hierarchical database to a relational database. The technique selected involved the cascading of hierarchical sequence fields into the relational schema to fully specify the parent-child relationships between hierarchical segments. We then described the data structures and implementation details necessary to integrate the schema transformer into the Language Interface Layer (LIL).

The design and implementation of the new language interface provides the means for accessing and manipulating a hierarchical database by the translation of SQL statements to their AB(hierarchical) equivalents. We discussed how it was possible to create a new, 'logical' language interface by modification of the relational interface and incorporation of the functionality of the hierarchical language interface into a framework that provides the cross-access capability. We then detailed the modifications necessary to the relational KMS and KC to implement the new language interface, and concluded by describing the translations of the SQL Select, Insert, Delete, and Update statements to the equivalent AB(hierarchical) transactions.

B. FINAL OBSERVATIONS

Figure 27 depicts the MMDS concept as a functional extension of MLDS. The earlier design [23] and implementation [24] of the capability to access a functional database using the network model and the CODASYL-DML data manipulation language, along with the work done in this thesis support and confirm the feasibility of the MMDS design. Additionally, this body of research provides the basis for designing alternate cross-access capabilities.

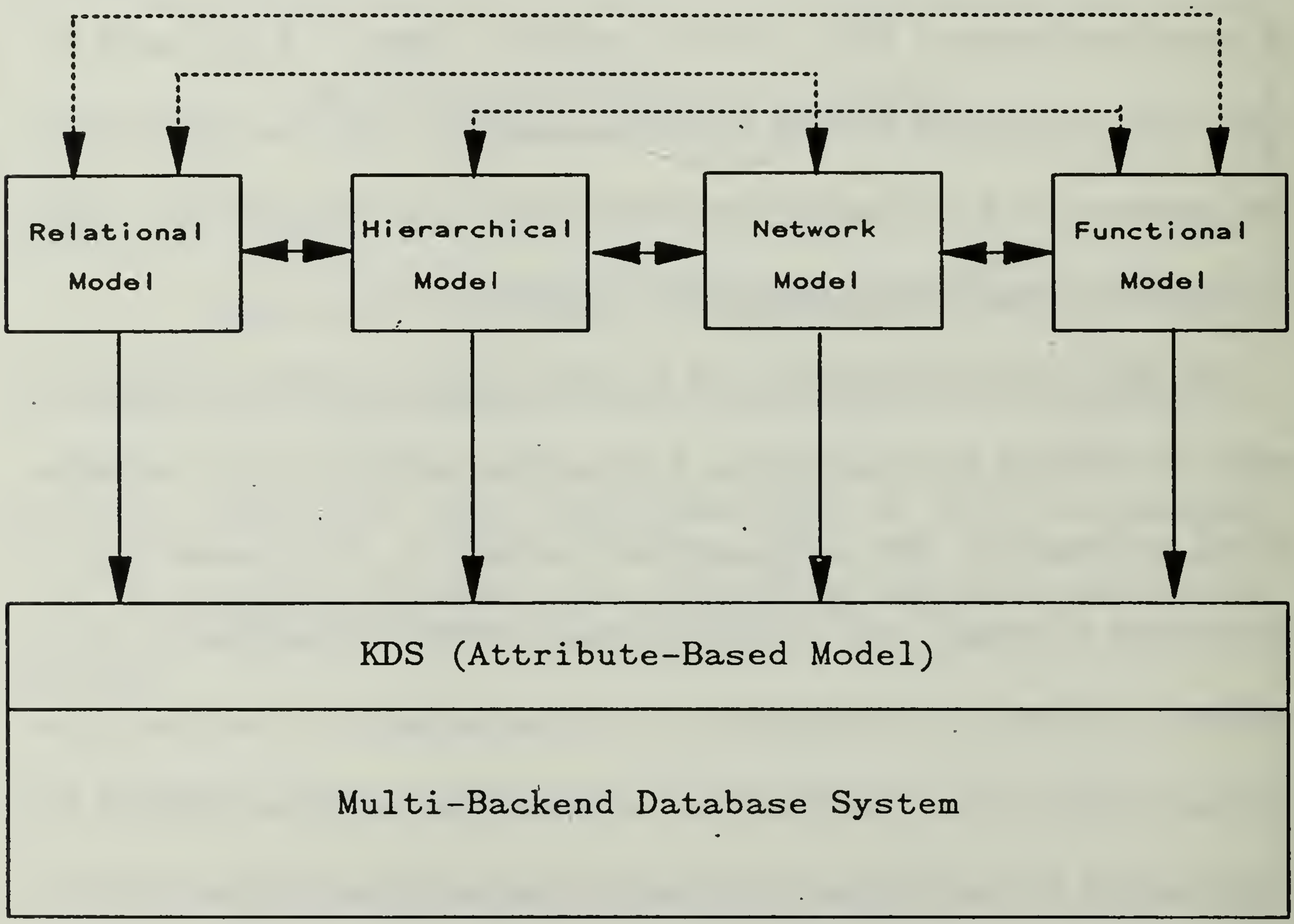
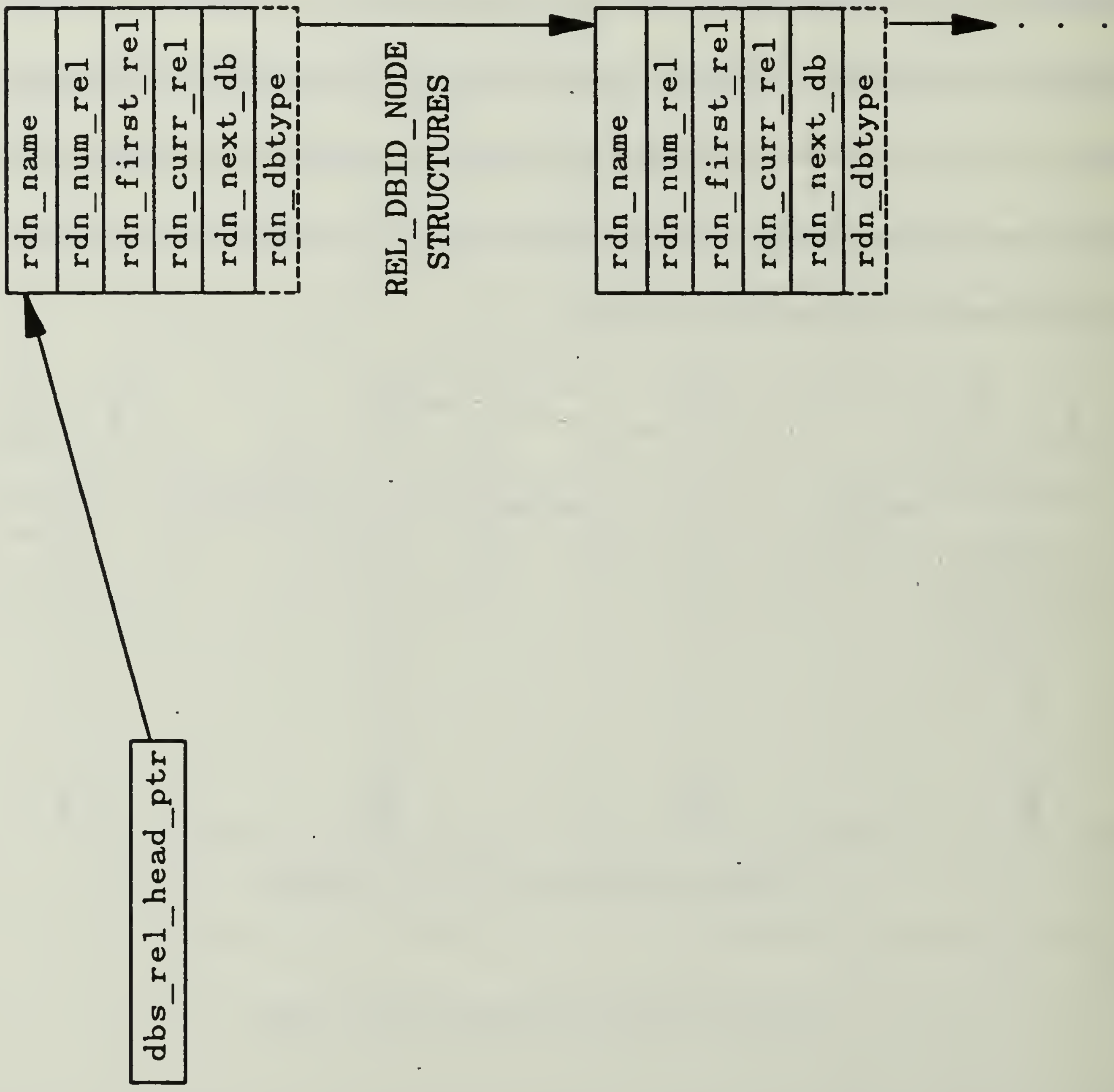
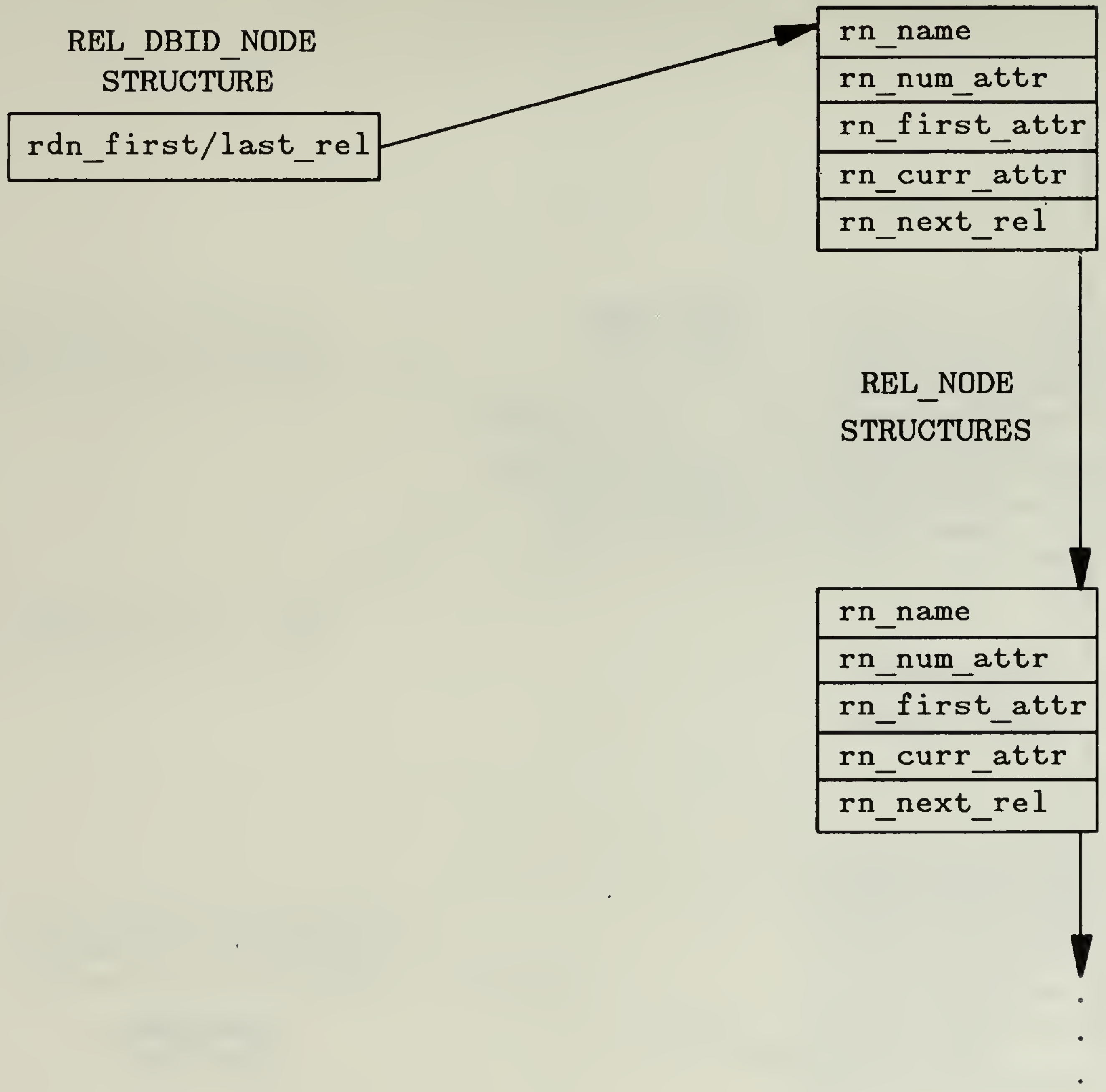


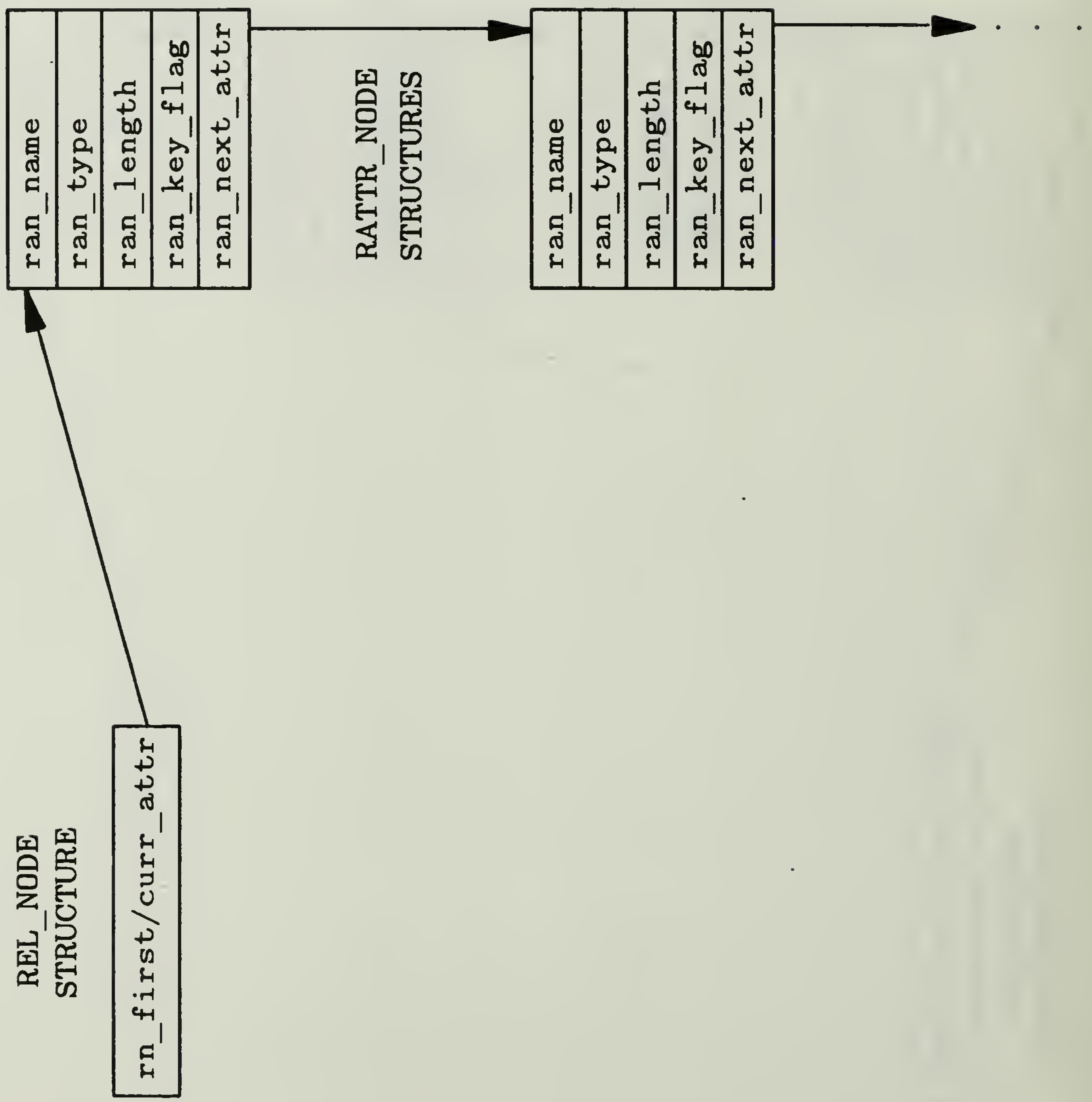
Figure 27. The Multi-Model Database System Concept

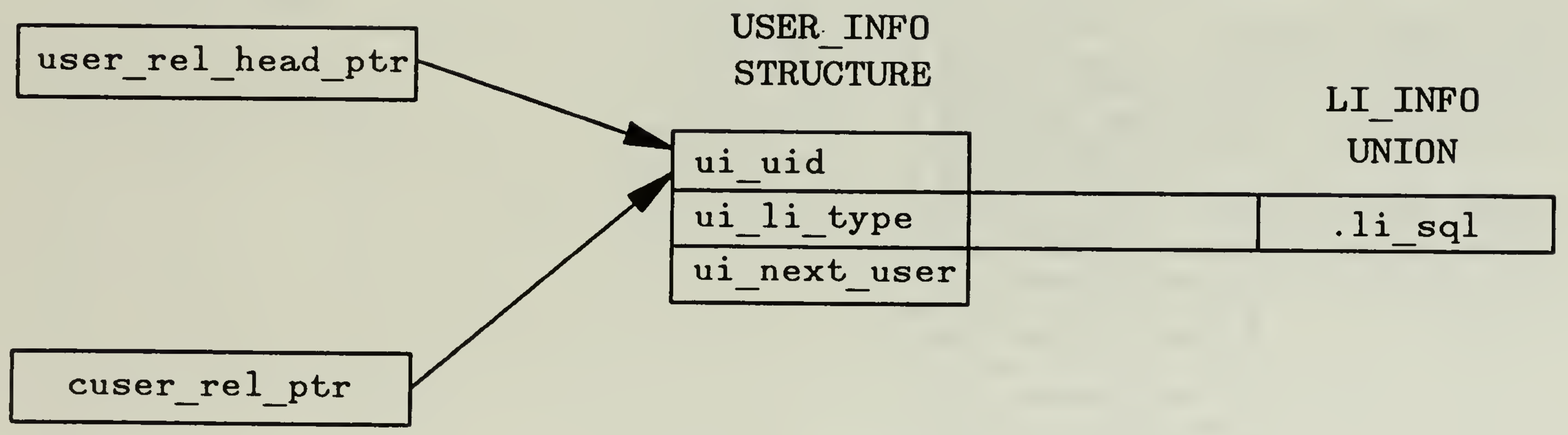
Currently planned thesis topics on the Multi-Model Database System include extensions on the functionality of the hierarchical and relational language interfaces, completion of the functional language interface, and additional cross-model access capability. The ongoing research and development efforts at the Laboratory for Database Systems Research indicate that a complete and fully-functional multi-model database system can be designed and implemented utilizing current hardware and software techniques and that the additional growth capacity, performance gains, and extensibility of such a system is a significant step in the area of database systems design.

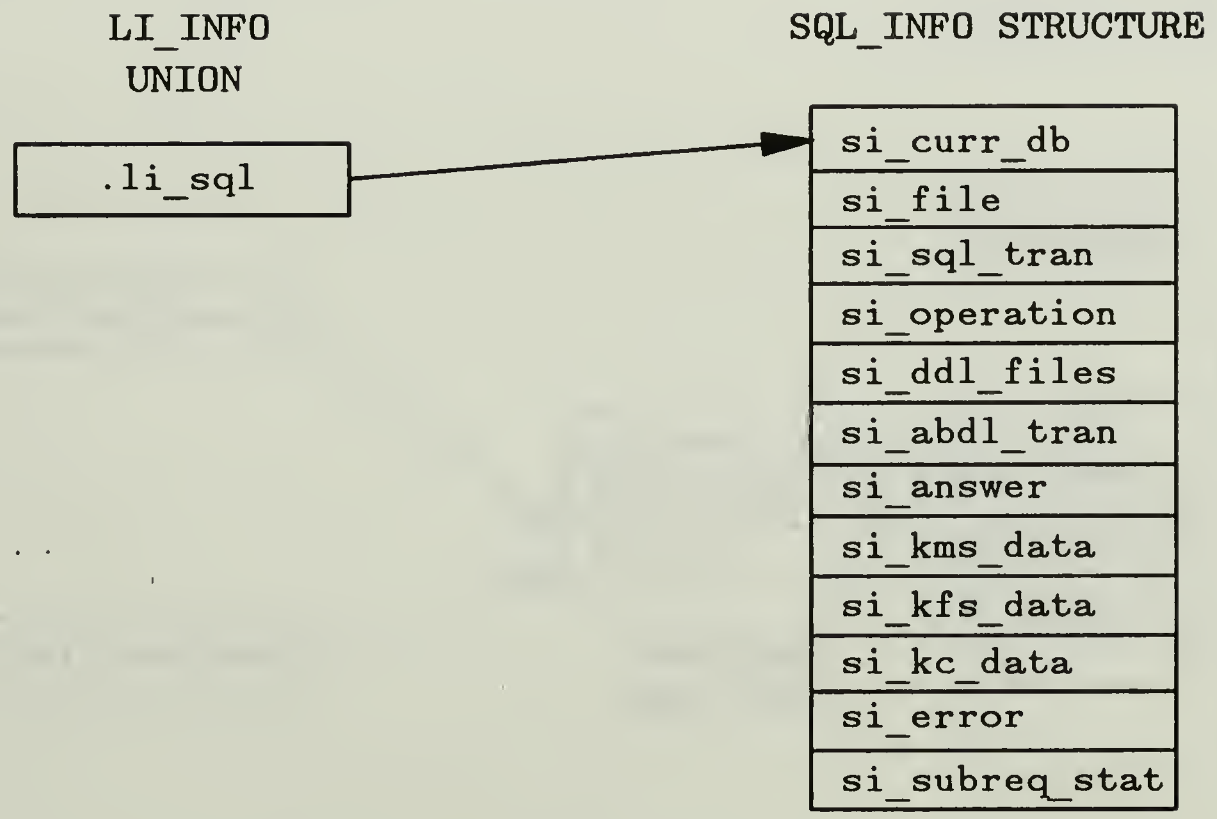
APPENDIX A - SCHEMATIC OF THE MAJOR DATA STRUCTURES

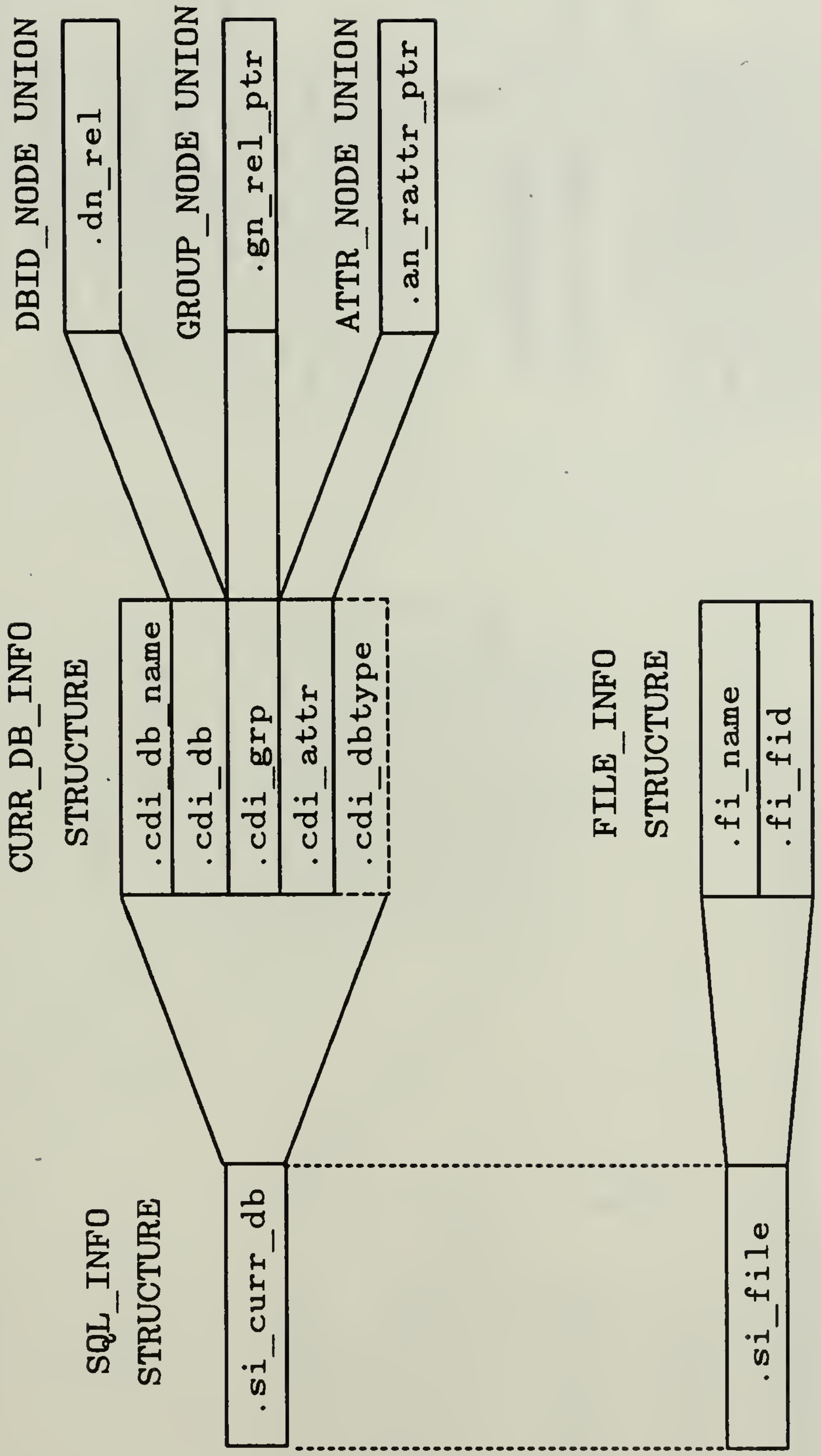


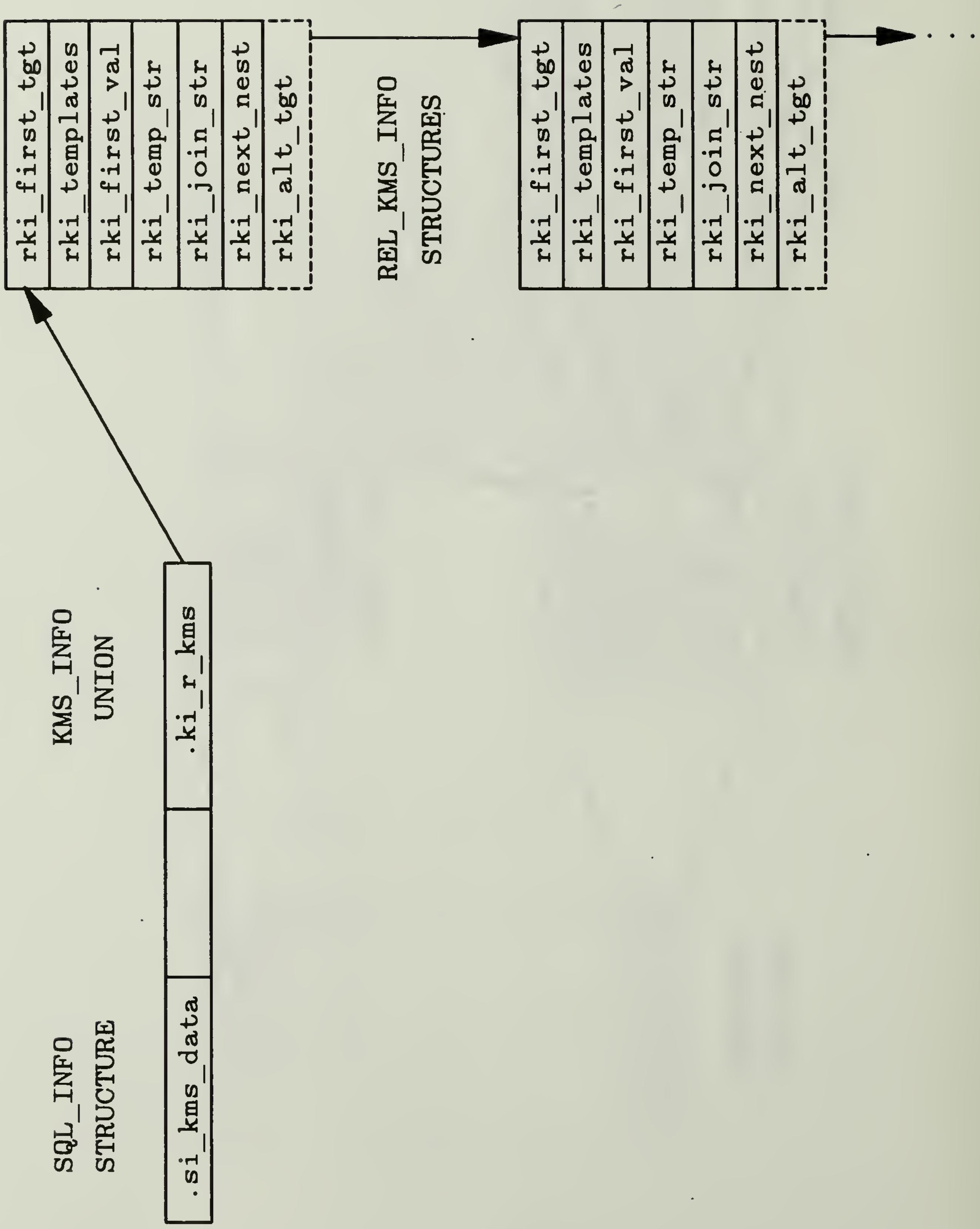

REL_DBID_NODE
STRUCTURE
rdn_first/last_rel
REL_NODE
STRUCTURES
rn_name
rn_num_attr
rn_first_attr
rn_curr_attr
rn_next_rel
rn_name
rn_num_attr
rn_first_attr
rn_curr_attr
rn_next_rel

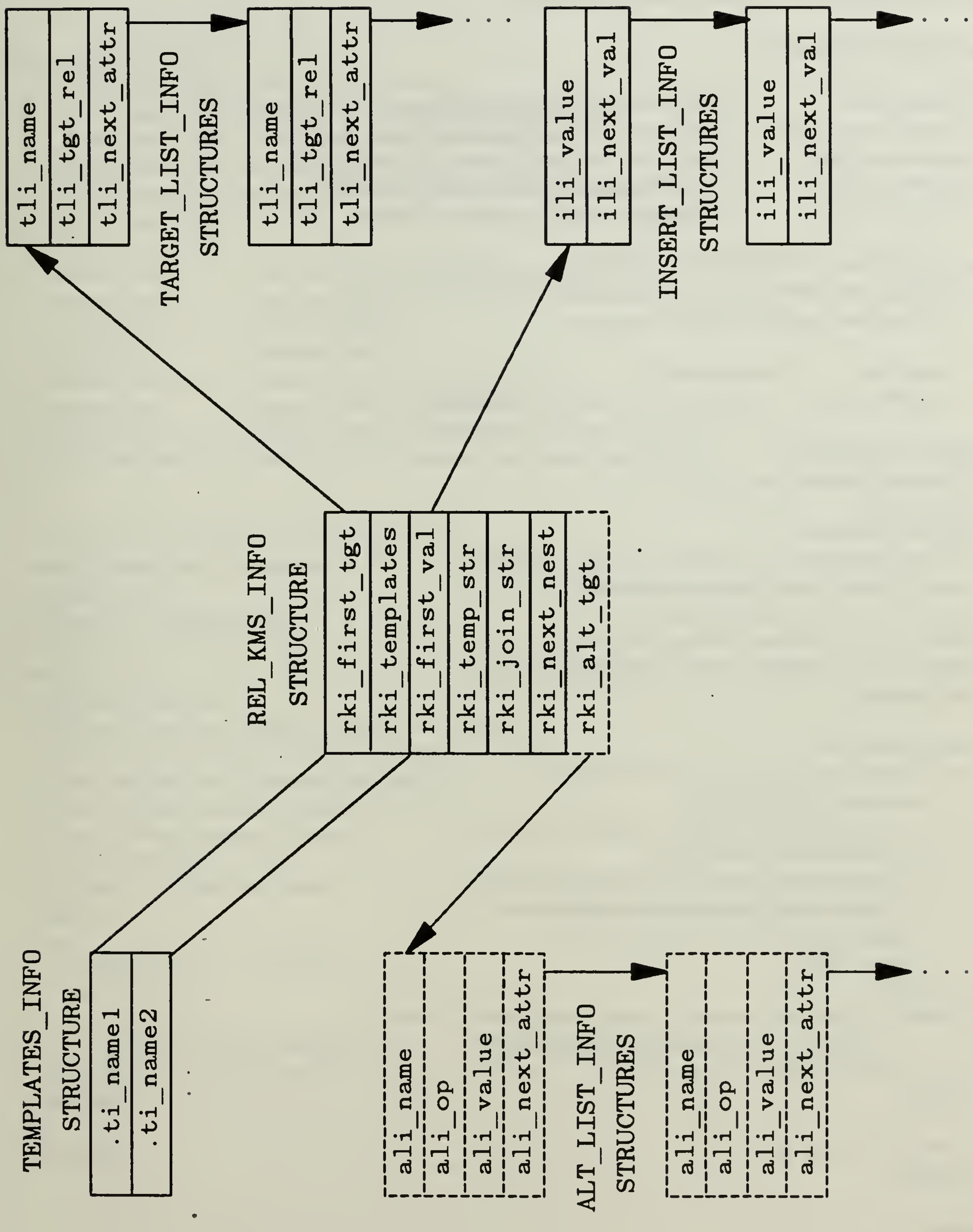

REL_NODE STRUCTURE
rn_first/curr_attr
RATTR_NODE STRUCTURES
ran_name
ran_type
ran_length
ran_key_flag
ran_next_attr
ran_name
ran_type
ran_length
ran_key_flag
ran_next_attr
. . .


user_rel_head_ptr
cuser_rel_ptr
USER_INFO STRUCTURE
ui_uid
ui_li_type
ui_next_user
LI_INFO UNION
.li_sql


LI_INFO
UNION
.li_sql
SQL_INFO STRUCTURE
si_curr_db
si_file
si_sql_tran
si_operation
si_ddl_files
si_abdl_tran
si_answer
si_kms_data
si_kfs_data
si_kc_data
si_error
si_subreq_stat


SQL_INFO STRUCTURE
.si_curr_db
.si_file
CURR_DB_INFO STRUCTURE
.cdi_db_name
.cdi_db
.cdi_grp
.cdi_attr
.cdi_dbtype
FILE_INFO STRUCTURE
.fi_name
.fi_fid
DBID_NODE UNION
.dn_rel
GROUP_NODE UNION
.gn_rel_ptr
ATTR_NODE UNION
.an_rattr_ptr


SQL_INFO STRUCTURE
.si_kms_data
KMS_INFO UNION
.ki_r_kms
rki_first_tgt
rki_templates
rki_first_val
rki_temp_str
rki_join_str
rki_next_nest
rki_alt_tgt
REL_KMS_INFO STRUCTURES
rki_first_tgt
rki_templates
rki_first_val
rki_temp_str
rki_join_str
rki_next_nest
rki_alt_tgt
. . .


tli_name
tli_tgt_rel
tli_next_attr
TARGET_LIST_INFO STRUCTURES
tli_name
tli_tgt_rel
tli_next_attr
ili_value
ili_next_val
INSERT_LIST_INFO STRUCTURES
ili_value
ili_next_val
REL_KMS_INFO STRUCTURE
rki_first_tgt
rki_templates
rki_first_val
rki_temp_str
rki_join_str
rki_next_nest
rki_alt_tgt
TEMPLATES_INFO STRUCTURE
.ti_name1
.ti_name2
ali_name
ali_op
ali_value
ali_next_attr
ALT_LIST_INFO STRUCTURES
ali_name
ali_op
ali_value
ali_next_attr

APPENDIX B - THE LIL PROGRAM SPECIFICATIONS

Note : italicized lines indicate MMDS modifications.

```
module SQL-INTERFACE

   db-list : list;             /* list of existing relational schemas        */
   head-db-list-ptr: ptr;  /* ptr to head of the relational schema list */
   current-ptr: ptr;         /* ptr to the current db schema in the list  */
   follow-ptr: ptr;          /* ptr to the previous db schema in the list */
   db-id : string;            /* string that identifies current db in use    */

proc LANGUAGE-INTERFACE-LAYER();
   /* This proc allows the user to interface with the system.  */
   /* Input and output: user SQL requests                          */

    stop : int;       /* boolean flag */
    answer: char;   /* user answers to terminal prompts */

  perform SQL-INIT();
  /* initialize pointers used in LIL */
  stop = 'false';
  while (not stop) do
    /* allow user choice of several processing operations */
    print ("Enter type of operation desired");
    print ("    (l) - load new database");
    print ("    (p) - process existing database");
    print ("    (x) - return to the to operating system");
    read (answer);

    case (answer) of
      'l':  /* user desires to load a new database */
            perform LOAD-NEW();
      'p':  /* user desires to process an existing database */
            perform PROCESS-OLD();
      'x':  /* user desires to exit to the operating system */
            /* database list must be saved back to a file    */
            store-free-db-list(head-db-list, db-list);
            stop = 'true';
            exit();
      default: /* user did not select a valid choice from the menu */
                print ("Error - invalid operation selected");
                print ("Please pick again")'
    end-case;

    /* return to main menu */
  end-while;

end-proc;
```

```
proc SQL-INIT();

end-proc;


proc LOAD-NEW();
   /* This proc accomplishes the following:                     */
   /* (1) determines if the new database name already exists,    */
   /* (2) adds a new header node to the list of schemas,         */
   /* (3) determines the user input mode (file/terminal),        */
   /* (4) reads the user input and forwards it to the parser, and */
   /* (5) calls the routine that builds the template/descriptor files */

   answer: int;          /* user answer to terminal prompts */
   more-input: int;      /* boolean flag */
   proceed: int;         /* boolean flag */
   stop : int;           /* boolean flag */
   db-list-ptr: ptr;     /* pointer to the current database */
   req-str: str;         /* single create in SQL form */
   ptr-abdl-list: ptr; /* ptr to a list of ABDL queries (nil for this proc)*/
   tfid, dfid: ptr;      /* pointers to the template and descriptor files */


 /* prompt user for name of new database */
 print ("Enter name of database");
 readstr (db-id);
 db-list-ptr = head-db-list-ptr;

 stop = 'false';
 while (not stop) do
   /* determine if new database name already exists */
   /* by traversing list of relational db schemas     */
   if (db-list-ptr.db-id = existing db) then
     print ("Error - db name already exists");
     print ("Please reenter db name");
     readstr (db-id);
     db-list-ptr = head-db-list-ptr;
   end-if;
   else
     if (db-list-ptr + 1 = 'nil') then
       stop = 'true';
     else
       /* increment to next database */
       db-list-ptr = db-list-ptr + 1;
   end-else;

 end-while;
```

```
  /* continue - user input a valid 'new' database name              */
  /* add new header node to the list of schemas and fill-in db name */
  /* append new header node to db-list                              */
  create-new-db(db-id);

  /* the KMS takes the SQL creates and builds a new list of relations   */
  /* for the new database. After all of the creates have been processed */
  /* the template and descriptor files are constructed by traversing    */
  /* the new database definition (schema).                              */

  more-input = 'true';
  while (more-input) do
    /* determine user's mode of input */
    print ("Enter mode of input desired");
    print ("    (f) - read in a group of creates from a file");
    print ("    (t) - read in a single create from the terminal");
    print ("    (x) - return to the main menu");
    read (answer);

    case (answer) of
      'f':  /* user input is from a file */
            perform READ-TRANSACTION-FILE();
            perform CREATES-TO-KMS();
            perform FREE-REQUESTS();
            perform BUILD-DDL-FILES();
            perform KERNEL-CONTROLLER();

      't':  /* user input is from the terminal */
            perform READ-TERMINAL();
            perform CREATES-TO-KMS();
            perform FREE-REQUESTS();
            perform BUILD-DDL-FILES();
            perform KERNEL-CONTROLLER();

      'x':  /* exit back to LIL */
            more-input = 'false';
      default: /* user did not select a valid choice from the menu */
            print ("Error - invalid input mode selected");
            print ("Please pick again");
    end-case;
  end-while;

end proc;
```

```
proc PROCESS-OLD();
  /* This proc accomplishes the following:              */
  /* (1) determines if the database name already exists,  */
  /*     as a Relational model. If not, other models are   */
  /*     checked, and if found, the schema is converted    */
  /*     to a relational schema.                           */
  /* (2) determines the user input mode (file/terminal),   */
  /* (3) reads the user input and forwards it to the parser */

     answer: int;          /* user answer to terminal prompts */
     found: int;           /* boolean flag to determine if db name is found */
     more-input: int;      /* boolean flag to return user to LIL */
     proceed: int;         /* boolean flag to return user to mode menu */
     db-list-ptr: ptr;     /* pointer to the current database */
     req-str: str;         /* single query in SQL form */
     ptr-abdl-list: ptr;   /* pointer to a list of queries in ABDL form */
     tfid, dfid: ptr;      /* pointers to the template and descriptor files */


   /* prompt user for name of existing database */
   print ("Enter name of database");
   readstr (db-id);
   db-list-ptr = head-db-list-ptr;

   found = 'false';
   while (not found) do
     /* determine if database name does exist     */
     /* by traversing list of relational schemas  */
     if (db-id = existing db) then
       found = 'true';
     end-if;
     else
     /* check if db name is defined in another model */
     perform CHECK-ALTERNATE-MODELS();
     else
       db-list-ptr = db-list-ptr + 1;
       /* error condition causes end of list('nil') to be reached */
       if (db-list-ptr = 'nil') then
         print ("Error - db name does not exist");
         print ("Please reenter valid db name");
         readstr (db-id);
         db-list-ptr = head-db-list-ptr;
       end-if;

     end-else;

   end-while;
  /* continue - user input a valid existing database name */
  /* determine user's mode of input */
  more-input = 'true';
  while (more-input) do
    print ("Enter mode of input desired");
    print ("    (f) - read in a group of queries from a file");
    print ("    (t) - read in a single query from the terminal");
    print ("    (d) - display the current database schema");
    print ("    (x) - return to the previous menu");
    read (answer);

    case (answer) of
      'f': /* user input is from a file */
          perform READ-TRANSACTION-FILE();
          perform QUERIES-TO-KMS();
          perform FREE-REQUESTS();

      't': /* user input is from the terminal */
          perform READ-TERMINAL();
          perform QUERIES-TO-KMS();
          perform FREE-REQUESTS();

      'd': /* display the database schema */
          perform SQL-TRAVERSE();

      'x': /* user wishes to return to LIL menu */
          more-input = 'false';

      default: /* user did not select a valid choice from the menu */
          print ("Error - invalid input mode selected");
          print ("Please pick again");
    end-case;

  end-while;

end-proc;

proc READ-TRANSACTION-FILE();
  /* This routine opens a create/query file and reads the requests */
  /* into the request list.  If open file fails, loop until valid   */
  /* file entered                                                    */

  while (not open file) do
    print ("Filename does not exist");
    print ("Please reenter a valid filename");
    readstr ( file);
  end-while;

  READ-FILE();
end-proc;
```

```
proc READ-FILE();
    /* This routine reads transactions from either a file or the          */
    /* terminal into the user's request list structure so that           */
    /* each request may be sent to the KERNEL-MAPPING-SYSTEM.*/

end-proc;


proc  READ-TERMINAL();
    /* This routine substitutes the STDIN filename for the read       */
    /* command so that input may be intercepted from the terminal */

end-proc;


proc CREATES-TO-KMS();
    /* This routine sends the request list of creates one by one      */
    /* to the KERNAL-MAPPING-SYSTEM                                   */

  while (more-creates) do
    KERNAL-MAPPING-SYSTEM();
  end-while;

end-proc;


proc  CHECK-ALTERNATE-MODELS();
  /* this routine calls other subroutines that check the Hierarchical,     */
  /* Network, and Functional schemas for the desired database name.  */
  /* If found, the schema is translated to a corresponding Relational    */
  /* schema and prepared for processing.                                    */

  perform TRAVERSE-DLI-SCHEMA();
  if found == true
    dbtype = HIE;
    perform TRANSLATE-DLI-TO-REL()
    /* initialize the data base. */
    sql_operation = CreateDB;
    Kernel_Controller();
  if found == FALSE
      {
      /* stub for future implementation of network model */
      }
   if (found == FALSE)
      {
      /* stub for future implementation of functional model */
      }
   } /* end check_alternate_models */
end-proc;
```

```
proc TRAVERSE-DLI-SCHEMA();
   /* This routine traverses the linked list of hierarchical          */
   /* database schemas in an attempt to locate the user-requested */
   /* database. If found, a pointer is returned to the schema.        */



proc TRANSLATE-DLI-TO-REL()

  /* this routine converts the hierarchical schema to a relational schema */

  /* the new rel database node is allocated and filled here with  */
  /* information from the hierarchical database node              */
  create new rel_dbid_node();
  strncpy(relational_db__name, hierarchical_db__name);
  number_of_relations = number_of_segments;
  dbtype = HIE;  /* identify db as hierarchical */
  previous_node_next_db = new_dbid_node; /* connect to rel db list */

  seg_ptr = hierachical_root_seg;
  create_flag = TRUE;
  up_flag    = FALSE;
  while (seg_ptr != NULL)
    if (create_flag == TRUE)
      /* the relation nodes are allocated and filled here */
      create_new_rel_node();
      strncpy(relation_name, segment_name);
      number_of_relational_attributes = number_of_hierarchical_fields;
      if (seg_ptr == hierarchical_root_seg)
        /* special case of first relation */
        dbid_node_first_rel = new_rel_node;
     else
       rel_node_next_rel = new_rel_node;

      hattr_ptr = seg_ptr->first_field;
      while (hattr_ptr != NULL)
        /* the attribute nodes are allocated and filled here */
        create_new_rattr_node();
        strncpy(attribute_name, field_name);
        attribute_type = field_type;
        attribute_length = field_length;
        key_flag = sequence_field_flag;
       if (hattr_ptr == seg_ptr->first_field)
          /* special case of first attribute */
          rel_node_first_attr = new_rattr_ptr;
       else
          rattr_node_next_attr = new_rattr_ptr;
       hattr_ptr = next_field;
       /* end attr loop */
```

```
        /* the sequence fields are cascaded at this point */
        ancestor_seg_ptr = seg_ptr->parent;
        while (anc_seg_ptr != NULL)
           create_new_rattr_node();
           strncpy(attribute_name, segment_name)
           attribute = field_type;
           attribute_length = field_length;
           attribute_key_flag = sequence_field_flag;
           attach_attribute_to_relation();
           ancestor_seg_ptr = anc_seg_ptr->parent;
        /* end if create_flag == true */

      create_flag = TRUE;
      if (up_flag == FALSE && seg_ptr->first_child != NULL)
         seg_ptr = seg_ptr->first_child;
      else
         if (seg_ptr->next_sibling != NULL)
           seg_ptr = seg_ptr->next_sibling;
           up_flag = FALSE;
        else
           seg_ptr = seg_ptr->parent;
           up_flag = TRUE;
           create_flag = FALSE;
           if (seg_ptr == hierarchical_db_root_seg)
           seg_ptr = NULL;
       /* end while seg_ptr != null   */
end-proc;
```

```
proc QUERIES-TO-KMS();
    /* This routine causes the queries to be listed to the screen.  */
    /* The selection menu is then displayed allowing any of the     */
    /* queries to be executed.                                      */

  perform LIST-QUERIES();
  proceed = 'true';

  while (proceed) do
    print ("Pick the number or letter of the action desired");
    print ("    (num) - execute one of the preceding queries");
    print ("    (d)   - redisplay the file of queries");
    print ("    (x)   - return to the previous menu");
    read (answer);
    case (answer) of

     'num' : /* execute one of the queries */
            traverse query list to correct query;
            perform KERNAL-MAPPING-SYSTEM();
            perform KERNEL-CONTROLLER();

     'd'   : /* redisplay queries */
            perform LIST-QUERIES();

     'x'   : /* exit to mode menu */
            proceed = 'false';

     default : /* user did not select a valid choice from the menu */
            print (" Error - invalid option selected");
            print (" Please pick again");
    end-case;

  end-while;

end-proc;


end-module;
```

APPENDIX C - THE KMS PROGRAM SPECIFICATIONS

```
module KMS ()

 perform parser()

end-module KMS


proc  yyparse ()

     /*  This proc accomplishes the following :                        */
     /*  (1) parses the SQL input requests and maps them to appropriate */
     /*      abdl requests, using LEX and YACC to build proc yyparse(). */
     /*  (2) builds the relational schema, when loading a new database.  */
     /*  (3) checks for validity of relation and attribute names within  */
     /*      the given db schema, when processing requests against an    */
     /*      existing database.                                          */
%{
     list: tgt-list               /* list of attribute names */
     list: templates              /* relation name(s) */
     list: insert-list            /* list of values for insertion op */
     list: alt_info               /* list of attributes, ops, and values */
     string: temporary-str        /* used for accumulation of query conjuncts */
     string: abdl-str             /* used for accumulation of abdl request */
     string: join-str             /* used for accumulation of join request */
     boolean: nested              /* signals a nested SELECT query */
     boolean: creating            /* signals a DbLoad - versus a DbQuery */
     boolean: or-where            /* signals an OR term in the WHERE clause */
     boolean: and-where           /* signals an AND term in the WHERE clause */
     boolean: set-member          /* signals set membership op, vice nested SEL */
     boolean: common-attr         /* signals COMMON attr predicate of JOIN op */
     boolean: rel1                /* signals curr predicate assoc'd w/1st join rel */
     boolean: rel2                /* signals curr predicate assoc'd w/2nd join rel */
     boolean: or-abdl-join        /* OR in 1st join retrieve request */
     boolean: or-kms-join         /* OR in 2nd join retrieve request */
     boolean: delete-all          /* signals deletion of all records in relation */
     boolean: key_attr            /* identifies a key attribute */
     boolean: leaf                /* identifies leaf segment in hierarchical schema */
     int: no_null_count           /* counts number of key attributes in a relation */
     int: target-list-length
     int: insert-list-length
     int: no-templates
     int: no-attributes
     int: attr-len
     char: attr-type
     char: db[]
     char: template[]
     char: attribute[]
%}
```

```
% start   statement

% token    /* LIST ALL TOKENS FROM "LEX", and their TYPE, HERE */

%%
/* The grammar rules that follow have been taken from :               */
/* "System R", Appendix II, by M.M. Astrahan in the ACM Trans-        */
/* actions on Database Systems, Vol. 1, No. 2, June 1976.             */
/* The rules are not shown in their entirety, however except for the  */
/* following exceptions, they were strictly adhered to in an effort   */
/* to facilitate future expansion of this program for SQL :           */
/* (1) all non-terminals are in lower-case,                           */
/* (2) all terminals (recognized by LEX/lex.yy.c) are in upper-case,  */
/* (3) some upper-case single-character letters appear throughout --  */
/*     they represent points in the grammar where allowances were     */
/*     made for optional terminals and non-terminals.                 */

statement:  query
            {
            nested = FALSE
            free all tgt/insert lists and temp-str (malloc'd vars)
            return
            }
         |  dml-statement
            {
            cat End-Of-Request ("]") to end of abdl-str
            free all tgt/insert lists and temp-str (malloc'd vars)
            return
            }
         |  ddl-statement
            {
            return
            }
            ;

dml-statement:  insertion
              | deletion
              | update
                ;

query:  query-expr
        ;

query-expr:  query-block
             {
             cat End-Of-Request ("]") to end of abdl-str
             }
             ;
```

```
query-block: select-clause  FROM  from-list
          {
          for (ea attribute name in tgt-list)
            if  (! join)
              if  NOT valid-attribute(db, template, attribute, attr-len)
              print ("Error - field name 'attribute-name' does not exist")
                perform yyerror()
                return
              end-if
            end-if
            else
              a join exists --  check that tgt-rel(s) match at least
              one from-list relation
              if  (match neither)
                print ("Error - 'attr' attr not in from-list relations")
                perform yyerror()
                return
              end-if
            end-else
          end-for
          cat "(" to abdl-str
          if  (join)
            cat "(" to join-str
          end-if
          if  (nested)
            fill temporary-str w/'*'s marking the length of the tgt attr
          end-if
          }

         A
          {
          cat ")" to abdl-str
          if  (! join)
            cat "('tgt-list')" to abdl-str
          end-if
          else
            cat "('tgt-list')" to abdl/join-str, as appropriate
            construct the rest of the abdl join request
            (ie, cat COMMON-str to abdl-str; cat join-str to abdl-str)
          end-else
          }

         B
          ;
A:  empty
      {
      cat "TEMP = 'relation-name'" to abdl-str
      }
 |  WHERE boolean
      {
      if  (! join) && (or-where)
        cat ")" to abdl-str
      end-if
      else if  (or-abdl-join)
        cat ")" to abdl-str
      end-elseif
      elseif  (or-kms-join)
        cat ")" to join-str
      end-elseif
      }
      ;

B:  empty
 |  GROUP BY  field-spec-list
      {
      cat "BY 'attribute-name'" to abdl-str
      }
 |  ORDER BY  field-spec-list
      {
      cat "BY 'attribute-name'" to abdl-str
      }
      ;

select-clause:  SELECT
                  {
                  if  (nested)
                    allocate another set of tgt/insert lists, temporary-str,
                    and abdl strings
                  end-if
                  copy "[ RETRIEVE  " to beginning of abdl-str
                  }
                C ;

C:  sel-expr-list
 |  MULTOP
      {
      /* retrieval of "all" attribute values desired */
      if (MULTOP value /= '*')
        print ("Error - asterisk(*) operator expected")
        perform yyerror()
        return
      end-if
      }
      ;
```

```
sel-expr-list:  sel-expr
             {
             copy first attribute name to tgt-list
             }

          |  sel-expr-list  COMMA  sel-expr
             {
             copy successive attribute name(s) to tgt-list
             }
           . ;


sel-expr:  expr
          ;


insertion:  INSERT INTO
           {
           copy "[ INSERT (" to beginning of abdl-str
           }
          receiver  COLON  insert-spec
           {

           /* If the current database is hierarchical, branch to */
           /* the alternate processing algorithm at this point   */
           perform INSERT-REL-TO-DLI()

           cat ")" to abdl-str
           }
           ;


receiver:  table-name
          {
          cat "<TEMP, 'relation-name'>" to abdl-str
          /* Get the number of key attributes in the current relation */
          call get_no_null_count()
          }

         D
          ;
```

```
D: empty
    {
    /* inserting info for "all" attribute values */
    copy all attribute names from schema to tgt-list
    if (target-list-length < 1)
      print ("Error - rel does not exist, or has no attr's")
      perform yyerror()
      return
    end-if
    }
 | LPAR field-name-list RPAR
    {
    for (ea attribute name in tgt-list)
      if NOT valid-attribute(db, template, attribute, attr-len)
        print ("Error - field name 'attribute-name' does not exist)
        perform yyerror()
        return
      end-if
    end-for
    } ;

field-name-list: field-name
                 {
                 target-list-length++
                 copy first attribute name to tgt-list
                 }
               | field-name-list COMMA field-name
                 {
                 target-list-length++
                 copy successive attribute name(s) to tgt-list
                 } ;

insert-spec: literal
             {
             if (length of tgt-list <> length of insert-list)
               print ("Error - not enough or too many values inserted")
               perform yyerror()
               return
             end-if
             for (ea attribute in tgt-list / ea value in insert-list)
               perform type-checking of attrribute-value pairs
               cat ",<'attribute-name', 'insert-value'>" to abdl-str
             end-for
             if (value not given for each key attribute)
               print (NONULL attributes in relation must be given specific values)
               perform yyerror()
               return
             end-if
             } ;
```

```
deletion: DELETE  table-name
          {
          copy "[ DELETE (  " to abdl-str
          copy 'table-name' to templates
          }
         E
          {
          if  (delete-all)
            cat "TEMP = 'table-name'" to abdl-str
          end-if
          cat ")" to abdl-str
          /* If deleting from a hierarchical db, allocate an alt_info structure, */
          /* intialize to NULL values, and attach to end of linked list.         */
          call alt_list_info_alloc()
          cat ('ZZZ' to name, op, value)

          /* If deleting from a hierarchical db */
          perform DELETE-REL-TO-DLI()
          }
          ;

E: empty
     {
     delete-all = TRUE
     }
 | WHERE  boolean
     {
     if  (or-where)
       cat ")" to abdl-str
     end-if
     }
     ;

update: UPDATE  table-name
         {
         copy "[ UPDATE (  " to beginning of abdl-str
         copy relation-name to templates
         }
        set-clause-list  F
         {
         cat ") 'set-clause-list'" to abdl-str
         /* If updating in a hierarchical db, allocate an alt_info structure, */
         /* intialize to NULL values, and attach to end of linked list.       */
         call alt_list_info_alloc()
         cat ('ZZZ' to name, op, value)

         /* If updating in a hierarchical db and attempting to change a Key field */
         print (UPDATE not allowed. The current implementation of DL/I)
         print (allows updates on NON-KEY fields only)
         } ;
F: empty
| WHERE boolean
    {
    if (or-where)
      cat ")" to abdl-str
    end-if
    }
    ;


set-clause-list: set-clause
                ;


set-clause: SET field-name EQ expr
            {
            if NOT validattribute(db, template, attribute, attr-len)
              print ("Error - field name 'attribute-name' does not exist")
              perform yyerror()
              return
            end-if
            if (updating a hierarchical db, check if updating is on a key field)
              call set_update_status()
            else
              copy "<'field-name = expr'>" to abdl-str
            end-else
            }
            ;


ddl-statement: create-table
               ;


create-table: CREATE
              {
              creating = TRUE
              locate db-id schema header
              }
             TABLE table-name COLON
              {
              no-templates ++
              create new template block
              enter 'relation-name' in template block
              }
             field-defn-list
              ;
```

```
field-defn-list: field-defn
                  {
                  no-attributes ++
                  }
               | field-defn-list  COMMA  field-defn
                  {
                  no-attributes ++
                  }
                  ;


field-defn: field-name  LPAR  type  G  RPAR
             {
             create new attribute block
             enter 'attribute-name' in attribute block
             }
             ;


type: CHAR  LPAR  INTEGER  RPAR
       {
       enter attribute type and length in attribute block
       }
     | INT  LPAR  INTEGER  RPAR
       {
       enter attribute type and length in attribute block
       }
     | FLOAT  LPAR  INTEGER  RPAR
       {
       enter attribute type and length in attribute block
       }
       ;


G: empty
    {
    set key-flag to '0' in attribute block
    }
  | COMMA  NONULL
    {
    set key-flag to '1' in attribute block
    }
    ;
```

```
boolean:  boolean-term
          {
          if  (! join)
            cat "-(TEMP = 'relation-name') and" to abdl-str
            cat temporary-str to abdl-str
          end-if
          }

        | boolean  OR
          {
          or-where = TRUE

          /* If deleting or updating in a hierarchical db, allocate an alt_info        */
          /* structure, intialize to NULL values, and attach to end of linked list.*/
          call alt_list_info_alloc()
          cat ('ZZZ' to name, op, value)

          if  (! join)
            abdl-str[11] = '('
            cat ") or ((TEMP = 'relation-name') and" to abdl-str
            copy empty-str to temporary-str
          end-if
          }

          boolean-term
          {
          if  (! join)
            cat temporary-str to abdl-str
          end-if
          else
            if  (current predicate assoc'd w/same rel as previous predicate)
              abdl/join-str[11] = '('
              cat ") or ((TEMP = 'rel-name') and" to abdl/join-str (as approp)
              cat temporary-str to appropriate str (abdl/join-str)
            end-if
            else
              abdl/join-str(as approp)[11 + 3] = '('
              cat "and" to appropriate str (abdl/join-str)
              cat temporary-str to appropriate str (abdl/join-str)
            end-else
            copy empty str to temporary-str
            or-where = FALSE
          end-else
          }
          ;
```

```
boolean-term:  boolean-factor
                {
                if  (join) && (! or-where)
                  determine rel that curr predicate is assoc'd with
                  if  (rel1) && (! common-attr)
                    cat "(TEMP = 'rel-name1') and" to abdl-str
                    cat temporary-str to abdl-str
                    cat " TEMP = 'rel-name2'" to join-str
                  end-if
                  if  (rel2) && (! common-attr)
                    cat "(TEMP = 'rel-name2') and" to join-str
                    cat temporary-str to join-str
                    cat " TEMP = 'rel-name1'" to abdl-str
                  end-if
                  if  (common-attr)
                    cat " TEMP = 'rel-name1/2" to abdl/join-str's
                  end-if
                end-if
                }
            |  boolean-term  AND
                {
                and-where = TRUE;
                if  (! join)
                  cat "and" to temporary-str
                end-if
                }
              boolean-factor
                {
                if  (join) && (! or-where) && (! common-attr)
                  if  (rel1)
                    abdl-str[11 + 3] = '('
                    cat ") and" to abdl-str
                    cat temporary-str to abdl-str
                  end-if
                  if  (rel2)
                    join-str[11 + 3] = '('
                    cat ") and" to join-str
                    cat temporary-str to join-str
                  end-if
                  copy empty-str to temporary-str
                  and-where = FALSE
                end-if
                }
                ;
boolean-factor:  boolean-primary
                  ;

boolean-primary:  predicate
                   ;
```

```
predicate: expr
      {
      if (! join)
        if NOT valid-attribute(db, template, attribute, attr-len)
          print ("Error - field name 'attribute-name' does not exist")
          perform yyerror()
          return
        end-if
        if (! and-where)
          allocate new temporary-str
        cat "('attribute-name' " to temporary-str
        and-where = FALSE
      end-if

      /* If deleting or updating in a hierarchical db, allocate an alt_info   */
      /* structure, set name to current attribute, and attach to linked list.*/
      call alt_list_info_alloc()
      cat (attribute to name)

      else
        save 'type' for later comparison during type-checking,
        in case this is the COMMON attribute predicate
      }
    comparison
      {
      if (nested)
        save attr name in case nest is actually a set membership op
      }
    table-spec
      {
      if (! join)
        cat ")" to temporary-str
      else
        if (common-attr)
          save values of 'expr', 'comparison', & 'table-spec'
          for the COMMON expr, and type-check the two attr's
        end-if
        if (! and-where) && (! or-where)
          allocate initial temporary-str
          copy "(" to temporary-str
        end-if
        else
          cat "(" to temporary-str
        end-else
        cat "'expr' 'comparison' 'table-spec')" to temporary-str
      end-else
      }
      ;
```

```
comparison:  comp-op
          {
          if  (! join)
            cat 'comp-op' to temporary-str
            if  (nested)
              copy type-op-code to abdl-str.rel-op
            end-if
          end-if
          }
          ;


comp-op:  EQ
        {
        /* If deleting or updating in a hierarchical db, set the  */
        /* current operation, and attach to end of linked list.  */
        cat (operation to op)
        }
     |  M  J
        {
        /* If deleting or updating in a hierarchical db, set the  */
        /* current operation, and attach to end of linked list.  */
        cat (operation to op)
        if  (nested)
          cat 'J' to 'M' and save
        end-if
        }
     |  L
        {
        /* If deleting or updating in a hierarchical db,    */
        /* and a nested operation is attempted - error.  */
        print (nested delete or update operation not allowed on HIE database)
        nested = TRUE
        }
        ;


J:  empty
 |  K
    {
    /* If deleting or updating in a hierarchical db,          */
    /* and a ALL/ANY operation is attempted - error.   */
    print (ALL/ANY operation not allowed on HIE database)
    nested = TRUE
    }
    ;
```

```
K:  ANY | ALL
    ;

L:  IN | NOT IN


M:  NE | RWEDGE | GE | LWEDGE | LE
    ;

table-spec: literal
          {
          if (! set-member)
            if ('literal[0]' = QUOTE)
              strip quotes from literal
              change literal to ALPHANUMFIRST
              literal-const = FALSE
            end-if
            cat result, or original literal, to temporary-str
            /* If deleting or updating in a hierarchical db, set the */
            /* attribute value, and attach to end of linked list.    */
            cat (attribute value to value)
            if (nested)
              set first-ptr to top of abdl-str list
            end-if
          end-if
          else
            set-member = FALSE
          end-else
          }
        | query-expr
          {
          increment ptr to next tgt/insert list, temp-str, and abdl-str
          }
        | LPAR query-expr RPAR
          {
          increment ptr to next tgt/insert list, temp-str, and abdl-str
          }
        | expr
          {
          common-attr = TRUE
          }
          ;
```

```
literal:  lit-tuple

     | LPAR  entry-list  RPAR
         {
         set-member = TRUE
         case (set-membership-op)
           3,5,8,10 :            /*  <=ANY, <ANY, >=ALL, >ALL  */
                cat 'max of value set' to temporary-str
           4,6,7,9  :            /*  >=ANY, >ANY, <=ALL, <ALL  */
                cat 'min of value set' to temporary-str
           1 :                   /*  NOT IN  */
             cat first value to temporary-str
             while  (other values exist)
               cat ") and ('attr-name' /= 'value'" to temporary-str
             end-while
           0,2 :                 /*  IN, /=ANY  */
             cat first value to temporary-str
             if  (more values exist)
               abdl-str[11] = '('
               or-where = TRUE
             end-if
             while  (other values exist)
               cat ")) or ((TEMPLATE = 'rel-name') and ('attr-name'"
               to temporary-str
               if  ( rel-op = IN )
                 cat " = " to temporary-str
               end-if
               else
                 cat " /= " to temporary-str
               end-else
               cat value to temporary-str
             end-while
         end-case
             ;



lit-tuple:  entry
      |  LWEDGE  entry-list  RWEDGE
         ;
```

```
entry-list:  entry
             {
             /* copy first value to insert-list */
             insert-list-length++
             if ('entry[0]' = QUOTE)
               strip quotes from entry
               change entry to ALPHANUMFIRST
             end-if
             copy result, or original entry, to insert-list
             }
          |  entry-list  COMMA  entry
             {
             /* copy successive value(s) to insert-list */
             insert-list-length++
             if ('entry[0]' = QUOTE)
               strip quotes from entry
               change entry to ALPHANUMFIRST
             end-if
             copy result, or original entry, to insert-list
             }
             ;

entry:  constant
          ;

expr:  arith-term
   |  expr  ADDOP  arith-term
       ;

arith-term:  arith-factor
          |  arith-term  MULT-OP  arith-factor
             ;

arith-factor:  H  primary
                 ;

H:  empty
 |  ADDOP
     ;

primary:  field-spec
       |  set-fn  LPAR  field-name  RPAR
       |  LPAR  expr  RPAR
       |  constant
           ;

field-spec-list:  field-spec
                    ;
```

```
field-spec: field-name
        | table-name DOT field-name
           {
           if (! valid-attribute(db, rel, attr, attr-len)
             print ("Error - 'rel.attr' is invalid combination")
             perform yyerror()
             return
           end-if
           if (join)
             if (! or-where) || ( (or-where) && (! and-where) )
               if (table-name = rel1)
                 rel1 = TRUE
                 rel2 = FALSE
               end-if
               if (table-name = rel2)
                 rel1 = FALSE
                 rel2 = TRUE
               end-if
             end-if
           end-if
           } ;

set-fn: AVG | MAX | MIN | SUM | COUNT ;

from-list: table-name
          {
          copy first relation name to templates
          if (tgt-list = null)
            fill tgt-list with "all" attribute names in the relation
          end-if
          }
        | from-list COMMA table-name
          {
          copy second relation name to templates
          join = TRUE
          allocate join-str
          } ;

empty: ;

constant: QUOTE I QUOTE
         {
         literal-const = TRUE
         perform type-checking
         }
       | INTEGER
         {
         perform type-checking
         } ;
```

```
I: IDENTIFIER
|  VALUE
    ;

field-name: IDENTIFIER
            ;

table-name: IDENTIFIER
            {
            if (! creating)
              if NOT valid-table(db, template)
                print ("Error - relation name 'table-name' does not exist")
                perform yyerror()
                return
              end-if
            end-if
            }
            ;
%%

end-proc yyparse



proc parser ()
  {
  if (! creating)
    allocate and initialize first tgt/insert lists, temporary-str, and abdl-str
    /* if an old abdl-str exists, free it first */
  end-if
  perform yyparse()
  reset all boolean and counter variables
  }
end-proc parser

proc yyerror (s)
  char *s
  {
  if (creating)
    set CreateDB-error-flag
    print ("Error msg - tell user which CREATE TABLE request was in error")
    free current schema (malloc'd vars)
  end-if
  else
    free all tgt/insert lists, temp-str, and abdl-strs
  end-else
  reset all boolean and counter variables
  }
end-proc yyerror
```

```
proc INSERT-REL-TO-DLI()
  /* This procedure translates a AB(relational) insert transaction */
  /* to an equivalent AB(hierarchical) insert transaction          */

  hierarchical_ptr = locate_dli_schema(db); /* head of hie db */
  relational_ptr = locate_rel_schema(db);   /* head of rel db  */

  /* search the hie schema for the segment/relation name */
  seg_ptr = hierarchical_root_seg;
  visit_flag = TRUE;  up_flag   = FALSE;  found = FALSE;
  while (seg_ptr != NULL && found == FALSE)
    if (visit_flag == TRUE)
      if(hierarchical_name == relational_name)
        found = TRUE;
    visit_flag = TRUE;
    if (found == FALSE)
      if (up_flag == FALSE && segment_first_child != NULL)
        seg_ptr = segment_first_child;
      else
        if (segment_next_sibling != NULL)
          seg_ptr = segment_next_sibling;  up_flag = FALSE;
        else
          seg_ptr = segment_parent;
         seg_ptr = segment_parent; up_flag = TRUE;  visit_flag = FALSE;
         if (seg_ptr == hierarchical_root_seg)
            seg_ptr = NULL;
  /* end while seg_ptr != null   */

  /* determine if RETRIEVE will be need, and if so, build it.   */
  /* note : A RETRIEVE is not needed at the root segment.       */
  while (segment_name != relation_name)
    rel_ptr = next_relation;
  if (seg_ptr->segment_parent != NULL) /* then retrieve is needed */
    strcat(ari_req, ") ]"); /* end the insert request */
    /* begin forming the RETRIEVE request */
    strcpy(new_req, "[ RETRIEVE  ((TEMP = ", segment_parent_name, ")");
    /* use the information in the insert request to form the new retrieve */
    while (field_name != attribute_name)
      strcat(new_req, " and (", attribute_name, " = ");
      /* add the attribute value */
      strcat(new_req, attribute_value, ")");

    /* add the target list to the retrieve request */
    strcat(new_req, ") (", relational_first_attr_name);
    no_req = no_req + 1; /* increase the number of ABDL requests */
    /* link the insert request to the retrieve request */
    first_req = retrieve_req;   next_req = insert_req;
  /* end if seg_ptr != null */

end-proc insert-rel-to-dli
```

```
proc DELETE-REL-TO-DLI()
  /* translates the SQL delete to AB(hierarchical) Delete transactions */

  /* initialize rel pointers */
  rdb_ptr = locate_rel_schema(db); /* head of rel db */
  hdb_ptr = locate_dli_schema(db); /* head of dli db */

  /* create the ancestor retrieve requests */
  while (!done)
    if (seg_ptr->hn_parent == NULL)
      done = TRUE;
    /* alloc and init a new abdl_str and a new tgt_list item */
    temp_sit_ptr = Sit_info_alloc();
    call init_sit_info ();
    operation = GhuOp;
    strcpy (abdl_req, "[ RETRIEVE  (TEMP = ", segment_name, ")");

    /* add the cascaded sequence fields and key field */
    while (attribute_name != segment_field_name)
      strcat (abdl_req, " and (", attribute_name, ")");
      for (i = 1; i <= attribute_length)
        strcat (abdl_req, Star);
      strcat (abdl_req, ")");
      rattr_ptr = next_attribute;

    /* add on specific query predicates if any */
    added_value = FALSE;
    while (temp_alt_ptr != end_alt_ptr)
      /* check of a specific query goes with this segment */
      hattr_ptr = seg_ptr_first_attr;
      entered = FALSE;
      while (hattr_ptr != NULL && !entered)
        if (ali_name == field_name)
          /* a specific query predicate matches an attribute in this segment */
          strcat (abdl_req, " and (", ali_name, op, value,")");
          added_value = TRUE; /* used to determine if additional brace needed */
          entered = TRUE;     /* used to break out of loop */
          /* end ali_name = field_name */
        else
          hattr_ptr = next_attr;
        /* end while hattr_ptr != null */
      temp_alt_ptr = ali_next_attr;
    if (extra parenthesis needed)
      abdl_req[LeadingLPAR] = '(';
      strcat (abdl_req, ")");

    /* add on the tgt list and by clause */
    strcat (abdl_req, " (", seg_first_field_name, ") BY ", seg_first_field_name, "]");
    seg_ptr = parent;
    /* end while !done */
/* build the Delete request for the specified relation/segment */

/* alloc and init a new abdl_str and a new tgt_list item */
call  Sit_info_alloc();
call init_sit_info ();
operation = DletOp;

/* formulate the first DELETE request */
strcpy (abdl_req, "[ DELETE ((TEMP = ", segment_name, ")");
while  (not at end of list tgt list)
   /* copy seq_fld attribute-value pairs to abdl_str */
   strcat (abdl_req, " and (", sii_name, "=");
   /* mark max length of attribute value */
   for  (i = 1;  attribute_length)
      strcat (abdl_req, Star);
   strcat (abdl_req, ")");
   tgt_ptr = next_attr;

   while (temp_alt_ptr != end_alt_ptr)
      hattr_ptr = first_attr->next_attr;
      entered = FALSE;
      while (hattr_ptr != NULL && !entered)
         if (ali_name == field_name)
            strcat (abdl_req, " and (", name, op, value, ")");
            added_value = TRUE;
            entered = TRUE;
         else
            hattr_ptr = next_attr;
      temp_alt_ptr = next_attr;
strcat (abdl_req, ") ]");

/* move the ptr to next set of specific predicates. Ptr will be null */
/* if no 'OR' involved or point to first predicate of next set        */
alt_ptr = ali_next_attr;

/* form the descendant Deletes to complete the Delete request */
call form_descendant_deletes ();

/* set up the sit_status structures */
call match();

/* call the Kernel Controller */
Kernel_Controller();

/* on return, clear the result files in case they are needed again */
close_buffs();
while (sit_ptr != NULL)
   partial_init_sit_info ();
   sit_ptr = next;
```

```
/* check if this is an 'OR' operation. If it is, revise the     */
/* initial set of retrieves as neccessary and call KC again. */
while (alt_ptr != NULL)
  /* reset necessary pointers */
  operation = ExecRetReq;    req_status = FIRSTTIME;
  /* set the boundary for the next set of specific predicates */
  end_alt_ptr = alt_ptr;
  while (ali_name != "ZZZ")
    end_alt_ptr = next_attr;
  /* move to the retrieve of the rel/seg being deleted */
  while (operation != DletOp)
    sit_ptr = next;

  /* modify all the retrieves */
  while (!done)
    if (!first_time && seg_parent == NULL)
      done = TRUE;

    /* identify the area of the modifications */
    while (abdl_req[fwd_count] != '\n')
      ++fwd_count;
    rev_count = strlen(abdl_req);
    while (abdl_req[rev_count] != '\n')
      --rev_count;

    /* copy the abdl req to the template in order to rebuild the request */
    strcpy (template, abdl_req);
    /* cut off the abdl req at the point of the first predicate */
    abdl_req[fwd_count] = '\0';

    /* search the rel schema for the segment/relation name */
    while (relation_name != segment_name)
      rel_ptr = next_relation;
    rattr_ptr = first_attr;

    /* add the cascaded sequence fields and key field */
    if (first_time && relation_name == segment_name)
      while (attribute_name != second_field_name)
        strcat (abdl_req, " and (", attribute_name, "=" );
        for (i = 1; attribute_length)
          strcat (abdl_req, Star);
        strcat (abdl_req, ")");
        rattr_ptr = next_attr;
    else
      while (attribute_name != first_field_name)
        strcat (abdl_req, " and (", attribute_name, "=" );
        for (i = 1; attribute_length)
          strcat (abdl_req, Star);
        strcat (abdl_req, ")");
        rattr_ptr = next_attr;
    /* add on specific query predicates if any */
    temp_alt_ptr = alt_ptr;
    while (temp_alt_ptr != end_alt_ptr)
      /* check if a specific query goes with this segment */
      if (first_time)
        hattr_ptr = first_attr->han_next_attr;
      else
        hattr_ptr = first_attr;
      while (hattr_ptr != NULL && !entered)
        if (ali_name == field_name)
          /* a specific query predicate matches an attribute in this segment */
          strcat (abdl_req, " and (", name, op, value, ")" );
          added_value = TRUE; /* used to determine if additional brace needed */
          entered = TRUE;     /* used to break out of loop */
        else
          hattr_ptr = next_attr;
        /* end while hattr_ptr != null */

      temp_alt_ptr = ali_next_attr;
    if (first_time)
      strcat (abdl_req, ") ]");
    else
      if (segment_parent != NULL || added_value)
        abdl_req[LeadingLPAR] = '(';
        strcat (abdl_req, ")");
      else
        abdl_req[LeadingLPAR] = ' ';
      /* add on the tgt list and by-clause of the request */
      i = strlen(abdl_req); j = rev_count;
      while (template[j] != '\0')
        abdl_req[i] = sit_ptr->Si_template[j];
        ++i; ++j;
      abdl_req[i+1] = '\0';

    free(Si_template);
    sit_ptr = Si_prev;
    if (!first_time)
      seg_ptr = segment_parent;
    first_time = FALSE;
    /* end while !done */

  /* move the ptr to next set of specific predicates. Ptr will be null */
  /* if no 'OR' involved or point to first predicate of next set      */
  alt_ptr = ali_next_attr;
  /* call the Kernel Controller */
  call kernel_controller();
  close_buffs();
  /* end while alt_ptr != null */
  /* after final call to KC, release ALL dli structures */
  /* before returning to normal SQL processing.         */

  /* clean up all the DLI structures */
end-proc   delete_rel_to_dli
```

LIST OF REFERENCES

[1] Demurjian, S. A. and Hsiao, D. K., "New Directions in Database-Systems Research and Development," Technical Report, NPS-52-85-001, Naval Postgraduate School, Monterey, California, February 1985.

[2] Hsiao, D. K. and Harary, F., "A Formal System for Information Retrieval from Files," *Communications of the ACM Vol. 13, No. 2*, (February 1970).

[3] Banerjee, J. and Hsiao, D. K., "DBC Software Requirements for Supporting Relational Databases," Technical Report, OSU-CISRC-TR-77-7, The Ohio State University, Columbus, Ohio, November 1977.

[4] Rollins, R., "Design and Analysis of a Complete Relational Interface for a Multi-Backend Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.

[5] Banerjee, J. and Hsiao, D. K., "The Use of a Database Machine for Supporting Relational Databases," *Proceedings 5th Workshop on Computer Architecture for Nonnumeric Processing* (August 1978).

[6] 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 Vol. SE-6, No. 1*, (January 1980).

[7] Banerjee, J. and Hsiao, D. K., "A Methodology for Supporting Existing CODASYL Databases with New Database Machines," *Proceedings of National ACM Conference* (1978).

[8] Macy, G., "Design and Analysis of an SQL Interface for a Multi-Backend Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, March 1984.

[9] Kloepping, G. R. and Mack, J. F., "The Design and Implementation of a Relational Interface for the Multi-Lingual Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[10] Weisher, D., "Design and Analysis of a Complete Hierarchical Interface for a Multi-Backend Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.

[11] Benson, T. P. and Wentz, G. L., "The Design and Implementation of a Hierarchical Interface for the Multi-Lingual Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1985.

[12] Wortherly, C. R., "The Design and Analysis of a Network Interface for a Multi-Backend Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

[13] Emdi, B., "The Implementation of a CODASYL-DML Interface for a Multi-Lingual Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

[14] Goisman, P. L., "The Design and Analysis of a Complete Entity-Relationship Interface for the Multi-Backend Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

[15] Anthony, J. A. and Billings, A. J., "The Implementation of an Entity-Relationship Interface for the Multi-Lingual Database System," M. S. Thesis, Naval Postgraduate School, Monterey, California, December 1985.

[16] Hsiao, D. K. and , M. J. Menon, "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.

[17] Hsiao, D. K. and , M. J. Menon, "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.

[18] Codd, E. F., "Relational completeness of Data Base Sublanguages," in *Data Base Systems*, (Prentice-Hall, 1972).

[19] Chamberlin, D. D. and Boyce, R.F., "SEQUEL: A Structured English Query Language," *Proc. ACM SIGFIDET Workshop* (May 1974).

[20] Date, C. J., in *An Introduction to Database Systems*, (Addison-Wesley, 1981), 3d edition .

[21] IBM,, *Information Management System IMS/360, Application Description Manual (Version 2)* (1971).

[22] Coker, H., Demurjian, S. A., Hsiao, D. K., Rodeck, B. D., and Zawis, J. A., "The Multi-Model Database System," Technical Report, Naval Postgraduate School, Monterey, California, July 1987.

[23] Rodeck, B., "Accessing and Updating Functional Databases Using CODASYL-DML," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1986.

[24] Coker, H., "Accessing a Functional Database Via CODASYL-DML Statements," M. S. Thesis, Naval Postgraduate School, Monterey, California, June 1987.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
5.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5000	1
6.	Professor David K. Hsiao, Code 52Hq Computer Science Department Naval Postgraduate School Monterey, California 93943-5000	2
7.	Professor Steven A. Demurjian Computer Science & Engineering Department The University of Connecticut 260 Glenbrook Road Storrs, Connecticut 06268	2
8.	Walter Zawis 1230 S.E. 12th Terrace Cape Coral, Florida 33904	2

Thesis
Z2485 Zawis
c.1 Accessing hierarchical databases via SQL transactions in a multi-model database system.