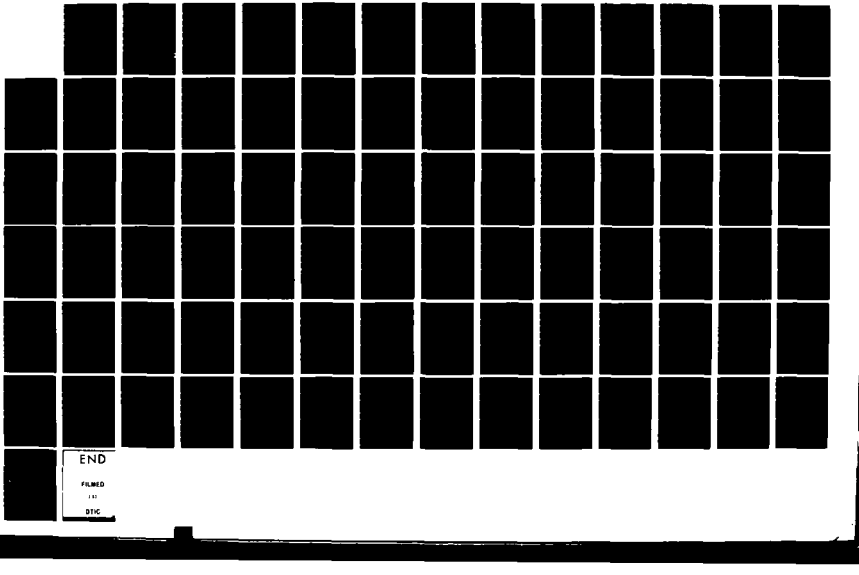
AD-A124 719 A MINICOMPUTER IMPLEMENTATION OF A DATA MODEL INDEPENDENT USER-FRIENDLY I..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. M D GUIDRY 1/1

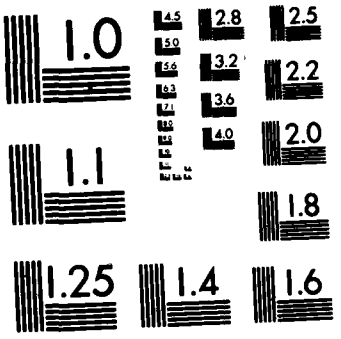
UNCLASSIFIED 15 DEC 82 AFIT/GCS/EE/82D-16 F/G 5/8 NL



END

FILMED

DTIC



DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY (ATC)

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/EE/82D-16

Maj. Lillie
Dr. Potoczny
Dr. Hartrum

A MINICOMPUTER IMPLEMENTATION OF A DATA MODEL INDEPENDENT, USER-FRIENDLY INTERFACE TO DATABASES

THESIS

AFIT/GCS/EE/82D-16 Michael D. Guidry
Capt USAF

Approved for public release; distribution unlimited....

AFIT/GCS/EE/82

A MINICOMPUTER IMPLEMENTATION OF A DATA MODEL INDEPENDENT, USER-FRIENDLY INTERFACE TO DATABASES

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University (ATC)
In Partial Fulfillment of the
Requirements of the Degree of
Master Of Science

by

Michael D. Guidry
Capt USAF

Graduate Information Systems

15 December 1982

Preface

With the advent of local area networks and the ability of users to access many different database management systems (DBMS), there is a need for a user-friendly interface which will allow users to query a database without worrying about which machine it is on or which DBMS it is stored under. Thus, Maj. Chuck Lillie, on the faculty of the AFIT/EN Electrical Engineering Department, proposed that such a system be developed as a Master's Degree thesis. This project was undertaken with the idea that a fully operational system would be delivered at termination; however, no such existing system could be found whose software and/or documentation could be used as a guide. It was decided that a base system be chosen and that the implementation part of this project consist of one major part of this system, and research would be done on the proposed capabilities that the base system did not possess.

Thanks are due to my thesis committee members for their comments during the development and preparation of this document. They were Maj. Chuck Lillie (advisor), Dr. Thomas Hartrum, and Dr. Henry Potoczny. Thanks are also due the AFIT/ENE technicians, especially Dan Zambon who kept the LSI-11's working. Finally, and certainly not least, I would like to thank my wife, Cathy, who managed to "survive AFIT" with me.

Table of Contents

List of Figures

List of Tables

Abstract

An interface system was proposed that would give the user the ability to query a database, regardless of the host DBMS, as long as the user's computer could communicate with the database's host computer.

Investigations were made into existing query resolution systems, and query definition structures and universal data manipulation languages were researched. With this background, a base query resolution system was chosen, and additional capabilities for an improved final query definition structure and a universal data manipulation language were proposed.

In this project, the syntactic analysis part of the query resolution system was implemented. The use of the improved final query definition structure within the query resolution system was defined, and a high level design of the UNIVERSAL_DML (including a pseudo-code representation of the resultant application program) is presented.

Chapter 1 Introduction

1.1 Background

More and more, people are finding reasons to use computers and Database Management Systems (DBMS) to help them solve their problems. Using a DBMS has some considerable advantages: the amount of redundancy in the stored data can be reduced, inconsistency in the stored data can be avoided, the data can be shared, security restrictions can be applied and standards and data integrity can be maintained. However, using a database, or a computer in general, does have some drawbacks. Computers can only do what users tell them to do, not what the users meant to tell them, nor what they wanted to tell them to do. This interaction yields very hostile feelings, sometimes, toward the computer. This problem is software related, and programs can be written so that the user will not feel like he is at the mercy of the machine. Software can be written that would show a menu of possible actions at any time, provide help in choosing an action (definition of the parameters and consequences for a particular action,) and provide meaningful, informative, error messages when necessary. Programs written in such a way can be classified as "user-friendly" since the user is able to percieve what is happening or why it happened and does not feel so hostile toward the machine.

Another of the problems with using computers is the

inability of some systems to work together (specifically the software aspect.) Consider database management systems; there are very many to choose from (DATAPRO lists over one hundred [ref 4].) Even though every DBMS manages data, one DBMS may not be able to access another's data. There are three different data model types commonly used in current DBMSs: network, hierarchical, and relational. Each type has a specific way in which it stores and accesses the data, and by which it defines the structure and relationships involved in the database. Examples of sample databases of each type are shown in Figures 1-1, 1-2, and 1-3.

S

S#	SNAME	STATUS	CITY
S1	SMITH	20	LONDON
S2	JONES	10	PARIS
S3	BLAKE	30	PARIS
S4	CLARK	20	LONDON
S5	ADAMS	30	ATHENS

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	NUT	RED	12	LONDON
P2	BOLT	GREEN	17	PARIS
P3	SCREW	BLUE	17	ROME
P4	SCREW	RED	14	LONDON
P5	CAM	BLUE	12	PARIS
P6	COG	RED	19	LONDON

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Figure 1-1. Examples of Relations in a Relational Data Model Representation of the Suppliers-and-Parts Database.

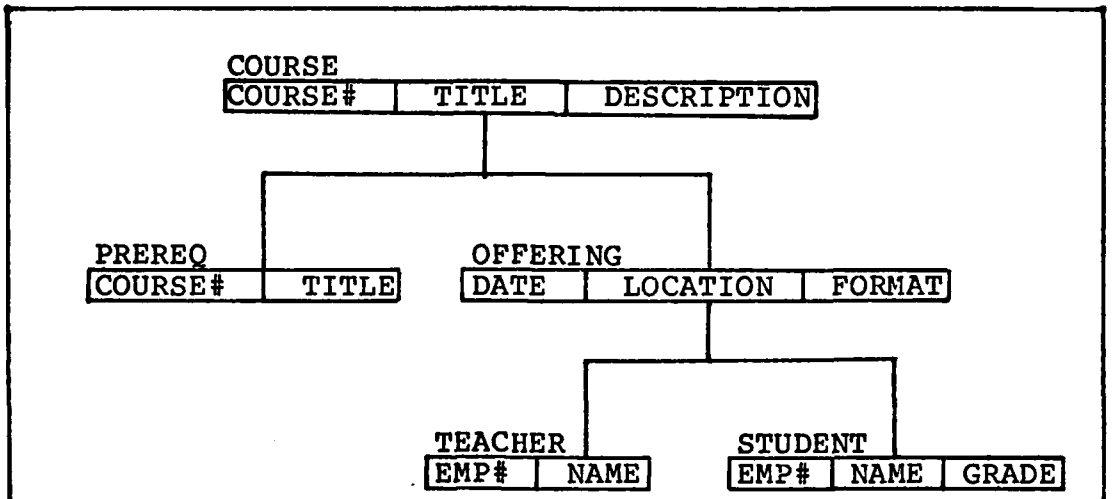


Figure 1-2. Example of Structure and Relation of the Segments of an Education Database Using a Hierarchical Data Model Representation.

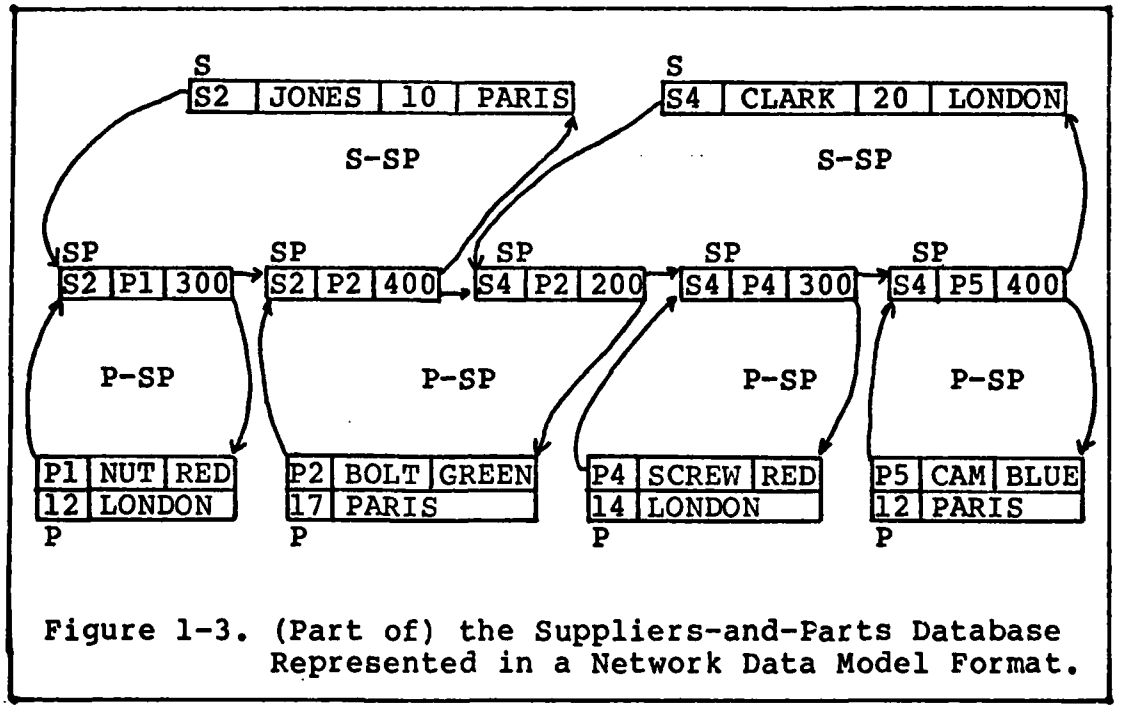


Figure 1-3. (Part of) the Suppliers-and-Parts Database Represented in a Network Data Model Format.

Problems occur not only among DBMSs of different data model types, but also when DBMS of the same type interact. It would be advantageous if a person using a single system could make meaningful queries against any DBMS. It would be a monumental task to write the programs for every possible DBMS that would interpret the necessary information from the data definition language, data dictionary, etc., so that a query might be formed. Development of these 100-plus user-friendly query resolution systems must be regarded as an impossible task. A better way to provide this capability would be to devise a way of representing, regardless of data model type, the structure and relationships involved in the database.

1.2 Statement of Problem

The purpose of this thesis is to 1) research the feasibility of defining queries in some universally translatable form, 2) research the feasibility of defining a data manipulation language that could perform queries against a database, regardless of data model type, and 3) implement a user-friendly query resolution system (using the UCSD Pascal programming language) on one of the DEC LSI-11 computers available in the Digital Engineering Laboratory (DEL) using the results provided by the research done for parts 1 and 2. If this can be achieved, then, not only would databases be accessible through a user-friendly interface on a minicomputer, but the user will not have to be concerned with the data model of the database he is

accessing.

1.3 Scope

The original scope of this thesis was to adapt an existing user-friendly system, design any additions or improvements as necessary according to the research done, and implement the system, AQERS (Another QuEry Resolution System), for testing with the Roth DBMS. The Roth DBMS was chosen because it is a relational system implemented on the DEL LSI-11 computer and is planned to be operational in time for the testing of AQERS.

1.4 General Approach

A literature search was made to find documents concerning the subjects that were to be researched. An effort was made to acquire those documents, and a list of all sources accessed or referenced is presented in the bibliography.

The next step was to identify some existing user-friendly interface systems and determine their potentials for this application. Three systems were identified for comparison: REGIS [ref 7] (Relational General Information System, currently in use within General Motors,) EUFID [ref 8] (End User Friendly Interface to Data Management Systems, developed by System Development Corporation,) and UTOPIA [ref 3] (developed through academic research efforts at the University of Southwestern Louisiana.) The main purpose of each of these systems is

to serve as a user-friendly man-machine interface between the user and the database. The pertinent information concerning the design and implementation of each system and why one was chosen over the others is the subject of Chapter 2 of this thesis.

Next, the question of a universally translatable form for a query definition was examined. The way hierarchical and network databases are pictorially depicted in books, such as An Introduction To Database Systems [ref 5 p. 300, 409], leads one to believe that these types of databases, and therefore queries made against them, can be defined with a tree structure. One article [ref 6 p. 375-376] was found confirming this idea for hierarchical systems. Trees are used to define queries in relational systems such as UTOPIA [ref 3 p. 73] and the Roth DBMS [ref 10 p. 52]. More information about the use of trees to define queries is presented in Chapter 3.

Using the concepts involved with defining the universally translatable query definition structure from Chapter 3, the possibility of defining a universal data manipulation language was examined. Research consisted of examining the available DBMS manuals to find out, given a starting tuple (in a relational DBMS), segment occurrence (in a hierarchical DBMS), or record occurrence (in a network DBMS) how one physically finds the next logically sequential tuple, segment occurrence, or record occurrence. The results showed that the algorithm for

finding the next sequential storage entity varied from DBMS to DBMS. A discussion on the development of such a data manipulation language is presented in Chapter 4.

1.5 Sequence of Presentation

This thesis is broken into six chapters. Following this chapter, the Introduction, there is a chapter on each of the three topics referenced in the Statement of Problem section of this chapter. Chapter 5 contains a description of AQERS and the software implementation involved in this project. Chapter 6 presents conclusions and recommendations.

Chapter 2 Choosing the Base Query Resolution System

2.1 Introduction

This chapter deals with the identification and comparison of some query resolution systems, one of which will serve as a base system for AQERS. Research provided these three user-friendly query resolution systems for comparison: REGIS (RElational General Information System), EUFID (End User Friendly Interface to Data Management Systems), and UTOPIA. This chapter will provide some background and an overview of each system, the criteria used in comparing the systems, and the results of the comparison.

2.2 The REGIS System

REGIS is an interactive system designed to provide convenient, powerful, and flexible information manipulation facilities for the storage, retrieval, and analysis of data. It is designed primarily for an interactive mode of operation in which the user extracts and analyzes information and continues analysis based upon the information just learned. Capabilities are integrated into the system to provide simple access to flexible graphical tools and statistical programs.

REGIS runs on an IBM 370/168 computer under the TSS operating system (the programming language of REGIS is not mentionned.) REGIS is based on the relational data model and is implemented in the REGIS Relational Data Management

System. The system is in use within the General Motors Corporation. The user must be familiar with simple terminology of the concepts and ideas involved, e.g. the data is made of rows ('tuples') and columns ('attributes') which compose rectangular tables of information ('relations') called sets. The user is shown how the command language works on these sets by means of a tutorial showing simple examples. The command language ia a non-procedural relational algebraic language which includes the most useful of the relational functions [ref 7].

2.3 The EUFID System

EUFID is a user-friendly, interactive interface to Data Management Systems (DMS) with the emphasis being on the user's use of natural language to communicate with the system. EUFID was designed by System Development Corporation to be easily transportable. It has been interfaced to a network type database (the World-Wide Data Management System on the Honeywell H-6000 computer) and a relational type database (the INGRES DMS on the PDP-11/70 computer.) It is implemented in both FORTRAN and C for ease of portability to other computers.

The EUFID system is composed of three main modules: QUESTION ANSWERING, SYNONYM EDITING, and HELP. The QUESTION ANSWERING module performs in the following manner: the user types in a question which is forwarded to the analyzer; the analyzer interprets the question and produces

a semantic representation of it; the mapper maps the routing of the flow of data from the database necessary to perform the query and generates an intermediate language representation of the question; the translator, which is DMS dependent, takes the intermediate language and generates the query statements for the DMS; finally, the DMS processes the query statements, accesses the database, and sends the answer back to the user. The SYNONYM EDITING module allows the user to create his own synonym dictionary to redefine words in the semantic dictionary. This supports the central idea of the use of natural language in that the user may redefine words that he is not comfortable with with words that are more familiar to him, e.g instead of using the system word 'OUTPUT', he might substitute the words 'LIST' or 'PRINT'. The HELP module is essentially an on-line User's Manual providing examples of queries which can and cannot be understood by EUFID and examples of EUFID and DMS error messages along with what they mean [ref 8].

2.4 The UTOPIA System

UTOPIA is an interactive query resolution system that gives the casual user, communicating in natural language, the ability to query multiple relational type databases. The system was designed at the University of Southwestern Louisiana as an educational research tool by a doctoral student and three assistants. It is based on the relational data model and implemented in the PL/1 programming language on the Honeywell H-6000 computer,

operating under the MULTICS operating system, to interface with the MULTICS Relational Data Store DBMS.

UTOPIA is composed of four main modules: QUERY, ANALYZER, SEMANTICS, AND RUN_PROC. QUERY is the calling program for the other three modules. QUERY asks for some sign-on information such as the name of the database to be queried, the user dictionary (DIALOG) to be used, and other security and accounting related data. QUERY then calls the ANALYZER module. Under ANALYZER, the user may modify his database's representation and user dictionary and he may input his query to the database. After the user states his query, ANALYZER checks it for syntactical correctness against the user and system dictionaries, and returns to the QUERY module with the query in an intermediate form. The QUERY module then calls SEMANTICS which analyzes the query against the database for semantic correctness, i.e. properly defined relations, properly defined attributes within the referenced relations, valid domains for relational and arithmetic expressions, etc. SEMANTICS also analyzes the query with respect to routing and implied access paths of data during execution of the query. After returning to QUERY, the RUN_PROC module is executed. RUN_PROC interprets the valid query into executable query language for the MULTICS Relational Data Store DBMS which then executes the query and presents the results to the user.

2.5 Comparing the Systems

The following criteria were used in comparing the three systems (listed in order of importance):

1 - It must be user-friendly.

2 - It must be implementable on a minicomputer.

3 - It must be based on the relational data model.

4 - The code must be available, at little or no cost, for modification and improvement.

5 - The code must be written in a well-known, high order language supporting structured programming techiniques.

6 - It must be able to interface with different data model type DBMSs.

7 - It must be able to interface with multiple databases.

These criteria represent the query resolution system that the statement of problem of this thesis would consider ideal. Therefore, it is doubtful that any system meeting all of these criteria can be found; however, one of the proposed systems may meet a large enough subset of the criteria to serve as a base system to which improvements and/or modifications could be made toward attaining the ideal.

First, match the specifications of REGIS against the criteria. It is user-friendly and relationally based. It is implemented on a large computer and no specification is made of the programming language used or its ability to interface with different data model type DBMSs or multiple databases. Since the system is owned by the General Motors Corporation, it is doubtful that the code would be

available at little or no cost, so only criteria 1 and 3 are met.

Matching the specifications of EUFID against the criteria, one finds criteria 1, 2, 5, and 6 are met explicitly. EUFID is not stated to be relationally based; however, it is described as table-driven, i.e. "To support a new application in EUFID, we implement a new set of tables" [ref 8 p. 380]. The contents of the tables closely resemble the UTOPIA system's Meta_Base Dictionary/Directory [ref 3 p. 35] and the user and database vocabularies [ref 3 p. 94]. No specification is made as to the ability to interface with multiple databases, and, again, since the system was commercially developed it is doubtful that the system would be available at little or no cost. EUFID meets over half the criteria, but the possibility that it is not relationally based is a serious possible shortcoming.

UTOPIA meets criteria 1, 3, 4, 5, and 7. Although the system is implemented on a large computer, the code is modularly written (using procedures and functions as functional units) and could probably be implemented on a minicomputer in some form, i.e. a program for each module with intermediate data stored in disk files. Criterion 6 is not possible with UTOPIA since the possible databases are restricted to relational databases (supported by the MULTICS Relational Data Store DBMS) [ref 3 p. 11].

Judging by the number of criteria met by each system,

either EUFID or UTOPIA would be preferable to REGIS. Choosing between EUFID and UTOPIA is more difficult. The concepts and basic flow of control of the query processing are very similar for both systems. This similarity is probably attributable to the influence of the RENDEZVOUS system [ref 2] on both EUFID [ref 8 p. 381] and UTOPIA [ref 3 p. 9]. The RENDEZVOUS system, one of the first natural language projects, was designed for very casual users, handled the difficult problems of how to interact with the user in the user's own words, and even helped the user form his query. Using a case by case analysis of both systems against the criteria, the following comparison is given:

- Both are user-friendly.
- EUFID has been implemented on a minicomputer and UTOPIA probably could be.
- UTOPIA is based on the relational data model and EUFID might be.
- Both are written in well-known high order languages supporting structured programming.
- EUFID can interface with different data model type databases, where UTOPIA can interface with multiple databases.

Using this information the two systems would be equally suited to serve as a base system; however, UTOPIA met one of the criteria that EUFID did not -- number 4 - availability of the code at little or no cost. The code

for UTOPIA is available for the cost involved with making a listing of the system on the computer and mailing the listing to this school. Because both systems are relatively equally suited to serve as the base system for AQERS, the cost involved becomes the deciding factor in choosing between them, and the final decision is to use UTOPIA.

2.6 Conclusion

In this chapter, three query resolution systems were compared to find the best one to serve as the base system for AQERS. An overview was given for each system. Criteria were stated for making the comparison, and the three systems were rated against these. The systems were compared by noting how many and which of the criteria each met. Of the three systems, UTOPIA was judged to be one of the two best and the most economically feasible.

Chapter 3 A Universally Translatable Query Definition

3.1 Introduction

One of AQERS features will be the ability to access a database of any of the three data model types, so the system must be able to communicate meaningful queries to the DBMS. This does not mean that AQERS will, necessarily, generate the particular DBMS's query language statements, but rather that the information necessary to define a query for any DBMS will be placed in some easily translatable form. This chapter will define a universal query definition structure, show what information is necessary to implement such a structure, demonstrate that UTOPIA, the system on which our query resolution system is based, is a sufficient system to handle the implementation, and give an example of how the implementation should work.

3.2 Defining The Query Definition Structure

Relational queries are clearly and easily defined in a tree structure. A parse tree structure [ref 12 p. 221, Fig. 6.1] could serve as a base structure for relational queries. Information would be necessary in the data dictionary to perform the data access path analysis, basically what the parse tree depicts, for queries to relational, hierarchical, and network type databases.

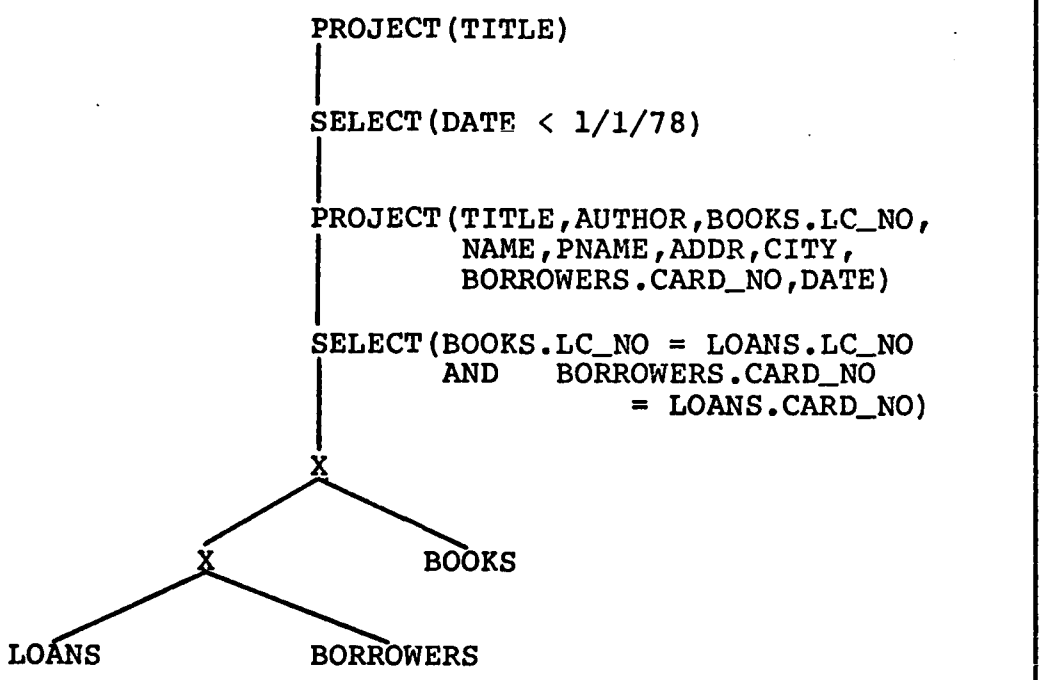


Figure 3-1. Parse Tree Representation of a Relational Query.

Among the many definitions contained in UTOPIA's data dictionary, several are necessary to perform the semantic and data access path analyses. The information used in semantic analysis is provided by the data base administrator. The RELATION_NAMES and DATA_ITEM_NAMES relations [ref 3 p.44,45] contain the information

Table I
Attributes of the RELATION_NAMES Relation

ATTRIBUTE NAME	DESCRIPTION
Relation ID (key)	Coded relation identifier
Relation name	The name of the relation

necessary to accomplish the mapping of the coded identifiers back into the administrator supplied names.

Table II
Attributes of the DATA_ITEM_NAMES Relation

ATTRIBUTE NAME		DESCRIPTION
Relation ID	(key)	Coded relation identifier
Data Item ID	(key)	Coded data item identifier
Data Item Name		The name of the data item

Table III
Attributes of the DATA_ITEM Relation

ATTRIBUTE NAME		DESCRIPTION
Item Name	(key)	The administrator assigned, system-wide, data name
Item Type		Identifies the data type, e.g. char, decimal, binary, etc.
Low Range		The lowest permissible value (used for range checking)
High Range		The highest permissible value
Item Description		DBA supplied natural language description of the data item
Item Status		Current Status, i.e., real, virtual, or proposed
Domain		The domain over which the item is defined
Relation Name		The relation for which the item is an attribute
Position		The relative position of the item in a tuple
Decimal Positions		The number of decimal positions (for a numeric item)
Duplicates		The number of items in the model which have a duplicate name
Owner		The owner of the data item
Creating Procedure		The name of the procedure that creates the data item
Class		Indicates if this is an actual item or the name of a Rel.
Key		Indicates if the item is part of the primary key
Number of Keys		The number of data items which make up the primary key

Any data item in a target relation, one from the database being queried, will have an entry in the DATA_ITEM relation [ref 3 p. 40].

The COMPARABILITY relation [ref 3 p.44] is used for data access path analysis and is loaded through interaction with the database administrator.

Table IV
Attributes of the COMPARABILITY Relation

ATTRIBUTE NAME		DESCRIPTION
Relation ID 1	(key)	Coded relation identifier
Data Item ID 1	(key)	Coded data item identifier
Relation ID 2	(key)	Coded relation identifier
Data Item ID 2	(key)	Coded data item identifier

The PATH_IDENTIFIER, PATH_NODE, and CONNECT relations [ref 3 p.45-48] also contain data access path information, but are loaded by the algorithms that perform the analysis.

Table V
Attributes of the PATH_IDENTIFIER Relation

ATTRIBUTE NAME		DESCRIPTION
Path Number	(key)	Concatenation of relation id's which identify the target relations of the access path
Duplicate Count	(key)	Counter used to uniquely identify an access path
Number of Paths		The number of access paths that exist for a particular combination of relations
Path Length		The number of nodes in an access path

Table VI
Attributes of the PATH_NODE Relation

ATTRIBUTE NAME		DESCRIPTION
Path Number	(key)	Concatenation of relation id's which identify the target relations of the access path
Duplicate Count	(key)	Counter used to uniquely identify an access path
Node Counter	(key)	Counter used to uniquely identify each node of each access path
Relation ID 1		Coded relation identifier for a relation in the node
Relation ID 2		Coded relation identifier for the other relation in the node
Root Relation		The relation from which a tuple is retrieved as the result of processing the node
Production Type		The relation comparability type
Connect Count		The number of data items which "connect" the two relations

Table VII
Attributes of the CONNECT Relation

ATTRIBUTE NAME		DESCRIPTION
Path Number	(key)	Concatenation of relation id's which identify the target relations of the access path
Duplicate Count	(key)	Counter used to uniquely identify an access path
Node Counter	(key)	Counter used to uniquely identify each node of each access path
Connect Counter	(key)	Counter used to uniquely identify each pair of connecting data items
Data Item ID 1		Coded data item identifier for the first connecting data item
Data Item ID 2		Coded data item identifier for the first connecting data item

A data access path for two relations R and S is the sequence of pairs of sets of attributes which "connect" R to S, or in mathematical notation:

> A sequence <(A1,B1) , ... , (An,Bn)> is a logical access path from R to S iff
> $\exists$ R1, ..., Rn+1, R1=R, Rn+1=S, and
> for i=1, ..., n, Ri is directly comparable to Ri+1 via (Ai,Bi) and
> Ri <> Rj for i <> j [ref 3 p.57].

The requirement that Ri <> Rj for i <> j guarantees that the path will be acyclic, which is very important if this analysis is to work on network type databases.

The main function of each of the three relations is to maintain some record of the uniqueness of each access path. Between specific relations, there may be multiple access paths, so in the PATH_IDENTIFIER relation, part of the primary key includes a counter used to identify a specific path. An access path may be composed of many nodes, so a node counter is used in the PATH_NODE relation to identify a specific node within a specific path. Two target relations of a node may be "connected" by many comparable data items, so a connect counter is used in the CONNECT relation to identify and maintain the number of connections possible for a given target relation. By analyzing the information in these relations, it is possible to find out if an access path exists between specified relations, and if so, which one is the shortest. The information placed in these relations is generated by two algorithms during data access path analysis [ref 3 p. 63].

UTOPIA stores attribute comparability relationships in the COMPARABILITY relation. The attribute comparability relationships define which relations have attributes in common, i.e. which data items in which relations have similar data item descriptions [ref 3 p. 55]. This concept can also be applied to network type databases since they also duplicate data items in the "link" records. Hierarchical databases, on the other hand, do not duplicate data items in the segments, so they need another means to define the link. Occurrence dependencies could be stored instead of the attribute comparability relationships. The occurence dependency would define the condition, that given two segments S1, S2, if an occurence of segment S1 exists, then it may have a pointer to an occurrence of segment S2. For example, consider the sample database structure as defined in Fig 1-2. Suppose the following occurrence dependencies are specifed: OFFERING --> TEACHER, and OFFERING --> STUDENT. This means for a given course offering there may be a list of the teacher(s) teaching the course and a list of student(s) taking the course. Since the COMPARABLITY relation expects a comparable attribute to be defined within the segments, the common attributes could either be pointer fields or some dummy attribute defined to be empty, either of which would make them common, maintain the integrity of the actual stored segments in the stored database, and still allow the data access path analysis to be done relative to the comparability of the segments. Keep

in mind that the purpose of this query definition is not to build the DBMS dependent query language statements, but to show the path (via which relations, segments, or records) that the data is to be accessed at any time during the query.

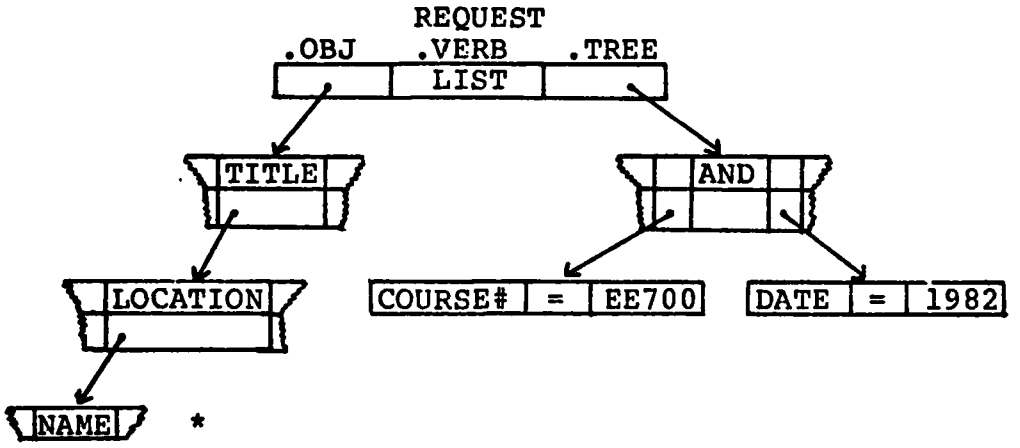
3.3 Relating The Query To The Tree

Next, we will analyze the basic parts of a query and show how each relates to constructing the query definition tree.

There are three basic components of a query. First, there must be a verb. This part tells the system what the user wants to do with the resultant data, e.g. STORE would place the data in a new file, PRINT would send the data to the line printer, SHOW would display the data on the CRT screen, and so on. Somewhere in the query, the user must specify what data he wishes the verb to act on. This part, called the object part, contains a list of data items which is the object of the verb. Most query languages require that a record (for network type databases), a segment (for hierarchical type), or a relation (for relational type) be specified to define a data item; however, our user-friendly system will find them itself through semantic analysis using the data dictionary, or if an item is undefined or ambiguously defined, through interaction with the user. The last component of a query is the qualification phrase. This

part consists of qualifications which the data items accessed during the processing of the query must meet in order for the respective record, segment, or relation occurrence to be considered for further processing; i.e. the qualification phrase specifies which record, segment, or relation occurrences are important for a given query.

Under UTOPIA, the query is transformed by the ANALYZER module into a tree structure as shown by the example in Figure 3-2.



QUERY -- LIST THE TITLE, LOCATION, AND NAME FOR COURSE# = EE700 AND DATE = 1982.

* During semantic analysis, the user will be asked if he wanted TEACHER's NAME or STUDENT's NAME. (TEACHER's NAME will be used for future reference.)

Figure 3-2. The Tree Generated by the ANALYZER Program.

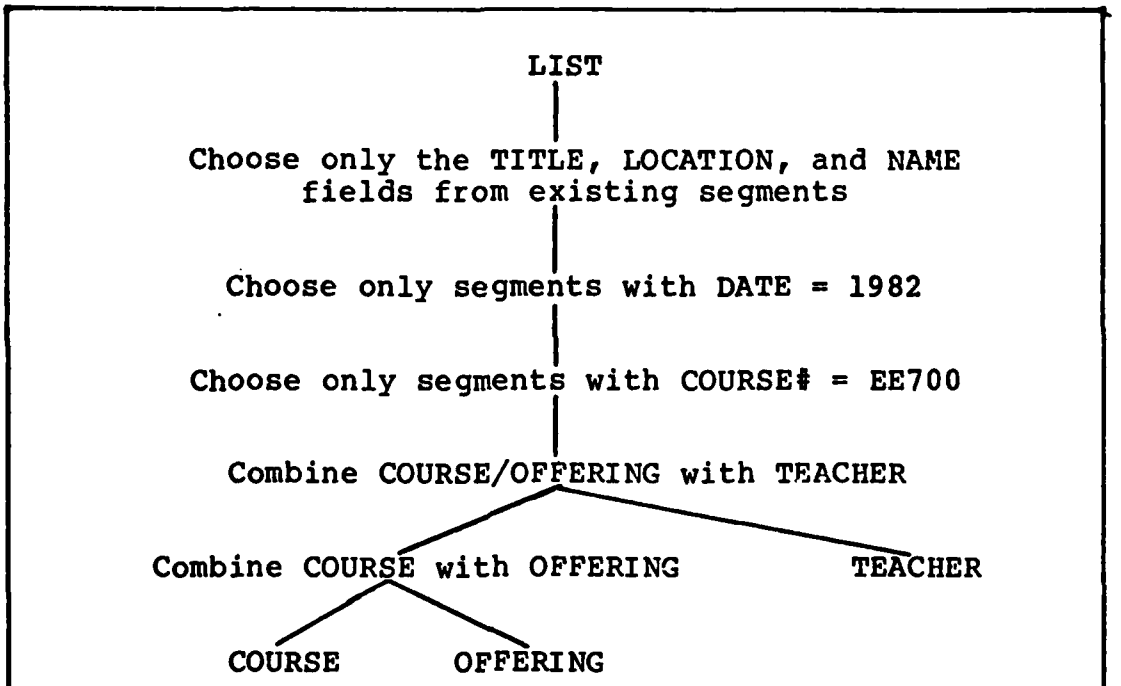
ANALYZER is described in detail in Chapter 5 of this thesis under section "5.3 Description of the Implementation".

UTOPIA describes how the access path detection and generation [ref 3 ch. 4] are accomplished for the relational model. The employment of occurrence dependencies or attribute comparability relationships provides the necessary information so that the same algorithms for generating the data access paths for the relational model can be applied to a UTOPIA data dictionary representing a hierarchical or network type database. UTOPIA uses two algorithms to perform the access path analysis [ref 3 p. 63]. Algorithm #1 uses the information contained in the COMPARE relation to generate a graph with labeled edges representing which nodes (target relations) are "connected" to which other nodes. This information is used to load the CONNECT relation. Algorithm #2 uses the graph to compute all of the paths from one node to all other nodes by concatenating the labels of the edges traversed from the starting node to all other nodes. The associated path represented by the concatenation of the node identifiers (relation id's which identify the target relations) is also generated. Algorithm #2 loads the necessary data into the PATH_IDENTIFIER and PATH_NODE relations.

Semantic analysis of the tree provided by ANALYZER yields, among other things, a list of records, segments, or relations that contain the items specified in the object and qualification parts of the query. Comparing this list with the possible access paths for the database, an access

path can be chosen, if one exists, which provides access to all of the records, segments, or relations in the list.

The access path, the object and qualification parts, and the verb are all the information needed to build the final query definition tree structure. An example of a query definition tree is provided in Figure 3-3 for the sample query described in Figure 3-2.



1. Assume the Occurrence Dependencies are:
   COURSE-->PREREQ, COURSE-->OFFERING,
   OFFERING-->TEACHER, and OFFERING-->STUDENT

2. Assume the access path to be used is:
   COURSE -- OFFERING -- TEACHER

Figure 3-3. The Query Definition Tree.

The access path defines the records, segments, or relations necessary, so they should appear first (at the

bottom of the tree, using postfix ordering). Next, the "data file length" to be used for subsequent processing should be narrowed down by processing the qualification statements which appear next up the tree. Next, only the data items in the object are necessary for the action of the verb, so an additional qualification statement would appear next up the tree which would reduce the "data file width" to only those data items. Finally, the verb would be placed at the top of the tree.

3.4 Conclusion

In this chapter, a universal query definition structure for AQERS has been defined. UTOPIA was shown to provide support for the implementation of such a structure. Finally, an example was provided to show the query at different stages during query resolution (completion of syntax checking and building the final query definition tree).

Chapter 4 A Universal Data Manipulation Language

4.1 Introduction

In Chapter 3, a universal query definition tree structure for AQERS was presented, and it was shown that our query resolution system can generate this query definition for queries against a database of any of the three data model types. The next step in the processing is the execution of the query against the database. This requires that a translator program take the query definition tree and create the actual query language program that accesses the database. These translator programs are DBMS dependent since the acceptable inputs and definitions vary from one DBMS to another. It would be necessary to have a translator program for each DBMS with which the query resolution system interacts. In this chapter, a data manipulation language (DML) system is presented that can process queries against a database, regardless of its data model type. The presentation consists of a discussion of the concept, a definition of the data structures used, a Structured English depiction of the data structures and algorithms defined, and a brief discussion of the foreseeable problems with implementing the system.

4.2 Discussion of the Concept

To reduce the need for terminologies for each of the data model types involved, terms are defined that will be

used throughout the remainder of this chapter: a tuple, segment occurrence, or record occurrence will be called a record; individual data item fields within a record will be known as elements; and, records (in network type systems), segments (in hierarchical systems), or relations (in relational systems) will be referred to as files.

This DML system would use an automated computer program generating system to create a source file for an application program that would use the database's files as data files and process the query given in the query definition tree. The programming language generated would have to support sophisticated record structures and define and access the database's and the program's data files. The program generator would have to be able to access the data dictionary and query definition tree in order to define the file and record structures needed. After the source file is created, the DML system would cause the application program source file to be compiled and executed, and return to the DML system when the application program's execution was finished.

The basic concept of the application program is to create a single file of valid sequentially stored records from all of the files specified in the object part of the query definition tree. Another intermediate file would be created which would contain only those records from the previous file which meet the qualifications specified in the query definition tree. Yet another intermediate file

would be built that would contain records whose only elements would be those specified in the object-part of the query definition tree. Finally, the verb would tell the system what to do with this final intermediate file. Each of these steps is discussed in detail.

The query definition tree defines which files will be used for the query. The DML system needs access to the data dictionary of the database to get information about the number, location, and types of elements in each record on each file so that the respective records and files can be defined within the application program. For each database file to be accessed, there should be a "while not end-of-file" loop. The loops should be nested according to a postfix retrieval ordering of the file nodes within the query definition tree. Before entering each loop, the file should be RESET to the first logical record in the file. Inside each loop, the next loop is executed followed by reading the next (READ_NEXT) logically sequential record in from the file. Inside the deepest nested loop, the records are concatenated and written to a SYSTEM_FILE. This file becomes the current SYSTEM_FILE.

The next step in building this single file is to check for the attribute comparability relationships that occur within each record. The current SYSTEM_FILE is scanned, and each record is checked to see if each set of comparable attributes within the record has the same value. If they do, then the record is written to a new

SYSTEM_FILE; otherwise, the record is considered an invalid record and skipped. On end-of-file on the current SYSTEM_FILE, the new SYSTEM_FILE becomes the current. This file represents the "natural join" of the target QUERY_FILEs.

The next part of the application program is the satisfying of the qualifier part of the query. The steps involved in this iterative process are: 1) retrieve the qualifier part of the query from the query definition tree, 2) RESET the current SYSTEM_FILE, 3) compare the specified value of the element in the first (second, third, ...) qualification against the value of the same element in the record, 4) if the record's element qualifies, then the record is written to a new SYSTEM_FILE, 5) READ_NEXT record from the current SYSTEM_FILE, and 6) if not end-of-file on the current SYSTEM_FILE, then repeat steps 2-6; else if end-of-file is true, then make the new SYSTEM_FILE the current SYSTEM_FILE, and repeat steps 2-6 for each qualification.

The next step in the application program is to build a file, from all of the qualifying records, whose only elements are those specified in the object part of the query definition tree. This is done by 1) RESETing the current SYSTEM_FILE, 2) putting the values of the elements specified by the object-part into the elements of a SYSTEM_RECORD and writing this SYSTEM_RECORD to a new SYSTEM_FILE, 3) READ_NEXT record from the current

SYSTEM_FILE, and 4) if not end-of-file, repeat steps 2-4; else if end-of-file, then make the new SYSTEM_FILE the current one.

The last part of the application program is the processing of the action of the verb, from the query definition tree, on the current SYSTEM_FILE. This could be handled by a case statement or an if-then-else statement in which all possible verbs are specified. When the verb matches a "case" or an "if", then the action would be carried out, e.g. SHOWing the file on the CRT, PRINTing the file on the line printer, or STOREing the file on the disk.

4.3 Definition of the Data Structures Involved

All of the information necessary to define the data structures involved in the algorithms is available in the query definition tree and the data dictionary. For each file used in the tree, a QUERY_FILE would be defined whose elements correspond in definition to those in the file. An assumption is made here that the name of the file in the tree is the same as the DBMS system name for that file. This may be an invalid assumption, and if so, another data dictionary table could be defined in which the DBA would provide the DBMS system name associated with each target file in the database. This relation could be used in conjunction with the query tree to define the names of the QUERY_FILEs. Two SYSTEM_RECORDs are necessary for any query. SYSTEM_RECORD_1 contains all the elements in all

the QUERY_FILEs so that their records can be concatenated and written to and read from a SYSTEM_FILE; SYSTEM_RECORD_2 contains only those elements specified in the object part of the query so that these elements can be written to and read from a SYSTEM_FILE in a record format. Three SYSTEM_FILEs, those into which intermediate data is placed, are necessary to perform any query. The first one is for the combination of all the QUERY_FILEs; the second one is used with the first during the creation of the natural join from the initial combination of the QUERY_FILEs and during the processing of the qualification part of the query. The second SYSTEM_FILE acts as the new SYSTEM_FILE when the first is the current SYSTEM_FILE and vice versa. The third SYSTEM_FILE is the file generated containing only those elements specified in the object part of the query definition tree and is necessary since it is of a different record type.

APPENDIX A provides a Structured English example of the application program described by the data structures and algorithms presented. The query and query definition tree used in this example are those defined in Figures 3-2 and 3-3, respectively. The example query is made against the database described by Figure 1-2, and it is assumed that the DBMS system names for the files are the same as those that appear in the tree.

4.4 Foreseeable Problems

Three problems are immediately obvious in discussing

this system: getting access to an acceptable automated computer program generating system, defining the DBMS dependent functions used within the algorithms, i.e. RESET and READ_NEXT, and being able to schedule the compilation and execution of a source file from a program and then returning to the scheduling program.

There is currently no automated computer program generating system that can generate the application program described using the database's data dictionary and the query definition tree as inputs. This problem could be solved by either buying such a system commercially or developing the system "in-house" via future class projects or thesis work.

The definition of the DBMS dependent function is a more complex problem. The functions depend upon how the particular DBMS physically stores the data in the files. The storage method may vary not only between the DBMSs, but between the files within a DBMS. Consider the storage methods of the following DBMSs, one from each data model type.

System R is a relational system. Relations in System R are represented as stored files. The file is identified at the Research Storage Interface by a numeric identifier called a relation ID (RID). The relational data system is responsible for mapping relation names to RIDs. Records in the stored file represent rows of the relations and are stored a string; however, each string consists of a prefix

containing control information followed by the stored representation of the tuple information. All elements within the record are stored as string type data (including floating point numbers) and variable length elements contain an indication of the element's length [ref 5 p. 173].

IMS is a hierarchical type DBMS. The records comprising a physical database can be stored in any one of four methods: the Hierarchical Sequential Access Method (HSAM), the Hierarchical Indexed Sequential Access Method (HISAM), the Hierarchical Direct Access Method (HDAM), and the Hierarchical Indexed Direct Access Method (HIDAM). The stored database segment occurrence contains the data -- exactly as the user expects -- along with a prefix which the user does not expect. The prefix contains control information for the record consisting of deletion flags, segment type code, pointers and so on depending on the type of access method chosen [ref 5 p. 313].

TOTAL is a network type DBMS. The data structures within a TOTAL database are of two types: master records (containing data elements which are not usually subject to change) and variable records (which link master records and contain more dynamic information). Referencing Figure 1-3, the records labeled P and S are master records, while the records labeled PS are variable records. The stored master records and variable records consist of control information as well as data; however, all of the elements within a

record are defined by the user and are thus definable in the data dictionary. Where a master record is stored within a file is determined by hashing the control key field in the record to an address and chaining, if necessary, to the next available space in the file. (The hashing is done by a proprietary hashing algorithm.) The variable records are stored sequentially in the variable data files. Variable records relating to the same master record are chained together in their file via pointers contained in the control information of the record [ref 1 p. 221] [ref 11 p. 33-38].

As shown by these examples, a set of RESET and READ_NEXT functions will have to be defined for each target database within each DBMS. These routines should consist of instructions in a language that are compatible with our application program and the target DBMS and should effect the return of the first logically sequential record in a given file (RESET) and each succeeding logically sequential record in the given file (READ_NEXT). The READ_NEXT routine should also be able to detect and set the end-of-file condition.

The final problem represents an impossible situation for the currently configured LSI-11's in the DEL. They can run in only a monoprogrammed environment (one program in memory for execution at a time). The ability to schedule the execution of a program, e.g. the compiler, from an originating program and then reenter the

originating program requires 1) an operating system which allows a multiprogramming environment and 2) more memory than is currently available.

The new operating system would have to allow a program to schedule another program and suspend its own execution, which would be restarted upon completion of the scheduled program. AQERS, with its large COMMON unit for the data dictionary, requires all of the current memory. In order to suspend the original program and compile and execute the application program, more memory would have to be available to load the compiler (or the operating system would have to dump memory out to a disk file and read it back in when the original program is to be restarted).

To solve this problem, an upgrade of the LSI-11 is necessary. The new system would need a more sophisticated operating system and more memory.

4.5 Conclusion

In this chapter a universal DML system has been defined. Algorithms were provided for combining the data files, selecting only those records which qualify according to the query, building a file with only the elements desired for the action of the verb specfied by the query, and performing the action of the verb. The data structures used with these algorithms were defined and a sample application program using Structured English was provided. Finally, some potential hinderances to implementation of

this DML system were discussed including additional software needed (i.e. an automated source code generating facility, DBMS dependent RESET and READ_NEXT functions, and an improved operating system) and additional hardware needed (more memory).

Chapter 5 The Query Resolution System

5.1 Introduction

The structure of AQERS resembles that of UTOPIA. The purpose of this chapter is to provide an overview of AQERS, a description of that part of AQERS that is implemented, ANALYZER, and a discussion of the testing facilties that were used (and are still available) for testing ANALYZER.

5.2 Overview of the System

This system is divided into two parts: those routines which perform the actual query resolution and those routines used by the administrators to create and maintain system information.

The QUERY program calls the other main programs in the system: ANALYZER, SEMANTICS, and UNIVERSAL_DML. The QUERY program asks the user to identify himself and the database he wishes to query. This information is used to identify which DIALOG the user will be using during syntactic analysis. If desired, the user's DIALOG may be updated at this time. The database identified is referenced to a table that defines where the target database exists and any other information that is necessary to interact with the host DBMS and computer. Messages are sent to that computer requesting the AQERS data dictionary for the target database. When the data dictionary is received, the QUERY program calls the ANALYZER system.

The ANALYZER system allows the user to input his query and analyzes it for syntactical correctness against the user, database, and system DIALOGs, as well as the system defined standard query constructs. The ANALYZER program is described in more detail later in this chapter. The output of the ANALYZER system is a tree representation called REQUEST that defines the verb of the query, the resultant data items which are the object of the verb, and a subtree which contains the relational and logical expressions comprising the qualifications for the query. Upon completion of the construction of this tree, ANALYZER returns to QUERY.

SEMANTICS is then called by QUERY. SEMANTICS performs semantic analysis on the query defined by REQUEST and chooses a logical data access path for the host files (relations, segments, or records) of the data items involved in the object part of the query. The semantic analysis includes verifying that the data item names exist in the database, finding out which host files contain each of the data items (including resolving any ambiguities involving the same item name in different files through interaction with the user). The domains of each argument in the qualification part are checked for validity (according to the data item definition in the data dictionary), and the arguments on both sides of the relational and arithmetic operators are checked for compatibility. After the data access path is defined (if

possible), the output from SEMANTICS is generated. This output consists of the query definition tree as described in Chapter 3, and following the completion of its construction, SEMANTICS returns to QUERY.

UNIVERSAL_DML is the next routine called by QUERY. Before UNIVERSAL_DML is called, QUERY sends messages to the target database's host computer requesting the database's file access routines, i.e. RESET and READ_NEXT. After receiving them, UNIVERSAL_DML is called. UNIVERSAL_DML, using a source code generating facility, creates an application program for the query defined by the query definition tree. UNIVERSAL_DML then causes this source code file to be compiled and executed. During execution, the application program interacts with the host DBMS of the target database via the RESET and READ_NEXT routines. This interaction yields a local data file whose records are a valid combination of the records in each of the files defined by the query definition tree. Subsequent files are built from this file whose contents represent the selection of only qualifying records and, finally, the projection of only selected data items. The action of the verb is performed on this final file. With the query completed, control is returned to QUERY, and the user may continue processing or exit the system.

Routines necessary for the creation of the data dictionary include routines that allocate and initialize the storage and call the remaining creation functions. The

different data dictionary tables have to be loaded with the basic structural and syntactic information from the target database schema. The administrator will also have to supply attribute comparability information, the location of the target database (and other information necessary to access it), and the names, as stored by the host DBMS, for each of the relations, segments, or records defined in the data dictionary so that the UNIVERSAL_DML will work properly. (These last two tables are not defined by UTOPIA and are not in the current implementaton of the data dictionary.) Finally, routines which generate and store the data access path information in the data dictionary are necessary.

To maintain the DIALOG files, routines are necessary to allow additions and deletions (changes can be effected with these two options) to each of seven files at each of three levels (user, database, and system). Recall that the user may update his DIALOG during execution of the QUERY program. The ability to repack files from which deletions are made must be available to keep used file space at a minimum.

5.3 Description of the Implementation

As part of this thesis effort, one portion of AQERS was implemented, i.e. ANALYZER -- the program which accepts the query and analyzes it for syntactical errors. The analysis is done via a set of files, called the DIALOG, and

several procedures.

The DIALOG is what allows ANALYZER to handle the "natural language" quality in queries. The DIALOG consists of seven files: PATTRN_MATCH, NOISE_PHRASE, VERBS, SEPARATOR, ATTRIBUTE_SYNONYM, OPERATOR, AND VALUE_SYNONYM. In each case except NOISE_PHRASE and SEPARATOR, the records contain substitute values for some part of the query which allows the query to be transformed from something the user wants to something the system can understand. Each record in a file has a two character prefix which identifies "whose DIALOG" that record belongs to. The record in a DIALOG file can exist at one of three levels: the user level, the database level, and the system level. Each of these levels defines a separate DIALOG within the same file. For example, a user's DIALOG consists only of those records in the file that have his prefix code (the INDEX_USER value in the program) in them, and the same is true for other users, different databases (the DB_INDEX value), and the system DIALOG. (There is only one system DIALOG, prefix code of '00', but there can be any number of user and database DIALOGs.)

The routines comprising ANALYZER are ANALYZER, the COMMON unit, the common procedures (POSTFIX and FIND_RIGHT_PAREN), COMMENT_DRIVER, FREE_OBJECT_LIST, FREE_TREE, PREPARE_PHRASE, FIND_PATTERN, NOISE_WORD_DELETE, VERB_SEARCH, FIND_PHRASE, FIND_EXPRESSION, FIND_VALUE_SYNONYM, BUILD_TREE_UP, RELATION_PHRASE_

ANALYZER, FIND_RELATIONAL_OR_LOGICAL_OPERATOR (FI_RL_LG_O), EVALUATE_EXPRESSION, BUILD_TREE, LIST_OBJECTS, and LIST_TREE. Some of these procedures contain internally defined procedures, usually because of UCSD Pascal compiler limitations on the size of a procedure, which will not be discussed. The size of the ANALYZER program and the AQERS COMMON unit required that segmentation be used, therefore several procedures are grouped together under module driver routines which form the segments. These routines are named MOD1DRIVER, MOD2DRIVER, MOD3DRIVER, REL_PH_ANL, AND FIN_VAL_SY. For more information on the segments see APPENDIX B, Bringing Up the ANALYZER Program.

The ANALYZER program is the starting point for processing a user's query. A TRACE may be turned on which allows the user to view the flow of control through the different procedures as well as the values of key variables during analysis of his query. The user may input comments about the system (COMMENT_DRIVER is not currently implemented), and the user may input his query.

A query is expected to have a verb and an object (unless the user chooses to use the default verb, LIST, in which case, the query needs only an object). For example, in the query "LIST ADDRESSES.", LIST is the verb and what the user wants to see listed is ADDRESSES (the object). A query may contain a qualification part separated from the object by a separator. For example, in the query "LIST ADDRESSES WHERE NAME=SMITH.", LIST is the verb, ADDRESSES

is the object, WHERE is the separator, and NAME=SMITH is the qualifier of the query so that only the addresses of the people named SMITH will be listed. Suppose the user has input a query. The analysis of the query follows these steps:

1. The pointers to the object and qualification parts in REQUEST are set to NIL by FREE_OBJECT_LIST and FREE_TREE.
2. The query is prepared for analysis by PREPARE_PHRASE. The system keys off of a blank to define entities within the query (e.g. item names, arithmetic, logical, and relational operators, numbers, etc.) so it makes sure that there are blanks separating these things. It also changes the exponentiation operator ("**") to "!" to remove ambiguity with the multiplication operator ("*").
3. FIND_PATTERN checks to see if any patterns in the valid records of the PATTRN_MATCH file occur in the query. If so, then patterns are replaced by the substitute values in the records.
4. NOISE_WORD_DELETE checks each entity in the query against the valid records of the NOISE_PHRASE file. When matches are found, the entity is deleted from the query.
5. VERB_SEARCH checks the query for a valid verb according to the valid records in the VERBS file. The first entity in the query is expected to be a

valid verb, and if it is not, the default value LIST is assumed to be the verb. If the first entity is a valid verb, then the substitute value in the record is used (e.g. LIST, PRINT, OUTPUT, and SHOW might all have the same substitute value of LIST) and the first entity in the query is deleted.

6. FIND_PHRASE checks each entity in the query against the valid records in the SEPARATOR file, looking for a separator. If a separator is not found then OBJECT_PHRASE (the object) is assumed to be the rest of the query. If a separator is found, then OBJECT_PHRASE becomes everything before the separator and RELATION_PHRASE (the qualification part) becomes everything after the separator.
7. FIND_EXPRESSION checks for any relational expression (delineated by parens) within OBJECT_PHRASE. If any are found, they are deleted from OBJECT_PHRASE and added to the object list.
8. FIND_VALUE_SYNONYM checks the entities in the OBJECT_PHRASE against the valid records of the VALUE_SYNONYM file. Matches are assumed to be adjectives that qualify some object element, so the entity is deletd and a value substitution is added to the subtree (via BUILD_TREE_UP) representing the qualification part of the query. For example, suppose "LOCAL ADDRESS" appears in OBJECT_PHRASE

and LOCAL appears in the VALUE_SYNONYM file. LOCAL would be deleted from OBJECT_PHRASE and some value substitution, e.g. "ADDRESS=DAYTON", would be added to the subtree.

9. FIND_OBJECTS checks for each entity in OBJECT_PHRASE against the valid records of the ATTRIBUTE_SYNONYM file to see if the entity has a synonym. Entities with valid suffixes (currently only "S" and "ES" are implemented) that were not found are checked again. If an entity is matched, then the synonym is added to the object list. If no match is made, the entity itself is added to the list. In no case will "AND" be added to the list. Additions to the list are made by BUILD_OBJECT_LIST.

10. If there is a RELATION_PHRASE, RELATION_PHRASE_ ANALYZER processes it. FIND_VALUE_SYNONYM is called to make any necessary substitutions in RELATION_PHRASE, and RELATION_PHRASE_ANALYZER finds where in the subtree the elements of RELATION_PHRASE will be added. BUILD_TREE is called to identify the elements and add them to the subtree. Arithmetic expressions are placed in postfix order (by POSTFIX) beford being added to the subtree by EVALUATE_EXPRESSION. The only valid logical operators are AND, OR, NAND, and NOR. the only valid arithmetic operators in POSTFIX are "!",

"*", "/", "+", "-", and the unary operators "+", and "-". FIND_RELATIONAL_OR_LOGICAL_OPERATOR (FI_RL_LG_O) matches each word in the phrase against the valid records of the operator file to find either a relational or logical operator. FI_RL_LG_O sets a return parameter whch BUILD_TREE uses to process a relation phrase or a logical phrase. BUILD_TREE processes the left and right sides of a phrase separately, recursively calling itself until the entire RELATION_PHRASE is processed.

11. Finally, if TRACE is active, the REQUEST object list and subtree information will be printed out by LIST_OBJECTS and LIST_TREE, respectively.

5.4 Testing the ANALYZER Program

The current implementation is very well suited to testing. Part of this was by design and part grew out of necessity. The TRACE option allows the user to follow the flow of control from one procedure to the next and watch the values of key parameters as the query resolution process takes place. A SEGMENT PROCEDURE must reside inside of a program in order to be compiled, and these programs are usually "dummy" programs consisting only of "BEGIN END." However, the dummy programs of this implementation are calling programs for the segment, and they allow the segments to be executed separately. The dummy programs allow an option of which procedure in the

segment is to be tested amd then request values for the appropriate input parameters for the chosen procedure. With this configuration, the tester can input any variation of input parameters to a given procedure until he is satisfied that the entire procdeure has been tested. In this mode of testing the TRACE option is always turned on. An additional program called TEST.TEXT was necessary to test the COMMON units procedures, POSTFIX and FIND_RIGHT_PAREN, since a unit does not require a program to be compiled. The TEST program works much like the dummy programs mentioned except no option is necessary (FIND_RIGHT_PAREN is so short that no separate testing was unnecessary).

Time ran out for coding before the procedures could be written that would allow the DIALOG files to be loaded and maintained. Code was added to the procedures that use the files so that a tester can define a file record to be used as if the record was found on the file itself. The tester must accustom himself to this process, since, depending on the procedure, the tester may be asked if he wishes to define a record once, three times (once for each level of DIALOG), once for each entity, or three times for each entity.

Each procedure was thoroughly tested in the segment test mode before being linked into the single code file mode for system testing. The benefit of testing a system like this is that for a given input parameter, if the

procedure works in the segment test mode, it should work in the system test mode. This is true because the same code file used in the segment test is the one linked into the system code file. (See APPENDIX B for more details about linking the system.) If the system test does not work for the same input values, it can be safely assumed that the problem is external to the segment being tested.

5.5 Conclusion

This chapter covers the proposed query resolution system, AQERS. An overview of how the system will work and what will be included in the system is provided. The implementation of ANALYZER is discussed including the DIALOG concept and implementation, the formation of queries for the system, the flow of control through the procedures during the syntactical analysis, and the testing facilities available for ANALYZER.

Chapter 6 Summary, Recommendations, and Conclusion

6.1 Summary

The purpose of this thesis was to provide an interface system, to be executed on a minicomputer, that would allow a user to query a database, without regard to the host DBMS, as long as the subject computers could communicate with each other. To this end, research was done on existing query resolution systems, query definition structures, and universal data manipulation languages.

In Chapter 2 of this thesis, three query resolution systems (i.e. REGIS, EUFID, and UTOPIA) were presented. An overview was provided for each of the systems. A set of judging criteria was presented, and each of the systems was rated against these criteria. The systems were then compared and UTOPIA was chosen to be the base system for this thesis effort since it was one of the two best and the least expensive to acquire of those two.

A query definition structure that can define a query of a database of any data model type is proposed in Chapter 3. The idea for the structure came from the parse tree structure which can be used to describe relational queries. The information necessary to build the structure as well as the information the structure would contain is defined. This structure would be generated during execution of the SEMANTICS part of the query resolution system and would be used as input to UNIVERSAL_DML.

The Universal Data Manipulation Language System (UNIVERSAL_DML), as described in Chapter 4, would generate a source code file that represents an application program for the query as defined by the query definition structure. UNIVERSAL_DML would also compile and execute this source code file, thereby completing the query resolution process. Algorithms are provided for generating the source code file, and there is a discussion of some problems with implementing the UNIVERSAL_DML system.

In Chapter 5, an overview of the proposed query resolution system, AQERS, is provided along with a description of the implementation of this effort, the ANALYZER program. The overview includes an analysis of the flow of control during the query resolution process, the inputs and outputs of each part of AQERS, and the programs that database administrators will need to support AQERS. The description of the implementation includes a discussion of the DIALOG concept which allows the system to resolve queries made using natural language, a brief discussion on the possible structure of queries for ANALYZER, and a detailed analysis of the flow of control during the syntactical analysis part of query resolution. Finally, a description of the testing facilities available in ANALYZER are provided, including the ability to monitor the flow of control from one procedure to the next, the ability to watch the values of key parameters during syntactical analysis, and the ability to input records, simulating

DIALOG files.

6.2 Recommendations

APPENDIX C is a hierarchical chart depicting the modules of AQERS and their current statii. The following is a list of future projects necessary to implement AQERS in its proposed form.

1. Develop the system for DIALOG file creation and maintenance. This system has not been designed; however, the record structures for each file are defined in the ANALYZER program, and the actual coding should be trivial. When coded, the actual files must be implemented in the ANALYZER program via Pascal "OPEN" and "CLOSE" commands in the procedures that use the files.

2. Perform a requirements analysis on the size of the arrays, relations, etc. in the data dictionary necessary to define a single database. The sizes used in UTOPIA are to handle up to 10 database declarations in a "Meta-base". In the AQERS implementation, each database would have its own data dictionary. Reducing the size of these storage elements would reduce the size of the COMMON unit, as implemented. Currently, the entire COMMON unit cannot be implemented and much of it is "commented out". Reducing the size of COMMON would allow larger segments or a combination of segments, in ANALYZER's case. Combining the MOD1DRIVER and MOD3DRIVER segments in ANALYZER would free a segment. The QUERY calling program could then be implemented as the main program, and SEMANTICS could be

implemented under this configuration.

3. Develop the data dictionary creation routines. These routines have not been designed; however, this function is already performed in UTOPIA. Using this code as a guide and remembering to include the two proposed tables not in UTOPIA (one for associating a machine with a database and one for associating the DBMS system names with the data dictionary names for the records, segments, or relations) (see Chapter 5.2 for more detail of these tables), this coding should not be difficult.

4. Develop the software necessary to communicate and transfer files (i.e. the data dictionary and the RESET and READ_NEXT procedures) between computers.

5. Convert (from the PL/1 programs in UTOPIA), develop, and implement the SEMANTICS part of AQERS. This should include the implementation of the universal query definition structure.

6. Improve the hardware and software configuration of the LSI-11 so that the UNIVERSAL_DML part of AQERS can be implemented. The changes would consist of more memory and an improved operating system (OS). The amount of additional memory needed would not be known until the sizes of the new operating system, the UNIVERSAL_DML system, and the compiler were known (all would have to be in memory at the same time). A virtual operating system would take care of the memory problem, but the new OS would at least allow a program (AQERS) to suspend itself while another program

(the query application program) is compiled and executed, and then restart itself upon completion of the scheduled program.

7. Develop the UNIVERSAL_DML system (including the source file generating facility) which would create, compile and execute an application program for a query defined by the query definition structure.

6.3 Conclusion

AQERS, when implemented, will be a very powerful tool in a network environment. From a mini-computer, a user will be able to query any database on any machine with which his computer can communicate. His query may be formed using a natural language, and the user will not be worried with which DBMS he must work with, what data model type it is, which application programs he must use, etc. All he will need to know is which database he wishes to access and what his query is. AQERS will be a very useful, user-friendly tool.

References

1. Cardenas, Alphonso F. Data Base Management Systems. Boston, Mass: Allyn and Bacon, Incorporated, 1979.

2. Codd, E. F. et al "RENDEZVOUS Version 1: An Experimental English-Language Query Formulation System for Casual Users of Relational Data Bases," IBM Research Report RJ2144 (January 1978).

3. Cousins, Thomas R. "A Methodology for the Inferential Derivation of Retrieval Semantics Utilizing a Relational View of a Meta-base," a dissertation presented to the University of Southwestern Louisiana, (December 1978).

4. -----"Datapro 70 The EDP Buyer's Bible," volume 3, section 70E. Delran, New Jersey: Mc Graw Hill Book Company, 1982.

5. Date, C. J. An Introduction to Database Systems (Third Edition). Reading, Mass: Addison-Wesley Publishing Company, 1981.

6. Hardgrave, Terry W. "Ambiguity in Processing Boolean Queries on TDMS Tree Structures: A Study of Four Philosophies," Proceedings of the 5th International Conference on Very Large Data Bases, p. 373-397, (1979).

7. Joyce, J. D. and N. N. Oliver "REGIS -- A relational information system with graphics and statistics," AFIPS Proceedings of the National Computer Conference (volume 45), p. 831-844, (1976).

8. Kameny, I. et al "EUFID: The End User Friendly Interface to Data Management Systems," Proceedings of the 4th International Conference on Very Large Data Bases, p. 380-391, (1978).

9. Lien, Y. Edmund "On the Equivalence of Database Models," Journal of the A.C.M. (volume 29 #2), (April 1982).

10. Roth, Mark A. "The Design and Implementation of a Pedagogical Relational Database System," a thesis presented to the Air Force Institute of Technology, Wright-Patterson A.F.B., Ohio, (Fall 1979).

11. -----TOTAL Training Notes for VAX Computers. Cincinnati, Ohio: CINCOM Systems, Incorporated, 1980.

12. Ullman, Jeffrey D. Principles of Database Systems. Rockville, Md: Computer Science Press, Incorporated, 1980.

Unreferenced Bibliography

1. Chen, Peter The Entity-Relationship Approach to Logical Database Design. Wellesley, Mass: Q.E.D. Information Services, Incorporated, 1977.

2. Hsiao, David K. and M. Jaishankar Menon "Design and Analysis of a Multi-Backend Database System for Performance Improvement, Functionality Expansion, and Capacity Growth," Ohio State University Computer and Information Science Research Center TR-81-7 (Part I, Chapter 3), (July 1981).

3. Kirby, J. and R. L. Kasyhap "An Approach for Communicating with a Data Base Using English Queries," IEEE Procedures of COMPSAC, p. 650-656, (1977).

4. Lozinskii, E. L. "On query-answering in relational data bases," AFIPS proceedings of the National Computer Conference (volume 48), p. 717-720, (1979).

5. -----MULTICS PL/1 Reference Manual (Series 60 Level 68). Honeywell Information Systems, Incorporated, June 1976.

6. Pollack, Seymour V. and Theodore D. Sterling A Guide to PL/1. New York: Holt, Rinehart, and Winston, Incorporated, 1969.

7. Schneider, Lowell I. and Michael Levin "Data Independent Accessing Methodology (DIAM) Distributed Access," Rome Air Development Center, Griffiss A.F.B., New York, (June 1981).

APPENDIX A

Structured English Example of the Application Program for a Sample Query

```
PROGRAM QUERY_APPLICATION_PROG;

VARIABLES

  (*        QUERY_FILE DECLARATIONS                          *)

  (* THERE ARE AS MANY QUERY_DATA FILES AS ARE USED          *)
  (* IN THE FILES PART OF THE QUERY DEFINITION TREE          *)

  COURSE     : FILE OF RECORD(COURSE#,TITLE,DESCRIPTION);
  OFFERING   : FILE OF RECORD(DATE,LOCATION,FORMAT);
  TEACHER    : FILE OF RECORD(EMP#,NAME);


  (*           SYSTEM_RECORD DECLARATIONS                    *)

  (* ONLY TWO SYSTEM_RECORDS ARE NECESSARY, ONE TO           *)
  (* COMBINE THE RECORDS OF ALL OF THE QUERY_DATA            *)
  (* FILES, AND ONE TO BUILD THE FILE WITH ONLY THE          *)
  (* OBJECT ELEMENTS IN IT                                   *)

  SYSTEM_REC_1     : RECORD(COURSE#,TITLE,DESCRIPTION,
                            DATE,LOCATION,FORMAT,EMP#,
                            NAME);
  SYSTEM_REC_2     : RECORD(TITLE,LOCATION,NAME);


  (*         SYSTEM_FILE DECLARATIONS                       *)

  (* THREE SYSTEM_FILES ARE NECESSARY TO PERFORM THE         *)
  (* QUERY: ONE TO COMBINE THE QUERY_DATA FILES, ONE         *)
  (* MORE TO CREATE THE NATURAL JOIN OF THE COMBINED         *)
  (* QUERY_FILES AND PERFORM THE QUALIFICATION CHECKS,       *)
  (* AND A THIRD ONE TO REDUCE THE NUMBER OF ELEMENTS        *)
  (* TO ONLY THOSE SPECIFIED IN THE OBJECT PART OF           *)
  (* THE QUERY.                                              *)

  SYSTEM_FILE_1    : SYSTEM_RECORD_1;
  SYSTEM_FILE_2    : SYSTEM_RECORD_1;
  SYSTEM_FILE_3    : SYSTEM_RECORD_2;


BEGIN   (*  PROCESSING  *)

   (* SETUP_SYSTEM_FILES MAKES FIRST PARAMETER READY
      FOR READING AND SECOND PARAMETER READY FOR
      WRITING    *)
```

```
SETUP_SYSTEM_FILES(DUMMY_FILE,SYSTEM_FILE_1);

RESET(COURSE);          (* RETURNS FIRST LOGICALLY
                           SEQUENTIAL RECORD IN
                           FILE  *)
WHILE NOT EOF(COURSE) DO
BEGIN
   RESET(OFFERING);
   WHILE NOT EOF(OFFERING) DO
   BEGIN
      RESET(TEACHER);
      WHILE NOT EOF(TEACHER) DO
      BEGIN
         SYSTEM_RECORD_1 := CONCAT(COURSE,OFFERING,
                                              TEACHER);
         WRITE(SYSTEM_FILE_1,SYSTEM_RECORD_1);

         (* READ_NEXT PROVIDES THE NEXT  *)
         (* LOGICALLY SEQUENTIAL RECORD  *)
         READ_NEXT(TEACHER);
      END;
      READ_NEXT(QUERY_FILE_B)
   END;
   READ_NEXT(QUERY_FILE_A)
END;   (* COMBINING THE RECORDS OF THE QUERY_FILES *)

SETUP_SYSTEM_FILES(SYSTEM_FILE_1,SYSTEM_FILE_2);
READ(SYSTEM_FILE_1,SYSTEM_RECORD_1);
WHILE NOT EOF(SYSTEM_FILE_1) DO
BEGIN
   (* CHECK_COMPARABLE_ATTRIBUTES IS A FUNCTION
      WHICH CHECKS ALL OF THE COMPARABLE
      ATTRIBUTES FOR AN INPUT RECORD AS DEFINED
      BY THE "COMPARABILITY" TABLE IN THE DATA
      DICTIONARY.  IT IS TRUE IF EACH SET OF
      COMPARABLE ATTRIBUTES IS EQUAL AND FALSE
      IF THEY ARE NOT   *)
   IF CHECK_COMPARABLE_ATTRIBUTES(SYSTEM_RECORD_1)
         = TRUE THEN  WRITE(SYSTEM_FILE_2,
                                SYSTEM_RECORD_1);
   READ(SYSTEM_FILE_1,SYSTEM_RECORD_1)

END;    (* NATURAL JOIN OF COMBINED QUERY_FILES *)


(* COMPARE RECORDS AGAINST QUALIFICATIONS OF THE   *)
(* QUERY DEFINITION TREE                           *)

GET_NEXT_QUALIFICATION;  (* RETURNS FIRST
                            QUALIFICATION WHEN CALLED
                            THE FIRST TIME  *)
WHILE (MORE_QUALIFICATIONS_TO_PROCESS) DO
BEGIN
   SETUP_SYSTEM_FILES(SYSTEM_FILE_2,SYSTEM_FILE1);
```

```
      READ(SYSTEM_FILE_2,SYSTEM_RECORD_1);
      WHILE NOT EOF(SYSTEM_FILE_2) DO
      BEGIN
         IF QUALIFICATION = SYSTEM_RECORD_1.SAME_ELEMENT
               THEN WRITE(SYSTEM_FILE_1,SYSTEM_RECORD_1);
         READ(SYSTEM_FILE_2,SYSTEM_RECORD_1)
      END;
      GET_NEXT_QUALIFICATION  (* RETURNS THE NEXT
                                  QUALIFICATION WHEN
                                  CALLED AGAIN  *)
   END;


   (* EXTRACT ONLY THOSE ELEMENTS AS SPECIFIED IN THE *)
   (* OBJECT PART OF THE QUERY DEFINITION TREE AND    *)
   (* PUT THEM IN A FILE                              *)

   (* MAKE CURRENT READY FOR READING AND SYSTEM_FILE_3 *)
   (* READY FOR WRITING                                *)
   SETUP_SYSTEM_FILES(SYSTEM_FILE_1,SYSTEM_FILE_3);

   READ(SYSTEM_FILE_1,SYSTEM_RECORD_1);
   WHILE NOT EOF(SYSTEM_FILE_1) DO
   BEGIN
      SYSTEM_RECORD_2.COURSE# :=
                     SYSTEM_RECORD_1.COURSE#;
      SYSTEM_RECORD_2.LOCATION :=
                     SYSTEM_RECORD_1.LOCATION;
      SYSTEM_RECORD_2.NAME := SYSTEM_RECORD_1.NAME;
      WRITE(SYSTEM_FILE_3,SYSTEM_RECORD_2);
      READ(SYSTEM_FILE_1,SYSTEM_RECORD_1)
   END;

   (* PERFORM THE VERB, AS SPECIFIED IN THE QUERY DEF *)
   (* TREE, ON THE OBJECT FILE                        *)

   GET_VERB; (* RETRIEVE VERB FROM QUERY DEFINITION
                  TREE          *)
   (* MAKE SYSTEM_FILE_3 READY FOR READING *)
   SETUP_SYSTEM_FILES(SYSTEM_FILE_3,DUMMY_FILE);

   IF VERB = 'PRINT' THEN
         OUTPUT(LINEPRINTER,SYSTEM_FILE_3);
   ELSE IF VERB = 'LIST' THEN
         OUTPUT(CRT,SYSTEM_FILE_3)
   ELSE IF VERB = 'STORE' THEN
         WRITE(CRT,'SYSTEM_FILE_3 IS STORED ON DISK')
   ELSE WRITE(CRT,VERB,' IS NOT AN IMPLEMENTED VERB');


END.  (* QUERY_APPLICATION_PROG  *)
```

APPENDIX B

User's Guide: Changing the System

Overview

Please refer to sections 3.3.1 and 3.3.2 of the UCSD Pascal Version II.0 reference manual and current program listings before trying to change the system.

The program is currently divided into a main body segment, five segment procedures and a unit. Each segment is contained in a dummy program in order to permit separate compilation. The unit contains all types, variables, and procedures which are global to more than one segment. Thus by including the unit in each dummy program, access is allowed to those elements. The format for each segment procedure is:

```
PROGRAM dummy-name;
USES COMMON;  (*the unit is named COMMON*)
SEGMENT PROCEDURE name(parameter-list);
  Local types, variable, and procedures;
BEGIN
  body of name;
END; (*name*)
BEGIN
END. (*dummy name*)
```

Since the segments are separately compiled, the parameter list of the segment procedure must contain all global variables accessed or modified by the procedure. The program or segment procedure which calls each segment procedure must have a dummy segment procedure with the same name, so that it may compile properly. The format for the

main body segment is:

```
PROGRAM main;
USES COMMON;
 local labels,types,constants, and variables;
SEGMENT PROCEDURE name1( --- );
BEGIN
END; (*name1*)
SEGMENT PROCEDURE name2( --- );
BEGIN
END; (*name2*)
.
.
.
other local procedures;
BEGIN
  body of main;
END. (*main*)
```

The format for segment procedures which use other segments is the same as the previous format for a segment procedure except dummy segment procedures are included as local procedures.

Each segment procedure has a particular segment number from 11 to 15 associated with it. The main body segment has number 1 and the unit has number 10. Other numbers are for Pascal use only. The way numbers are assigned to the segment procedures is in first compiled, first numbered order. Thus, in the above format for the main body segment name1 would be assigned 11, name2 assigned 12, etc. Therefore, in each dummy program used to define a segment procedure, an appropriate number of dummy segments must exist before the defined segment to ensure that the segment count is the same. Thus, for example, the format for segment name2 would be:

```
PROGRAM dummy name;
USES COMMON;
SEGMENT PROCEDURE dummy name1;
BEGIN
END; (* dummy name1 *)
SEGMENT PROCEDURE name2( --- );
  local labels, types, etc.
BEGIN
  body of name2
END; (*name2*)
BEGIN
END.  (*dummy name*)
```

The unit -- COMMON -- is compiled and placed in the system library using the librarian program, LIBRARY.CODE. (See section 4.2 of the UCSD Pascal manual.) When each program is compiled, the unit is retrieved from the system library and used in the program. However, each program must still be linked with the system library in order to bind the external variable and procedure references into the unit. After each program is compiled and linked, then the librarian may be used to put all the segments together into one code file. Each code file containing the segment is retrieved and linked into the proper space of the final code file using its assigned segment number into the overall file.

The UCSD Pascal INCLUDE directive (see section 1.6.1 of the UCSD Pascal Manual) allows text files to be included in a segment without inserting the actual source code. The location of the include files, relative to the segment procedure, becomes very important since the disk that the include file is on must be stated in the segment procedure source code.

If a particular segment, including the main body segment, is to be changed then the steps to be followed are:

1) Change the source code for the segment.
2) Compile the program containing the segment.
3) Link the code file to the system library.
4) Using the librarian create a new overall file passing all unchanged segments to the new file and linking in the changed segment.

If the unit has to be changed then after compiling it and placing it into the system library, each program must be recompiled, linked to the system library, and then put together with the librarian.

Bringing Up the ANALYZER Program

To bring up the Data Definition Facility the following steps should be followed:

1. First, compile COMMON.TEXT, making sure that its include files (POSTFIX and FIND_RIGHT_PAREN) are available such that the object code resides in COMMON.CODE.
2. Execute the System Librarian by typing "X" at the system level. The following prompt line will appear on the CRT:

   EXECUTE WHAT CODE FILE --->

3. Enter "LIBRARY".

4. Next, the following prompt will appear:

```
OUTPUT CODE FILE --->
```

5. Enter a "*" or "SYSTEM.LIBRARY".

6. Next, the user will be prompted as follows:

```
LINK CODE FILE --->
```

7. Enter #COMMON.CODE where # is #4/#5 and the screen will appear as follows:

```
LINK CODE FILE ---> #COMMON.CODE

0-        0  4-        0  8-        0 12-        0
1-        0  5-        0  9-        0 13-        0
2-        0  6-        0 10-COMMON     14-        0
3-        0  7-        0 11-        0 15-        0

OUTPUT CODE FILE ---> *
```

The user may now link the COMMON segment into segment 10 of the output code file by typing a "10", (<sp>) and "10". In addition to linking this segment into the SYSTEM.LIBRARY, which resides on the system disk, the two segments PASCALIO and DECOPS must also be linked back into the System library, SYSTEM.LIBRARY. In order to do this, type "N" for new. This indicates to the librarian that segments from a new code file are to be put into the output code file. The librarian will respond with the following prompt:

```
LINK CODE FILE --->
```

8. Enter a "*" or "SYSTEM.LIBRARY".

9. To link PASCALIO, type a "2", followed by a (<sp>), then a "2".

10. To link DECOPS, type a "3", followed by a (<sp>), then a "3".

Now, the SYSTEM.LIBRARY should look like the following:

```
0-              0 4-        0  8-          0 12-        0
1-              0 5-        0  9-          0 13-        0
2-PASCLIO    1824 6-        0 10-COMMON      14-        0
3-DECOPS     2092 7-        0 11-          0 15-        0
```

To exit from the Librarian do the following:

11. Type "Q" for QUIT.

12. ENTER (<cr>) to exit LIBRARIAN.

At this point, the ANALYZER software can be compiled. This software resides in the following TEXT files:

ANALYZER - the executive for the ANALYZER program.

MOD1DRIVER - the file that contains the source code for the MODIDRIVER segment consisting of a module driver and the following INCLUDE TEXT files: COMMENT_DR, FRE_OBJ_LS, FREE_TREE, and PREP_PHRAS.

MOD2DRIVER - the file that contains the source code for the MOD2DRIVER segment consisting of a module driver and the following INCLUDE TEXT files: FIND_PATTE, NOIS_WRD_D, VERB_SEARC, FIND_PHRAS, and FIND_EXPR.

MOD3DRIVER - the file that contains the source code for the MOD2DRIVER segment consisting of a module driver and the following INCLUDE

TEXT files: LIST_OBJEC, LIST_TREE, BLD_OBJ_LS, and FIND_OBJEC.

REL_PH_ANL - the file that contains the source code for the REL_PH_ANL segment consisting of the REL_PH_ANL procedure and the following INCLUDE TEXT files: FI_RL_LG_O, EVAL_EXPR, and BUILD_TREE.

FIN_VAL_SY - the file that contains the source code for the FIN_VAL_SY segment consisting of the FIN_VAL_SY procedure and the BLD_TRE_UP INCLUDE TEXT file.

To link the ANALYZER source code together, continue with the following steps:

13. Compile ANALYZER.TEXT, MOD1DRIVER.TEXT, MOD2DRIVER.TEXT, MOD3DRIVER.TEXT, REL_PH_ANL.TEXT, and FIN_VAL_SY.TEXT making sure that their INCLUDE files are available.

14. Once all the source files have been compiled properly into their appropriate CODE files, they can be linked together into one CODE file using the LIBRARIAN. To invoke the LIBRARIAN, repeat steps 2 thru 4. Once, the prompt has been displayed, enter a valid CODE file name.

15. Next, the following prompt will be displayed:

    LINK CODE FILE --->

16. Enter the code file for ANALYZER. To link the

ANALYZER program into the output code file link segment 1 to segment 1. Next, enter a "N" for new. This will indicate to the LIBRARIAN the fact that segments from a new code file are to be linked into the output file. For the new code file enter the code file for MOD1DRIVER. To link this segment into the output code file, link segment 11 to segment 11. Continue this process, linking MOD2DRIVER into segment 12, MOD3DRIVER into segment 13, REL_PH_ANL into segment 14, and FIN_VAL_SY into segment 15.

17. To finish the linking process, PASCALIO, DECOPS, and COMMON must all be linked into this output code file. To do this type "N" and for the LINK CODE FILE enter "*" or SYSTEM.LIBRARY. To link these segments into the output code file, link segment 2 to segment 2, segment 3 to segment 3, and segment 10 to segment 10. The output code file should look like the following:

```
0-            0  4-  0 8-              0 12-MOD2DRIVER
1-ANALYZER       5-  0 9-              0 13-MOD3DRIVER
2-PASCALIO 1824  6-  0 10-COMMON         14-REL_PH_ANL
3-DECOPS   2092  7-  0 11-MOD1DRIVER     15-FIN_VAL_SY
```

18. The final step is to link the output code file by executing the LINKER. To do so, type an "L" at the Pascal system level. The following prompts will be displayed:

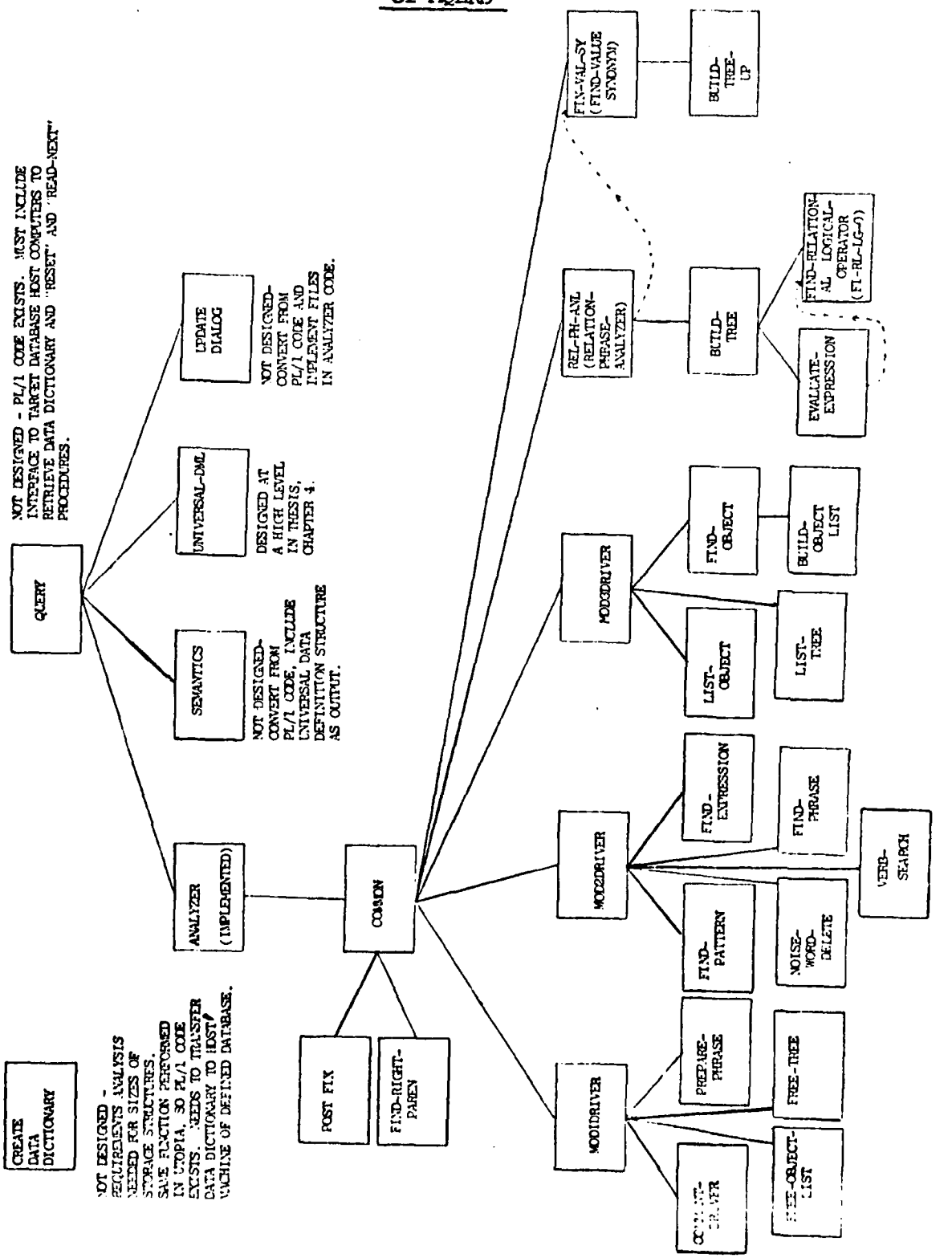
```
Host file?    enter output code file that contains all
              segments to be linked.
Lib file?     enter (<cr>).
Map file?     enter any valid file name.
Output file?  enter ANALYZER.CODE or some other
                valid code file name, this will be the
                executable code file.
```

Bringing Up Segments

Using the procedures mentioned above in step 18, a segment can be linked for execution. This is posible because the dummy programs in ANALYZER's segments are not simply "BEGIN END.", but calling programs for the segment procedures. After the segment is compiled, perform step 18 above. Instead of providing the code file that contains all of the segments to be linked to the host file prompt, give it the name of the segment code file to be executed. When prompted for a lib file, return a "*" or SYSTEM.LIBRARY. (You will be asked for another lib file but give it a (<cr>) to signify there are no more.) Give the same responses as specifed in step 18 for the last two prompts, and your segment is now executable.

APPENDIX C

Hierarchical, Block Diagram of the Modules (Including Statii) of AQERS



Vita

Michael D. Guidry was born on April 11, 1955 at San Antonio, Texas. He graduated from Mc Allen High School at Mc Allen, Texas in May, 1973. He attended the University of Texas at Austin on a four year Air Force ROTC scholarship. Upon graduation in May, 1977, he received a Bachelor of Arts degree in Computer Science and a reserve commission in the Air Force. He went on active duty in July, 1977, as was assigned to Headquarters 21st Air Division, Hancock Field, New York, where he spent four years in the Division of Program Maintenance. In June, 1981, he entered the School of Engineering at the Air Force Institute of Technology.

Permanent Address: 330 Clovis Place
San Antonio, Tx 78221

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFIT/GCS/EE/82D-16	2. GOVT ACCESSION NO.: AD-A124719
3. RECIPIENT'S CATALOG NUMBER:	
4. TITLE (and Subtitle): A MINICOMPUTER IMPLEMENTATION OF A DATA MODEL INDEPENDENT, USER-FRIENDLY INTERFACE TO DATABASES	5. TYPE OF REPORT & PERIOD COVERED:
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Michael D. Guidry, Capt USAF	8. CONTRACT OR GRANT NUMBER(s):
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Air Force Institute of Technology (AFIT/EE), Wright-Patterson AFB, Ohio 45433	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS:
11. CONTROLLING OFFICE NAME AND ADDRESS:	12. REPORT DATE: December 1982
	13. NUMBER OF PAGES: 80
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office):	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES

Approved for public release IAW AFR 190-17.

26 JAN 1983

Lynn Wolaver, Development

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

User-friendly, Query Resolution System, Data Model Independent, UCSD Pascal, Universal Data Manipulation Language

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

An interface system was proposed that would give the user the ability to query a database, regardless of the host DBMS, as long as the user's computer could communicate with the database's host computer.

Investigations were made into existing query resolution systems, and query definition structures and universal data manipulation languages were researched. With this background, a base query resolution system

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

was chosen, and additional capabilities for an improved final query definition structure and a universal data manipulation language were proposed.

In this project, the syntactic analysis part of the query resolution system was implemented. The use of the improved final query definition structure within the query resolution system was defined, and a high level design of the UNIVERSAL_DML (including a pseudo-code representation of the resultant application program) is presented.