VIADUCT: AN INTERACTIVE, VERY-HIGH-LEVEL DATA MANIPULATION LANGUAGE FOR A MICROCOMPUTER-BASED DATABASE SYSTEM

BY PETER THEODORE WOOD

A thesis prepared under the supervision of
Associate Professor S.R. Schach
in fulfilment of the requirements for the
Degree of Master of Science
in Computer Science
at the University of Cape Town.

September 1982.

ABSTRACT

A very-high-level data manipulation language for a database system is one in which the user specifies in non-procedural terms the operations that are to be performed on the data stored in the database; the actual method by which the operations are executed does not concern the user. VIADUCT provides such an interface to a microcomputer-based database system known as MDBS. Thus VIADUCT allows a microcomputer user lacking in computer sophistication to interact with, and derive the benefits of, a powerful database management system.

Additional security restrictions and integrity constraints usually found only on mainframe database management systems are provided by VIADUCT through the mechanism of a subschema generator.

A primary consideration in the microcomputer context is that of ease-of-use. To this end VIADUCT provides the user with on-line help facilities during command formulation as well as interactive prompting if incomplete commands are entered. As far as possible the user is kept unaware of the structure of the database system. This is achieved by means of automatic path finding, selection, and traversal techniques.

ACKNOWLEDGEMENTS

Firstly, I wish to thank Professor Schach for the encouragement and helpful suggestions and advice that he provided throughout the duration of this research. His constant enthusiasm was infectious and his frequent witticisms were always greatly appreciated.

My thanks are extended to Dr M.H. Linck who provided the acronym VIADUCT (a Very-high-level, Interactive Data manipulation language developed at the University of Cape Town).

I am grateful to the C.S.I.R. for their financial assistance during this two year period, and to Microcom (Pty) Ltd for their generosity and helpfulness.

Finally, I would like to express my appreciation for the encouragement, patience, and understanding shown by my mother and by Corinne, to both of whom this thesis is dedicated.

CONTENTS

CHAPTER 1: **INTRODUCTION**

This chapter provides the background to the thesis and the objectives that were to be achieved in the design and development of VIADUCT. The chapter concludes with a brief outline of the structure and content of the thesis.

1.1 **Background**

Most so-called database management systems currently available for microcomputers are little more than data retrieval systems limited in power and storage capacity and providing few, if any, of the traditional features of database systems. This thesis was initiated when a microcomputer database management system known as MDBS, which incorporated a number of features previously found only on mainframe and mini computer systems, was released. The MDBS package certainly had the power and features of larger database systems, but as a consequence it was more complex and difficult to use than most other microcomputer software packages.

Thus it was decided that an easy-to-use interface to the MDBS database system, similar to a traditional query language processor, should be developed to allow the unsophisticated microcomputer user to interact with a powerful database system.

MDBS subsequently released their own query language processor but this did not invalidate the research and development of VIADUCT since the two systems differ in basic design objectives.

The objectives that were set for the design of VIADUCT are discussed in the following section.

1.2 Objectives

The primary objective of this research was to develop an easy-to-use query language processor for a microcomputer based database system. Many query language processors, especially those designed for network model database systems, require the user to specify explicitly the path to be followed through the database schema for executing the command entered. This demands that the user have detailed knowledge of the structure of the database schema being used. In the microcomputer field, where users are often unfamiliar with computer terminology and techniques, the requirement of explicit path specification for every command is totally unacceptable.

For the above reasons VIADUCT was designed to free the user from the need to have any knowledge of the structure of the database; instead VIADUCT must automatically find the relevant paths through the schema. The criterion of ease-of-use required that on-line help facilities be provided. In addition, the user was to be prompted for any necessary information that he had not included in his formulation of the query.

VIADUCT was to provide all the commands necessary for data manipulation so that it could be used for almost all database tasks. This, in turn, would reduce (and perhaps even remove) the need for the writing of specific application programs to

access the database.

During the course of using and evaluating MDBS it became apparent that major database features omitted from MDBS should be included in VIADUCT. Thus secondary objectives in the design of VIADUCT were to improve database security restrictions and enhance the provision of data integrity. The facilities necessary to implement these were incorporated in a pseudo-subschema generator which generates restricted views of the database similar to a conventional database subschemata.

The above objectives are justified and discussed in greater detail in a subsequent chapter on design criteria (Chapter 6). This allows the necessary theoretical topics to be covered first as described below.

1.3 Thesis outline

This work essentially consists of two parts: background material, and a detailed description of VIADUCT itself. After a short introduction to the features of VIADUCT (Chapter 2), three chapters are dedicated to theoretical and background material. Chapter 3 reviews the features of database management systems in general and those of the network model in particular. In Chapter 4 the microcomputer database system MDBS is described and evaluated. Chapter 5 outlines the microcomputer hardware and software environment in which VIADUCT was developed.

The description of VIADUCT itself which then follows is divided

into six chapters. Chapter 6 justifies and discusses the criteria considered in the design of VIADUCT. Chapters 7 to 10 describe the features of VIADUCT and details of their implementation: the pseudo-subschema construct utilized by VIADUCT (Chapter 7), the command line parser (Chapter 8), the path finding and traversal algorithms employed by VIADUCT (Chapter 9), and the command processor (Chapter 10).

Chapter 11 summarizes the features of VIADUCT by describing a database schema currently in use. Conclusions are drawn and directions for further work are outlined in Chapter 12.

CHAPTER 2: **SYSTEM OVERVIEW**

In order to familiarize the reader with some of the basic features of VIADUCT, a simple database system is introduced, followed by a sample terminal session using the package. The emphasis in this chapter is on the useability and external appearance of the system, rather than the underlying concepts involved. This is done at the risk of using terms which are defined or explained only in subsequent chapters; however, it is hoped that this will not detract from the spirit of this chapter.

2.1 **A simple database**

A diagram of the database used as an example in this chapter is given in Figure 2.1, and represents a simple education database. This is a slightly modified version of the database used in the MDBS Primer [Hols80]. It consists of information concerning and relating TEACHERs, STUDENTs, SUBJECTs, COURSEs, SECTIONs, and ACTIVITies. Each of these consists of a number of pieces of information; for example, each TEACHER has a NAME and AGE, while a SUBJECT has both a NUMBER and a TITLE associated with it.

The relationships between these entities can be described in a simplistic manner as follows. The groups of TEACHERs, STUDENTs, SUBJECTs, SECTIONs, and ACTIVITies are directly accessible to the user, this being represented by the fact that they are connected to the entity named SYSTEM. Each SUBJECT consists of a number of COURSEs which, in turn, comprise a number

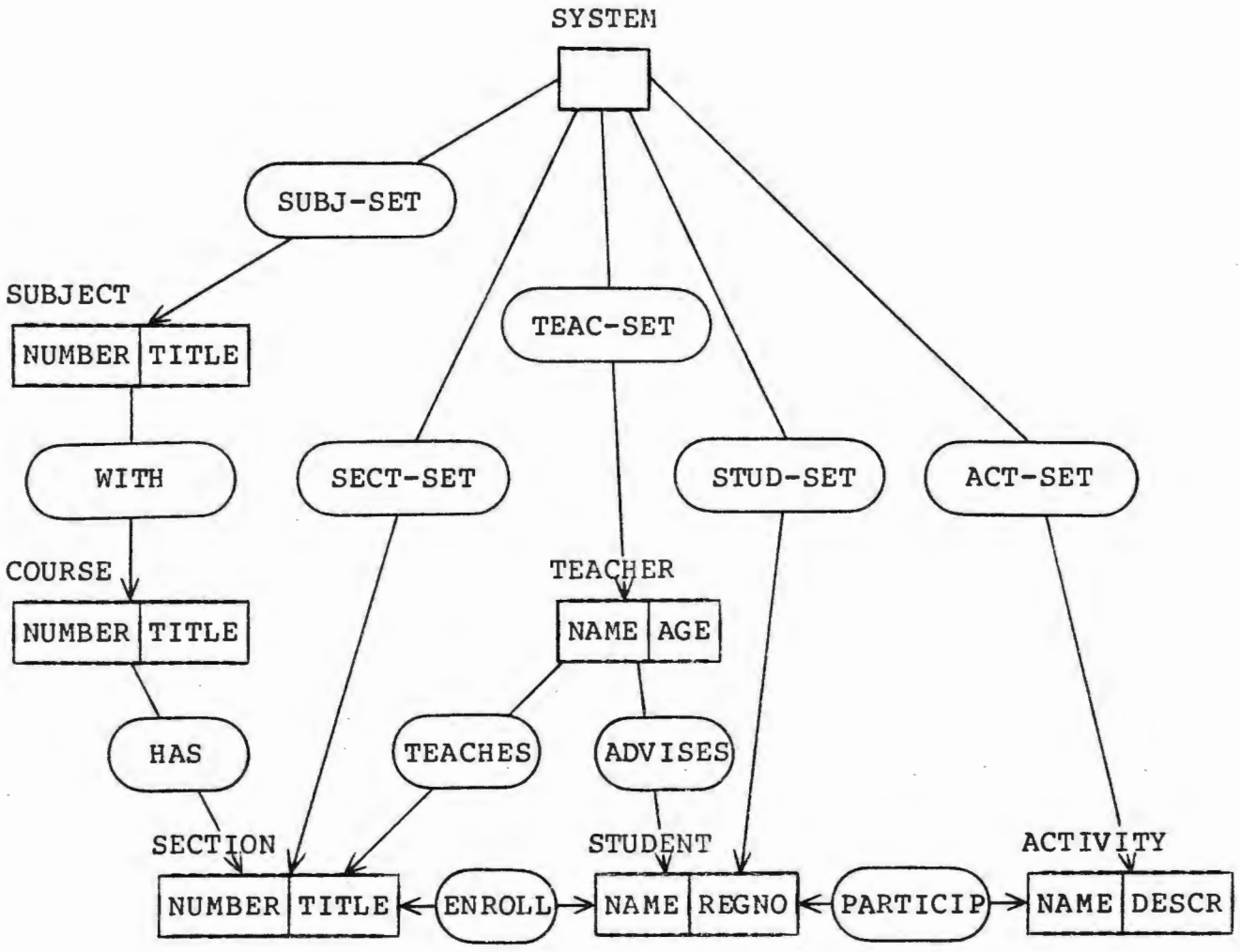


Figure 2.1. Diagram of a simple database.

of SECTIONs. A TEACHER teaches possibly many SECTIONs and also advises a group of STUDENTs. The STUDENTs enroll in SECTIONs, each of which, conversely, has many STUDENTs on its enrollment list. Each STUDENT also participates in a number of ACTIVITies, where each ACTIVITY has many STUDENTs who participate in it.

The last two relationships, those between SECTIONs and STUDENTs, and STUDENTs and ACTIVITies, can be thought of as many-to-many (as represented in Figure 2.1 by the double arrows), while all the others may be termed one-to-many.

2.2 **Sample terminal session**

For this example, it is assumed that the database depicted in Figure 2.1 has already been loaded with a certain amount of information. VIADUCT enables the user to perform a number of operations on this information including: obtain a list of the NAMEs and AGEs of all TEACHERs; create a new SECTION; update the TITLE of a particular COURSE; remove a STUDENT from an ACTIVITY in which he currently participates; delete a TEACHER who has resigned; or add a STUDENT to a SECTION for which he has enrolled. For the purposes of this overview two more complex examples will be discussed, allowing some of the more sophisticated features of VIADUCT to be highlighted.

The first response required from a VIADUCT user is his user identification (userid) followed by his password, both of which would have been allocated to him by the database designer. On acceptance of these entries, the screen used in the so-called

expert entry mode is displayed (Figure 2.2). Assuming that the user is unacquainted with the system, he may enter a question mark ('?'), which will result in the help mode screen being displayed (Figure 2.3).

As indicated by the help screen menu, an entry of 'Y' or 'y' results in an option being chosen, while any other entry causes the system to skip to the next option. At any time, the user may enter an exclamation mark ('!') to return to the expert mode of entry at the current position in the request. Alternatively, additional help may be solicited by the entry of a question mark in response to an option. These alternatives are covered in detail in Chapter 8.

Assuming that the user is in help mode (Figure 2.3), and that he wishes to create a record for a new STUDENT who is to be advised by a TEACHER whose name is 'Brown', the following is a possible entry sequence. The user types 'N' as the cursor is next to the LIST function. In response the cursor moves down to the UPDATE function. Another 'N' response causes the cursor to move down to CREATE. The user now selects the CREATE function by entering 'Y' at this point. This results in the display of all the records which the user is allowed to create. In the same way as for the functions, the cursor moves from SUBJECT towards ACTIVITY in response to 'N' inputs, until the user selects the STUDENT record by typing 'Y' when the cursor is next to STUDENT. At this stage the screen appears as in Figure 2.4.

During the formulation of a command in help mode, the corresponding expert mode input is displayed at the top of the

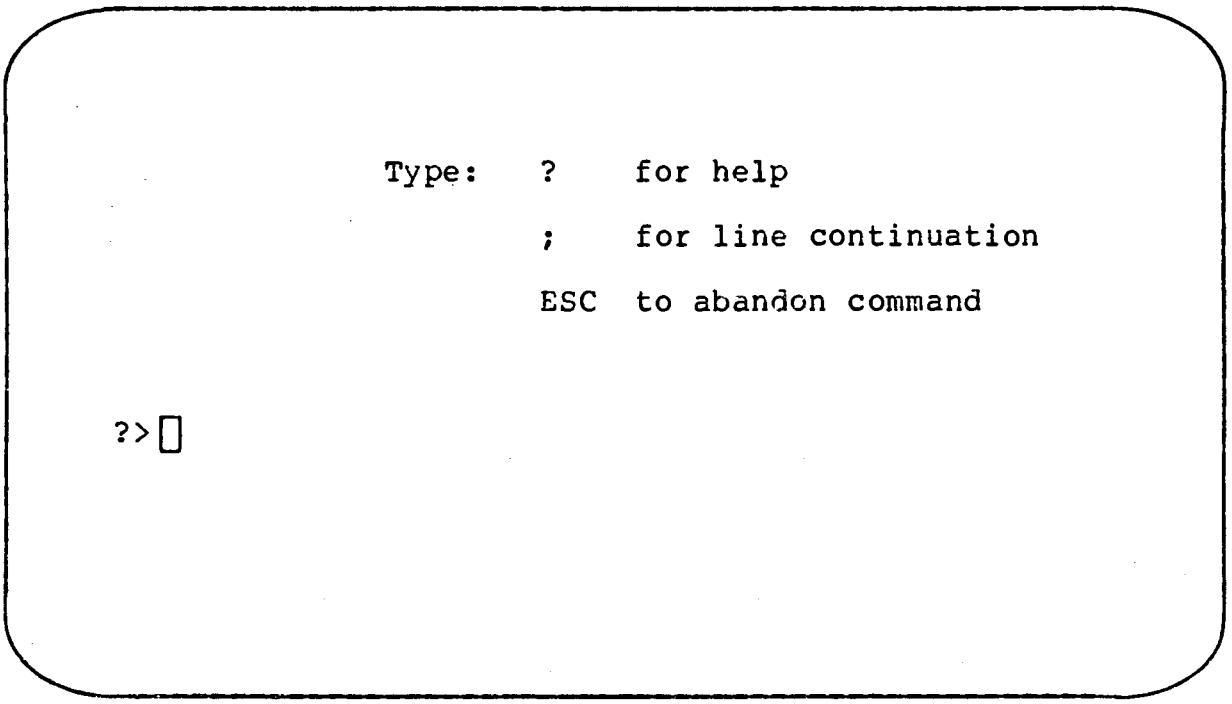


Figure 2.2. Initial screen for expert mode.

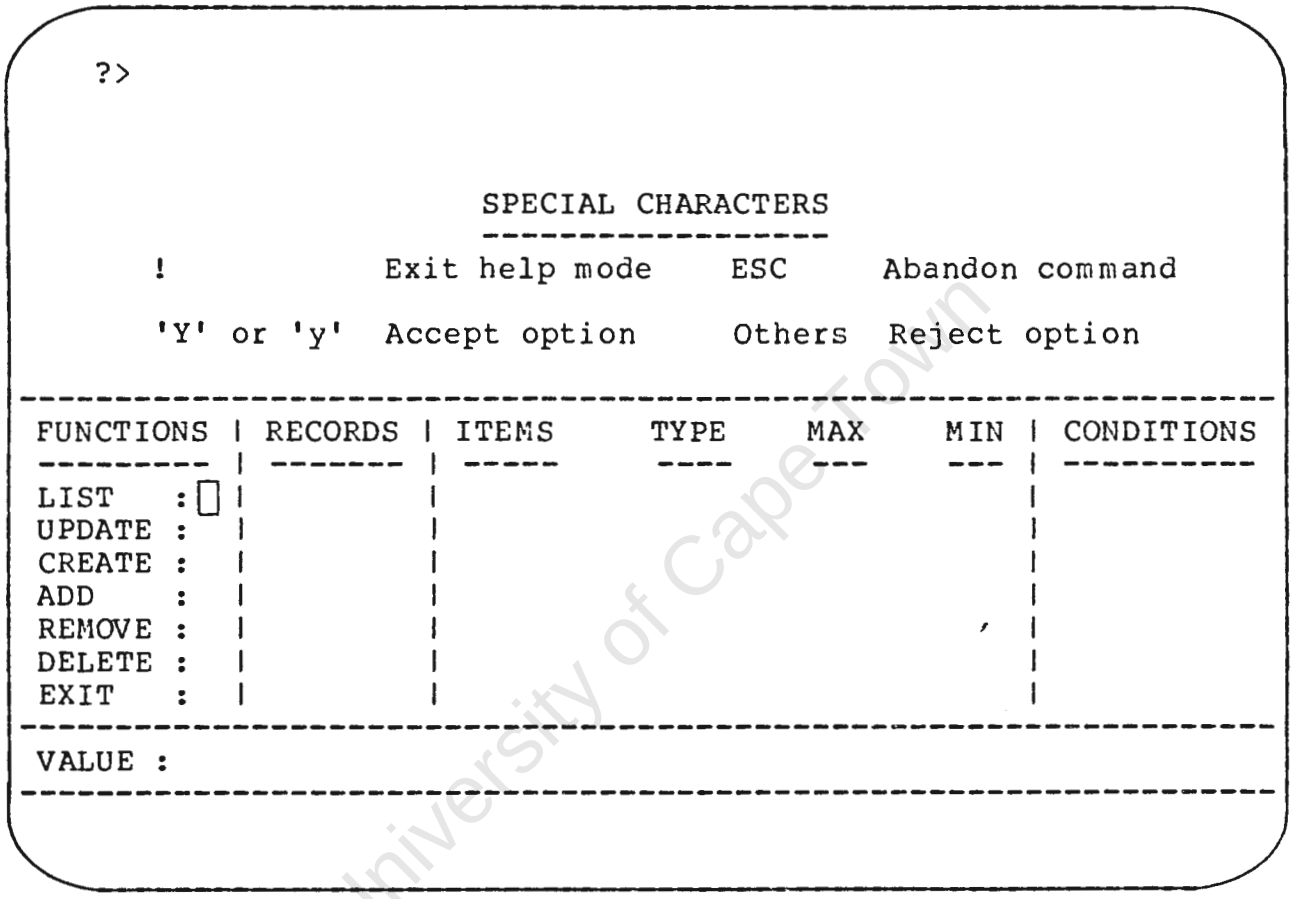


Figure 2.3. Help mode screen layout.

```
?>CREATE STUDENT

                    SPECIAL CHARACTERS
                    ------------------
    !           Exit help mode    ESC     Abandon command

  'Y' or 'y'    Accept option     Others  Reject option

-------------------------------------------------------------------------
FUNCTIONS | RECORDS    | ITEMS     TYPE     MAX     MIN | CONDITIONS
--------- | -------    | -----     ----     ---     --- | ----------
LIST  :N  | SUBJECT :N|                                  |
UPDATE:N  | COURSE  :N|                                  |
CREATE:Y  | SECTION :N|                                  |
ADD   :   | TEACHER :N|                                  |
REMOVE:   | STUDENT :Y|                                  |
DELETE:   | ACTIVITY: |                                  |
EXIT  :   |           |                                  |
-------------------------------------------------------------------------
CONDITIONS ?:
-------------------------------------------------------------------------
```

Figure 2.4. Creating a student record (help mode). (User responses in boldface)

screen, thus acting as a reminder of what has been entered, as well as teaching the expert mode of entry to the inexperienced user. Thus CREATE STUDENT appears at the top of the screen in response to the user's replies so far (Figure 2.4).

In order to enter the fact that the STUDENT is to be advised by a particular TEACHER, the user types 'Y' in response to the CONDITIONS prompt (at foot of Figure 2.4), which results in the record names being redisplayed (Figure 2.5). On selecting the TEACHER record, the various items comprising that record are shown. Since the item NAME is to be tested, and must be equal to the value 'Brown', the user responds as shown in Figure 2.5. No more conditions are required so an 'N' is entered next to the remaining TEACHER items and all other record names. This completes the process of command entry in help mode.

VIADUCT now enters its execution phase of operation, which results in the dialogue shown in Figure 2.6. The user is first prompted for values for each of the items which make up a STUDENT record. The record is then created and automatically added (or connected) to the TEACHER record with the value 'Brown' in its NAME item, since this was specified by the user in his request. The user is then asked whether this STUDENT record is to be added to any SECTIONs or ACTIVITies. Since a STUDENT can participate in many ACTIVITies, this prompt is repeated until an 'N' is entered.

As a second example, the user may wish to list all the TEACHERs who are younger than 30 and who teach the SUBJECT science, along with the COURSEs and SECTIONs that they teach. Assuming that

```
?>CREATE STUDENT WHERE TEACHER.NAME EQ BROWN


                    SPECIAL CHARACTERS
                    ------------------
    !           Exit help mode     ESC     Abandon command

    'Y' or 'y'  Accept option      Others  Reject option

-------------------------------------------------------------------------
FUNCTIONS | RECORDS    | ITEMS      TYPE     MAX     MIN | CONDITIONS
--------- | -------    | -----      ----     ---     --- | ----------
LIST  :N  | SUBJECT  :N| NAME     :Y                     |  EQ :Y
UPDATE:N  | COURSE   :N| AGE      :N                     |  LT :
CREATE:Y  | SECTION  :N|                                 |  GT :
ADD   :   | TEACHER  :Y|                                 |  NE :
REMOVE:   | STUDENT  :□|                                 |
DELETE:   | ACTIVITY:  |                                 |
EXIT  :   |            |                                 |
-------------------------------------------------------------------------
VALUE ?: Brown<CR>
-------------------------------------------------------------------------
```

Figure 2.5. Creating a student record (help mode). (User responses in boldface)

```
?>CREATE STUDENT WHERE TEACHER.NAME EQ BROWN
------------------------------------------------------------
Enter STUDENT-NAME: Bloggs<CR>
Enter STUDENT-REGNO: 96143<CR>
---  Record created
---  Record added to TEACHER-NAME Brown

Add to record SECTION, set ENROLL ?: N

Add to record ACTIVITY, set PARTICIP ?: Y
Enter ACTIVITY-NAME: Squash<CR>
---  Record added to ACTIVITY-NAME Squash
Add to record ACTIVITY, set PARTICIP ?: N

                Type <RETURN> to continue:
```

Figure 2.6. Execution phase of creating a student record. (User responses in boldface)

the user has experience with the system and is in expert mode (Figure 2.2), he might enter:

LIST TEACHER.NAME,TEACHER.AGE, COURSE.NUMBER, COURSE.TITLE ;
SECTION.NUMBER, SECTION.TITLE ;
WHERE SUBJECT.TITLE EQ SCIENCE, TEACHER.AGE LT 30 <CR>

(<CR> denotes the entry of a carriage return.)

If at any stage the user had entered a term not recognized by VIADUCT, it would immediately have informed the user of the fact by means of an appropriate error message, and thereafter displayed the request up to the point at which the error was detected and awaited further input. The expert mode user is given three attempts to correct the syntax of his request, after which help mode is automatically entered. Since the syntax of the above query is correct, the system executes the request after soliciting additional information from the user (Figure 2.7).

Firstly, the system informs the user that more than one path through the database connects the records specified in the query. If the user prefers not to choose a particular path, the system automatically chooses the shortest one. In this example, the first path corresponds to the desired request, namely, for TEACHERs who teach SECTIONs, while the second represents the TEACHERs who advise STUDENTs who are enrolled in the SECTIONs.

Ascending or descending order can be specified for the values to be listed. Following the user's responses to these prompts, the values in the database satisfying the entered query are displayed. Where possible, duplicate values are excluded from the listing, thus enhancing readability and emphasizing control

```
?> LIST  TEACHER.NAME, TEACHER.AGE, COURSE.NUMBER, COURSE.TITLE
         SECTION.NUMBER, SECTION.TITLE
   WHERE SUBJECT.TITLE EQ SCIENCE, TEACHER.AGE LT 30
-------------------------------------------------------------------

Alternate paths exist. Do you wish to choose one (Y/N) ?: Y
                        2 Paths exist
1. SUBJECT WITH COURSE HAS SECTION TEACHES TEACHER
2. SUBJECT WITH COURSE HAS SECTION ENROLL STUDENT ADVISES TEACHER
Enter path number : 1

Send listing to console, printer, or disk [C/P/D] ?: C

Ascending or descending COURSE-NUMBER [A/D] ?: A
Ascending or descending SECTION-NUMBER [A/D] ?: D
Ascending or descending TEACHER-NAME [A/D] ?: A

TEACHER        TEACHER  COURSE  COURSE       SECTION  SECTION
NAME           AGE      NUMBER  TITLE        NUMBER   TITLE
Brown          26       202     Chemistry    2022     Organic
                                             2021     Inorganic
Jones          29       205     Physics      2051     Waves

Number of occurrences is  3

                Type <RETURN> to continue :
```

Figure 2.7. Execution phase of a list command.
(User responses in boldface)

breaks.

The purpose of this chapter has been to give the reader an overview of the VIADUCT system. Two examples of reasonable complexity, operating on a simple, but not unrealistic, database design were used to achieve this. Further examples are discussed in Chapters 8, 10, and 11.

Reference

[Hols80] Holsapple C. **A Primer on Data Base Management Systems.** Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

CHAPTER 3: **DATABASE CONCEPTS**

Chapter 2 presented an informal introduction to a specific application of a certain database management system. This chapter outlines the concepts of database systems in general, and the network method of organization in particular. Query language processors are reviewed, as is the level of sophistication of current microcomputer database systems. No claim is made as to the completeness of this overview; rather, features relevant to subsequent chapters are emphasized. The reader desiring more detailed information is referred to the many texts available on the subject [Date81, Mart77, Ullm80].

3.1 **General**

In recent years, the term 'database' has too frequently been inappropriately applied to various record-keeping systems. This critisism is particularly applicable in the microcomputer field, where the naivete of the user can be readily exploited.

There does not, in fact, appear to be any minimum set of requirements which a file system must fulfil in order to be termed a database system. Date [Date81] and Ullman [Ullm80] define a database as simply the operational data of an enterprise, and the database management system as being the software permitting applications to access that data.

However, both Date and Ullman emphasize that a database should be an integrated system consisting of interrelated data, which can be concurrently shared by a number of users. Features which are stressed by these authors are those of integrity, security, and data independence.

Operational data is an extremely valuable asset of any enterprise, and the presence of inaccurate or inconsistent data will seriously affect the level of integrity of that data. The amount of inaccurate data can be reduced by placing constraints on the values which data items can assume, thereby at least excluding the existence of absurd entries. Update anomalies [Ullm80] can arise through the existence of redundant data, but these anomalies can generally be avoided through reduction of data to normal forms [Ling81, Tsou80]. The concurrent usage of a database system can also produce anomalies if adequate locking protocols are not employed [Lehm81]. Support routines, necessary for the protection from system crashes and subsequent recovery, must be provided to ensure that the data retains its integrity.

As a result of the integration and sharing of data, security restrictions must be applied to the database system. These must prevent unauthorised access, both wilful and unintentional, to the database as a whole as well as to areas of the database not available to the current user. These restrictions are applied through user identification, the extensive use of passwords, and the definition of external views of the database. Data encryption can also be used, particularly when it is necessary to take precautions against the possibility of wire-tapping.

The prevention of inferences being made as a result of unauthorized extraction of information from statistical databases, has been the subject of much recent research [Denn79].

A primary objective of a database management system is to provide data independence [Date81]. If the database is to be shared by many applications of varying nature, it is essential that the data be stored in a manner which is independent of those applications. Another advantage of data independence is that, to a certain extent, it allows applications to remain unchanged when the physical, and even logical, structure of the underlying database system is altered. This may be necessitated by changing requirements, or by the natural growth of the system. One method of achieving this is to keep the structural information of the database in what is known as a data dictionary, which is often stored in the database itself.

As always, there is a price to pay for these features, and a compromise often has to be reached. It is important to evaluate the trade-off between the level of security enforcement, the amount of data integrity checking, the extent of data independence, and the resulting performance degradation.

The ANSI/SPARC study group on Data Base Management Systems [ANSI75] defined three levels of abstraction in a database system: the internal, conceptual, and external levels. The internal level is used to describe the physical storage layout of the database system; the conceptual level represents the logical description of the database, the view conceptualised by the users; and the external level provides for restricted logical

views of the data required by differing applications.

There will usually be one person, known as the database administrator (DBA), who is responsible for the design and maintenance of these various levels and views of the database system. It will also be his task to ensure that security restrictions are enforced, that the integrity and performance of the database are monitored, and that any necessary changes are implemented.

There are three commonly accepted types of representation or data models: the hierarchical, network and relational systems. The hierarchical model can be viewed as a tree structure, the network model as a graph structure, and the relational model as a collection of tables, or relations [Codd70]. Each has its own merits and drawbacks; however, attention will be focused on the network model, since VIADUCT was written for an implementation using this approach.

3.2 The network model

The network database model is an implementation of the recommendations of the CODASYL Data Base Task Group (DBTG), as set out in its initial report published in April 1971 [CODA71] and subsequently amended. The latest version is dated January 1981 [CODA81]. The original report proposed the use of three languages in the description and manipulation of database systems: the schema data description language (schema DDL), the subschema data description language (subschema DDL), and the data

manipulation language (DML). The schema DDL specifies the conceptual level, or logical appearance, of a network database system. The subschema DDL, which corresponds to the external level of the database system, describes a restricted view of the schema for applications programs and provides an interface to the database for the various host language processors. The DML is embedded in a host programming language (such as COBOL) and, through the invocation of a particular subschema, allows manipulation of the database system.

3.2.1 **Database structure**

The logical structure of a network database system is defined using three basic constructs: record types, data items, and set types. As in a straightforward file structure, each record type embodies a collection of entities known as data items. The data comprising the database consist of instances or occurrences of these record types. Relationships between the record types are represented by set types which impose a hierarchical structure on the data whereby one record type is an owner of a set type and another is a member. In CODASYL terms, such a set type signifies a one-to-many association between the record types, that is, a set occurrence consists of one occurrence of the owner record type and many (or possibly zero) member record occurrences.

Referring to the data structure diagram of the schema introduced in Section 2.1 (Figure 2.1), TEACHER, SUBJECT, and STUDENT are examples of record types, while NUMBER and TITLE are the data

items comprising the record type COURSE. The relationship between the record types COURSE and SECTION is represented by the set HAS, which specifies that each COURSE has (possibly) many SECTIONs associated with it.

In extended network models, that is, supersets of the CODASYL recommendations, relationships other than one-to-many may be allowed. Many-to-one associations are the inverse of one-to-many relationships, and so represent the situation where many owner record occurrences are associated with a single member record occurrence. Combining one-to-many and many-to-one relationships results in a many-to-many relationship, as illustrated in the schema of Section 2.1 by the sets PARTICIPATE and ENROLL. For example, each STUDENT may ENROLL in many SECTIONs, and each SECTION has an ENROLLment of many STUDENTs. One-to-one relationships are also useful in certain circumstances, although many implementations require this restriction to be enforced by the application program itself.

Many-to-many relationships are often obscured by the existence of some connection record type, or intersection data. Indeed, in systems adhering to the CODASYL recommendations, this is the only way in which many-to-many relationships can be represented, even if this requires the declaration of a dummy record type. Returning to the sample schema, a many-to-many relationship effectively exists between TEACHERs and COURSEs, but the record type SECTION transforms this into two one-to-many relationships.

Multiple associations may exist between two record types, this being represented by many set types connecting the corresponding

record types (Figure 3.1(a)). A single record type can be the owner of several set types (the hierarchical structure of Figure 3.1(b)) or, conversely, can be a member of many set types. In the latter case, the member record type is known as a connection record type and represents what is sometimes termed a confluent hierarchy (Figure 3.1c). Furthermore, a particular set type may have multiple member record types (Figure 3.1(d)) or multiple owner record types (Figure 3.1(e)). All the sets in these figures may be of any of the types previously mentioned, and can of course be combined to form a complex network structure. Some implementations of the network model allow the definition of recursive set types, where one record type is both owner and member of the same set type (Figure 3.1(f)). These sets can be used when it is necessary to represent a hierarchy within the occurrences of a particular record type, such as within the employees of a company.

There has been some controversy as to whether relationships other than one-to-many should be allowed in network models. Bachman [Bach77] feels that there is inadequate justification for the CODASYL restrictions, while Ullman states that the problem of many-to-many sets 'is not conveniently solved' [Ullm80, p. 87].

A particular type of set, known as a singular set, is provided for entry into the database structure. Singular sets each have only one set occurrence and all have the same owner record type (designated 'SYSTEM') of which there is only one occurrence. The example of Section 2.1 contains five singular sets: SUBJ-SET, SECT-SET, TEAC-SET, STUD-SET, and ACT-SET.

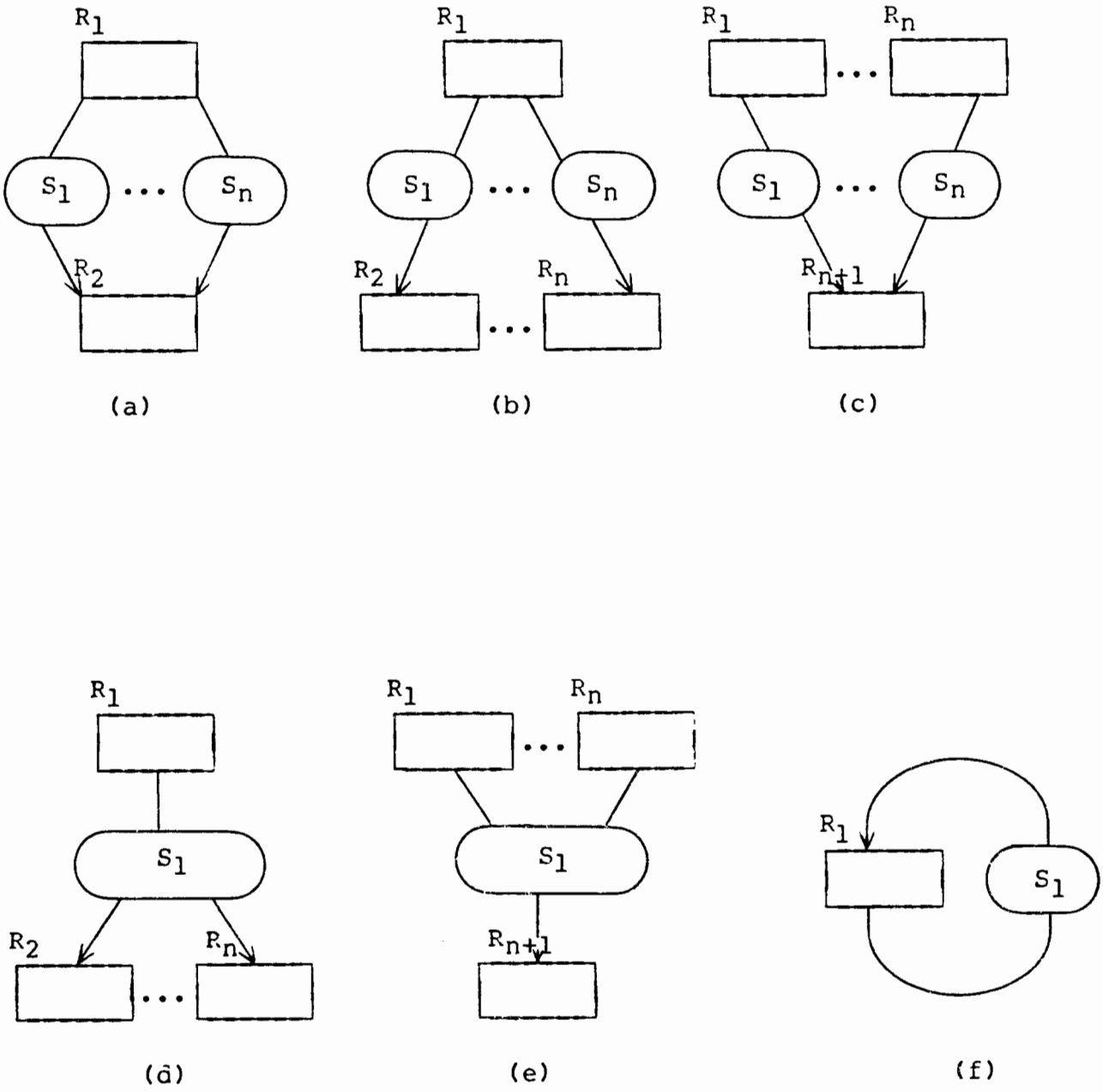


Figure 3.1. Bachman diagrams [Bach69] representing network model set structures.

The data description language (DDL) is used to describe all these record types and set types, as well as the relationships between them, by specifying the owner and member record types for each set type and the nature of the association (e.g. one-to-many). All these specifications make up the schema, which is also used to allocate sections of the database to certain physical areas or realms on the storage medium. In DMS 1100 [Sper78], for example, these areas correspond to mass storage files.

The DBA is responsible for defining a subschema for each different user or application. These subschemata allow certain record types, set types, data items, and areas to be excluded from the particular application's view of the database. Different access strategies may also be specified as may additional security restrictions and integrity constraints (see Sections 3.2.3 and 3.2.4).

3.2.2 Database manipulation

Databases may be manipulated by means of four basic operations: (a) the creation of record occurrences, (b) the deletion of record occurrences, (c) the updating of values of data items within record occurrences, and (d) the ability to find and retrieve record occurrences based on some selection criterion. For network database systems, it is also desirable to be able to update the associations between record occurrences. This is achieved by allowing record occurrences to be inserted into (connected to) and removed from (disconnected from) set occurrences.

Before discussing these operations, it is necessary to describe the method by which navigation through a network database is performed. Each record occurrence has a unique identifier assigned to it by the system at the time of its creation. This is known as its database key. Apart from allowing the system to identify a record occurrence uniquely, these database keys are used as 'currency indicators', to keep track of which record occurrences are being referenced at various levels in the database. A currency indicator usually exists for each record type, each set type, and each realm in the database; in addition, there is a currency indicator for the most recently accessed record occurrence in the entire database, known as the 'current of run-unit'. The currency indicators are updated automatically by the database system as record occurrences are found, created, or deleted. In this way, positional markers are kept for the user.

When a record occurrence is created, it must be linked to some set occurrence to establish its relationship to other record occurrences in the database. This can be done either automatically by the database management system, or manually by the user who must then issue an insert or connect command to the system. An entry in the schema for each set type specifies whether this 'insertion membership class' is to be AUTOMATIC or MANUAL.

It is also possible to specify a set retention clause in the schema. This indicates whether an occurrence of a record type, which is a member of the particular set type, may be removed or disconnected from a set occurrence. If the retention class is

FIXED, then no record occurrence can ever be removed from its corresponding set occurrence, while if it is MANDATORY, none may be removed from the set as a whole. A retention class of OPTIONAL specifies that record occurrences may be removed from the entire set type.

If a record occurrence is to be located in the database, the schema must provide a means for specifying how this is to achieved. This is done by declaring a LOCATION MODE for each record type in the database. In CODASYL, the four possibilities are: DIRECT, CALC, INDEX SEQUENTIAL, and VIA SET. To locate a record occurrence using DIRECT mode, the database key corresponding to that occurrence must be supplied to the system by the program. If the LOCATION MODE is CALC, then a DBA-supplied procedure is used as a hashing function on certain items within the record type, to determine the record's location. When INDEX SEQUENTIAL mode is specified, the value of the key must be supplied to the system. If VIA SET is to be used, then it depends on what ordering has been specified for the members of the named set in the schema (ORDER clause).

There are a number of different set orderings which can be specified: SORTED, LAST, FIRST, PRIOR, or NEXT. If the occurrences are to be sorted, then a key data item, or combination of items, must be specified in the schema. When finding record occurrences by matching data item values or groups of values, it is important to know whether occurrences with duplicate values can be expected or whether they have been forbidden by an appropriate entry in the schema.

When a record occurrence is stored in the database, it is important that all the set occurrences in which it participates as an automatic member have been selected, so that the record can be connected to its owner record occurrences. It is also necessary to isolate the correct set occurrence when finding a record occurrence whose location mode is via set. The SET OCCURRENCE SELECTION clause in the schema is used for this purpose.

In CODASYL terms, a set occurrence is uniquely identifiable by its owner record occurrence, and so the method of selection is defined as either CURRENT OF SET or LOCATION MODE OF OWNER. In the first case, this means that the application program is responsible for ensuring that the owner record occurrence is current-of-set, and in the second, that the correct set occurrence can be found by using the location mode of the owner record type. If this is DIRECT, CALC with no duplicates, or INDEX SEQUENTIAL with no duplicates, then the owner record occurrence can be accessed directly. It is possible, of course, that the LOCATION MODE of the owner record type is via some other set, in which case the SET OCCURRENCE SELECTION clause for that set must be examined, and so on. In this way, a path through the database is established, which terminates only when an owner record can be accessed directly, or the system entry point is reached.

It is important that the application program assigns values to all items used as keys in a path prior to the execution of a command which results in the automatic traversal of the path by the system. After the correct set occurrence has been selected,

the required record occurrence is located according to the order clause specified in the schema.

The deletion of record occurrences must be approached with great caution. Since a hierarchy of data exists in the database, it is possible that the deletion of a record occurrence can result in the effective deletion of all its member record occurrences. In CODASYL, the specification of a DELETE ALL clause in the schema for a particular record type has this effect, while the DELETE ONLY clause permits deletion of a record occurrence only if it has no member occurrences.

3.2.3 Security restrictions

In order to prevent unauthorized access to a database system, users are usually required to identify themselves to the system by some means. A further degree of security is introduced through issuing passwords to users. Before the user can perform any operations on the database itself, he must invoke a subschema to which he has been assigned access. This provides security for the data not declared in the invoked subschema.

Most of the other security restrictions in the network model rely on the concepts of access control locks and access control keys. These can be applied to the various functions which operate on the schemas, subschemas, areas, records, sets, and even data items. An access control lock is declared in the schema and/or subschema, and may be defined as a literal (similar to a password), a data item (in which case it is known as a lock

item), or a procedure (possibly to interrogate some other part of the database to see if the request should be granted). Access control keys are declared in the application program, and are used to 'open' the locks.

For additional security when storing or transmitting data item values, encryption procedures can be specified in the schema. This is achieved by declaring DBA-supplied ENCODE and DECODE procedures for any data item type in the schema which is to be stored in an encrypted form.

3.2.4 **Integrity constraints**

There are many entries in the schema and subschema definitions which are used to enforce integrity in the database. The very existence of subschemata ensures that the integrity of certain data cannot be affected by particular users. Integrity is also achieved through the application of a number of security restrictions, some of which were discussed in the previous section.

The access control locks described above can also be used to check the integrity of input data, in so far as it can be checked. However, there exist other schema entries included expressly for the purpose of imposing integrity constraints on the data. CALL PROCEDUREs can be specified for the various functions, in the same way as for access control locks. These procedures are then called upon to check the integrity of the data when the particular function is used. In addition, data

items can be restricted to certain values, or to lie within certain bounds by means of a CHECK clause.

As mentioned in Section 3.2.2, duplicate data item values can be forbidden, thereby avoiding the entry of multiple (and possibly inconsistent) copies of the same data. The set membership class of record types allows the DBA to impose some control over the way in which record occurrences are allowed to relate to one another.

Sometimes it is desirable that a member record type contain a data item that is also defined in its owner record type. This might be to allow a certain ordering to be placed on the member record occurrences, or to avoid having to retrieve an owner record occurrence to determine the value of the data item. This would require the introduction of redundancy, and possible inconsistency, into the database. Provision for avoiding this is made through the declaration of VIRTUAL data items in the schema. These are simply pointers to the data items from which they derive their values, so inconsistencies cannot arise. This is known as 'controlled redundancy'.

The problem of data sharing and concurrent updates is addressed by the DML. Before any area of the database can be accessed it must be opened in some USAGE-MODE (either retrieval or update) with possible protection schemes or exclusive usage to disallow other users access to that area. On modifying a record occurrence, the user is informed if another user has updated the occurrence since it was first selected. This situation can be dealt with through the use of the KEEP, FREE, and REMONITOR

statements as defined in the original CODASYL report [CODA71]. This solution is far from satisfactory as exemplified by the fact that the 1981 CODASYL report [CODA81] specifies a record and realm locking protocol system in place of the original recommendations.

The CODASYL reports do not specify the type of support routines necessary for audit trails, or recovery procedures, and so DBMS manufacturers have provided their own. These include LOGging transactions, the specification of the frequency of recovery points on the audit trail tape, and other data saving techniques using LOOK clauses. Recovery can be performed by means of ROLLBACKs, either complete or selective.

3.3 Query language processors

The data manipulation language (DML) described in the previous section requires embedding in some host programming language, which means that the user has to be familiar not only with the structure of the database system itself, but also the particular programming language being used. In order to provide for the needs of non-programmers as well as for ad hoc queries, self-contained query language processors (QLPs) were developed. These are often more appropriately named higher-order data sublanguages as they provide the user with a set of higher-level operators that are not usually restricted to query facilities, but include commands allowing database modification as well. As a result of this further level of abstraction, the QLP is usually less procedural in nature than the corresponding DML.

The recognition that human efficiency is a primary element in the efficiency of computer systems has led to a number of studies and reports on the human factors involved in the design and use of QLPs [Reis80, Schn80, Welt81]. Major concerns of these surveys have been whether the languages under review are easy to learn, use, and remember. These are important considerations, particularly as the language may often be used by inexperienced users and, even then, possibly only intermittently. Controlled experiments have been carried out to evaluate the performance of programmers and non-programmers when learning and using various query languages [Reis81].

Factors which influence the ease-of-use of query languages include their syntactic form, their level of procedurality, and the underlying data model. Most query languages adopt a linear syntax (left-to-right, top-to-bottom), using keywords to identify constructs; examples are SQL, formerly SEQUEL [Cham74, Cham80], and MDBS.QRS [MDBS81]. Some QLPs, such as Query-By-Example (QBE) [Zloo75] use a two-dimensional syntax in which position is significant, while others, such as QUEASY [Chri78] and HI-IQ [Gerr75], employ a method of user prompting for query specification.

Procedurality is not a clearly defined concept; it can be thought of as the amount of information that has to be supplied to a system for informing it <u>how</u> to arrive at a solution. The data model for which a query language is designed seems to influence its level of procedurality: languages based on the relational model tend to be less procedural in nature than those based on the hierarchical and network models. This can be attributed to

the set-oriented approach of the relational model as opposed to the record-oriented approach of the others. It is generally accepted that non-procedural query languages are easier to use than procedural ones, although it is thought that more complex queries may be handled better in a procedural language [Welt81]. It is still the subject of some dispute whether a query language having a keyword or a position oriented syntax is easier to learn and use.

Researchers in the field of artificial intelligence have taken the ease-of-use criterion of query languages a step further and have produced QLPs which accept natural language input. Examples of these include ROBOT [Harr78] and EQS [Mart78], and the differences between these and the more usual query languages are outlined in [Jero78].

Query languages usually provide a limited set of commands for retrieving and modifying information stored in the database; however, SQL, for example, also permits the alteration of the schema (or conceptual level) of the database. Most QLPs have been developed as stand-alone processors, although there exist derivatives of both SQL and QBE that can be imbedded in host languages, thereby lending uniformity to the database system.

Two network model query language processors, QLP 1100 [Sper79] and MDBS.QRS [MDBS81] may be compared from the viewpoint of procedurality. Both QLP 1100 and MDBS.QRS have a keyword oriented syntax and they accept pseudo English input. However to the user they appear to differ with respect to their levels of procedurality. QLP 1100 is treated as a language processor and,

as such, has its own subschema definition in which the access paths to be used by the QLP when traversing the database to solve any query must be specified. This allows the QLP to appear less procedural to the user; however the procedurality has simply been shifted into the subschema. MDBS.QRS, on the other hand, requires that the user explicitly specify with each command the path to be used to answer that query. This restriction is discussed in the next chapter where the MDBS package is described in some detail.

3.4 Databases on microcomputers

Sophisticated database management systems have been available for many years on mainframe and mini computers but it is only recently, with the increasing cost effectiveness of micro-computers and their subsequent proliferation in small enterprises, that the need for comparable packages for these machines has arisen.

Unfortunately, many packages currently available for 8-bit micro-computers are advertised as database management systems, but are no more than tools for the manipulation of data banks. They do not provide most of the desirable features of database systems as outlined in Section 3.1. Current prices of these so-called database management systems for single user microcomputer systems range from about $25 to $1500.

The vastly differing levels of sophistication among the database management systems implied by the above price range have prompted

a number of surveys [Barl81, Ever81]. The conclusion that can be drawn from these is that database systems on microcomputers are still in their infancy, and most should rather be termed data handling programs or key-to-disk systems.

References

[ANSI75] ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report FDT, **ACM SIGMOD bulletin**, Vol 7, No 2, 1975.

[Bach69] Bachman, C.W. Data Structure Diagrams. **Data Base** (journal of **ACM SIGBDP**), Vol 1, No 2 (Summer 1969).

[Bach77] Bachman, C.W. Why restrict the modelling capability of CODASYL data structure sets, **AFIPS Conference Proceedings**, Vol 46, 1977, pp 69-75.

[Barl81] Barly K.S. and Driscoll J.R. A Survey of Data-Base Management Systems for Microcomputers, **Byte**, Vol 6, No 11 (November 1981), pp 208-234.

[Cham74] Chamberlin, D.D. and Boyce, R.F. SEQUEL: A Structured English Query Language, IBM Technical Report RJ1394, 1974.

[Cham80] Chamberlin, D.D. A survey of user experience with the SQL data sublanguage, IBM Research Report R92769(35322), San Jose, California, May 1980.

[Chri78] Christensen, M.A. QUEASY: The design and implementation of a Managerial Information System for Casual Users, **Proceedings ACM Annual Conference**, 1978, pp 230-233.

[CODA71] Data Base Task Group of CODASYL Programming Language Committee, Report (April 1971).

[CODA81] CODASYL Committee. CODASYL Data Description Journal of Development, Ottawa, Canada, 1981.

[Codd70] Codd, E.F. A Relational Model of Data for Large Shared Data Banks. **Commun. ACM**, Vol 13, No 6 (June 1970).

[Date81] Date C.J. **An Introduction to Database Systems**, 3rd Edition, Addison-Wesley, Reading, Massachusetts, 1981.

[Denn79] Denning, D.E., Denning, P.J., and Schwartz, M.D. The tracker: A threat to statistical database security. **ACM Trans. Database Syst.**, Vol 4, No 1 (March 1979), pp 76-96.

[Ever81] Everest G.C. and Lawrence C.T. Comparative Study of Database Management Systems on Microcomputers, **Proceedings ACM SIGMOD Workshop on Small Database Systems**, Orlando, Florida, October 13-15, 1981, pp 77-89.

[Gerr75] Gerritson, R. A Preliminary System for the Design of DBTG Data Structures. **Commun. ACM**, Vol 18, No 10 (October 1975), pp 551-557.

[Harr78] Harris, L.R. The Robot System: Natural language processing applied to data base query, **Proceedings ACM Annual Conference**, 1978, pp 165-172.

[Jero78] Jerold-Kaplan, S. On the difference between natural language and high level query languages, **Proceedings ACM Annual Conference**, 1978, pp 27-38.

[Lehm81] Lehman, P.L. and Yao, S.B. Efficient Locking for Concurrent Operations on B-Trees. **ACM Trans. Database Syst.**, Vol 6, No 4 (December 1981), pp 650-670.

[Ling81] Ling, T., Tompa, F.W., and Kaneda, T. An improved third normal form for relational databases. **ACM Trans. Database Syst.**, Vol 6, No 2 (June 1981), pp 329-346.

[Mart77] Martin J. **Computer Data-Base Organization**, 2nd Edition, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.

[Mart78] Martin, W.A. Some comments on EQS, a near term natural language database query system, **Proceedings ACM Annual Conference**, 1978, pp 156-164.

[MDBS81] MDBS.QRS - Query System/Report Writer, Version 1.02. Micro Data Base Systems Inc., Lafayette, Indiana, 1981.

[Reis80] Reisner, P. Human Factors Studies of Database Query Languages: A Survey and Assessment, **ACM Computing Surveys**, Vol 13, No 1 (March 1980).

[Schn80] Schneidermann, B. **Software Psychology: Human Factors in Computer and Information Systems**, Winthrop, Cambridge, 1980 (Chapter 8).

[Sper78] Sperry Univac. Data Management System (DMS 1100), Level 8R1, Data Administrator Reference. Sperry Univac, St. Pauls, Minnesota, 1978.

[Sper79] Sperry Univac. Query Language Processor (QLP 1100), User Reference Manual UP-8615. Sperry Univac, St. Pauls, Minnesota, 1979.

[Tsou80] Tsou, D.M. and Fischer, P.C. Decomposition of a relational scheme into Boyce-Codd normal form. **Proceedings ACM Annual Conference**, 1980, pp 411-417.

[Ullm80] Ullman, J.D. **Principles of Database Systems**, Computer Science Press, Rockville, Maryland, 1980.

[Welt81] Welty, C. and Stemple, D.W. Human Factors Comparison of a Procedural and a Nonprocedural Query Language. **ACM Trans. Database Syst.**, Vol 6, No 4 (December 1981), pp 626-649.

[Zloo75] Zloof, M.M. Query-by-Example, **AFIPS Conference Proceedings**, Vol 44, 1975, pp 431-438.

CHAPTER 4: **THE MDBS PACKAGE**

In this chapter the database system on top of which the VIADUCT package was implemented is described and evaluated in the light of the principles and objectives of database sytems introduced in Chapter 3, as well as the recommendations of the CODASYL committee.

4.1 **An overview of the system**

The database management system used in this research is known as MDBS. It is produced by Micro Data Base Systems Inc., Lafayette, Indiana. Since it was first made available in 1980, MDBS has been hailed as the only 'true' database management system available for microcomputers. At a price of about $1500, it is aimed at professional programmers and not microcomputer hobbyists. A number of reviews of MDBS have appeared in microcomputer journals [Land81a, Land81b, Mach82], and the package has featured favourably in surveys on database systems for microcomputers [Barl81, Ever81].

The MDBS package is based on the network model outlined in Section 3.2, and runs on any machine using the CP/M single-user operating system (introduced in the next chapter). It provides both restrictions and extentions to the CODASYL recommendations; these differences are discussed in Section 4.4.

MDBS provides only two of the three levels of implementation recommended by the ANSI/SPARC Study Group (Section 3.1). The

first is the Data Description Language (DDL) which is used to specify both the schema and a limited type of subschema. The DDL and its analyzer are discussed in the next section. The second component of the package is the set of Data Management System (DMS) routines, one for each statement of the Data Manipulation Language (DML). The DMS and DML are outlined in Section 4.3.

Additional programs in the package include a Query/Report System (QRS), a Recovery Transaction Logging (RTL) system, and a Dynamic Restructuring System (DRS). The former is discussed in Section 4.5, and the latter two in Section 4.6.

During the course of this research a new and more sophisticated version of the MDBS package, known as MDBS III, became available. The major improvements introduced in this version are outlined in Section 4.7.

4.2 The MDBS Data Description Language (DDL)

As an example, the DDL text corresponding to the schema of Figure 2.1 is presented in Figure 4.1. This schema resides in a normal text file and is created using the text entry facilities of the DDL processor. The schema is used only when generating the data dictionary in an initial, empty database, and should subsequently be removed from the view of users, since it contains readable password entries (which are encrypted in the database itself).

All the data stored in the database resides in a single file

```
FILES   EDUCATON.DB    1    512
DRIVE   1      50
PASSWORDS
        JOE                  020 020     SECRET
        FRED                 010 010     PRIVATE

RECORD  SUBJECT              010 010
ITEM    NUMBER    BIN   2    010 020
ITEM    TITLE     CHAR 20    010 010

RECORD  COURSE               010 010
ITEM    NUMBER    BIN   2    010 020
ITEM    TITLE     CHAR 20    010 010

RECORD  TEACHER              020 020
ITEM    NAME      CHAR 20    020 020
ITEM    AGE       BIN   1    020 020

RECORD  SECTION              010 010
ITEM    NUMBER    BIN   2    010 020
ITEM    TITLE     CHAR 20    010 010

RECORD  STUDENT              010 010
ITEM    NAME      CHAR 20    010 020
ITEM    REGNO     BIN   2    010 020

RECORD  ACTIVITY             020 020
ITEM    NAME      CHAR 10    020 020
ITEM    DESC      CHAR 20    020 020

SET     SUBJ-SET  AUTO 1:N   010 010
                             SORTED    NUMBER
OWNER   SYSTEM
MEMBER  SUBJECT

SET     TEAC-SET  AUTO 1:N   020 020
                             SORTED    NAME
OWNER   SYSTEM
MEMBER  TEACHER

SET     ACT-SET   AUTO 1:N   020 020
                             SORTED    NAME
OWNER   SYSTEM
MEMBER  ACTIVITY

SET     STUD-SET  AUTO 1:N   010 010
                             SORTED    NAME
OWNER   SYSTEM
MEMBER  STUDENT
```

Figure 4.1. A sample DDL file.

```
SET     SECTIONS AUTO 1:N 010 010
                                SORTED    NUMBER
OWNER   SYSTEM
MEMBER  SECTION

SET     WITH     MAN  1:N 010 010
                                SORTED    NUMBER
OWNER   SUBJECT
MEMBER  COURSE

SET     HAS      MAN  1:N 010 010
                                SORTED    NUMBER
OWNER   COURSE
MEMBER  SECTION

SET     TEACHES  MAN  1:N 020 020
                                SORTED    NUMBER
OWNER   TEACHER
MEMBER  SECTION

SET     ENROLL   MAN  N:M 010 010
        SORTED   NUMBER         SORTED    NAME
OWNER   SECTION
MEMBER  STUDENT

SET     ADVISES  MAN  1:N 020 020
                                SORTED    NAME
OWNER   TEACHER
MEMBER  STUDENT

SET     PARTICIP MAN  N:M 020 020
        SORTED   NAME           SORTED    NAME
OWNER   STUDENT
MEMBER  ACTIVITY

END
```

Figure 4.1 (Cont'd). A sample DDL file.

(named EDUCATON.DB in Figure 4.1), which can be spread over up to eight disk drives. A number of different users can be granted use of the system by including their userids and corresponding passwords in the schema. In Figure 4.1 two userids, JOE and FRED, with passwords SECRET and PRIVATE respectively, are declared. Each user's view of the schema is represented by his read and write access levels. These are the numeric quantities listed between each userid and password in Figure 4.1. Access levels may also be specified for each record type, data item, and set type in the schema. A maximum of 255 different access levels may be declared, and a user has (read or write) access to all objects declared with an access level less than or equal to his own access level. The write access level (the second of the two entries) must be greater than or equal to the read access level (the first entry) for every object in the schema. The default access level is zero. These access levels are the only method whereby a primitive type of subschema may be declared for an MDBS database.

The declaration of users in the schema is followed by the declaration of the record types (of which there may be up to 255) to be used in the database, and their associated data items (up to 255 per record type). Record type and data item names may be up to eight characters in length. Associated with each data item is its type and length (in bytes). Types that are catered for are: character strings of fixed length N (CHAR N), single precision integers (BIN 1), double precision integers (INT or BIN 2), real numbers (REAL 4), and Boolean values (LOG). Items may also be declared to be repeating items. Such items may repeat either a fixed number of times (i.e. a static array), or a

variable number depending on the value of another data item declared previously in the record type. In the MDBS DDL listing in Figure 4.2 the item DEP-NAME repeats a maximum of five times depending on the value of the item DEPENDS.

The last objects to be declared in the DDL file are the set types, whose names also comprise up to eight characters. Automatic or manual set membership of member record occurrences may be specified by means of AUTO and MAN entries respectively. The set type may be declared as one-to-many (1:N), many-to-one (N:1), many-to-many (N:M), or one-to-one (1:1). Multiple owner and member record types are permitted for each set type. Recursive set declarations such as the set MANAGES in Figure 4.2 are also allowed in MDBS.

In the case of 1:N sets, an ordering may be placed on the member record occurrences, while for N:1 sets an order may be specified for the owner record occurrences. For N:M sets, orderings may be placed on both owner and member record occurrences. Obviously, ordering has no significance in the case of 1:1 sets. The orderings which may be specified are: SORTED, LIFO (equivalent to LAST), FIFO (equivalent to FIRST), NEXT, PRIOR, and IMMATerial, which is the default and system defined. In the case of a sorted ordering, a key from the corresponding owner or member record type must be named. If it is not, the whole record is used as the sort key.

This concludes the brief description of the MDBS DDL. For more detail, the reader is referred to the DDL manual [MDBS79a].

For each set type declared in the schema, MDBS retains two currency indicators: the current owner and current member record occurrence of that set type. There is also a current record occurrence for each record type, as well as a current of run-unit. In CODASYL implementations the currency indicators for all sets in which a record type participates are automatically updated when an occurrence of that type is referenced, whereas in MDBS this is not necessarily so. Because of this there are a number of commands in the DML for specifically setting currency indicators.

The DML commands can be divided into nine categories: create, insert, delete, find, retrieve, modify, remove, assignment, and utility commands.

There are two create commands: create record (CR) and create record and store data in it (CRS). The created record occurrence is automatically added (see below) to any set in which it has been declared as an automatic member. It is the responsibility of the application programmer to ensure that the correct record occurrences are the current owners of the set types involved. The created record becomes the current of run-unit and current of the corresponding record type.

The commands to add (or connect) a record occurrence to a set occurrence are: add current of run-unit to set (ACS), and add member to set (AMS). In the second case a record type, of which the record to be added is the current occurrence, must be specified, as well as the set type involved. Again it must be ensured that the correct owner occurrence is selected. The

position of the new member occurrence in the set depends on the ordering specified in the schema (see Section 4.2). The new member becomes the current of run-unit and the current of the set type.

There are four delete commands all of which have the same effect; they simply operate on different currency indicators. They are: delete record based on current of run-unit (DRC), current member (DRM), current owner (DRO), and current record (DRR). In each case, all set occurrences of which the deleted record was an owner are deleted (i.e. all members are lost), and for all set occurrences in which the deleted record was a member, the record occurrence is removed (see below) from the set. All currency indicators which referenced the deleted record occurrence are set to null (an undefined value).

The commands to find (locate) a record occurrence are the most numerous. Since there is no means of finding a record occurrence directly in MDBS, all find commands reference the set type of which the desired record type is either an owner or member. There are commands to find the first member (FFM), last member (FLM), first owner (FFO), and last owner (FLO) of the set type. The definition of first or last is determined by the set ordering specified in the schema. In order to traverse a set, there are commands to find the next or previous member (FNM, FPM) relative to the current member, and to find the next or previous owner (FNO, FPO) relative to the current owner. Again, the next or previous occurrence is dependent on the set ordering. It is also possible to find a member or owner occurrence based on the value of its sort key (FMSK, FOSK), in which case the first

occurrence whose sort key matches the value specified is found. In all the above cases, if the desired record occurrence is found, the current of run-unit and either current owner or current member are updated accordingly. If it not found, the corresponding currency indicators are set to null, and the end-of-set status (i.e. -1) is returned by the DMS. The current of record type is not affected by any of the above commands.

In order to retrieve all the information from a record occurrence specified in one of the currency indicators, the following commands are provided: get data from current of run-unit (GETC), current member (GETM), current owner (GETO), and current record (GETR). Individual fields may be retrieved by specifying a field name to one of the following commands: get field from current of run-unit (GFC), current member (GFM), current owner (GFO), or current record (GFR).

The commands to modify information in the database are similar to the retrieval commands. Either a complete record is modified (PUTC, PUTM, PUTO, PUTR), or an individual field is changed (SFC, SFM, SFO, SFR).

The remove commands that are provided are: remove the current member from the specified set (RMS), in which case the member occurrence previous to the current one is linked to the member following the current member; or remove all set members (RSM), which means that the current owner of the set is severed from all its members. In neither case are any record occurrences actually deleted from the database, although it is possible that they become inaccessible as a result of not being members of any

other set type.

Assignment commands refer to the assignment of values to currency indicators, thereby allowing for the traversal of the database network structure. Basic assignments are: assign to the current owner or member of a set type the database key of the current of run-unit (SOC, SMC) or the current record of some type (SOR, SMR); the converse of the latter is to assign to the current record of some type the database key of the current owner or member of a set type (SRO, SRM). To move 'down' the network (i.e. away from the SYSTEM entry point), the current owner of one set occurrence can be assigned to be the same as the current member of another set occurrence (SOM), while to move 'up', the current member of one set occurrence is determined from the current owner of another (SMO). In order to traverse the database while remaining at the same depth, commands are provided to establish the current member of one set type as the current member of another set type (SMM) and to establish the current owner of one set type as the current owner of another set type (SOO). Of course, all these commands may be applied only to valid relationships defined in the schema: in each case the record types concerned must participate in both the set types referenced in the command.

Utility commands include those to open and close the database (OPEN, CLOSE), return run statistics for the database (STAT), and get the owner and member count of a set (GOC, GMC) - that is, the number of owner or member occurrences of a set occurrence. It is also possible to retrieve the record type of the current of run-unit or current owner or member of a set type (GTC, GTO,

GTM), as well as to check the record type represented in these currency indicators (CCT, COT, CMT).

For each of the commands in the DML appropriate error codes are returned if exceptional conditions occur or if security restrictions are violated in the course of a call to the DMS. Further details of the DML commands can be found in the DMS manual [MDBS79b].

4.4 Differences between CODASYL and MDBS

4.4.1 Restrictions

It is obvious from the schema used as an example (Figure 4.1) that the MDBS DDL capabilities are far weaker than those of its CODASYL counterparts. Rather than list all the clauses omitted by MDBS, only the more important omissions are discussed here.

The first restriction is the inability to specify a true subschema, thereby limiting the level of data independence that can be achieved. As a consequence of this different versions of the DMS routines are available, one for each host programming language supported. The views that can be represented are strict subsets of the actual schema, and are hierarchically structured as determined by the nature of the access levels. This means that, for example, no disjoint views of the database can be defined.

The schema does not provide for the assignment of areas within

the database, and so the whole database is either open or closed at any one time. Because of this there are, of course, no area currency indicators.

It is not possible to define LOCATION MODEs of record types in MDBS, neither do methods for SET OCCURRENCE SELECTION exist (Section 3.2.2). The default record occurrence LOCATION MODE is VIA SET, and SET OCCURRENCE SELECTION is effectively through LOCATION MODE OF OWNER (VIA SET), so it is not possible to access a record occurrence directly. This has serious speed implications, especially with regard to large database systems. It also means that the application program itself must explicitly traverse paths to record types, updating currency indicators accordingly, as this cannot be performed automatically by the system.

Specification of key items in MDBS is limited to either a single item, or all the items within a record type. This means that the items cannot be arbitrarily concatenated to form keys, and this, along with the fact that duplicate keys cannot be explicitly disallowed, makes integrity harder to enforce. (Arbitrary keys can in fact be achieved by isolating the items to be used as a key in a separate record type, and making it the owner of a 1:1 set type whose member record type comprises the remaining items. An example of this appears in Section 10.4.2.)

All items have to be explicitly stored by the application program as no VIRTUAL or derived (calculated) values are allowed in MDBS. This, along with the fact that no integrity checks (such as in a CHECK clause) can be supplied by the DBA, means that it is the

responsibility of the application programmer to ensure that the integrity of the data is not violated.

If a user has write access to a record type, he may also delete any occurrence of that type. Thus the MDBS access levels do not provide as comprehensive protection as do the access control locks of CODASYL. The user may also remove (disconnect) an occurrence from any set in which it participates, as all set membership is OPTIONAL in MDBS and cannot be defined to be either FIXED or MANDATORY (Section 3.2.2).

The deletion of a record occurrence in MDBS results in the effective deletion of all its member occurrences, so it is again the responsibility of the user to ensure that no information is lost when performing such a command.

A number of DML commands are excluded from MDBS because of the restrictions applied by the DDL. For example, no commands for locating record occurrences directly or for finding record occurrences with duplicate keys are included. Since MDBS is designed for a single user system, there is no need for commands relevant to area and record locking protocols (Section 3.2.4).

4.4.2 **Extensions**

The main extension to the CODASYL recommendations supported by MDBS is the provision of a wider variety of set types. All the types introduced in Section 3.2.1 (1:N, N:1, N:M, 1:1) are available to the database designer. As a result of the

inclusion of many-to-many and many-to-one sets, two currency indicators are required for each set type - the current owner and current member record occurrence. In CODASYL there is a single current-of-set currency indicator, which may be the owner of the set or one of its members. MDBS also allows the use of recursive sets (Figures 3.1(f) and 4.2).

In MDBS, more than one record type may be declared to be the owner of a particular set type (Figure 3.1(e)), and more than one to be the member (Figure 3.1(d)), whereas CODASYL permits only the latter case.

Turning to the DML, MDBS allows most functions to operate on the record occurrence represented by a number of the currency indicators. CODASYL usually requires that a record occurrence is the current-of-run-unit before an operation may be performed on it.

4.5 **The MDBS Query/Report System (QRS)**

The query language processor supplied with the MDBS package is known as the Query/Report System (QRS). As mentioned in Section 3.3, this is a keyword oriented language which accepts pseudo-English as input. Although MDBS claims that the QRS is non-procedural, the inclusion of a compulsory PATH clause (see below) requires the user to be well acquainted with the structure of the database before using QRS. As a result, the PATH clause has been criticized for detracting from the ease-of-use of the system [Land81b]. QRS is a stand-alone processor which operates

directly on database files without requiring any additional interface. When executing, it occupies approximately 44K bytes of main memory, thus leaving about 12K bytes for buffer space.

The syntax of a QRS command is as follows:

<COMMAND> <FIND clause> <CONDITIONAL clause> <PATH clause>

There are four commands having this syntax: LIST, WRITE, CHANGE, and STATS. The purpose and operation of each of these commands as well as the form and use of the other syntactic elements is outlined below.

The LIST command is used to generate a report based on the items appearing in the FIND clause. The item values which will be listed are dependent on the conditions specified in the CONDITION clause and the sets chosen for the PATH clause. The report is sent to either the console or the printer: the columns are headed appropriately. A statistical analysis of any numeric data listed is produced at the end of the report.

The WRITE command operates in the same way as the LIST command, except that the output is sent to a user-specified disk file and is not formatted in any way. This file can then be used as input to other software packages such as word processors or sort packages.

The CHANGE command is used to modify the value of a data item in the database. This may be accomplished by assigning a new absolute value to the item, or by modifying it in accordance with an arithmetic expression. Depending on the conditions specified, the change may affect only a single record occurrence

in the database or any number that satisfy the conditions. Only one data item type may be modified at a time, and modification of a sort key is prohibited.

STATS produces a statistical analysis of any numeric items supplied in the FIND clause, subject to the conditions specified. For each data item included in the FIND clause the statistics computed are: the maximum, minimum, and average values, the standard deviation, variance, and sum of the values, and the number of observations. For character data only the number of observations is recorded.

The FIND clause is used to specify which data item types are to be operated on by a command. The clause comprises one or more terms, optionally separated by commas, where each term is either a data item type or an arithmetic expression involving one or more types. In the case of the LIST command, the order of the item types specified in the FIND clause determines the format of the report produced. In addition, control breaks may be introduced into a report by specifying the data items to be checked for value changes. This is done after the full list of data items and is preceded by the keyword BY. Using the schema of Figure 4.1, and assuming prior entry of the LIST command, an example of a FIND clause is: TEACHER.NAME, STUDENT.NAME

The CONDITION clause is optional and, when used, must be preceded by the keyword FOR. The clause is used to place restrictions on the values of items which are to be utilized in the execution of the command. It comprises any number of relational expressions, combined with Boolean operators (AND, OR, XOR, NOT) to form a

Boolean expression. Each relational expression consists of two terms (as defined above), separated by one of the relational operators: =, NE, <, >, LE (<=), GE (>=). For character data, wildcard and match-one expressions may be included through the use of the symbols '$' and '?' respectively. An example of a CONDITION clause (using the schema of Figure 4.1) is: FOR TEACHER.AGE LT 30 AND STUDENT.NAME = 'W$'

The PATH clause, identified by the keyword THRU, is compulsory for every command and is used to specify which sets must be traversed in order to execute the command entered. This requires the user to identify which sets are needed to connect all the data item types referenced in the FIND and CONDITION clauses. Furthermore, the path specified must start with a SYSTEM-owned set, form a fully connected path, and the sets must be entered in the correct order. If a set has to be traversed from its member record type to its owner record type, then its name must be preceded by a '>' symbol in the PATH clause. The PATH clause that would have to be included for a query involving the FIND and CONDITION clauses in the previous examples might be: THRU TEAC-SET, ADVISES. There are a number of possible alternate paths, for example: THRU STUD-SET, >ADVISES.

From the above it can be seen that the QRS is fairly procedural in nature as it is imperative that the user has a copy of the schema description available to him. If there is more than one path connecting the data items referenced in the command (as in the above example), the user's choice may have serious performance implications of which he is unaware.

In addition to the four commands having the above syntax, there are a number of utility commands. These provide inter alia for the specification of report formatting information, the definition of macro commands and synonyms for the data item types in the database. Queries stored in a text file on disk can be read and executed as if they had been entered from the console and parts of the data dictionary can be viewed. The QUIT command terminates execution of the QRS.

Further details of the QRS are available in the MDBS.QRS User Manual [MDBS81a].

4.6 Other features of MDBS

Two additional utilities are supplied with the MDBS package. These are the Recovery/Transaction Logging System (RTL) and the Dynamic Restructuring System (DRS).

The RTL provides the recovery procedures necessity of which was outlined in Section 3.2.4, while the DRS allows for the changing requirements of database installations as mentioned in Section 3.1. Neither of these utilities is of relevance to this thesis; more information is available in the appropriate reference manuals [MDBS80a, MDBS80b].

4.7 MDBS III

During the course of this research a more sophisticated version

of the MDBS package, known as MDBS III, became available. However, MDBS emphasize that it is a separate product aimed at a higher-priced market, and that they will continue to support the previous version, which is now known as MDBS I. Furthermore, MDBS I is upwardly compatable with MDBS III.

MDBS III incorporates a number of extensions and improvements to MDBS I. It is available for a wider range of processors and in two forms: a machine language form, and one written in the C programming language [Kern78] for portability between systems utilizing the Unix operating system [Ritc74]. The package is now able to support multi-users and, depending on the configuration and maximum number of users desired, ranges in price from $3 600 to $52 800.

Apart from the inclusion of multi-user capabilities, the package is now host language independent, includes CALC record location mode, permits the declaration of areas, and includes date and time data types. Features enhancing integrity include: range checks on item values, duplicate key exclusion, and fixed set retention (Section 3.2.2). Security improvements include: more flexible access rights, and the possibility of encrypting data item values.

Additions to the DML include active and passive lock and unlock instructions for use in multi-user environments, enhanced find commands, user-defined currency indicators, commands for areas, and commands for performing Boolean operations between set occurrences. Full details of these and other improvements can be found in the MDBS III User's Manual [MDBS81b].

References

[Barl81] Barly K.S. and Driscoll J.R. A Survey of Data-Base Management Systems for Microcomputers. **Byte**, Vol 6, No 11 (November 1981), pp 208-234.

[Ever81] Everest G.C. and Lawrence C.T. Comparative Study of Database Management Systems on Microcomputers. **Proceedings ACM SIGMOD Workshop on Small Database Systems**, Orlando, Florida, October 13-15, 1981, pp 77-89.

[Kern78] Kernighan, B.W. and Ritchie, D.M. **The C Programming Language.** Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.

[Mach82] Machrane, B. A Review of MDBS. **Microsystems**, Vol 3, No 3 (May/June 1982), pp 28-33.

[MDBS79a] MDBS.DDL User's Manual. Micro Data Base Systems Inc., Lafayette, Indiana, 1979.

[MDBS79b] MDBS.DMS User's Manual. Micro Data Base Systems Inc., Lafayette, Indiana, 1979.

[MDBS80a] MDBS.RTL - Recovery/Transaction Logging System. Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

[MDBS80b] MDBS.DRS - Dynamic Restructuring System. Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

[MDBS80c] MDBS - CP/M Machine Language Interface Notes. Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

[MDBS80d] MDBS - PL/I Interface Notes. Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

[MDBS81a] MDBS.QRS - Query System/Report Writer, Version 1.02. Micro Data Base Systems Inc., Lafayette, Indiana, 1981.

[MDBS81b] MDBS III User's Manual. Micro Data Base Systems Inc., Lafayette, Indiana, 1981.

[Land81a] Landgarten, H. MDBS, Part I: The Database Manager. **Lifelines**, Vol 2, No 2 (July 1981), pp 10-15.

[Land81b] Landgarten, H. MDBS, Part II: QRS. **Lifelines**, Vol 2, No 4 (September 1981), pp 9-14.

[Ritc74] Ritchie, D.M. and Thompson, K. The UNIX Timesharing System. **Commun. ACM**, Vol 17, No 7 (July 1974), pp 365-375.

CHAPTER 5: **THE HARDWARE/SOFTWARE ENVIRONMENT**

This chapter describes the hardware and software environment in which VIADUCT was developed. This has implications with regard to the machines on which the package may be implemented as well as constraints on its portability.

5.1 **The hardware**

A South African made microcomputer system donated to the University of Cape Town by its manufacturer, Microcom (Pty) Ltd, was used for this research. The computer, known as the Micro-64, consists of a processor, two single-sided 8" floppy-disk drives each with a maximum storage capacity of 600K bytes, a CRT terminal, and an 80 character-per-minute dot matrix printer.

The processor is built around a South African defined bus structure known as the SABUS [Micr81]. The utilization of a bus structure allows for the modular construction of the machine and provides ease of expansion. Furthermore, it also facilitates fault finding and any subsequent repair of the system. The processor comprises a number of modules each of which plugs into the bus. These are: a central processing unit (CPU) board, four 16K byte memory cards, a communications card, and a floppy-disk controller board.

The main component of the CPU board is an 8-bit Zilog Z80A microprocessor chip, which runs at a clock frequency of 4 MHz and

is capable of addressing 64K bytes of memory. Also on the board is a 2K EPROM (Erasable Programmable Read Only Memory) which is used for bootstrap purposes.

Each memory card consists of 16K bytes of static RAM (Random Access Memory). Any 2K block may be disabled to allow for monitors and bootstrap EPROMs. These cards contain facilities allowing them to be used in bank switched memory environments required for the support of 8-bit multi-user operating systems [Digi81a].

The communications card provides two serial RS232C data ports and one parallel data port for connection to peripheral devices. The transfer rates of the serial ports are switch selectable and range between 50 and 19 200 baud.

The capabilities of the floppy-disk controller are closely linked to those of the operating system being used (see next section). In the present configuration, the floppy-disk controller can control up to eight drives, these being any combination of 5 1/4" or 8" floppy-disk drives. Each of these drives may be either single or double-sided and may record information in any of four different densities.

The system can be upgraded by replacing the CPU board with one containing a more powerful Intel 8088 or 8086 16-bit microprocessor. For a multi-user configuration, more memory cards can be added to provide additional program space, while extra communications cards can be included if more than two terminals are required. Modules allowing the addition of hard-

disk systems are currently being developed.

5.2 **The operating system**

The operating system available on the Micro-64 was CP/M (Control Program for Microcomputers) version 2.2, supplied by Digital Research Inc., Pacific Grove, California [Digi78a]. With over 200 000 users worldwide, CP/M has become the de facto operating system standard for microcomputers based on the Intel 8080 (or 8085) and Zilog Z80 microprocessors. This can no doubt be attributed to its timely development, extremely low purchase price (initially $75, now $150), portability, and large users group. Another factor influencing its popularity is the widespread availability of over 500 application software products.

CP/M was developed in 1973 by Dr. Gary Kildall (now head of Digital Research) for the Intel 8080 microprocessor to exploit the (then new) floppy disk storage medium and the possibility that, because of declining hardware costs, a machine could be dedicated to a single engineer or programmer. During the course of the next few years, CP/M was implemented on a number of different computer systems. This task was made simpler by the fact that the operating system comprises two parts: an invariant portion and a variant portion. The invariant portion, the Basic Disk Operating System (BDOS), is written in PL/M. The variant portion, known as the Basic Input/Output System (BIOS), contains all device dependent code. It is written in assembly language and consequently must be customised by the hardware vendor. As

distributed, CP/M is configured for the Intel MDS800 microcomputer development system.

Details of the operating system not relevant to this work are omitted from this section. The interested reader is referred to the CP/M manuals [Digi78a-Digi78d] and the many articles and books available [Hoga81, Zaks80].

There is a third component of the operating system, the Console Command Processor (CCP), whose function is to interpret commands entered by the user. A minimal number of CP/M commands are permanently resident in memory; the rest, including executable user programs (COM files), are transient commands which reside on disk, and are loaded into memory and executed when their file names are entered from the console.

CP/M resides in the top portion (highest addresses) of the available memory and, depending on the facilities included in the BIOS, occupies about 7K bytes of storage. The base page of memory (the first 256 bytes) is also reserved for use by CP/M. This means that CP/M can run in systems of size ranging from a minimal 16K configuration to a full 64K system, the only difference being the amount of memory remaining to run user programs. This area of memory is known as the Transient Program Area (TPA).

Up to 16 logical disk drives, each having a maximum storage capacity of 8 Megabytes, can be controlled by CP/M. As a result a 24 Megabyte hard disk, for example, must be divided into three 8 Megabyte logical drives.

The development of CP/M has not remained static and there now exist a number of derived operating systems: MP/M [Digi81a], which is a multi-user, multi-tasking operating system, and CP/NET, a combination of CP/M and MP/M. CP/NET allows communication over a network of machines where there is one master machine running MP/M and a number of slave machines running CP/M. CP/M and MP/M are now both available for the Intel 8086 (or 8088) 16-bit microprocessors; these versions are known as CP/M-86 and MP/M-86 respectively.

5.3 **The Database Management System**

MDBS, the database management system chosen as the base for this research, was the most comprehensive implementation available that could run under the CP/M operating system. It is described in some detail in Chapter 4.

5.4 **The programming languages**

VIADUCT is written mostly in PL/I with approximately 80 lines of Intel 8080 assembly language. The version of PL/I used is based on the ANSI General Purpose (G) subset specified by the PL/I standardization comittee. It is known as PL/I-80 and is supplied by Digital Research, Pacific Grove, California [Digi80a-Digi80c].

When a decision had to be made as to the choice of programming language for VIADUCT, there were a number of possibilities: a

variety of Basic compilers and interpreters, an implementation of PL/I, and Intel 8080 assembly language. Of these, PL/I was the obvious choice when the aspects of power and portability were considered. The interface between VIADUCT and the MDBS database system had to be written in Intel 8080 assembly language even though a version of the DMS routines which could be called directly from PL/I-80 became available [MDBS80b]. The reasons for this are given in Section 10.2.

Although a number of Pascal and C compilers were subsequently released, PL/I-80 remained a good choice because of its powerful systems programming facilities essential for this type of package, and for reasons of portability discussed in the next section. PL/I-80 has been well received in reviews [Gilb81, Derf81], one of which uses it as a reference point for the benchmarking of over 50 language implementations [Gilb81].

PL/I-80 is a three pass compiler which can operate in 40K bytes of main memory, although 64K of memory is needed for the compilation of programs of reasonable size. The compiler supports the compilation of separate modules which must then be linked with the PL/I library routines to produce an executable file. The linker [Digi80c] provides the facility to link modules as overlays so that programs larger in size than the physical memory available can be executed. An overlay manager is responsible for the loading of the overlays when referenced at run time.

5.5 **Portability issues**

In its present form VIADUCT can be run on any machine which supports version 2 of the CP/M operating system. It could equally well run under any of the CP/M compatible operating systems (e.g. TurboDos, I/OS), or as a single task under the multi-user version of CP/M, namely MP/M. In this latter case the buffer space available would be considerably reduced, resulting in more disk accesses and consequently slower execution speed. Details of memory and mass storage requirements can be found in the user manual (Appendix B). Constants declared in VIADUCT determine the size of the largest database schema that can be accomodated by the package. Details of these constants are discussed in Section 7.2.

There are essentially four components of the package that have to be considered when discussing the porting of VIADUCT to a new environment. These are: the operating system (CP/M), the database management system (MDBS), the implementation language (PL/I-80), and the assembly language routines (Intel 8080).

CP/M has been implemented on almost every 8-bit microcomputer available, including the Apple which has an incompatable processor (the Motorola 6502) and thus requires a special CPU board containing a Zilog Z80 processor (the Microsoft softcard). IBM offers CP/M-86 as an alternate operating system for its personal computer, and has also made it available on the IBM Displaywriter. CP/M simulators have even appeared on machines such as the DEC PDP-11 [Greu82]. The single-sided, single-density 8" floppy-disk format used by CP/M is an IBM standard (3704),

thus facilitating the transfer of packages between machines.

Although VIADUCT provides many features not available in MDBS, the MDBS DDL itself was not altered in any way. This means that VIADUCT can be used with any existing MDBS database. All data stored in the database by VIADUCT is in a form compatible with that used by the MDBS QRS, so that both packages may be used on the same database without the need for any modification.

MDBS I will run on any CP/M-based machine, while MDBS III, with which MDBS I is upward compatible (Section 4.7), is available for a number of different machines and operating systems. These include the UNIX and Xenix operating systems running on Intel 8086, Zilog Z8000, or PDP-11 processors; RT/11, RSTS, and RSX-11M on the PDP-11; and CP/M-86, MP/M-86, IBM PCDOS and OASIS-16 on the Intel 8086.

The Subset G implementation of PL/I has been adopted by DEC, Data General, Wang, and Prime. Furthermore, Subset G is upward compatible with the full implementation of PL/I so that VIADUCT should compile on any machine running either the full or Subset G implementation of PL/I. Digital Research are soon to release their version of PL/I for the Intel 8086 processor which will be known as PL/I-86. Even if the language features implemented are not identical to those in PL/I-80, PL/I-86 should provide upward compatibility from PL/I-80.

The Intel 8080 assembly language used in VIADUCT is restricted to a single assembler source file (INTEL.ASM). The 80 lines of text in this file would have to be translated into the

appropriate assembly language and re-assembled before the package could be used on a machine utilizing a processor other than an Intel 8080 (or 8085) or Zilog Z80. It is felt that the inclusion of these few lines of assembly language code does not seriously affect the portability of VIADUCT particularly bearing in mind automatic assembly language translators such as XLT86 [Digi81b].

Some features of the Soroc IQ120 terminal (such as the clear screen function and cursor addressing facilities) are used by VIADUCT. The routines supporting these features are isolated in a PL/I module (SOROC.PLI) which should be altered if a terminal incompatible with the Soroc is to be used.

References

[Derf81] Derfler, F.J. and Greene, W.H. PL/I-80 two-pass compiler from Digital Research. **Infoworld**, Vol 3, No 28 (November 30, 1981), pp 56-57.

[Digi78a] An Introduction to CP/M Features and Facilities. Digital Research Inc., Pacific Grove, California, 1978.

[Digi78b] ED: A Context Editor for the CP/M Disk System - User's Guide. Digital Research Inc., Pacific Grove, California, 1978.

[Digi78c] CP/M Assembler (ASM) User's Guide. Digital Research Inc., Pacific Grove, California, 1978.

[Digi78d] CP/M Dynamic Debugging Tool (DDT) User's Guide. Digital Research Inc., Pacific Grove, California, 1978.

[Digi80a] PL/I-80 Reference Manual. Digital Research Inc., Pacific Grove, California, 1980.

[Digi80b] PL/I-80 Applications Guide. Digital Research Inc., Pacific Grove, California, 1980.

[Digi80c] Link-80 Operator's Guide. Digital Research Inc., Pacific Grove, California, 1980.

[Digi81a] MP/M II Operating System - System Implementor's Guide Digital Research Inc., Pacific Grove, California, 1981.

[Digi81b] XLT86 User's Guide. Digital Research Inc., Pacific Grove, California, 1981.

[Gilb81] Gilbreath, J. A High Level Language Benchmark. **Byte**, Vol 6, No 9 (September 1981), pp 180-198.

[Greu82] Greuner, G. Simulating CP/M on a mini pays off in available software. **Electronic Design**, (July 8, 1982).

[Hoga81] Hogan, T. **CP/M User's Guide.** Osborne/McGraw-Hill, Berkeley, California, 1981.

[Micr81] Micro-64 System Documentation. Microcom (Pty) Ltd, Cape Town, 1981.

[Zaks80] Zaks, R. **The CP/M Handbook With MP/M.** Sybex, Berkeley, 1980.

CHAPTER 6: **DESIGN CRITERIA**

This chapter expands upon the objectives of VIADUCT presented in Chapter 1. The primary design criteria discussed in this chapter are ease-of-use, the maintenance of data integrity, and the enforcement of security restrictions. Finally an overview of the structure of VIADUCT is presented.

6.1 **Ease-of-use**

The most important criterion in the design of VIADUCT was that of ease-of-use. This attribute is especially desirable in the microcomputer field where users tend to be less proficient in data processing terminology and their companies may not have the financial means to employ data processing personnel. In fact, the user of a microcomputer may, in some instances, be a one-man company.

In order to enhance the ease-of-use of a database manipulation language, it was felt that the language should be non-procedural in nature and require as little input from the user as possible. Where necessary the user should be prompted for missing or clarifying information: a method similar to that employed in QUEASY and HI-IQ (Section 3.3). Furthermore, it should be possible to state simple commands simply, syntax should remain consistent and unified over all the commands of the language, and good error detection with meaningful error messages should be provided. These criteria are similar to those listed by Zloof

as objectives for designing a user-friendly data manipulation language for nonprogrammers [Zloo78]. The package should cater for both the casual and sophisticated user with on-line help facilities being provided. The provision of all the necessary database manipulation functions reduces the need for hosting in a general purpose programming language; it should be possible for a user to use only VIADUCT for all his data manipulation purposes.

The complexity of the underlying database schema may have a noticable effect on the amount of interaction required from the user. However it is hoped that, through the intelligent definition of subschemata, users need be presented with only those entities with which they are familiar. In this way, users who interact with a small portion of the database would be presented with a restricted (and consequently simpler) view of the database structure. Furthermore, VIADUCT users are presented with only those functions which they are permitted to execute and only those record and item types on which they may operate, thereby avoiding the display of superfluous (and possibly confusing) information. The methods used to achieve these objectives are discussed in the next chapter.

The objective of non-procedurality precludes the use of any form of path specification as present for example in MDBS.QRS (Section 4.5). VIADUCT automatically selects the path required to execute the command entered; if more than one such path exists and they are of equal merit, the user is asked which is to be taken. Even at this stage the user is not forced to make a decision. If the user prefers not to make a choice or does not know which path to choose, the system selects the shortest of the

paths or, if there are paths of equal length, chooses one arbitrarily. Practical examples of this operation are given in Chapters 2 and 9, while the implementation details are discussed in Chapter 9.

The provision of two modes of input, namely expert and help mode (introduced in Chapter 2) enhances the flexibility of the system by providing for the needs of both sophisticated and casual users. Help mode is completely menu-driven in an attempt to alleviate the trepidation felt by first-time computer users. The user is permitted to alternate freely between the two modes of input, thereby allowing the expert to view portions of the data dictionary if needed and the inexperienced user to attempt expert mode.

In expert mode, VIADUCT analyses each symbol comprising a command as it is entered, rather than only on termination of entry of the command. This allows for immediate error trapping and removes the tedium of having to re-enter an entire command when the system detects a syntax error. On detecting an error VIADUCT displays an appropriate error message followed by the command line up to the last correct symbol and awaits further input. The user is allowed three attempts at correcting any single error after which help mode is entered automatically. Command format and acceptance are dictated by the design of the parser; this is described in Chapter 8.

VIADUCT requires the user to enter only the minimal information necessary to identify the operation to be performed after which, in the execution phase, any additional information needed to

execute the command is solicited from the user (Section 10.3). Alternatively the user may enter all the required information at the time of command entry in which case no prompts are generated by the system in the execution phase. The first example presented in Chapter 2 illustrates this feature. Instead of responding with an 'N' to the CONDITIONS prompt the user could have responded 'Y' and then specified values for each of the items comprising the STUDENT record being created as well as for the ACTIVITY in which the STUDENT was to participate.

6.2 **Integrity constraints**

Since VIADUCT is a general purpose data manipulation language and not simply a query language, it is desirable that more rigorous integrity constraints are placed on permissible operations and on the item values that may be entered. Once again this is of special importance for a microcomputer system where total responsibility for the maintenance of integrity should not be given to the user. It was felt that the MDBS DDL (Section 4.2) does not provide adequate measures against the possibility of inaccurate entries by the user, and hence violation of data integrity.

A number of features not present in MDBS were included in VIADUCT in an attempt to provide improved data integrity. These included information required by CODASYL such as whether duplicate keys are allowed in sorted sets, and integrity constraints on the range of item values similar to the CODASYL CHECK clause (Section 3.2.4). The validity of data item types is

checked by means of the PL/I runtime system incorporated in the parser (Chapter 8). A feature similar to the CODASYL concept of virtual storage is also implemented in VIADUCT. This allows for the propagation of item values from one record type to record types at a lower level in the hierarchy. For reasons of portability the item values themselves rather than pointers to the values (CODASYL) are stored in the record types. However this redundancy of data is controlled by VIADUCT thus avoiding inconsistencies and reducing the number of item values the user must enter.

All this additional information is stored in the user's pseudo-subschema (Section 7.2) and is obtained from the user or database designer at pseudo-subschema generation time (Section 7.3). This method was chosen in preference to augmenting the DDL description, the advantage being that the DDL format is left unaltered. The user may therefore continue to use MDBS.QRS as well as any existing or future DML programs.

6.3 **Security aspects**

It was felt that the MDBS read and write access levels provided inadequate security measures for the database system (Section 4.2). To improve this, the concept of processing options for record types was introduced. These are similar in nature to the processing options used by IMS [Date81, p. 292].

Every record type has associated with it a set of processing options corresponding to the operations the user is allowed to

perform on that record type. Data item types also have processing options associated with them although only the list and the update function are of relevance to them. The combination of all the processing options for every record type to which the user has access gives the set of functions the user is allowed to perform on the database. This means that, during the process of command entry, only those functions that the particular user is allowed to execute and only those record types and data item types on which he may operate with each function are presented to him.

This allows greater flexibility in the design of pseudo-subschemata for each user of the system. A user may be permitted to create an occurrence of a record type but not delete it, or may be allowed to list the entire contents of record occurrences but update only one data item within each record.

The use of processing options allows security violations to be detected at command entry time rather than only at the time of execution as is the case with most CODASYL systems [Date77, p. 386].

6.4 VIADUCT structure

The VIADUCT package consists primarily of two programs totalling approximately 4000 lines of PL/I. The first (named GENPSS and described in Section 7.3) is used to generate a pseudo-subschema for each user of the system and store these on disk for subsequent use (Section 7.1). The second program (named VIADUCT)

retrieves from disk the pseudo-subschema corresponding to a userid entered at the keyboard and responds to user commands by accessing the database with reference to the user's view as defined in his own pseudo-subschema.

VIADUCT itself consists of a number of modules. These can be logically divided into the main control module, the module which parses the commands and sets up an internal intermediate representation of each command (Chapter 8), and the modules which interpret and execute the stored command (Chapter 10). There is one module associated with each of the functions that can be performed in VIADUCT, namely: LIST, UPDATE, CREATE, ADD, REMOVE, DELETE, and EXIT. (These function names were chosen to be consistent with those used in the NDBS DML.) The structure of the function modules is described in Section 10.1. All of the modules other than the main module are defined as overlays (Section 5.4). This is done to allow the DMS to keep as many database pages as possible resident in memory and thereby improving execution speed.

The main control module of VIADUCT is now described. In order to identify himself to VIADUCT, a user must enter his userid and password. For reasons of security, the password is not echoed to the terminal screen. This feature requires the inclusion of an assembly language routine in VIADUCT since PL/I-80 does not provide a facility for non-echoed input. The userid entered by the user identifies the file corresponding to the user's pseudo-subschema which is then read into memory.

The entered userid and password are passed to the DMS routines in

order to open the database in one of two modes. If the user has processing options in his pseudo-subschema for functions additional to the LIST function, the database is opened in MODIFY mode; otherwise it is opened for retrieval only. This provides for additional security in the event of a system crash.

Following the successful opening of the database (which requires that the user has specified a valid password) input and output buffers are dynamically allocated for each record type as defined in the user's pseudo-subschema. Each of these buffers is of length equal to that of the corresponding record type. The buffers are used to store item values that are present in a command entered by the user (Section 8.4) and to store values that are retrieved during a database query (Section 9.5).

Control is now transferred to the parser. On the termination of command entry, the parser returns the function chosen by the user to the main module which calls the appropriate execution module. The process of calling the parser and function modules is repeated until the user selects the EXIT function at which point the database is closed and execution of VIADUCT is terminated.

References

[Date77] Date, C.J. **An Introduction to Database Systems.** 2nd Edition, Addison-Wesley, Reading, Massachusetts, 1977.

[Date81] Date, C.J. **An Introduction to Database Systems.** 3rd Edition, Addison-Wesley, Reading, Massachusetts, 1981.

[Zloo78] Zloof, M.M. Design Aspects of the Query-By-Example Data Base Management Language. In **Databases: Improving Usability and Responsiveness** (ed., B. Schneidermann). Academic Press, New York, 1978.

CHAPTER 7: THE PSEUDO-SUBSCHEMA

This chapter describes the pseudo-subschema construct utilized by VIADUCT. Justification is given for the necessity of such a construct, its structure and content are described, and the procedure for its generation is outlined.

7.1 Justification

In order to implement the features introduced in the previous chapter without altering the MDBS DDL file in any way, some form of additional internal data structure was required. This was further necessitated by the fact that the MDBS data dictionary (stored in the database itself) proved to be inaccessible, so that the DDL text itself was the only source of database description available for use by VIADUCT. For convenience, the DDL text of Figure 4.1 defining the education database is reproduced as Figure 7.1.

To require that the DDL file would always be available on-line when VIADUCT was run would negate all security precautions as the passwords associated with each userid are kept in the DDL file in readable text form (see Figure 7.1). If an internal representation of the schema was created on every execution of VIADUCT an unacceptable delay in program load time would result.

For these reasons a separate view of the database schema, known as a pseudo-subschema (pss), is generated for each userid

```
FILES   EDUCATON.DB    1    512
DRIVE   1      50
PASSWORDS
        JOE                  020 020     SECRET
        FRED                 010 010     PRIVATE

RECORD  SUBJECT              010 010
ITEM    NUMBER    BIN   2    010 020
ITEM    TITLE     CHAR 20    010 010

RECORD  COURSE               010 010
ITEM    NUMBER    BIN   2    010 020
ITEM    TITLE     CHAR 20    010 010

RECORD  TEACHER              020 020
ITEM    NAME      CHAR 20    020 020
ITEM    AGE       BIN   1    020 020

RECORD  SECTION              010 010
ITEM    NUMBER    BIN   2    010 020
ITEM    TITLE     CHAR 20    010 010

RECORD  STUDENT              010 010
ITEM    NAME      CHAR 20    010 020
ITEM    REGNO     BIN   2    010 020

RECORD  ACTIVITY             020 020
ITEM    NAME      CHAR 10    020 020
ITEM    DESC      CHAR 20    020 020

SET     SUBJ-SET  AUTO 1:N  010 010
                             SORTED    NUMBER
OWNER   SYSTEM
MEMBER  SUBJECT

SET     TEAC-SET  AUTO 1:N  020 020
                             SORTED    NAME
OWNER   SYSTEM
MEMBER  TEACHER

SET     ACT-SET   AUTO 1:N  020 020
                             SORTED    NAME
OWNER   SYSTEM
MEMBER  ACTIVITY

SET     STUD-SET  AUTO 1:N  010 010
                             SORTED    NAME
OWNER   SYSTEM
MEMBER  STUDENT
```

Figure 7.1. DDL file of the education database.

```
SET     SECT-SET AUTO 1:N 010 010
                              SORTED   NUMBER
OWNER   SYSTEM
MEMBER  SECTION

SET     WITH     MAN  1:N 010 010
                              SORTED   NUMBER
OWNER   SUBJECT
MEMBER  COURSE

SET     HAS      MAN  1:N 010 010
                              SORTED   NUMBER
OWNER   COURSE
MEMBER  SECTION

SET     TEACHES  MAN  1:N 020 020
                              SORTED   NUMBER
OWNER   TEACHER
MEMBER  SECTION

SET     ENROLL   MAN  N:M 010 010
        SORTED   NUMBER       SORTED   NAME
OWNER   SECTION
MEMBER  STUDENT

SET     ADVISES  MAN  1:N 020 020
                              SORTED   NAME
OWNER   TEACHER
MEMBER  STUDENT

SET     PARTICIP MAN  N:M 020 020
        SORTED   NAME         SORTED   NAME
OWNER   STUDENT
MEMBER  ACTIVITY

END
```

Figure 7.1 (Cont'd). DDL file of the education database.

declared in the schema and each is stored as a file on disk. These separate views provide additional security as each pss contains only that information relevant to the particular user's access rights. Passwords are not stored in the pss and it is not necessary for the DDL file corresponding to the database being used to be on-line. Instead VIADUCT requires a user's own pss to be available on disk before that user can access the database.

Consideration was given to the idea of storing the pseudo-subschema as part of the database itself. This was conceptually appealing and would allow the user to access portions of the data dictionary in the same way as he would other data in the database. Unfortunately implementing this would require the database files to be set up by VIADUCT so that it could augment the schema with the necessary record, item, and set types. This would mean that VIADUCT could not be used on existing databases, a fact that was deemed unacceptable. Furthermore, the speed of execution would be degraded as a result of having to access the database each time to check the validity of a command entered by the user.

7.2 Structure

The pseudo-subschema corresponding to each userid is stored in a separate PL/I disk file. It consists of a single record the structure of which is given in Figure 7.2. A decision had to be taken with regard to a suitable data structure by which to represent the pss. The number of valid schemata is countably

infinite. In order to cater for this wide range of possible schemata a dynamic data structure was initially used to represent the pss; however this was subsequently altered to a static representation for the reasons given below:

(a) Because of its use of pointers, a dynamic structure must always be loaded into the same place in memory. Since the pss is stored in a file and read into memory at run time, some type of relocatable data structure is a prerequisite.

(b) Stored as a static structure the pss can be manipulated from PL/I-80 as a normal disk file, since its records are of a fixed length and type.

(c) The static structure allows VIADUCT to be modularised, since its common (EXTERNAL) data is STATIC - a requirement of PL/I-80. A consequence of this is that overlays can be utilised to reduce overall program size, a factor of primary concern in a microcomputer application.

(d) Using the static structure the finding of paths between record types was greatly simplified, thus allowing a faster and more elegant solution to be attained.

The decision was therefore taken to store the information pertaining to record, set, and data item types necessary for the operation of VIADUCT in the form of (static) arrays of structures. The relationships between the set and record types are represented by means of an incidence matrix (described below). This method of representation has its own disadvantages, especially the fact that the maximum allowable number of set and record types has to be predefined. This results in wasted storage in many cases, a situation worsened by the inevitable sparseness of the incidence matrix. However, this shortcoming

```
%REPLACE
        max_records     BY  8,
        max_sets        BY 11,
        max_items       BY  3,

        /* The structure of the pseudo-sub-schema follows */

DCL     pss                                         FILE,

        1 pseudo_sub_schema STATIC EXTERNAL,
          2 db_name                                 CHAR  (12),
          2 incidence_matrix
                (1:max_records, 1:max_sets)         FIXED (7),
          2 proc_options                            BIT   (8),
          2 number_records                          FIXED (7),
          2 record (1:max_records),
            3 record_name                           CHAR  (8),
            3 number_record_owners                  FIXED (7),
            3 number_record_members                 FIXED (7),
            3 record_options                        BIT   (8),
            3 record_size                           FIXED (15),
            3 number_items                          FIXED (7),
            3 item (1:max_items),
              4 item_name                           CHAR  (8),
              4 item_options                        BIT   (8),
              4 item_type                           BIT   (3),
              4 item_size                           FIXED (7),
              4 max                                 CHAR  (4),
              4 min                                 CHAR  (4),
              4 depending_record                    FIXED (7),
              4 depending_item                      FIXED (7),
              4 repeat_factor                       FIXED (7),
          2 number_sets                             FIXED (7),
          2 set (1:max_sets),
            3 set_name                              CHAR  (8),
            3 number_set_owners                     FIXED (7),
            3 number_set_members                    FIXED (7),
            3 set_mode                              BIT,
            3 set_type                              BIT   (2),
            3 owner_order                           BIT   (3),
            3 member_order                          BIT   (3);
```

Figure 7.2. PL/I-80 definition of the pseudo-subschema record structure.

can be considerably reduced by tailoring the package to each particular database system. This is easily achieved by modifying the constants declared in the pss definition file, and then recompiling the modules comprising VIADUCT.

Refering to Figure 7.2 the pss contains the following primary components: the database file name ('db_name'), the incidence matrix, the set of processing options ('proc_options'), the number of record ('number_records') and set types ('number_sets') accessible to the user, and arrays defining each of these record and set types. Those components requiring further explanation are described below.

An incidence matrix [Pfal77, p. 169] is a means of representing a directed graph. Let G be a directed graph with point (vertex) set $P = \{p_i\}$ and edge set $E = \{e_k\}$. Then the incidence matrix of the graph G is an integer array $B_G = [b_{ki}]$, $k = 1,\ldots,|E|$, $i = 1,\ldots,|P|$ such that for each element $e_k = (p_i,p_j) \in E$, array element $b_{ki} = -1$, $b_{kj} = +1$ and $b_{km} = 0$ for all $m \neq i,j$. That is, the element b_{ki} is nonzero if and only if the point p_i is incident to the edge e_k.

In order to represent the relationships between the record and set types in the pss, the concept of an incidence matrix is extended as follows. Whereas a directed edge e_k connects point p_i to point p_j, set s_k may have a number of owner record types $\{r_i \mid i \in I_k\}$ (corresponding to p_i), and a number of member record types $\{r_j \mid j \in J_k\}$ (corresponding to p_j). Let R denote the total number of record types in the pseudo-subschema, and S the total number of set types. The incidence matrix B of the

pss T is then an S X R integer array $B_T = [b_{ki}]$ such that for each set type s_k, array element $b_{ki} > 0$ for all $i \in I_k$, $b_{kj} < 0$ for all $j \in J_k$, while $b_{km} = 0$ for all $m \notin I_k \cup J_k$. Thus for each element b_{ki} of the incidence matrix a positive value indicates that the i'th record type is an owner of the k'th set type and a negative value indicates that the i'th record type is a member of the k'th set type.

One problem that arises from the the use of an incidence matrix is how to represent a recursive set. In VIADUCT this was solved by implication: if a set type has an owner record type but no member record type in the incidence matrix it is assumed to be a recursive set. Further problems associated with recursive sets are discussed in Section 9.4.

A schematic representation of the incidence matrix corresponding to the relationships defined in the schema of Figure 7.1 for the user JOE is given in Figure 7.3. The incidence matrix record and set subscripts correspond to those used in the pss arrays in which the record and set information is stored. Apart from defining the relationships between the record and set types, the elements of the incidence matrix serve an additional purpose, namely to identify the key of a sorted set. If a relationship exists between a particular record and set type and the ordering of the set type (whether owner or member) is sorted, the absolute value of the corresponding incidence matrix element gives the subscript of the item type within the record type which is to be used as the key. More specifically, an absolute value of 1 is used to denote that the whole record is the key, while any other absolute value (other than 1 or 0) is one greater than the

		1. SYSTEM	2. SUBJECT	3. COURSE	4. TEACHER	5. SECTION	6. STUDENT	7. ACTIVITY
1.	SUBJ-SET	+1	-2					
2.	TEAC-SET	+1			-2			
3.	SECT-SET	+1				-2		
4.	STUD-SET	+1					-2	
5.	ACT-SET	+1						-2
6.	WITH		+1	-2				
7.	HAS			+1		-2		
8.	TEACHES				+1	-2		
9.	ENROLL					+2	-2	
10.	ADVISES				+1		-2	
11.	PARTICIP						+2	-2

Figure 7.3. Schematic representation of the incidence matrix corresponding to the schema of Figure 7.1. (Blank entries represent zeroes.)

corresponding key item subscript. This is possible since MDBS allows either the complete record or only a single item type to be the key in any one set type.

The processing options field is defined as an 8-bit string. The first seven bits of this string are used to denote whether a user has access rights to the function corresponding to that position in the string. For this purpose the functions are numbered as follows: 1 = LIST, 2 = UPDATE, 3 = CREATE, 4 = ADD, 5 = REMOVE, 6 = DELETE, 7 = EXIT. If a bit is set (equal to one) the user has access rights to the corresponding function.

The PL/I-80 type of FIXED (7) defines integers in the range -128 to 127. This means that with the standard configuration of VIADUCT a pss can accomodate schemata containing up to 127 record types each with a maximum of 127 item types, and up to 127 set types.

The array element within the pss describing the i'th record type, i.e. 'record(i)', is itself a structure containing a number of fields. The name of the record type as defined in the MDBS DDL ('record_name') is followed by the number of sets owning the record type ('number_record_owners') and the number of sets owned by the record type ('number_record_members'). Next in the definition are the user's processing options for the record type ('record_options'), the size of the record type in bytes (up to 32767), the number of item types comprising the record type, and an array of item type descriptions, one for each item type defined in the record type.

The information stored for the n'th item type within the i'th record type (i.e. 'item(i,n)') includes the name of the item type ('item_name'), the processing options associated with it ('item_options'), its data type ('item_type'), size in bytes ('item_size'), and the maximum and minimum values it may assume. The encoding of the data item types is shown in Figure 7.4. The maximum and minimum values are defined as being four bytes in length (CHAR (4)). This is to allow the storage of real numbers (which occupy four bytes) where the item is of type REAL. The storage of numeric values in a character field is performed by means of the type templates to be described in Section 8.4.

If an item is a derived (VIRTUAL) item (Section 6.2), the subscripts of the record type and item type from which the value is to be derived are stored in the 'depending_record' and 'depending_item' fields respectively. If the item is a repeating item the maximum number of repetitions is stored in 'repeat_factor'. When the number of repetitions is dependent on the value of some other item type in the record type the subscript of this item type is stored in the 'depending_item' field. If the item type is neither derived nor repeating all these fields ('depending_record', 'depending_item', 'repeat_factor') will contain zero.

The information relevant to each set type that is stored in the pss includes: the name of the set type ('set_name'), the number of record types owning the set type ('number_set_owners'), the number of record types owned by the set type ('number_set_members'), whether the set membership is automatic or manual ('set_mode'), the type of relationship represented by

```
DCL
        ((auto          INIT ('0'B),          /* Set modes */
        manual          INIT ('1'B)) BIT (1),

        (one_one        INIT ('00'B),         /* Set types */
        one_n           INIT ('01'B),
        n_one           INIT ('10'B),
        n_m             INIT ('11'B)) BIT (2),

        (immat          INIT ('000'B),        /* Set orderings */
        sorted          INIT ('001'B),
        fifo            INIT ('010'B),
        lifo            INIT ('011'B),
        prior           INIT ('100'B),
        next            INIT ('101'B),
        sort_duplicates INIT ('111'B)) BIT (3),

        (ch             INIT ('000'),         /* Item types */
        int             INIT ('001'),
        real            INIT ('010'),
        bin             INIT ('011'),
        log             INIT ('100')) BIT (3)) STATIC EXTERNAL;
```

Figure 7.4. PL/I-80 declarations for the encoding of set modes, set types, set orderings, and data item types.

the set type (e.g. one-to-many) ('set_type'), the order of owner record occurrences ('owner_order'), and the order of member record occurrences ('member_order'). The encoding used for the set modes, types, and orderings are given in Figure 7.4. The owner order for sets of type one-to-many or one-to-one and the member order for sets of type many-to-one or one-to-one automatically default to immaterial ('immat'). As can be seen in Figure 7.4, an additional ordering ('sort_duplicates') has been added to those available in MDBS. This ordering permits the existence of duplicate key values in sorted sets. Either this ordering or that of 'sorted' (in which duplicates are forbidden) may be specified at the time of pss generation (Section 7.3).

This concludes the description of the components comprising the pseudo-subschema.

7.3 **Generation**

A stand-alone program GENPSS must be executed in order to generate a pseudo-subschema corresponding to each particular userid defined in the MDBS DDL. It is assumed that the generation of pseudo-subschemata is performed by an experienced user: usually the database designer himself.

GENPSS first requires the user to enter the userid for which he wishes to generate a pss. The access rights corresponding to this userid defined in the MDBS DDL are used to identify which record, item, and set types are to be included in the pss: all

types with a read access level greater than that of the userid are automatically excluded from the pss.

A sample execution of GENPSS using the schema of Figure 7.1 to generate a pss corresponding to the userid FRED (whose restricted view of the schema is depicted in Figure 7.5) is given in Figure 7.6. The generation of a pss for the userid JOE (as used in the examples in Chapter 2) would be similar except that all record and item types are accessible to that userid and all processing options would be included in his pss.

The processing options associated with each record type in the schema are initially set up according to the user's read and write access levels. If a user has write access to a record type his processing options for that record type default to all possible functions. However, if the user has only read access to the record type his processing options default to the LIST and EXIT functions only. For item types, the LIST and UPDATE functions are the only applicable processing options since these are the only functions that operate on item types (Section 8.1). If the user has write access to an item type he is permitted to perform both these functions on the item type, but if he has only read access he is restricted to LISTing the values of that item type. However, if the user has neither write nor read access to a record or item type, that type is totally excluded from the pss. For example, the record type ACTIVITY in the schema of Figure 7.1 does not appear in Figure 7.5 or in the generation process depicted in Figure 7.6.

The processing options for a particular record or item type are

specified by entering in the appropriate column in any order the initial letters of all functions the user is to be allowed to perform on that record or item type (Figure 7.6). GENPSS checks that the options entered do not conflict with the access rights declared in the MDBS schema. The default processing options described above can be chosen by entering a carriage return instead of a list of options. If no options are to be entered, that is, the record or item type is to be excluded totally from the user's view, a zero is entered. This provides for the definition of disjoint views rather than the strictly hierarchical views offered by MDBS.

In addition to being used to specify processing options, the screen in Figure 7.6 is used to enter the maximum and minimum values an item type is allowed to assume. From Figure 7.6 it can be seen that maximum and minimum item values are requested only for numeric item types (REAL, INT, BIN 1, and BIN 2). For character data (CHAR N) the maximum value defaults to the length of the character string and the minimum value to one. Maximum and minimum values are not applicable to Boolean item types (LOG). For numeric item types the default values (which are used if the operator enters only a carriage return) are the maximum and minimum values that can be represented by the appropriate type in PL/I-80. In the case of a depending item the maximum repeat factor is used as the maximum value for the controlling item type. As an example, the default maximum for the item DEPENDS in Figure 4.2 would be five.

As can be seen from Figure 7.6 the GENPSS user is asked to specify whether duplicate key item values are to be allowed for

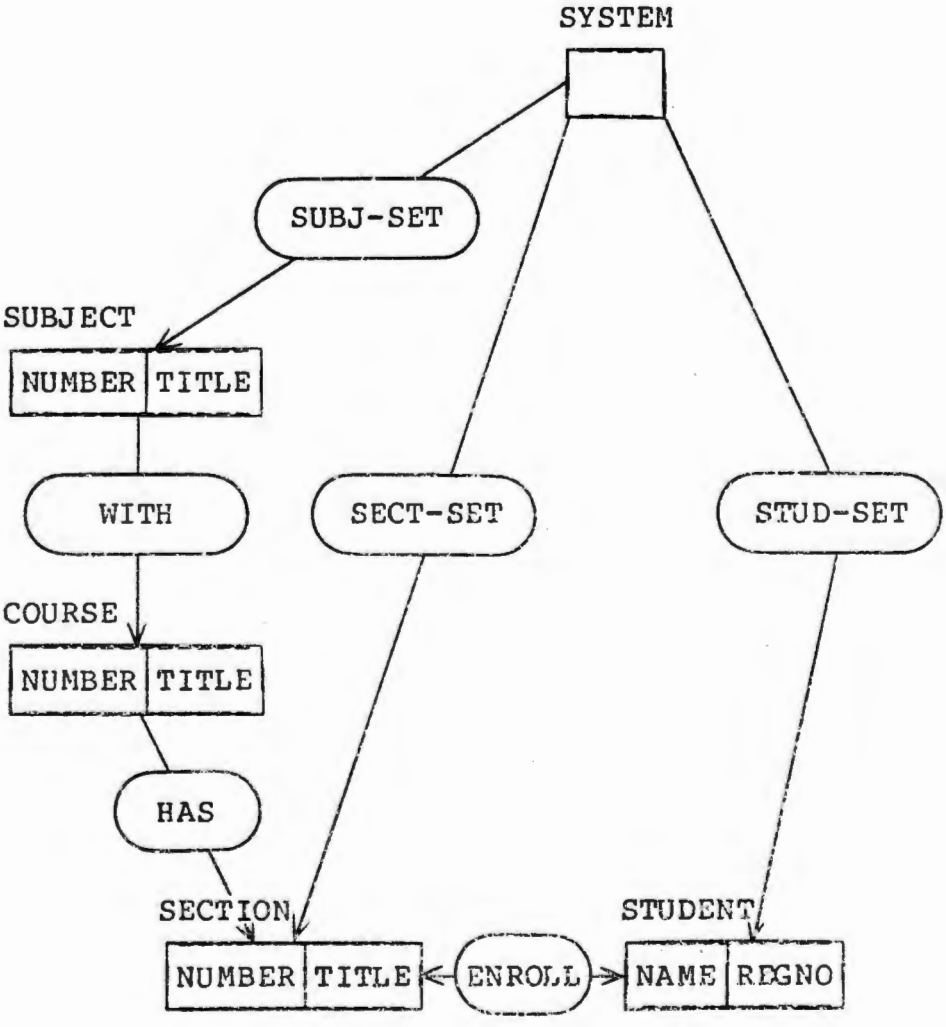


Figure 7.5. Restricted view of the schema of Figure 7.1 corresponding to userid FRED.

```
A>GENPSS<CR>
Pseudo-SubSchema Generator

            Enter userid : FRED<CR>
```

For the following record types enter the user's processing options, and for the item types enter the processing options and the maximum and minimum item values.

RECORD NAME	OPTIONS (LUCARD)	ITEM NAME	OPTIONS (LU)	MAX VALUE	MIN VALUE
SUBJECT	LUC	NUMBER	L	999	1
		TITLE	LU		
COURSE	LUC	NUMBER	L	999	1
		TITLE	LU		
SECTION	LUC	NUMBER	L	9999	1
		TITLE	LU		
STUDENT	L	NAME	L		
		REGNO	L	20000	1

Figure 7.6. A sample execution of GENPSS using the schema of Figure 7.1. (Userid FRED) (User responses are in boldface.)

```
For the following sorted sets enter whether
 duplicate key values are to be permitted.

      SET              KEY              Y/N
      ---              ---              ---

SUBJ-SET         SUBJECT-NUMBER         N
SECT-SET         SECTION-NUMBER         N
STUD-SET         STUDENT-NAME           Y
WITH             COURSE-NUMBER          N
HAS              SECTION-NUMBER         N
ENROLL           SECTION-NUMBER         N
ENROLL           STUDENT-NAME           Y


Are there any derived (virtual) items (Y/N) ? : N


Pseudo-subschema generated - file name FRED.PSS

A>
```

Figure 7.6.(Cont'd). A sample execution of GPSS using the schema of Figure 7.1. (Userid FRED) (User responses are in boldface.)

each of the sorted sets in the pss. For many-to-many sets with both owners and members specified as sorted, entries for each key are required from the user (e.g. the set ENROLL). If the user responds with a carriage return at any stage, the default option of allowing duplicates is chosen.

If there are to be any derived item types in the pss the user must respond to the last question of Figure 7.6 with a 'Y'. Since there are no derived items in this schema the user responds with an 'N' which results in the generation of the pss on disk and termination of GENPSS. If, however, the user had responded with a 'Y' he would have been prompted for the names of the derived item types and associated record types, as well as the names of the record and item types from which the values of the derived items are to be taken. In each case GENPSS checks that the two items are of the same type. An example of a schema with derived item types is given in Section 10.4.2.

If during the course of generating a pss the maximum number of record, item, or set types as declared in Figure 7.2 should be exceeded, GENPSS will terminate with an appropriate error message. To compile such a pss GENPSS and VIADUCT must be reconfigured accordingly.

Reference

[Pfal77] Pfaltz, J.L. **Computer Data Structures.** McGraw-Hill, New York, 1977.

CHAPTER 8: **THE PARSER**

This chapter describes the parser, one of the major components comprising VIADUCT. The responsibility of the parser is to ensure that commands formulated by the user are syntactically correct. To facilitate this, the user is offered two modes of input: expert mode (Section 8.2) and help mode (Section 8.3). To permit the processor modules (Chapter 9) to operate on commands once they have been completely entered, the parser must set up some form of intermediate command representation (Section 8.4).

8.1 General features

The parser module of VIADUCT consists of approximately 1000 lines of PL/I-80 code. It calls two external routines: an assembly language routine for character input (20 lines), and a common PL/I-80 routine (shared with the processor routines) to read and validate item values (60 lines). Each of these routines is discussed below. The parser is set to expert mode when it is read into memory at system initialization; after the execution of each command the parser returns to the mode that was being used at the termination of entry of the previous command.

In expert mode the command syntax of VIADUCT is keyword oriented and non-procedural, while in help mode command formulation is menu-driven. In neither case does the parser require that the entered command be complete (for execution purposes) but only

that it be syntactically correct. The user is allowed to interchange freely between the two modes of input. In either mode a command may be abandoned at any stage of entry by pressing the escape key on the terminal (ESC).

Each VIADUCT command consists of three components: the function to be performed, the record or item(s) on which the entered function is to operate (operand clause), and the constraints to be imposed on item values during the execution of a command (selection clause). The command format governs both the validity of commands entered by the user in expert mode and the sequence of menus displayed by VIADUCT in help mode. The complete command syntax is given in Appendix A.

As stated in the previous chapter, there are seven functions that the VIADUCT user may perform: LIST, UPDATE, CREATE, ADD, REMOVE, DELETE, and EXIT. The functions CREATE, DELETE, ADD, and REMOVE operate on a single record type, while UPDATE operates on a single data item type; LIST operates on a number of data item types. For all functions the selection clause is optional: any additional information required will be prompted for by the processor. If present, the selection clause is a list of relational expressions involving data item types and values separated by commas and preceded by the key word WHERE. VIADUCT interprets the selection clause as the conjunction of the logical expressions, that is, logical AND connectives are assumed between the expressions (see below). The relational operators supported are: equal (EQ), not equal (NE), greater than (GT), and less than (LT). In all instances the first operand of the expression is a specification of an item type (see below) and the

second is an item value. The ordering of the expressions is not significant. An example of a selection clause is:

```
WHERE TEACHER.AGE LT 25, ACTIVITY.NAME EQ SQUASH
```

The restriction of the logical connectives to AND operators was done for reasons of command syntax simplicity. The inclusion of arbitrary combinations of both logical and arithmetic relational operators would overcomplicate the syntax and possibly confuse the average VIADUCT user. It was felt that the infrequency of occurrence of such complex constraints did not warrant their inclusion in the syntax. Furthermore, simple OR constraints can always be implemented as separate queries.

All commands are parsed with reference to the user's view of the database as defined in his pseudo-subschema. Only those functions declared in the user's view are valid in expert mode or are displayed in help mode. Once a permissible function has been entered or selected by the user, all record or item types specified by the user in the operand clause of the command are checked for availability under that function. The record and item types available to the user in the selection clause of the command are those to which the user has access under the LIST function. This is because, in order to specify the values of item types in a selection clause, the user must know, and therefore be permitted to LIST, these values.

Details specific to each of the two input modes are discussed in the next two sections. Section 8.4 outlines how the parser represents each command in an intermediate form suitable for subsequent execution by the processor.

8.2 **Expert mode**

The initial screen displayed by VIADUCT to the user in expert mode was presented to the reader in Chapter 2 (Figure 2.2). After displaying this screen the parser awaits input from the user. The user's input is read by the parser one character at a time and is analyzed after the receipt of each symbol. A symbol is defined to be either a string of non-blank characters terminated by a blank or special character (see below), or a string of characters delimited by quote marks ('). The special characters used by the parser are period ('.'), comma (','), question mark ('?'), semi-colon (';'), escape ('ESC'), and carriage return (CR). The context and interpretation of each of these characters is covered elsewhere in this chapter. In certain instances (outlined below) a lookahead of one symbol is required before analysis of a symbol can be accomplished.

An assembly language routine had to be used to provide VIADUCT with character-at-a-time input facilities since PL/I-80 does not support such input. The inclusion of an assembly language routine for this purpose is essential for the following reasons: (a) the PL/I-80 string input feature permits a maximum string length of 80 characters which would not be sufficient for more extensive commands; (b) character-at-a-time input allows for the soonest possible error detection, and subsequent correction by the user requires minimal reentry on his part; (c) if the user types '^C' (CONTROL key and C simultaneously) at the beginning of a string in PL/I-80 the operating system reboots (Section 5.2). This would result in the database being left in an open and possibly inconsistent state. With the character-at-a-time input

of VIADUCT the entry of '^C' at any stage of the command results in the database being closed (i.e. all memory resident database pages are written to disk) and the termination of VIADUCT execution.

Commands may be entered in upper case, lower case, or any combination. All function names, record names, item names, and keywords (WHERE, EQ, NE, GT, LT) are automatically converted to upper case, while the case of characters comprising item values entered by the user is preserved. Blanks are of relevance only as separators.

Record types entered in a command are recognised by their names as defined in the schema (and pseudo-subschema). A simple linear search [Knut73] of the array of record types in the pseudo-subschema is performed to obtain the array subscript of any record type entered.

The specification of an item type in either the operand or selection clauses may be done by entering the name of the record type to which the item type belongs followed by a period and the name of the item type. However, if the item name is not ambiguous, the entry of the item name will suffice. The following example using the pss generated for the user JOE of Figure 7.1 illustrates this feature:

CREATE TEACHER WHERE TEACHER.NAME EQ BROWN, AGE EQ 25<CR>

The above command is not ambiguous since the item AGE is not named in any record type other than TEACHER. The user could equally well have substituted TEACHER.AGE for AGE in the command

without affecting the interpretation by the parser. However, if the record name TEACHER qualifying the item type NAME had been omitted, the following would have occurred:

```
CREATE TEACHER WHERE NAME EQ
                          ^ *** Ambiguous item name
CREATE TEACHER WHERE
```

This example also serves to illustrate the action taken by the parser on detection of a syntax error: the appropriate error message is displayed beneath the offending symbol and the command string up to last correct symbol is displayed ready for further input from the user. As mentioned in previous chapters, the user is allowed three attempts at correcting any erroneous symbol after which, if the third attempt is still unsuccessful, help mode is automatically entered at the menu corresponding to the current position in the command string.

Note that in the above example one-symbol lookahead has been used. This is necessary when an item specification is expected by the parser since, in order to be able to interpret a symbol as either a record name or item name, the parser must check whether the next symbol is a period or not.

If the next symbol entered by the user is a period the parser checks that the preceding symbol is a valid record type (as explained previously). When the user enters another symbol the parser assumes that it corresponds to an item type within the record type and performs a linear search on the corresponding array of items to obtain the subscript of the item type.

If, however, the user does not enter a period after the first symbol of an anticipated item specification, the parser assumes that the first symbol is an unambiguous item type name. To check this, the parser must search the array of record type definitions for a matching item type name. Once such a name has been found the parser must continue its search to ensure that the same name does not exist in any other record type within the given pseudo-subschema. The position of a record or item type in its array is used by the parser in setting up the command representation for the processor modules (Section 8.4).

Search time could be reduced by sorting the record type names in the pseudo-subschema alphabetically and using a binary search [Knut73] instead of a linear search to locate a record type name. However, this method could not be used to search for an item type within a record type since the ordering of item types within a record types is significant. With the present search method the user does not experience any noticeable delay in command input.

The parser subjects any item values specified in the selection clause of a command to certain validation procedures. The type of the data item is checked by means of the appropriate PL/I-80 conversion routine. If an error is detected an '*** Invalid item type' error message is output and reentry of the item value by the user is requested. If the item is of the correct type but its value falls outside the bounds specified in the pseudo-subschema (maximum and minimum values) an error of '*** Item value out of range' is generated.

On-line help is available to the expert user at any stage in the

formulation of his command. The entry of a question mark results in the help mode screen and the menu corresponding to the point at which the question mark was entered being displayed. For example, assume that the expert user (userid JOE) enters the following:

LIST STUDENT.NAME, REGNO WHERE ACTIVITY.?

The user types a question mark because he cannot remember the name of the item type he wants as a selector. As a result, the help screen of Figure 8.1 is displayed. The user may now either continue in help mode by selecting the option he desires or may return to expert mode at the point which he left it by entering an exclamation mark ('!').

If, however, the user with userid FRED (in Figure 7.1) had attempted to enter the above command, the following would have occurred:

LIST STUDENT.NAME, REGNO WHERE ACTIVITY.

^ *** Invalid record name

LIST STUDENT.NAME, REGNO WHERE

This is because the ACTIVITY record type is not defined in the user's pseudo-subschema and so VIADUCT has no knowledge of its existence.

A semi-colon can be used to signal line continuation to the parser. The entry of a semi-colon results in VIADUCT issuing a prompt on the next line and awaiting further input. At any stage the user may abandon entry of the current command by pressing the escape key on the terminal. The termination of command entry in expert mode is signaled by the user typing a carriage return.

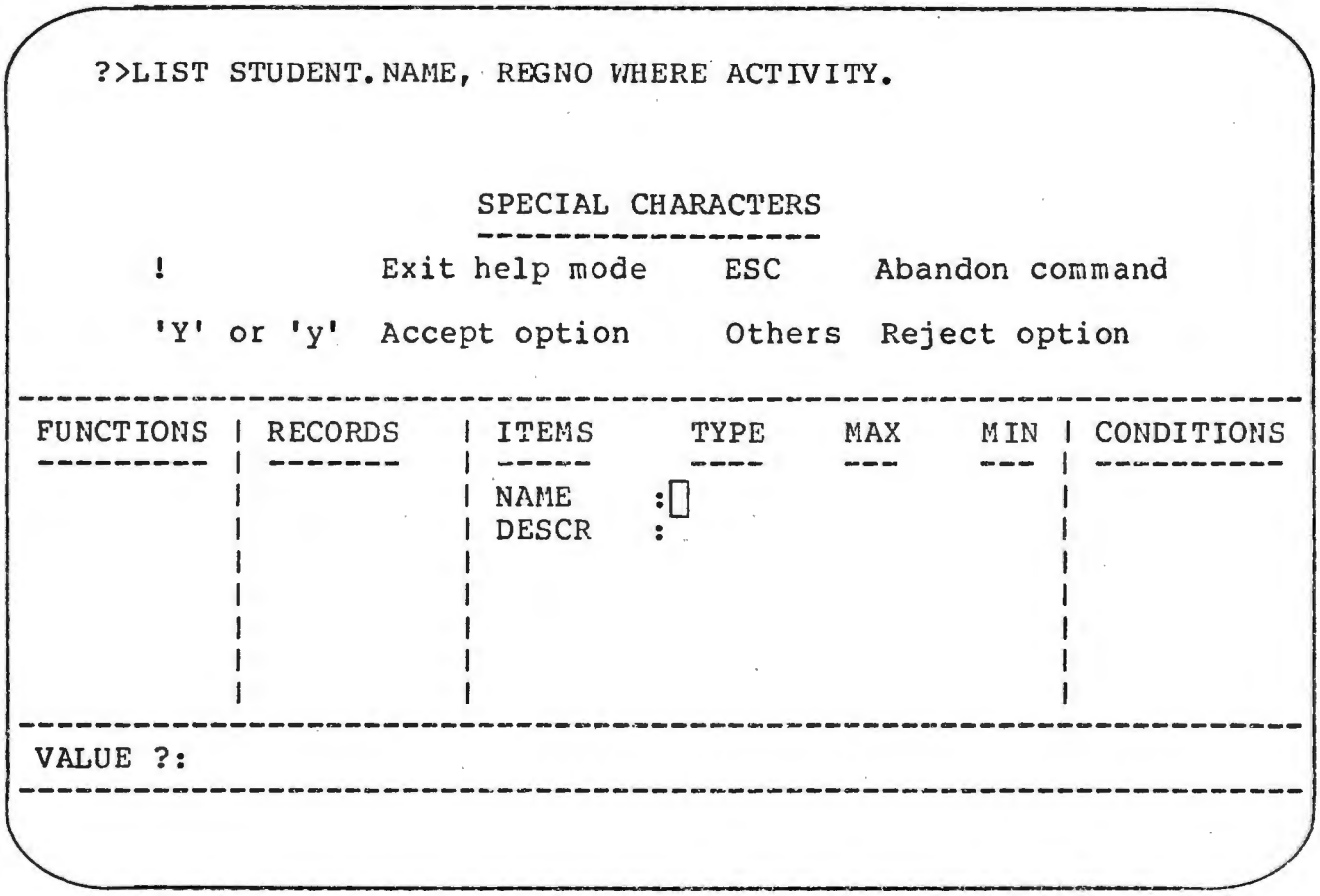


Figure 8.1. On-line help available in expert mode.

8.3 Help mode

The help mode of command entry is provided primarily for use by the inexperienced user. As a result it is designed to require a minimal number of keystrokes in order to specify a complete VIADUCT command. However, as shown in the previous section, it is also useful to the expert user who requires assistance at some stage during the formulation of a command.

It is assumed that the reader is familiar with the format of the help screen as it was introduced in Chapter 2. The example presented in that chapter covered a number of the facilities provided in help mode. A different example is given here to outline some additional features available in help mode.

For the purposes of this section it is assumed that the user has the restricted view of the database as defined by the userid FRED in the schema presented in Figure 7.1. As can be seen from Figure 8.2 only those functions defined in the user's view (see Section 7.3) are displayed in help mode.

The sample command to be entered in this example is equivalent to the following expert mode entry:

```
UPDATE SECTION.TITLE ;
        WHERE SUBJECT.NUMBER EQ 10, ;
              COURSE.NUMBER EQ 12, ;
              SECTION.NUMBER EQ 122<CR>
```

As shown in Figure 8.2, the user starts by entering 'N' next to the LIST function. Now that the cursor is next to UPDATE the

user may type a question mark to view those record types which contain item types he may update. Having checked that he may update items within SECTION records, the user types 'Y' next to the UPDATE function (the cursor has not moved).

Since the user does not wish to update items within SUBJECT or COURSE records, he types 'N' next to SUBJECT and COURSE. After typing 'Y' next to SECTION the only item that may be updated by this user, namely TITLE, is displayed. If the user wishes to find out the maximum permissible length of a TITLE, he may now enter a question mark which provides a display of the corresponding item's type and maximum and minimum values. Typing 'Y' next to the TITLE item (again the cursor has not moved) causes the cursor to move to the CONDITIONS prompt. Figure 8.3 depicts the screen displayed after the user again types 'Y'. The command displayed at the top of the help mode screen corresponds to the user's responses and is updated after each entry by the user.

The record and item names displayed in Figure 8.3 (i.e. those for which conditions can be specified) correspond to those available to the user under the LIST function (as explained in Section 8.1).

After the user has entered a value for the SUBJECT NUMBER in Figure 8.3 the cursor moves to the COURSE record option. The remaining conditions for the above example are specified in a similar manner. All item values entered are subjected to the same validation checks as in expert mode (Section 8.2).

```
?>UPDATE SECTION.TITLE WHERE SUBJECT.NUMBER EQ 10

                    SPECIAL CHARACTERS
                    ------------------
    !           Exit help mode    ESC     Abandon command

    'Y' or 'y'  Accept option     Others  Reject option

-------------------------------------------------------------------------
FUNCTIONS | RECORDS    | ITEMS      TYPE     MAX     MIN | CONDITIONS
--------- | -------    | -----      ----     ---     --- | ----------
LIST  :N  | SUBJECT :Y | NUMBER  :Y                      |  EQ :Y
UPDATE:Y  | COURSE  :  | TITLE   :N                      |  LT :
CREATE:   | SECTION :  |                                 |  GT :
EXIT  :   | STUDENT :  |                                 |  NE :
          |            |                                 |
          |            |                                 |
-------------------------------------------------------------------------
VALUE ?: 10<CR>
-------------------------------------------------------------------------
```

Figure 8.3. Specification of a selection clause in help mode. (User responses in boldface)

In help mode it is not necessary to perform a linear search to establish the existence and subscript of a record or item type. This is because the parser determines the subscript from the cursor position at which the user enters 'Y', taking into account the possibility that a number of record or item types could be present in the user's pseudo-subschema but excluded from his view under the present function.

If the user types a question mark next to any name in the record menu, the items accessible to the user in that record and under the chosen function are displayed along with their types and maximum and minimum values. A question mark may also be entered when the cursor is next to the VALUE prompt. This results in the display of the attributes of the item type whose value is to be entered.

If at any stage when the user is required to make a choice from a menu he types 'N' next to every option, the cursor simply moves back to the first option. This procedure will be repeated until the user either makes a choice or presses the escape key on the terminal.

In help mode either an 'N' response to the CONDITIONS prompt or the exhaustion of record type options for which conditions can be specified signals the termination of command entry to the parser.

8.4 Intermediate command representation

All the information contained in each VIADUCT command entered by

the user must be transferred to the appropriate processor module for the successful execution of the command. The information required can be summarized as follows: the function to be performed; the record type(s) in the operand clause together with its (their) items (if specified); if a selection clause exists, the records and items as well as the relational operators and item values comprising each expression.

The function to be performed is returned to the main module of VIADUCT by the parser. Each function is represented by an integer since PL/I-80 does not support enumeration types (as does Pascal [Jens75], for example). The integers assigned to each function were given in Section 7.2 and their PL/I-80 declaration is shown in Figure 8.4(a).

All references to record, set, and item types are via the subscripts as defined by their positions in the respective arrays declared in the pseudo-subschema. These positions are dependent on the ordering of the types in the original MDBS DDL text file. A record subscript is returned as a parameter to the main module by the parser. This subscript has the following interpretations: for the CREATE, DELETE, ADD, and REMOVE functions, it represents the record on which the function is to be performed; for the UPDATE function, it represents the record containing the item whose value is to be updated; for the LIST function it has no real significance and is arbitrarily chosen as representing the record type containing the last of the items specified in the operand clause.

The item types referenced in the operand and selection clauses

are represented in two 2-dimensional Boolean arrays: 'operand_item' and 'select_item' respectively (Figure 8.4(b)). If an item type appears in the operand clause its corresponding element in the array 'operand_item' is set to true. Likewise, each item type present in the selection clause of a command has its 'select_item' entry set to true. On each invocation of the parser the elements of these arrays are reinitialized to false.

Each relational operator in the selection clause is stored as an integer in another 2-dimensional array 'proviso'. The assignment of integers to operators is given in Figure 8.4(c).

Any item values specified in the selection clause of a command are stored in the input buffer of their record type as allocated by the main module (Section 6.4). Each input buffer is necessarily an untyped sequence of memory locations since it must be dynamically allocated at run time and cannot comply with all possible record formats. For this reason all access to buffers must be performed by so-called type templates.

The type templates are implemented in PL/I-80 as BASED variables [Digi80], one for each type to be represented. Storage for each of the variables is never ALLOCATEd so they can be used to overlay arbitrary memory locations by assigning values to their controlling pointer variables. This is allowed in PL/I since pointers are not bound to data types. Figure 8.5(a) shows the declaration of the type templates all of which are BASED at a memory location defined by the contents of the pointer 'p'.

In order to store an item value in the correct position (as

```
%REPLACE  list   BY 1,
          update BY 2,
          create BY 3,
          add    BY 4,
          remove BY 5,
          delete BY 6,
          exit   BY 7;
```

(a)

```
DCL  operand_item(1:max_records,1:max_items) BIT,
     select_item(1:max_records,1:max_items)  BIT;
```

(b)

```
%REPLACE  equal        BY 1,
          not_equal    BY 2,
          less_than    BY 3,
          greater_than BY 4;
DCL  proviso(1:max_records,1:max_items) FIXED (7);
```

(c)

Figure 8.4. PL/I-80 declarations for command representation.

defined in the original schema) within the input buffer of the corresponding record type, it is necessary to implement a form of pointer arithmetic. The element of the 'input_buffer' pointer array (Figure 8.5(b)) corresponding to the record type involved (e.g. 'input_buffer(r)') contains the starting address of that record type's input buffer. The offset of the item type whose value is to be stored in the input buffer can be calculated by summing the lengths (in bytes) of all the item types preceding it in the definition of the record type. Now it is necessary to assign to pointer variable 'p' (on which the type templates are BASED) the sum of the value of the pointer 'input_buffer(r)' and the calculated offset. This is done by redefining the input buffer as an array 'data' BASED at 'q' (Figure 8.5(a)):

```
q = input_buffer (r);
```

Using the PL/I function ADDR which returns the memory address of the variable in its argument, the pointer 'p' is set to point to the correct memory location.

```
p = ADDR ( q -> data ( offset ) );
```

It is now possible to place an item value in the correct position in the input buffer by assigning a value to the appropriate BASED data type.

From recent letters in ACM SIGPLAN Notices it would appear that the above method of pointer arithmetic has been used elsewhere for some time [Metz82, Stac82]. Although such methods are undesirable as they are contrary to the practice of strong typing, they were chosen in preference to methods based on precompilation [Scha80] which would allow strong typing to be consistently implemented within VIADUCT. Precompilation would

```
DCL  (p, q)                          PTR,
     data(0:max_record_length)       BIT (8)     BASED (q),
     alphanumeric                    CHAR        BASED (p),
     single_integer                  FIXED (7)   BASED (p),
     double_integer                  FIXED (15)  BASED (p),
     real_number                     FLOAT       BASED (p),
     boolean                         BIT         BASED (p);
```

(a)

```
DCL  input_buffer(1:max_records)     PTR,
     output_buffer(1:max_records)    PTR;
```

(b)

Figure 8.5. PL/I-80 declarations relevant to command representation.

MDBS	PL/I-80
CHAR	CHAR
BIN 1	FIXED (7)
BIN 2	FIXED (15)
INT	FIXED (15)
REAL	FLOAT
LOG	BIT

Figure 8.6. Comparison of MDBS and PL/I-80 data types.

require VIADUCT to be customized for each pseudo-subschema that was to be generated for every database schema. It was felt that this was unacceptable for reasons of ease-of-use.

Figure 8.6 shows the MDBS data types and their PL/I-80 counterparts. The only instance of incompatability found between the two sets of data types was that MDBS stores two-byte integers (INT or BIN 2) with the high-order byte first whereas PL/I-80 stores the low-order byte first in its equivalent data type (FIXED (15)). In order to maintain compatability with MDBS products VIADUCT swaps the bytes of all two-byte integers before storing them in the database and similarly after retrieving them from the database.

The following two chapters demonstrate how the output from the parser is interpreted in order to process a command entered by the user.

References

[Digi80] PL/I-80 Language Reference Manual. Digital Research Inc., Pacific Grove, California, 1980.

[Jens75] Jensen, K. and Wirth, N. **Pascal User Manual and Report**, 2nd Edition. Springer-Verlag, New York, 1975.

[Knut73] Knuth, D.E. **The Art of Computer Programming**, Vol 3, Sorting and Searching. Addison-Wesley, Reading, Massachusetts, 1973.

[Metz82] Metz, S.J. Correspondence. **ACM SIGPLAN Notices,** Vol 17, No 2 (February 1982).

[Scha80] Schach, S.R. A portable trace for the Pascal heap. **Software - Practice and Experience,** Vol 10, No 6 (June 1980), pp 421-426.

[Stac82] Stachour, P.D. Correspondence. **ACM SIGPLAN Notices,** Vol 17, No 3 (March 1982).

CHAPTER 9: **PATH FINDING AND TRAVERSAL ALGORITHMS**

In this chapter both the restricted and generalized path finding and database traversal algorithms are described. These algorithms are central to the operation of the processor modules described in the following chapter.

9.1 **Introduction**

The restricted path finding algorithm is used by all the processor functions to find a path from the SYSTEM record type to any other single record type in the schema. The restricted traversal algorithm is then used by the create, add, remove, and delete functions to isolate a single record occurrence at the end of a path found by the restricted path finding algorithm. The two generalized algorithms are used by the list and update functions to find a path connecting any two record types in the schema, since these functions can be specified with arbitrary combinations of record and item types in a command. All six functions are described in detail in Chapter 10.

The traversal algorithms utilize various DMS functions to find record occurrences (such as 'fmsk') and update currency indicators (such as 'som'). In order to aid the reader in following these traversal algorithms, the PL/I-80 declaration of the MDBS DMS function names and the source of each acronym as defined in the MDBS DMS manual [MDBS79] is given in Appendix B. For further explanation of these functions the reader is referred to Section 4.3. The parameters required for each DMS function

are given in Appendix C. Each parameter is in fact either the address where a type name is stored or the address of a buffer containing record data or item data. The underlined parameters in Appendix C represent values that are returned by the DMS. A description of the interface to the database is deferred until Section 10.2.

Before describing the path finding and traversal algorithms it is necessary to introduce the data structures used to store all the possible paths found in any one invocation of the path finding routine. These paths must be stored so that after the termination of the algorithm one of the paths may be selected for traversal. The PL/I-80 declaration of the path data structures are given in Figure 9.1. The paths are defined as a two-dimensional array allowing 'max_paths' paths each of length 'max_path_length'. Each element of a stored path refers either to a record or a set subscript, the interpretation being made as follows:

```
For path(i,j), i = 1,2,...,number_of_paths,
               j = 1,2,...,path_length(i),
    If mod(j,2) = 0 then path(i,j) is a set subscript
                    else path(i,j) is a record subscript
```

that is, alternate elements refer to sets and records. This is of course the 'natural' way to describe a path through a network database.

A 'work_path' is used to store path information while the schema is being traversed. The maximum number of alternate paths that can accomodated in any one search is declared as a constant (which can be modified as required). The maximum path length is

```
        /* The path constants */
%REPLACE
        max_paths       BY   6,
        max_path_length BY  13;


      /* The path vector declarations */
DCL     (work_path(1:max_path_length),
        path(1:max_paths,1:max_path_length),
        path_length(1:max_paths),
        number_of_paths)                FIXED (7) STATIC EXTERNAL;

     /* The arrays used to flag paths */
DCL   (record_eligible(1:max_records),
      set_eligible(1:max_sets))          BIT STATIC EXTERNAL;
```

Figure 9.1. PL/I-80 path data structure declarations.

defined to be one less than twice the maximum number of record types in the schema. This caters for the worst case in which every record type in the schema is on the path.

9.2 The restricted path finding algorithm

The restricted path finding algorithm ('find_res_path') is given in Figure 9.2. Before the algorithm is invoked all records and sets are assigned to be eligible for selection and the path length is initialized to zero. Given a single record type (the source) which is passed to the particular function module by the main module, the algorithm finds all paths connecting that record type to the SYSTEM record type that can be found moving only up the schema hierarchy, that is, from set member to set owner. To do this, the rows and columns of the incidence matrix (Section 7.2) are traversed recursively, always choosing set types in which the current record type (as defined by the parameter 'source') is a member (i.e. the corresponding incidence matrix entry is negative), and record types which are owners (i.e. the incidence matrix entry is positive) of the previously chosen set types.

While the schema is being searched the candidate path under consideration is kept in the 'work_path'. Once a path to SYSTEM has been found, the work path is copied to the next unused 'path' data structure and the search continues. The algorithm terminates when all paths have been found or the maximum number of alternate paths has been exceeded (a situation checked by the routine 'store_path').

```
recursive procedure find_res_path (source, path_length)
                    returns ( boolean );

declare   source,                          /* record type */
          s,                               /* set subcript */
          r,                              /* record subscript */
          path_length          integer,
          found                boolean;

begin

  if source = system
    then found := true;
    else
      begin
        path_length := path_length + 1;
        work_path[path_length] := source;
        record_eligible[source] := false;
        path_length := path_length + 1;
        for s := 1 to number_of_sets do
          begin
            if incidence_matrix[s,source] < 0 and set_eligible[s]
              then
                begin
                  set_eligible[s] := false;
                  work_path[path_length] := s;
                  for r = 1 to number_of_records do
                    if incidence_matrix[s,r] > 0 and record_eligible[r]
                      then if find_res_path (r, path_length)
                             then store_path (path_length);
                end
                set_eligible[s] := true;
          end
        record_eligible[source] := true;
        found := false;
      end
  return found;

end find_res_path;
```

Figure 9.2. The restricted path finding algorithm.

Once all the paths between a record type and the SYSTEM record type have been found and stored, it is necessary to select one for traversal (unless only one path exists). A path of length three signifies that only a singular set is involved (as the path must be of the form record-set-SYSTEM). If only one path is of length three then this path is chosen. However, if there is more than one such path or all paths are of length greater than three, the user is asked which path is to be taken (the reader is referred to the example of Section 2.2). If the user prefers not to make a selection, the shortest path is chosen; if there is more than one such shortest path, one is chosen arbitrarily.

9.3 **The restricted path traversal algorithm**

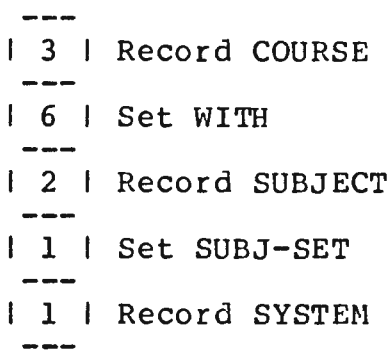
The restricted path traversal algorithm is used in conjunction with the restricted path finding algorithm when a particular occurrence of the record type at the end of a path must be identified. In order to do this, the selected path is scanned in reverse order, that is, from SYSTEM to the record type involved. Since a single record occurrence is to be isolated at the end of a linear hierarchical path, an occurrence must be selected at each level (or in each set) in the hierarchy. If the user has not specified sufficient information in the selection clause at command entry to do this, he is now prompted for the identifying information. Any superfluous information specified in the selection clause regarding items not on the chosen path is ignored by the path traversal algorithm.

The key item is always used to identify a record occurrence on

the path if the set in which it participates is sorted. If the key item value was not input at command entry it is now solicited from the user. If duplicate key values have been allowed and are present in a sorted set or the set is not sorted, the user must identify which record occurrence is to be selected. In this case VIADUCT reviews the occurrences comprising the set by outputting the contents of each in turn to the console and then asking the user to choose the one which is to be used. As this process can possibly be very time consuming, it is obviously preferable for the user to select paths containing sets which are sorted or to define schemata where every set is sorted.

On each occasion that a record occurrence is selected at a given level in the hierarchical path, the currency indicator of the following set on the path in which the record type participates as an owner is updated explicitly (using 'som'). A simple example taken from the education database will serve to illustrate the operation of the traversal algorithm.

Suppose that the user wishes to perform some function on a particular COURSE record occurrence. The path found by the restricted path finding algorithm is shown in Figure 9.3(a). The integers in the path array correspond to the record and set subscripts as defined in the incidence matrix for the education database (Figure 7.2). Figure 9.3(b) shows the sequence of database calls executed to find a particular COURSE occurrence. The key values for SUBJECT NUMBER (10) and COURSE NUMBER (12) would either be retrieved from the selection clause of the command or be obtained from the user during the traversal.



(a)

```
error_status = dms ( fmsk, SUBJ-SET, 10, null );
if error_status ~= 0 then exit;
error_status = dms ( som, WITH, SUBJ-SET, null );
if error_status ~= 0 then exit;
error_status = dms ( fmsk, WITH, 12, null );
if error_status ~= 0 then exit;
```

(b)

Figure 9.3. Path finding and traversal to a COURSE record.

9.4 The extended path finding algorithm

In contrast to the restricted path finding algorithm which finds paths between one record type and SYSTEM, this path finding algorithm ('find_ext_path') finds paths from any one record type (the 'source') to any other record type (the destination, 'dest') in the schema. In order to do this, the algorithm must perform an exhauastive search of the schema to find such paths since, unlike in the restricted form, it is not known which of the two record types is at a higher level in the schema. Thus a record or set is eligible for selection simply if it has a non-zero entry in the incidence matrix. The algorithm terminates when either the destination record type is reached or the maximum number of alternate paths is exceeded. The extended path finding algorithm is given in Figure 9.4.

The Boolean arrays 'record_eligible' and 'set_eligible' in Figure 9.4 are used to ensure that cycles are not generated. The fact that cyclic paths cannot be followed by VIADUCT means that queries which involve the traversal of recursive sets cannot be processed. An example of the problems associated with cyclic paths is given in Section 9.5.

The paths which must be found by 'find_ext_path' are those connecting all record types referenced either explicitly (by name) or implicitly (by item) in both the operand and selection clauses of the command.

One record type (that passed to the function module by the main module) is arbitrarily chosen as the source record type. All

```
recursive procedure find_ext_path (source, dest, path_length)
                    returns ( boolean );

declare   source,                    /* source record type */
          dest,                      /* destination record type */
          s,                         /* set subcript */
          r,                        /* record subscript */
          path_length          integer,
          found                boolean;

begin

  path_length := path_length + 1;
  work_path[path_length] := source;
  if source = dest
    then found := true;
    else
      begin
        record_eligible[source] := false;
        path_length := path_length + 1;
        for s := 1 to number_of_sets do
          begin
            if incidence_matrix[s,source] ≠ 0 and set_eligible[s]
              then
                begin
                  set_eligible[s] := false;
                  work_path[path_length] := s;
                  for r := 2 to number_of_records do
                    if incidence_matrix[s,r] ≠ 0 and record_eligible[r]
                      then if find_ext_path (r, dest, path_length)
                             then store_path (path_length);
                end
                set_eligible[s] := true;
          end
        record_eligible[source] := true;
        found := false;
      end
  return found;

end find_ext_path;
```

Figure 9.4. The extended path finding algorithm.

other record types referenced in the command are considered destination record types. The algorithm is invoked a number of times in order to find the paths between the source and each destination record type. As each path is selected the participating record and set types are flagged as being on the final path. Each of the separate paths from the source to one destination will be linear by construction; however, when combined into the overall final 'path' the resulting structure may form a tree structure rather than a linear list.

On each invocation of the algorithm it is possible that alternate paths between the two record types exist. In its present form VIADUCT asks the user to specify which path is to be taken in each case (see Figure 2.7). As in the restricted form, the shortest path is used if the user prefers not to make a choice. At a later stage the facility to perform path analysis with reference to the other record types specified in the command will be added to VIADUCT.

When paths between all the record types referenced in a command have been found an entry point to the final path is sought. If none of the record types involved is a member of a singular set, the path finding algorithm ('find_res_path') of Section 9.2 (Figure 9.2) must be invoked to find a path to the SYSTEM record type. As an example, if the user entered the command

LIST COURSE.TITLE<CR>

a path from COURSE to SYSTEM would be found and all COURSEs for all SUBJECTs would be listed. On the other hand, if more than one record type on the final path participates in a singular set, the best of these must be chosen.

This choice of entry point is critical to the amount of processing required to execute a command. This is because the complete path must be traversed for every record occurrence found in the entry set. The best possible entry set is one that is sorted (with duplicates forbidden) and in which a test for equality on the key item is to be performed. The choice by VIADUCT of such a set will result in a considerable saving in database processing time (since the DMS function 'fmsk' can be used). If the key item of a sorted set cannot be used, the next best entry set is one in which some condition has been placed on certain item values in the member record occurrences. This will still result in a substantial saving in the number of occurrences retrieved from the database in order to execute a command. Thus VIADUCT attempts to find an entry point that will result in the minimal amount of database processing.

To illustrate some of the complexities involved in path analysis, an example is taken from the education database. Suppose the following command is entered:

LIST TEACHER.NAME, SECTION.NAME, STUDENT.NAME<CR>

For convenience the relevant portion of the education database is reproduced in Figure 9.5. If TEACHER is arbitrarily chosen by VIADUCT as the source record type, then SECTION and STUDENT would be the destination record types. The path finding algorithm would find two paths connecting TEACHER and SECTION, and two connecting TEACHER and STUDENT. Refering to the incidence matrix of the education database (Figure 7.2), these paths would be represented as shown in Figure 9.6.

Since the above query has three different meaningful

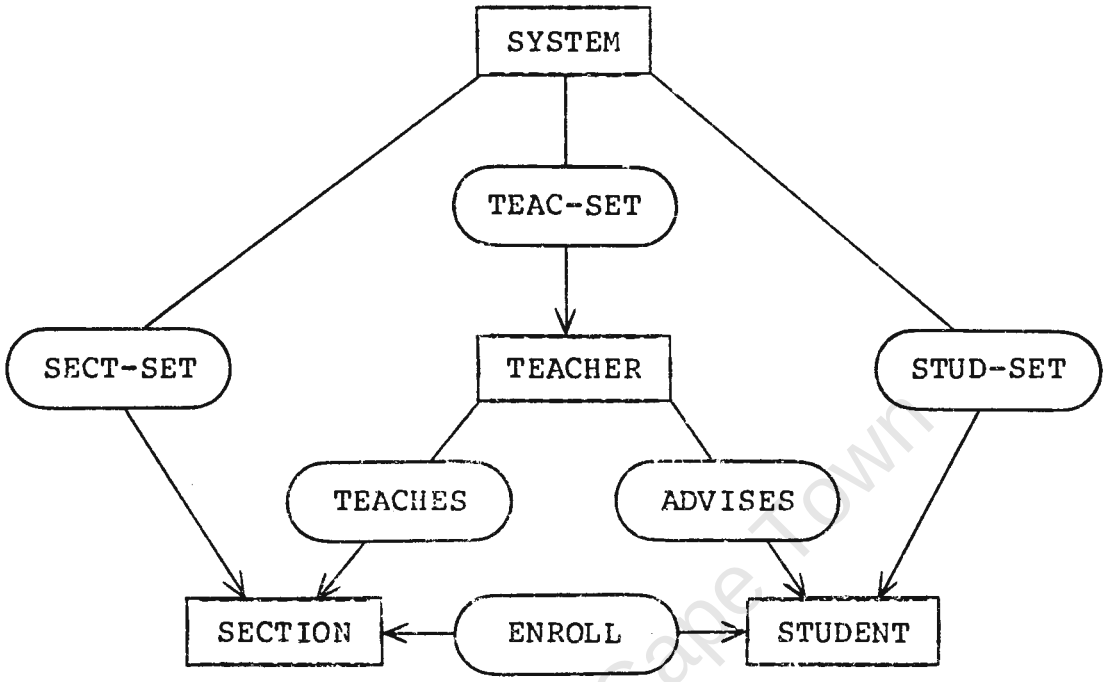


Figure 9.5. Portion of the education database.

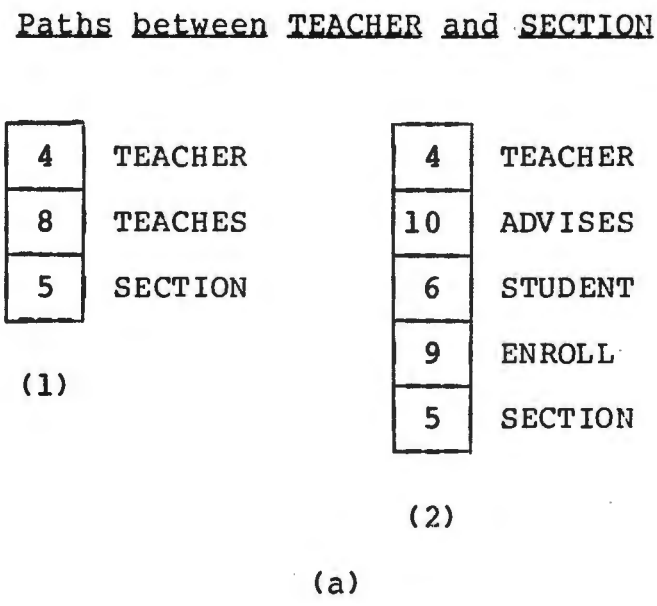


Figure 9.6. Alternate paths found for a particular query.

interpretations VIADUCT cannot choose a combination of paths automatically. These interpretations are listed below with the combination of paths from Figure 9.6 that would have to be selected by the user to ensure the correct interpretation of the query:

(i) TEACHERs along with the SECTIONs they teach and the STUDENTs they advise. (Paths (1a) and (1b))

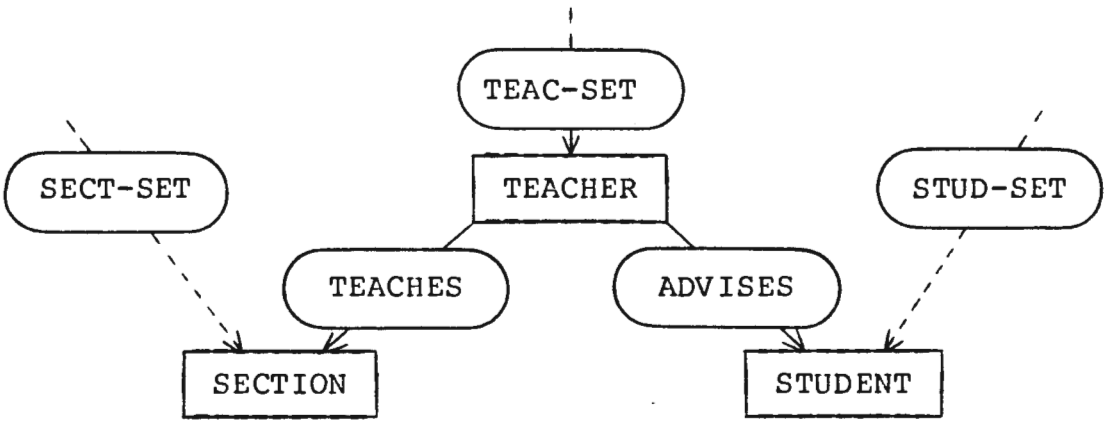
(ii) TEACHERs along with the SECTIONs they teach and the STUDENTs enrolled in those SECTIONs. (Paths (1a) and (2b))

(iii) TEACHERs along with the STUDENTs they advise and the SECTIONs in which those STUDENTs are enrolled. (Paths (2a) and (1b))

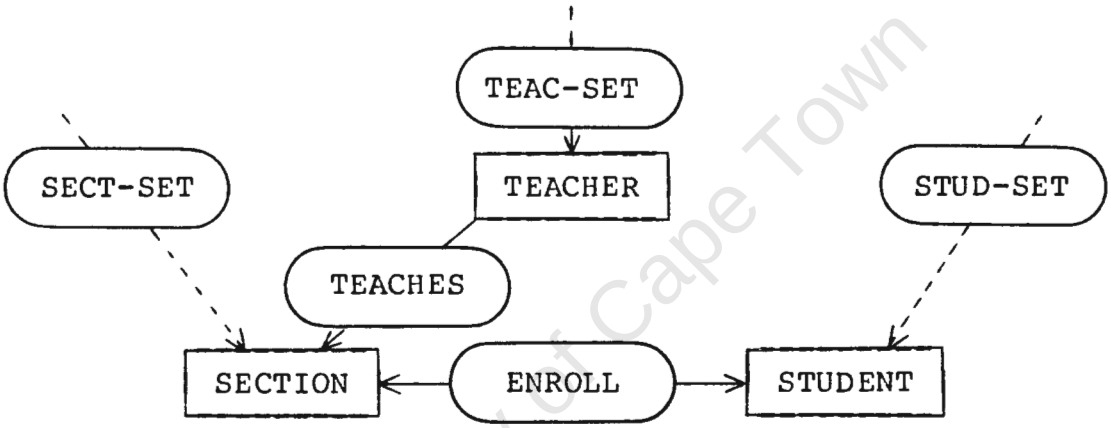
The combination of paths (2a) and (2b) is cyclic and as such would be prohibited by VIADUCT.

In addition to the above complexities, the final path deduced in each of the above three cases contains three possible entry points (Figure 9.7). In all cases two of the entry points result in linear paths while one results in a non-linear path. Irrespective of which entry point is chosen by VIADUCT for any one of the above three paths, the result in each case must be consistent, that is, the answer obtained via each of the three entry points must be identical. Assuming that the user chose the combination of paths (1a) and (2b) in Figure 9.6 (corresponding to interpretation (ii) above and Figure 9.7(ii)), the three paths resulting from the choice of different entry points are shown in Figure 9.8.

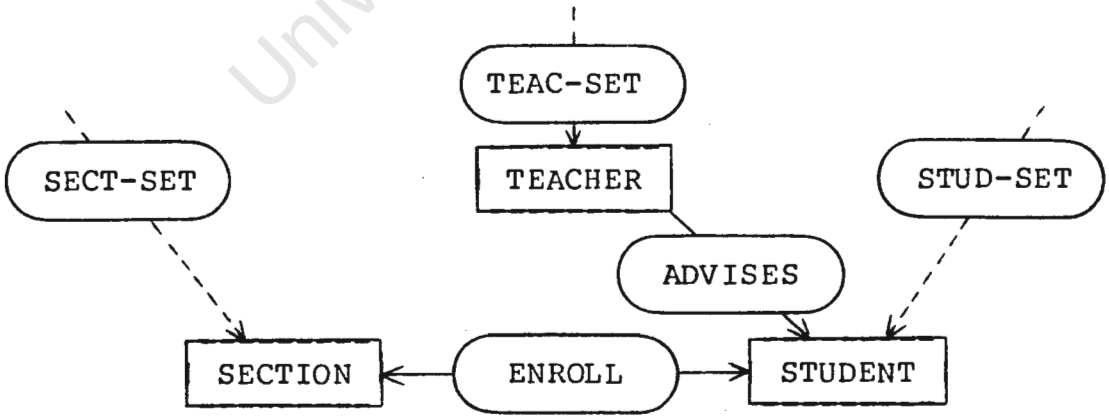
Since no conditions favour the choice of any particular entry point, one can be selected arbitrarily by VIADUCT. For the two



(i)

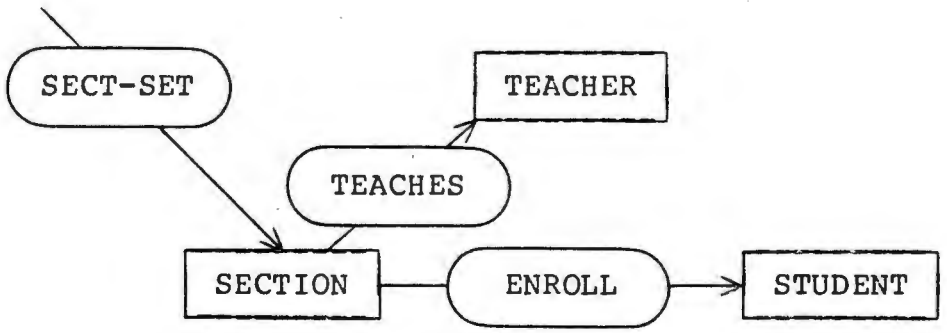


(ii)

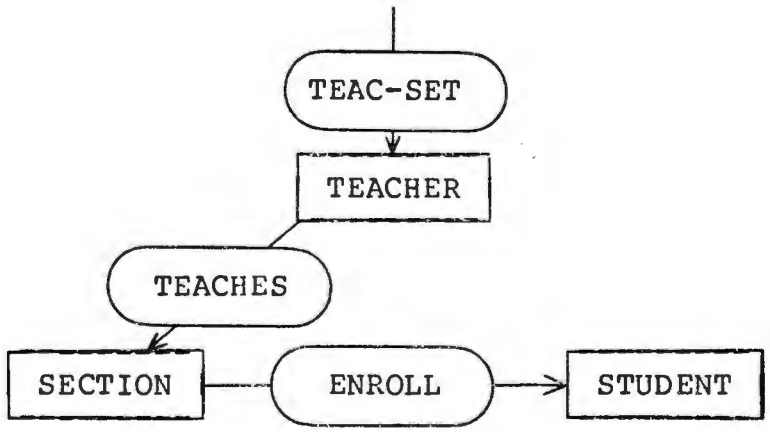


(iii)

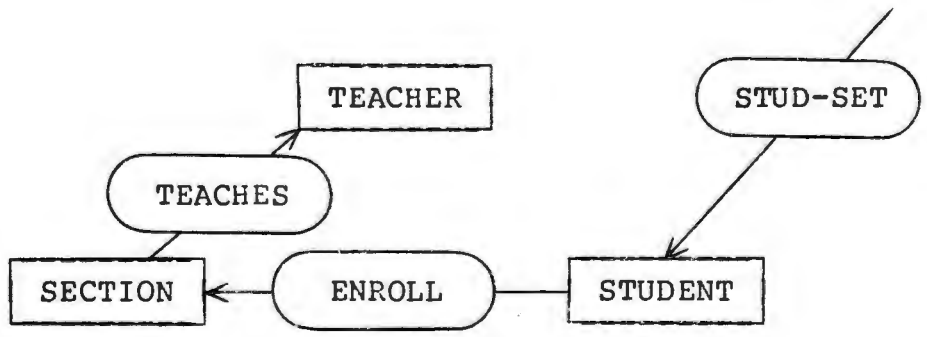
Figure 9.7. Three different paths connecting three record types. (The broken lines designate the possible entry sets.)



(a)



(b)



(c)

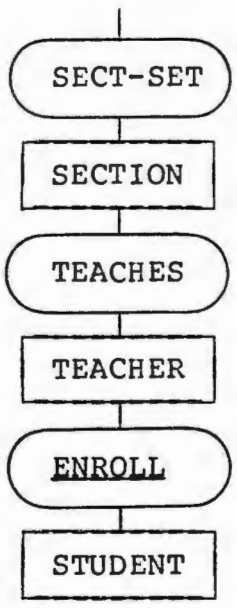
Figure 9.8. The dependence of path on choice of entry point. (Arrows now show the direction of path traversal.)

entry points which result in linear paths (Figure 9.8(b) and (c)) it can be seen that the subordinate record types (those further along the path) are dependent on the record types which precede them on the path; however, for the entry point producing the non-linear path (Figure 9.8(a)) this dependency is not immediately apparent but must nevertheless be enforced during path traversal.

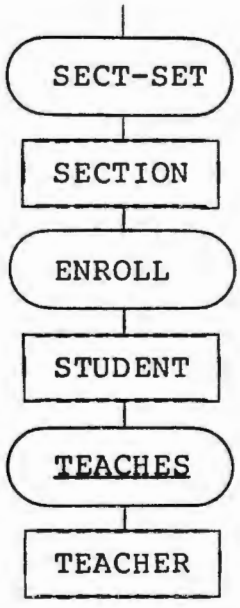
Unfortunately, although conventional recursive algorithms used for path finding traverse a path in the correct sequence, the traversal is performed on a depth-first basis. That is, all satisfying occurrences on a single branch of the path are retrieved before another branch is traversed. This method would be satisfactory for linear paths but for non-linear paths this would result in the branches being interpreted as logical OR connectives. However, the above non-linear path must be traversed with AND connectives between the branches to interpret correctly the dependency between the participating record types.

In order to traverse successfully a non-linear query graph with AND connectives, the following procedure is followed. The non-linear path is converted to an equivalent linear path by traversing the incidence matrix to obtain the correct sequence of records and sets for path traversal. This linear sequence is stored in a one-dimensional path array. For the non-linear path of Figure 9.8 (a) the two possible resultant linear paths are given in Figure 9.9 (a)(i) and (ii).

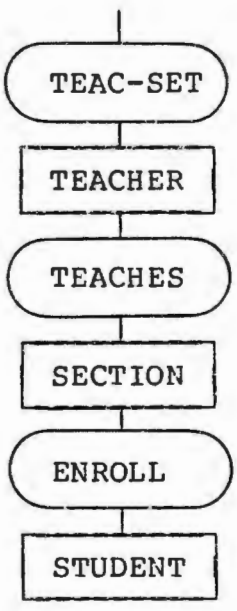
These two paths do not correspond to either of the naturally linear paths (Figure 9.9(b) and (c)) nor even to any paths of the other interpretations of the original query. This is because



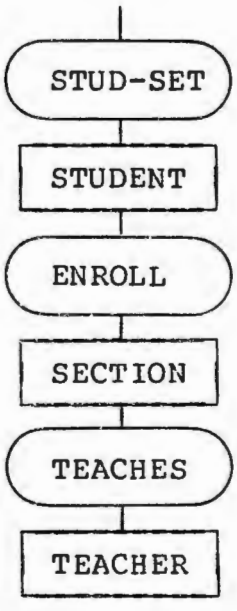
(a)(i)



(a)(ii)



(b)



(c)

Figure 9.9. Linear paths corresponding to the paths of Figure 9.8.

whereas in a natural linear path sets are preceded and followed by their participating record types, this is not necessarily true in a converted non-linear path. For example, as a result of linearization the underlined set names in Figure 9.9(a)(i) and (ii) do not participate in any relationship with their preceding record types. The procedure required to traverse such a path (and in fact used to traverse all paths) is described in the next section.

9.5 The extended path traversal algorithm

There are four methods by which any set on the path can be traversed. If a test for equality is to be performed on the key item of a sorted set, the first occurrence of a record type with that key value can be found directly ('fmsk'). If duplicates are allowed in the sorted set, these can be retrieved using the routine to find duplicate keys (Section 10.3) until no more duplicates are found. However, if the set is not sorted or an equality test is not to be performed on the key item of a sorted set, all the occurrences in the set must be traversed (using, for example, 'ffm' and 'fnm') . The fourth case requires no traversal of a set since only one record occurrence is available. This is when a 1:N set is being traversed in the direction of its owner record type, an N:1 set is being traversed in the direction of its member record type, or a 1:1 set is being traversed in either direction.

As each satisfying record occurrence is found, its data is transferred to its output buffer. A satisfying record

occurrence is one in which the values of all item types conform to any restrictions placed on them by the user. Thus logical AND connectives are assumed between the conditions specified for item types within a single record type. If the record occurrence passes these conditions, the path must be navigated by updating currency indicators and then traversing sets further along the path.

In VIADUCT, the traversal algorithm is implemented by means of two mutually recursive procedures, 'traverse' (Figure 9.10) and 'navigate' (Figure 9.11). The 'traverse' procedure is used to traverse the occurrences of a set, exploiting any restrictions placed on the key item in order to speed up the traversal. It also determines in which direction the set is to be traversed by consulting information obtained from the user as to whether ascending or descending order is desired. Once a record occurrence has been isolated, the 'navigate' procedure is called. This procedure applies the restrictions which have been placed by the user on record occurrences retrieved and returns directly to 'traverse' if these conditions are not met. Otherwise all relevant currency indicators are updated and the next set on the path is passed to the 'traverse' procedure.

A non-linear path which has been converted to a linear path is traversed correctly by using the information regarding those sets which have been flagged as being on the path (Section 9.4). When each record occurrence found by 'traverse' passes the conditions placed on it, 'navigate' updates the currency indicators for all sets in which that record type participates and which are on the path. This ensures that when a branch of

```
recursive procedure traverse ( ss, many_occurrences, i );

declare (ss, r, i, first, next, find)   integer,
        (many_occurrences, successful)  boolean,
        key_ptr                         pointer,
        order                           bit (3);

begin

r := path[i];  i := i + 1;
if incidence_matrix[ss,r] > 0
    then
        begin
            if original_order[ss]
                then begin  first := ffo;  next := fno;  end
                else begin  first := flo;  next := fpo;  end
            find  := fosk;   order := set[ss].owner_order;
        end
    else
        begin
            if original_order[ss]
                then begin  first := ffm;  next := fnm;  end
                else begin  first := flm;  next := fpm;  end
            find  := fmsk;   order := set[ss].member_order;
        end
if key_is_to_be_tested ( r, ss, order, key_ptr )
  then
    begin
      if dms (find,addr(set_name[ss]),key_ptr,null)
        then
          begin
            navigate ( ss, r, i );
            if order = sort_duplicates
              then
                while find_next_duplicate( ss, r ) do
                  navigate ( ss, r, i );
          end
    end
  else
    begin
      error_status := dms (first,null,addr(set_name[ss]),null);
      if many_occurrences
        then
          while error_status = 0 do
            navigate ( ss, r, i );
            error_status := dms (next,null,addr(set_name[ss]),null);
          end
        else
          if error_status = 0
            then navigate ( ss, r, i );
    end

end traverse;
```

Figure 9.10. Extended path traversal algorithm ('traverse').

```
recursive procedure navigate ( ss, r, i );

declare (ss, r, s, i, function)          integer,
        (many_occurrences, successful)   boolean;

begin

if items_selected_in ( r )
  then
    begin
      if incidence_matrix[ss,r] > 0
        then function := geto;
        else function := getm;
      error_status := dms(function, addr(set_name[ss]),null,null);
      if conditions_passed ( r )
        then transfer_items ( r );
        else return;
    end

for s := 1 to number_of_sets do
  if s ≠ ss and incidence_matrix[s,r] ≠ 0 and set_on_path[s]
    then
      begin
        if incidence_matrix[s,r] > 0 /* owner */
          then if incidence_matrix[ss,r] > 0
                 then function := soo;
                 else function := som;
          else if incidence_matrix[ss,r] > 0
                 then function := smo;
                 else function := smm;
        error_status := dms (function,addr(set_name[s]),
                                      addr(set_name[ss]),null);
      end

s := path[i];
if end_of_path
  then perform_function;
  else
    begin
      many_occurrences := set_type[s] = n_to_m;
      if incidence_matrix[s,r] > 0
        then
         many_occurrences := many_occurrences or set_type[s] = one_to_n
        else
         many_occurrences := many_occurrences or set_type[s] = n_to_one
      traverse ( s, many_occurrences, i+1 );
    end

end navigate;
```

Figure 9.11. Extended path traversal algorithm ('navigate').

the query graph which has been moved further down the path as a result of linearization is to be traversed, the currency indicators for the first set of that branch are correctly assigned.

In the case of the example introduced in the previous section, suppose that path (a)(i) of Figure 9.9 is derived by VIADUCT as a result of traversing the incidence matrix. As each satisfying SECTION occurrence is found, it is assigned to be the current member of TEACHES and the current owner of ENROLL. When a satisfying TEACHER occurrence is found no currency indicators are updated since none of the sets in which it participates (apart from the set TEACHES which is currently being traversed) is on the path. Whenever the set ENROLL is to be traversed the required SECTION occurrence is already the current owner of the set so the traversal proceeds correctly. The code skeleton executed to satisfy the above query is given in Figure 9.12. It can be similarly shown that path (a)(ii) of Figure 9.9 gives the correct output.

To conclude this section an example of the problems associated with permitting cyclic queries, and hence recursive sets, is given. Returning to the MDBS DDL of Figure 4.2, assume that the user enters the following command:

LIST NAME WHERE NAME EQ JONES, DEPENDS GT 2<CR>

First it must be established to which set (STAFF or MANAGES) the query refers. If it is assumed that it refers to the set MANAGES and that MANAGES is in fact defined to be N:M, then the query is still ambiguous: it could refer to those employees who manage Jones, or to those who are managed by Jones. The

```
error_status = dms ( ffm, null, SECT-SET, null );
while error_status = 0 do
  if conditions_passed ( SECTION )
    then
      transfer_items ( SECTION );
      error_status = dms ( smm, TEACHES, SECT-SET, null );
      error_status = dms ( som, ENROLL, SECT-SET, null );
      error_status = dms ( geto, TEACHES, null, null );
      if conditions_passed ( TEACHER )
        then
          transfer_items ( TEACHER );
          error_status = dms ( ffm, null, ENROLL, null );
          while error_status = 0 do
            error_status = dms ( getm, ENROLL, null, null );
            if conditions_passed ( STUDENT )
              then
                transfer_items ( STUDENT );
                list_items;
            error_status = dms ( fnm, null, ENROLL, null );
          end
  error_status = dms ( fnm, null, SECT-SET, null );
end
```

Figure 9.12. Code skeleton for traversing path (a)(i) of Figure 9.9.

resulting complexity of command specification is contrary to the stated design goals of VIADUCT. For this reason cyclic paths (such as the combination of paths (2a) and (2b) in Figure 9.6), and hence recursive sets, are excluded from consideration in VIADUCT.

Reference

[MDBS79] MDBS.DMS User's Manual. Micro Data Base Systems Inc., Lafayette, Indiana, 1979.

CHAPTER 10: THE PROCESSOR

This chapter describes those modules responsible for the execution of a VIADUCT command. Apart from describing the function modules themselves this involves discussing the various utility routines used in VIADUCT.

10.1 Overview

As outlined in Section 8.4, the main module calls the processor module corresponding to the function returned to it by the parser. Thus there are six function modules: one for each of the executable functions. The function modules and their associated common modules (see below) total some 3000 lines of PL/I.

There are a number of routines which are required by all of the processor modules. These routines are grouped together in a common module which is described before any of the function modules (Section 10.3).

The six function modules can be divided into two groups: those that operate on record types (ADD, CREATE, REMOVE, and DELETE) (Section 10.4), and those that operate on item types (LIST and UPDATE) (Section 10.5). As a result of the similarities among the functions within a group, there are a number of routines that are common to each group of functions. These are discussed within the relevant sections before each function is described in detail. The relationship between all the modules comprising the

processor is depicted in Figure 10.1. (The path finding and traversal algorithms described in the previous chapter are contained in the appropriate common modules.)

Essential to the operation of VIADUCT is the assembly language interface to the database system. This routine and its parameter passing convention are discussed in the next section.

A number of the examples in this chapter are derived from the education database used in previous chapters. In addition, sections of a database schema to be described in detail in Chapter 11 are introduced to illustrate some of the more advanced features of VIADUCT.

10.2 The database interface

This section describes the assembly language interface to the MDBS DMS routines and the calling conventions employed. This interface was written before a PL/I-80 interface to the database system was available; however, when one did become available it was found to be unsuitable for the purposes of VIADUCT. This was because the calling convention of the interface required that DML commands be passed to the database as character strings passed by value [MDBS80a]. Because of the generalized nature of VIADUCT, this convention was found to be unacceptable.

The assembly language interface consists of 30 lines of Intel 8080 assembler code. It is assembled using the relocating macro assembler supplied with PL/I-80 [Digi80] and linked to the

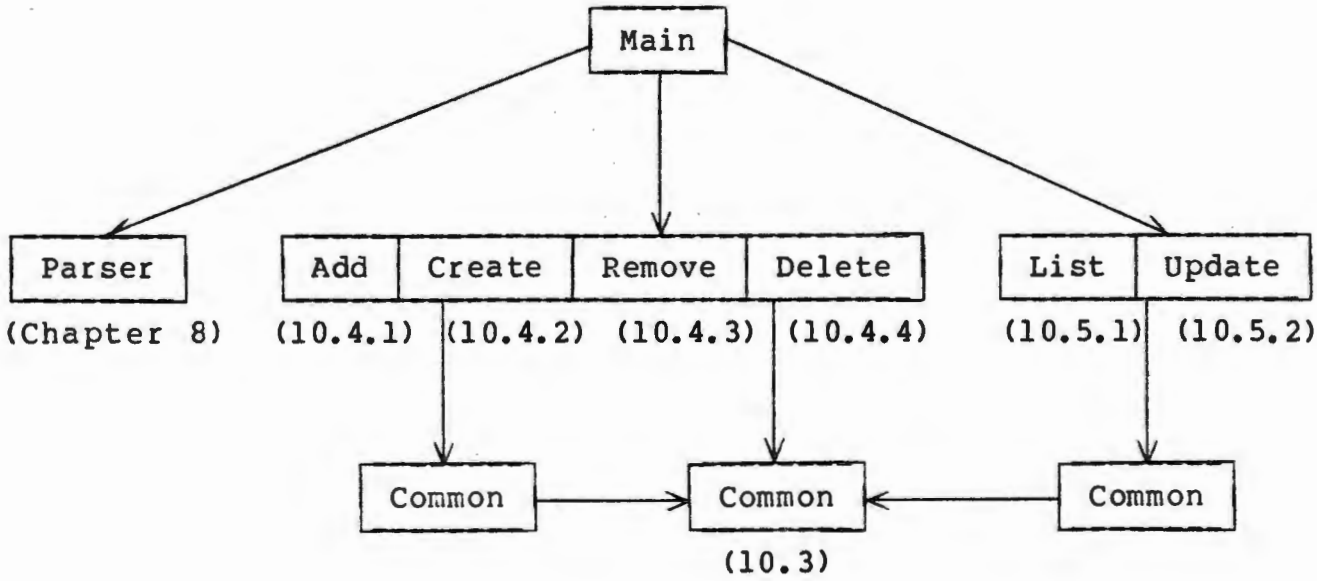


Figure 10.1. Calling relationship between processor modules. (The section numbers in which each module is described are shown in parentheses.)

compiled PL/I-80 modules.

The assembly language routine is called from the PL/I modules as a conventional externally defined function. Four parameters are passed to the routine. The parameters are: the DMS function (an integer) and three pointer parameters referencing various items of information required by the DMS routines. The use of pointers allows parameters of differing types to be passed to the DMS. This is necessary since the DMS requires the same parameter to be used in different functions for passing the addresses of set types, record types or data areas.

After a database call the routine returns the error status, an integer whose value is associated with a particular error condition. An error status of zero signifies that no error has occurred, while a status of minus one means that the end of a set has been encountered. The list of errors and possible causes is given in the MDBS DMS manual [MDBS79].

The assembly language routine is responsible for interpreting the PL/I-80 parameter passing conventions [Digi80] and assigning the correct addresses to the Intel 8080 registers as defined in the assembly language interface manual [MDBS80b] before performing a DMS call. For functions requiring information to be retrieved from the database the DMS returns the address of an internal buffer (subsequently referred to as the DMS buffer) to the assembly language routine. In these instances the interface returns this address to the PL/I module as the third pointer parameter. The sample command executions presented in this chapter use the MDBS DMS function acronyms (Appendix B) and

parameters listed in Appendix C.

10.3 Common routines

In addition to the restricted path finding algorithm described in Section 9.2 there are a number of utility routines that are common to all six functions modules. These routines are responsible for the input, output, and comparison of item values as well as the finding of record occurrences with duplicate keys.

The 'item_input' routine is used to prompt the user for an item value which is required for command execution but was not specified at the time of command entry (the corresponding element of the array 'select_item' (Section 8.4) contains the value false). The item value entered by the user is subjected to the same validation prccedures (to check data integrity) as enforced in the parser (Section 8.2) by calling a routine which is common to the parser and processor modules. As in the parser the value is stored in the input buffer of the corresponding record type (Section 8.4).

The 'item_input' routine takes appropriate action if the item value to be input is in fact a derived item (see Section 10.4.3) or a repeating item. In the case of a repeating item the correct number of instances of the item type are obtained from the user. If the number of repetitions is dependent on another item within the record type, this value must already have been input (Section 4.2) and so can be used to control the number of prompts to the user.

The 'item_output' routine uses type templates (Section 8.4) to retrieve the value to be output from the correct position in the output buffer of the corresponding record type. The value is output either to the console, a disk file, or the printer, depending on which has been selected (see Section 10.5.2). Derived items do not require special treatment and the correct number of values are output for any repeating item.

The 'compare' routine compares the value of an item type stored in an input or output buffer with that stored in the same position in the DMS buffer. The comparison to be performed (EQ, NE, LT, or GT) is stored as an integer in the array 'proviso' (Section 8.4). A type template is used only for comparisons between real numbers since comparisons between items of other types make use of the fact that these can be compared byte by byte from left to right. The Boolean result of the comparison is returned to the calling procedure.

The 'find_next_duplicate' routine retrieves the next record occurrence of a set declared with duplicate key values permitted. Before this routine is called a key value match has already been found and the value is in the correct output buffer. Since the set is sorted, the next record occurrence will have a key value either equal to or greater than the current key value. 'Find_next_duplicate' returns a Boolean result indicating whether another duplicate key exists or not.

10.4 Add, create, remove, and delete

10.4.1 The add function

The add function is used to add a record occurrence which is already stored in the database to a number of set occurrences in which it does not currently participate but is permitted to do so. The set types considered are 1:N sets of which the operand record type is a member, N:1 sets of which it is an owner (logically equivalent to the first case), and N:M sets of which it is either an owner or member. From a practical viewpoint this last case has been implemented as a combination of the previous two; strictly speaking N:M sets have neither an owner nor member since both record types participate in the set on an equal basis.

The add function does not differentiate between the possible membership classes (Section 3.2.2) of a set to which a record occurrence can be added. A record occurrence does not have to be added to a MANUAL set since it is possible that the occurrence has been removed from an AUTOMATIC set previously (Section 10.4.3). This is possible because neither MDBS nor VIADUCT implement FIXED set membership explicitly. However, in Section 11.1 a possible method for enforcing FIXED set membership in VIADUCT is described.

Having found a path to the operand record type using the restricted path finding algorithm, it is necessary to traverse that path by means of the restricted path finding algorithm to isolate the specific record occurrence of the type to be added. Thereafter, the sets to which it is to be added must be

considered. Obviously the set used in the selection of the record occurrence is not an eligible set type since the occurrence is already a member of that set. The eligible sets are chosen in one of two ways: either the user has indicated implicitly during command entry that a specific set is to be used by entering an item value for the other record type (either owner or member) participating in that set type, or in response to a question asked at execution time the user has explicitly responded that the set is to be selected (see Figure 2.6). For every set that is selected by one of these methods a path is found and traversed to the corresponding owner or member record. If the specified occurrence is not present in the database an appropriate error message is generated. In addition, if the set is sorted and duplicate keys are prohibited a check is made to ensure that this integrity constraint will not be violated by the addition of the selected record occurrence.

The syntax of the add function demonstrates consistency with respect to the symmetry of N:M sets. For example, using the education database, the following two commands are equivalent:

(a) **ADD STUDENT WHERE ACTIVITY.NAME EQ SQUASH, ;**
STUDENT.NAME EQ JONES<CR>

(b) **ADD ACTIVITY WHERE STUDENT.NAME EQ JONES, ;**
ACTIVITY.NAME EQ SQUASH<CR>

Thus it is not important by which method the user formulates this command as the resulting operation on the database will be the same: the STUDENT named Jones will be linked with the ACTIVITY squash. However, the sequence of database calls executed to achieve this in each case will be different as shown in Figure 10.2. For explanatory purposes all examples containing sample

database calls will use the actual values of the items or records rather than pointers to these values. The statements in square brackets in Figure 10.2 are executed if in (a) duplicate STUDENT NAMEs are disallowed in set PARTICIP or in (b) duplicate ACTIVITY NAMEs are prohibited in set PARTICIP. The reason that both methods add the record occurrence as a member of the set ('ams') is that the DMS does not provide a function to add an occurrence as the owner of a set. The DMS function 'src' is used to establish the occurrence found as the current of record type ('fmsk' does not do this) since the function 'ams' operates on the current of record type.

10.4.2 **The create function**

The create function is used to create a new occurrence of the operand record type in the database. Before a record occurrence of any type can be created an owner record occurrence must be identified for every set of which the record occurrence to be created is an AUTOMATIC member. This is because a record occurrence is added by the DMS to all sets in which it participates as an automatic member as soon as it is created. If an owner record occurrence is not identified for each of these sets, either an error will result because there is no current-owner-of-set, or the newly created occurrence may be linked to the incorrect owner occurrence. The owner occurrences are identified by finding and traversing a path to each owner record type.

All sets in which the operand record type is a manual member or

```
error_status = dms ( fmsk, STUD-SET, JONES, null );
if error_status ~= 0 then exit;
error_status = dms ( src, null, null, null );
error_status = dms ( fmsk, ACT-SET, SQUASH, null );
if error_status ~= 0 then exit;
error_status = dms ( src, null, null, null );
error_status = dms ( smr, PARTICIP, ACTIVITY, null );
error_status = dms ( sor, PARTICIP, STUDENT, null );
[ error_status = dms ( fosk, PARTICIP, JONES, null );
  if error_status = 0 then exit; ]
error_status = dms ( ams, ACTIVITY, PARTICIP, null );
```

(a)

```
error_status = dms ( fmsk, ACT-SET, SQUASH, null );
if error_status ~= 0 then exit;
error_status = dms ( src, null, null, null );
error_status = dms ( fmsk, STUD-SET, JONES, null );
if error_status ~= 0 then exit;
error_status = dms ( src, null, null, null );
error_status = dms ( sor, PARTICIP, STUDENT, null );
error_status = dms ( smr, PARTICIP, ACTIVITY, null );
[ error_status = dms ( fmsk, PARTICIP, SQUASH, null );
  if error_status = 0 then exit; ]
error_status = dms ( ams, ACTIVITY, PARTICIP, null );
```

(b)

Figure 10.2. Two methods of linking a STUDENT with an ACTIVITY.

the owner of an N:M or N:1 set must also be considered as possible recipients of a newly created record occurrence. (Membership of an N:M set can never be automatic since it is not possible for the DMS to add a record occurrence to multiple owner record occurrences; there is only one current-owner-of-set.) For this purpose, the operation of the create function is similar to that of the add function. If there are no sets in which the record occurrence to be created participates as an AUTOMATIC member and the user does not select any sets of which it is a MANUAL member to which it can be added, the record occurrence is not created since it would be inaccessible in the database.

As in the add function, all sorted sets (without duplicates) to which the record occurrence is to be added are checked to ensure that duplicate keys will not exist after creating the record occurrence. Once these checks have been performed, any outstanding item values for the record occurrence to be created (i.e. not set to true in 'select_item') are now obtained from the user and placed in the input buffer. The occurrence is then created in the database by calling the relevant DMS function.

Finally, the occurrence is added to all the MANUAL sets which were selected previously. This process is identical to that performed by the add function.

An example using the education database will serve to illustrate the above procedure. Suppose that the user enters:

```
CREATE STUDENT WHERE TEACHER.NAME EQ BROWN, ;
                     SECTION.NUMBER EQ 122, ACTIVITY.NAME EQ SQUASH, ;
                     STUDENT.NAME EQ JONES, REGNO EQ 9999<CR>
```

This example is illustrative since it requires most of the features of the create module to be employed during its execution. This is because the STUDENT record type is an automatic member of a 1:N set (STUD-SET), a manual member of a 1:N set (ADVISES), a manual member of an N:M set (ENROLL), and the owner of an N:M set (PARTICIP).

Figure 10.3 shows the sequence of calls made to the database management system during the execution of this command. The first nine statements establish the occurrences to which the STUDENT record is to be linked. These occurrences are then assigned to be the current owner ('sor') or member ('smr') of each set involved by updating the relevant currency indicators. The statements in square brackets are included if duplicate key values are disallowed in all sets in which STUDENT NAME is defined as the key item. The STUDENT occurrence is then created after which it is added to each of the chosen sets in which it is a MANUAL member or owner.

Since the education database does not include any derived items, it is necessary to introduce a new example to illustrate this feature of VIADUCT. Figure 10.4 shows a schematic representation of a portion of a database schema whose DDL description is given in Figure 10.5.

This schema extract displays a number of features not present in the education database. There are two sets in which the whole member record is used as a key (sets FROM and SUPPLIES), there is a 1:1 set (PARTCOST), and there are two derived items, PARTNO and SUPPNO in the record type PARTSUPP. It should be noted that the

declaration of derived types is performed in the process of pseudo-subschema generation (Section 7.3). The two derived items are each derived from the items with identical names in the owner record types, PART and SUPPLIER. (It is not required that the derived and contributing items have the same names, only that they are of the same type.)

The structure of the PART and SUPPLIER relationship can be justified in the following way. Each PART can be supplied by many SUPPLIERs and each SUPPLIER supplies possibly many PARTs. This association would tend to suggest the use of an N:M relationship in the database. However, each SUPPLIER supplies a PART at a single COST so the N:M relationship is broken by this intersection data (Section 3.2.1). By simply allowing both PART and SUPPLIER to own a number of COST record occurrences (two 1:N sets), it cannot be ensured that each SUPPLIER supplies each PART at only one COST. This integrity constraint could be enforced by introducing the key items of PART and SUPPLIER (PARTNO and SUPPNO) into the COST record and making their combination the key item (with no duplicates allowed) in both 1:N sets. In this way duplicate PART-SUPPLIER combinations (and consequently multiple COSTs) can be avoided in the database. However, since it is not possible to have a number of items comprising the key in MDBS (as opposed to the whole record), the COST record must be divided as shown in Figure 10.4. To ensure that a single COST exists for each PART-SUPPLIER combination, a 1:1 set is utilized.

The user may wish to enter into the database the fact that an existing SUPPLIER now supplies a PART which is on the inventory list but until now has been supplied only by different SUPPLIERs.

```
error_status = dms ( fmsk, TEAC-SET, BROWN, null );
if error_status ~= 0 then exit;
error_status = dms ( src, null, null, null );
error_status = dms ( fmsk, SECT-SET, 122, null );
if error_status ~= 0 then exit;
error_status = dms ( src, null, null, null );
error_status = dms ( fmsk, ACT-SET, SQUASH, null );
if error_status ~= 0 then exit;
error_status = dms ( src, null, null, null );
[ error_status = dms ( fmsk, STUD-SET, JONES, null );
  if error_status = 0 then exit; ]
error_status = dms ( sor, ADVISES, TEACHER, null );
[ error_status = dms ( fmsk, ADVISES, JONES, null );
  if error_status = 0 then exit; ]
error_status = dms ( sor, ENROLL, SECTION, null );
[ error_status = dms ( fmsk, ENROLL, JONES, null );
  if error_status = 0 then exit; ]
error_status = dms ( smr, PARTICIP, ACTIVITY, null );
[ error_status = dms ( fosk, PARTICIP, JONES, null );
  if error_status = 0 then exit; ]
error_status = dms ( crs, STUDENT, JONES 9999, null );
error_status = dms ( ams, STUDENT, ADVISES, null );
error_status = dms ( ams, STUDENT, ENROLL, null );
error_status = dms ( ams, ACTIVITY, PARTICIP, null );
```

Figure 10.3. Sequence of database calls executed to create a STUDENT record occurrence.

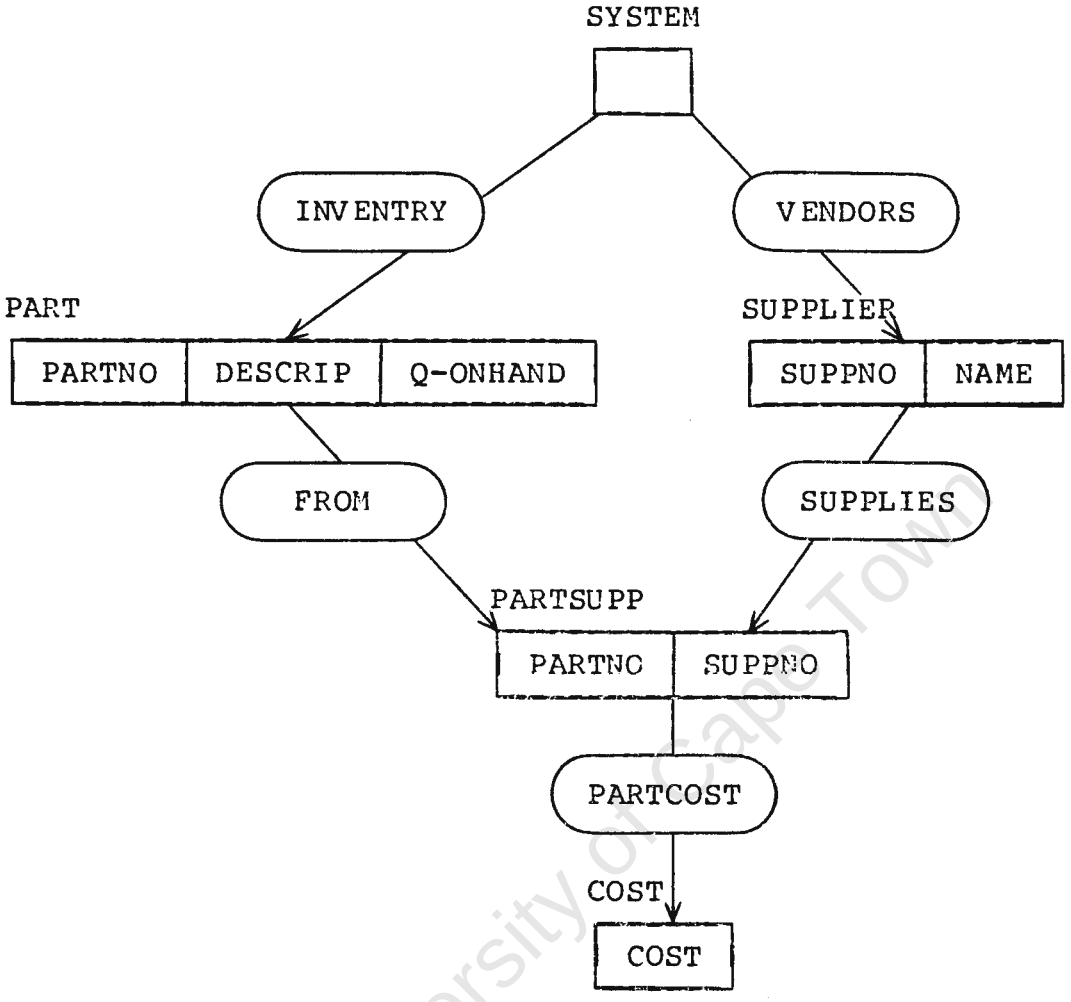


Figure 10.4. Diagram of a PARTs and SUPPLIERs schema.

```
RECORD PART
ITEM   PARTNO    CHAR  6
ITEM   DESCRIP   CHAR 20
ITEM   Q-ONHAND  BIN   2

RECORD SUPPLIER
ITEM   SUPPNO    CHAR  6
ITEM   NAME      CHAR 20

RECORD PARTSUPP
ITEM   PARTNO    CHAR  6
ITEM   SUPPNO    CHAR  6

RECORD COST
ITEM   COST      REAL  4

SET    INVENTRY  AUTO 1:N
                                SORTED   PARTNO
OWNER  SYSTEM
MEMBER PART

SET    VENDORS   AUTO 1:N
                                SORTED   SUPPNO
OWNER  SYSTEM
MEMBER SUPPLIER

SET    FROM      AUTO 1:N
                                SORTED
OWNER  PART
MEMBER PARTSUPP

SET    SUPPLIES  AUTO 1:N
                                SORTED
OWNER  SUPPLIER
MEMBER PARTSUPP

SET    PARTCOST  AUTO 1:1
                                IMMAT
OWNER  PARTSUPP
MEMBER COST
```

Figure 10.5. Portion of the MDBS DDL of the schema depicted in Figure 10.9.

To do this the user might enter the following:

CREATE PARTSUPP WHERE PART.PARTNO EQ LS173, ;

SUPPLIER.SUPPNO EQ 12345<CR>

The above command would be executed by VIADUCT without any additional information being required from the user. The PART.PARTNO and SUPPLIER.SUPPNO values are used to identify owner record occurrences for the PARTSUPP occurrence to be created. Because PARTSUPP.PARTNO and PARTSUPP.SUPPNO are derived items, VIADUCT checks whether values for the contributing items (PART.PARTNO and SUPPLIER.SUPPNO respectively) have been input. Since this is the case (corresponding elements of 'select_item' are set to true), these values are copied from the input buffers of the contributing records into the PARTSUPP input buffer. As no other item values are required in PARTSUPP, a record occurrence is created. If the user does not enter values for PART.PARTNO and SUPPLIER.SUPPNO in the command, these values are solicited from him before a PARTSUPP occurrence is created since these are the items from which the PARTSUPP record is composed.

The user could now create a COST record occurrence for the newly created PARTSUPP occurrence. This would be performed in a similar manner:

CREATE COST WHERE PART.PARTNO EQ LS173, ;

SUPPLIER.SUPPNO EQ 12345, COST EQ 23.45<CR>

The correct PARTSUPP record occurrence would be found by the path traversal algorithm using the fact that it is composed of derived items. Since COST is the only item comprising the record COST, the COST occurrence would be created and automatically linked to

the PARTSUPP occurrence found (since membership in the set PARTCOST is AUTO).

Unfortunately a COST record occurrence cannot be created without previously having created a PARTSUPP occurrence. Since a 1:1 relationship exists between the two record types, the provision of a facility whereby a PARTSUPP occurrence is automatically created when a COST occurrence is created might be desirable. This would have the effect of further shielding the user from the physical representation of the data since he would be unaware of the existence of the PARTSUPP record type.

The sets in which PARTSUPP participates as a member are defined as AUTOMATIC since any PARTSUPP occurrence must be linked to its correct owner occurrences at the time of its creation in order to maintain integrity. If the sets had been defined as MANUAL, it would be the responsibility of the user to instruct VIADUCT to add the PARTSUPP occurrence to the PART and SUPPLIER occurrences. To ensure that the integrity of the derived items is not violated subsequently, a form of FIXED set membership (Section 3.2.2) would have to be enforced (Section 11.1).

10.4.3 **The remove function**

The remove function is used to remove an existing record occurrence from any number of sets in which it currently participates. The relationship between the record type and sets from which it can be removed correspond to those which are considered by the add function. It must be ensured, however,

that the record occurrence is not removed from the only set in which it participates as this will result in the occurrence (and possibly other occurrences) becoming inaccessible in the database. This is a consequence of the inability to access record occurrences directly in MDBS; all access to record occurrences is via set and so it is essential that all record occurrences in the database remain connected to some set occurrence.

The sequence of operations performed by the remove function is similar to that of the add function: a path is found and traversed to the record occurrence to be removed; then the owner or member record types of all eligible sets are checked to see if they have been selected. Before the record occurrence can be removed from a set it must be established that it participates in that set. If the set under consideration is the one used to isolate the record occurrence, this does not present a problem (unlike the case of the add function). The error status returned from the DMS functions used to update currency indicators can be used to determine whether the record occurrence is a member or owner of the set being considered.

Having established that the record occurrence is a member (or owner) of the set type, it is not necessary to identify the owner (or member) record occurrence except in the case of an N:M set. This is because the member of a 1:N set (or the owner of an N:1 set) clearly determines the owner (or member) of the set occurrence. However, for the sake of command consistency (see Section 6.1) the user is required to identify the owner or member occurrence in each case. This occurrence can be used to ensure

that in the above cases the correct set occurrence from which the operand record is to be removed is chosen.

Since, as noted in the case of the add function, there is no DMS function to remove the owner occurrence from an N:M or N:1 set, all commands requiring such an action are implemented as the removal of the member occurrence instead. The remove function displays the same symmetry as the add function with regard to N:M sets. Thus the following two commands are functionally equivalent:

(a) **REMOVE STUDENT WHERE SECTION.NUMBER EQ 122 ;**
STUDENT.NAME EQ JONES<CR>

(b) **REMOVE SECTION WHERE STUDENT.NAME EQ JONES ;**
SECTION.NUMBER EQ 122<CR>

10.4.4 The delete function

The delete function is used to delete from the database an existing record occurrence of the type specified in the operand clause of a command. As such, it is the most trivial function to implement and execute but has the potential for causing extensive loss of data in the database.

As mentioned in Section 4.3, MDBS does not check whether the record occurrence to be deleted has any member occurrences (there is no equivalent of the CODASYL DELETE ONLY clause). All member record occurrences are deleted along with their owner record occurrence if they are not members of other set occurrences.

For the above reason, after the record occurrence to be deleted has been identified by finding and traversing a path to it, it is assigned to be the owner of all sets of which it is declared as an owner record type in the schema. For each of these sets the error status returned by the DMS currency assignment functions should reflect the end-of-set condition. If this is not the case, the user is informed that member occurrences exist in the sets for which a zero error status is returned. In this situation the user is asked to confirm that the deletion is in fact to proceed (Figure 10.6).

The delete function could display more intelligence by ascertaining whether the member occurrences of the record occurrence to be deleted do in fact participate in other set occurrences and therefore would not be lost as a result of the deletion.

10.5 List and update

10.5.1 The list function

The correct operation of the list function relies on the path finding and traversal algorithms described in the previous chapter. Additional features of the list function include the production and format of the output; the example of Figure 2.7 demonstrates most of these features.

The user is asked if the output generated is to be sent to the console, the printer, or a user-named disk file. Output to a

```
DELETE SUBJECT WHERE SUBJECT.NUMBER EQ 10<CR>
    error_status = dms ( fmsk, SUBJ-SET, 10, null );
    error_status = dms ( som, HAS, SUBJ-SET, null );
    Since error_status = -1
COURSE occurrences exist.
Do you still wish to delete (Y/N) ?: Y
    error_status = dms ( drm, SUBJ-SET, null, null );
Record deleted.
```

Figure 10.6. Example of a deletion.
(database calls are shown indented)

disk file is useful in that the information can subsequently be processed by other software packages such as word processors, sort programs, or financial planning packages. An optional heading facility is provided and in the case of printer output this heading is printed along with a page number at the top of each page. If the output is sent to the console, the listing is interrupted after the screen has been filled, allowing the user to examine the output before allowing the listing to continue.

As shown in Chapter 2 (Figure 2.7), if the user has specified that the key item of a sorted set is to be listed, he is asked whether the output is to be ordered in ascending or descending key sequence. The example in Chapter 2 served also to illustrate the feature of the list function whereby information retrieved higher up a path is output only once and is not repeated every time conditions are met at the end of the path. This makes the output more readable, especially since the order of item types listed across the page (or screen) is determined by their ordering on the path rather than the order in which they were specified in the operand clause of the command.

An example of the listing of items from a record type which is one of multiple members of a set is given in Section 11.2. Further examples involving the list function are also given in Section 11.2.

10.5.2 **The update function**

The update function uses the generalized path finding and traversal algorithms (Chapter 9). This allows for multiple occurrences of an item to be updated in a single command. On each occasion during path traversal that the conditions specified by the user in the selection clause are met, the user is asked to enter a new value for the item that was referenced in the operand clause of the command.

The following examples using the PARTs and SUPPLIERs database of Section 10.4.2 will illustrate this procedure. The command:

UPDATE COST WHERE PART.PARTNO EQ LS173, ;

SUPPLIER.SUPPNO EQ 12345<CR>

would result in the user being prompted for one new COST value (Figure 10.6). However, the following command would require a number of entries from the user:

UPDATE COST WHERE SUPPLIER.SUPPNO EQ 12345<CR>

In this case the user would be required to update the values of as many occurrences of the COST item as the number of PARTs supplied by the SUPPLIER with SUPPNO 12345.

VIADUCT does not permit the updating of derived items because this would constitute a violation of integrity: the source item value would not have been similarly updated. However, the values of the source items from which others are derived are allowed to be updated. In this case, the path is extended to include all record types containing items derived from the source item. These derived items are then updated automatically by VIADUCT when the source item value is changed.

```
error_status = dms ( fmsk, INVENTRY, LS173, null );
if error_status = 0
  then
    error_status = dms ( som, FROM, INVENTRY, null );
    if error_status = 0
      then
        PARTSUPP.PARTNO <- PART.PARTNO                [derived item]
        PARTSUPP.SUPPNO <- SUPPLIER.SUPPNO            [derived item]
        error_status = dms ( fmsk, FROM, LS173 12345, null );
        if error_status = 0
          then
            error_status = dms ( som, PARTCOST, FROM, null );
            error_status = dms ( getm, PARTCOST, null, null );
Current contents are : 12.34
Enter new COST : 13.99<CR>
            error_status = dms ( sfm, PARTCOST, null, null );
Item updated.
```

Figure 10.6. Updating of a single COST.
(The code executed is shown indented.)

References

[Digi80] Link-80 Operator's Guide. Digital Research Inc., Pacific Grove, California, 1980.

[MDBS79] MDBS.DMS User's Manual. Micro Data Base Systems Inc., Lafayette, Indiana, 1979.

[MDBS80a] MDBS - CP/M Machine Language Interface Notes. Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

[MDBS80b] MDBS - PL/I Interface Notes. Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

CHAPTER 11: **A CASE STUDY**

This chapter uses a representative portion of an actual database schema to summarize the features of VIADUCT described in previous chapters. In Section 11.1 the essential role of the pseudo-subschema is highlighted; in Section 11.2 further examples of VIADUCT commands are introduced.

11.1 **Database description**

The schema of the database system used in this case study is depicted in Figure 11.1 while its associated MDBS DDL is listed in Figure 11.2. This database is a slight simplification of an actual MDBS schema implemented by the author for a stock control application.

As can be seen from Figure 11.2, four views of the database corresponding to the userids STOCKROOM, PURCHASING, SELLING and MANAGEMENT have been defined. The portion of the database relating PARTs and SUPPLIERs is identical to that introduced in Section 10.4.2 except that access levels have now been placed on the item types.

Because of the hierarchical nature of the MDBS access rights it is not possible to define disjoint views of the database. (The reader will recall that this was a major motivation for implementing VIADUCT.) Thus the fact that the record types CUSTOMER, DEBIT, CREDIT, and ORDER have been totally excluded

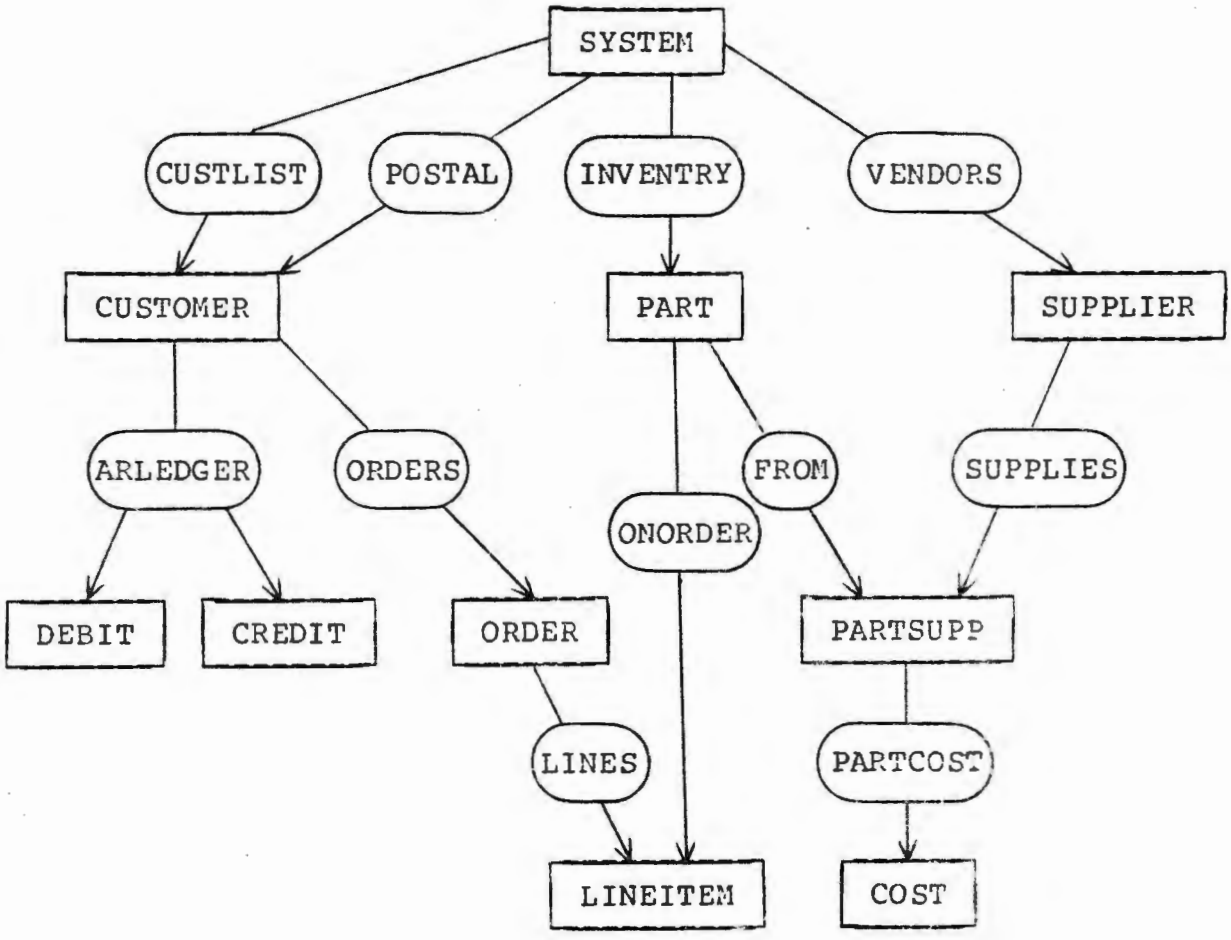


Figure 11.1. Database schema for the case study.

```
FILES   BUSINESS.DB    1    512
DRIVE   1      50
PASSWORDS
        STOCKROOM              010 010      STORE
        PURCHASING             020 020      COSTS
        SELLING                030 030      PRICES
        MANAGEMENT             030 030      CARTE-BLANCHE

RECORD  CUSTOMER               030 030
ITEM    NAME       CHAR 20     030 030
ITEM    ACCTNO     BIN   2     030 030
ITEM    POSTCODE   CHAR  4     030 030

RECORD  PART                   010 010
ITEM    PARTNO     CHAR  6     010 020
ITEM    DESCRIP    CHAR 20     010 020
ITEM    Q-ONHAND   BIN   2     010 010

RECORD  SUPPLIER               010 020
ITEM    SUPPNO     CHAR  6     010 020
ITEM    NAME       CHAR 20     010 020

RECORD  DEBIT                  030 030
ITEM    DATE       CHAR  6     030 030
ITEM    AMOUNT     REAL  4     030 030

RECORD  CREDIT                 030 030
ITEM    DATE       CHAR  6     030 030
ITEM    AMOUNT     REAL  4     030 030

RECORD  ORDER                  030 030
ITEM    DATE                   030 030
ITEM    PONUMBER   CHAR  6     030 030

RECORD  PARTSUPP               010 020
ITEM    PARTNO     CHAR  6     010 020
ITEM    SUPPNO     CHAR  6     010 020

RECORD  LINEITEM               010 030
ITEM    QUANTITY   BIN   2     010 030

RECORD  COST                   020 020
ITEM    COST       REAL  4     020 020

SET     CUSTLIST   AUTO 1:N    030 030
                               SORTED   ACCTNO
OWNER   SYSTEM
MEMBER  CUSTOMER

SET     POSTAL     AUTO 1:N    030 030
                               SORTED   POSTCODE
OWNER   SYSTEM
MEMBER  CUSTOMER
```

Figure 11.2. MDBS DDL of the schema of Figure 11.1.

```
SET     INVENTRY AUTO 1:N 010 010
                              SORTED    PARTNO
OWNER   SYSTEM
MEMBER  PART

SET     VENDORS  AUTO 1:N 010 010
                              SORTED    SUPPNO
OWNER   SYSTEM
MEMBER  SUPPLIER

SET     ARLEDGER MAN  1:N 030 030
                              SORTED    DATE
OWNER   CUSTOMER
MEMBER  DEBIT
MEMBER  CREDIT

SET     ORDERS   MAN  1:N 030 030
                              SORTED    DATE
OWNER   CUSTOMER
MEMBER  ORDER

SET     ONORDER  MAN  1:N 010 030
                              IMMAT
OWNER   PART
MEMBER  LINEITEM

SET     LINES    MAN  1:N 030 030
                              FIFO
OWNER   ORDER
MEMBER  LINEITEM

SET     FROM     AUTO 1:N 010 020
                              SORTED
OWNER   PART
MEMBER  PARTSUPP

SET     SUPPLIES AUTO 1:N 010 020
                              SORTED
OWNER   SUPPLIER
MEMBER  PARTSUPP

SET     PARTCOST AUTO 1:1 020 020
                              IMMAT
OWNER   PARTSUPP
MEMBER  COST

END
```

Figure 11.2 (cont'd). MDBS DDL of the schema of Figure 11.1.

from the view of the STOCKROOM and PURCHASING users means that users who do have access to the above record types (i.e. SELLING and MANAGEMENT) automatically have access to all information accessible to the STOCKROOM and PURCHASING users (since SELLING and MANAGEMENT must have read/write access levels exceeding those of STOCKROOM and PURCHASING). As a result of this there is no record type that is inaccessible to the SELLING user, so he has the same access rights as MANAGEMENT.

A further problem arises as a result of the fact that the STOCKROOM user has write access to the item Q-ONHAND. In MDBS this allows him to create or delete any PART occurrence. A similar problem arises with any user who has write access to any item within a record type.

Features of VIADUCT available to the user at pseudo-subschema generation time (Section 7.3) can be used to overcome the above limitations. The inability to define disjoint views can be removed by specifying that the SELLING user cannot perform any functions on SUPPLIER, PARTSUPP, or COST record types. In this way two views which overlap only with respect to the PART and LINEITEM record types can be defined for the PURCHASING and SELLING personnel (Figures 11.3 and 11.4 respectively).

The problem of unauthorised users having create and delete rights was overcome in VIADUCT by granting use of the CREATE and DELETE functions to only those users who are permitted by management to use them. The precise record types he may CREATE or DELETE are defined for each such authorised user. Thus the STOCKROOM user has been granted access to only the LIST and UPDATE functions; he

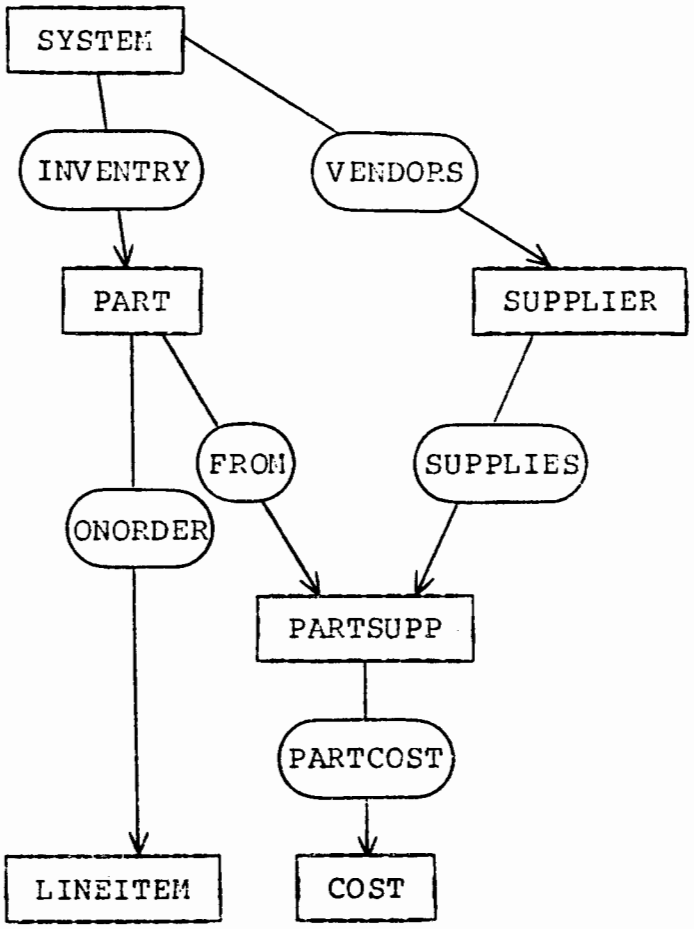


Figure 11.3. PURCHASING view of the database schema.

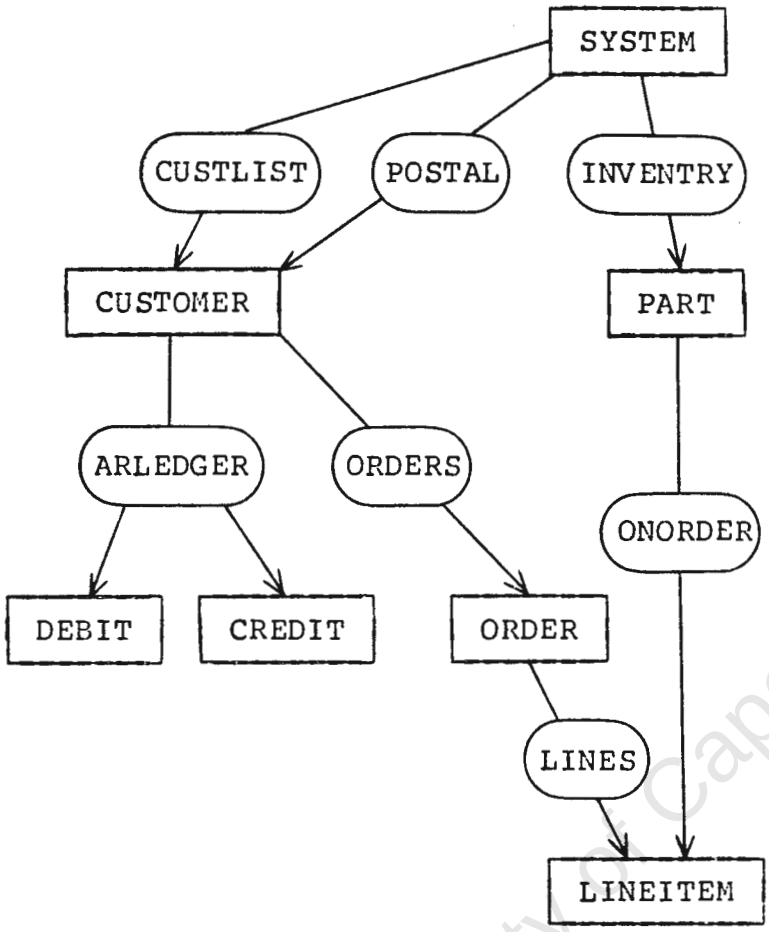


Figure 11.4. SELLING view of the database schema.

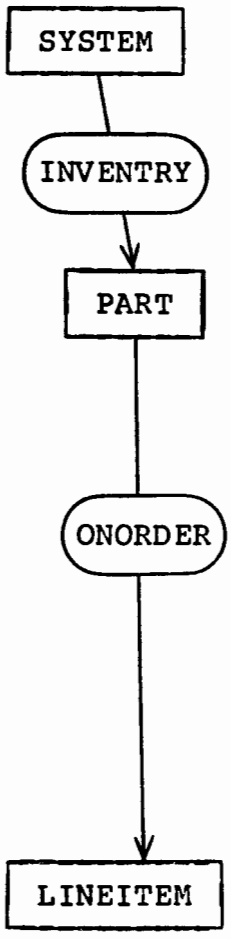


Figure 11.5. STOCKROOM view of the database schema.

is permitted to LIST PARTs and the QUANTITies in which they have been ordered and is allowed to UPDATE only the item Q-ONHAND (Figure 11.5).

The PURCHASING user is permitted to perform all the functions of the STOCKROOM user. In addition he is able to CREATE and UPDATE PART record occurrences as well as being able to CREATE, UPDATE and LIST SUPPLIER, PARTSUPP, and COST record occurrences. As described in Section 10.4.2 both the items comprising PARTSUPP are derived and so cannot be updated (Section 10.5.3). The problem of inconsistencies that might arise as a result of users REMOVing PARTSUPP occurrences from their correct owners (Section 10.4.2) have been avoided by not allowing any users to REMOVE or ADD PARTSUPP occurrences. (This enforces a method of FIXED set membership for the PARTSUPP record type in the sets FROM and SUPPLIES.)

As mentioned above the SELLING user does not have access to the SUPPLIER, PARTSUPP, or COST record types. He is permitted to LIST all item types in his view, UPDATE all items except those comprising PARTs, CREATE all record types excluding PARTs, and DELETE DEBITs, CREDITs, ORDERs, and LINEITEMs. (The ADD and REMOVE functions are not applicable to this view of the schema.)

Only the MANAGEMENT user has the capability to DELETE CUSTOMERs, PARTs, and SUPPLIERs. In addition to this he is allowed to perform all the functions granted to the other users. Thus his view of the database encompasses the entire schema (Figure 11.1).

As stated in Section 10.4.2 the conditions necessary to enforce

integrity with respect to the relationship between PARTs and SUPPLIERs have been enforced by VIADUCT through the declaration of derived items in the pseudo-subschema. Further integrity constraints have been placed on the information in the database by prohibiting duplicate key values in all sorted sets except in the set POSTAL which orders CUSTOMERs by their postal codes, and those sets which are ordered by an item DATE (ARLEDGER and ORDERS). Meaningless item values such as negative quantities and costs have also been forbidden by specifying realistic minimal values for all numeric item types during the generation of the pseudo-subschemata.

11.2 **Database manipulation**

In order to summarize the features of VIADUCT commands, three examples using the LIST function are given below. The LIST function has been chosen because its use can result in the most complex database traversal sequences; furthermore, a number of examples utilizing the other VIADUCT functions have been given in previous chapters. For all three examples given below it is assumed that entry is by a MANAGEMENT user working in expert mode.

The first example is to LIST all CUSTOMERs who have ordered PARTs supplied by a particular SUPPLIER. This requirement would be entered using the following command:

(1) **LIST CUSTOMER.NAME WHERE SUPPLIER.SUPPNO EQ 12345<CR>**

Although the resultant path for this query is linear, it involves traversing almost the entire database and in so doing requires

the use of all the DMS currency assignment functions.

VIADUCT has a choice of four entry points to solve the above query: CUSTLIST, POSTAL, INVENTRY, or VENDORS (see Figure 11.1). Since the key item of the set VENDORS, namely SUPPLIER.SUPPNO, is to be tested, this set is automatically chosen as the entry point. The code executed to traverse the resulting path is given in Figure 11.6.

The second example involves listing the DEBITs accrued by all CUSTOMERs in a certain postal code area. Such a query might be:

(2) **LIST CUSTOMER.NAME, DEBIT.DATE, DEBIT.AMOUNT ;**
WHERE POSTCODE EQ 7700<CR>

Finding a path connecting the record types CUSTOMER and DEBIT does not present a problem. Since the key item of the set POSTAL (item POSTCODE) rather than that of CUSTLIST (item NAME) is to be tested, POSTAL is chosen as the entry set. The fact that the set ARLEDGER has multiple member record types (DEBIT and CREDIT) means that the occurrences of different types are interspersed through the set. Thus, for this query, VIADUCT must ensure that only the DEBIT occurrences are retrieved by checking the type of each member occurrence of the set ARLEDGER as it is found. The code necessary to satisfy the above query is given in Figure 11.7.

The final example is to LIST the SUPPLIERs and corresponding COSTs of the PART with DESCRIPtion 'NUT'. This would be entered as follows:

(3) **LIST SUPPLIER.NAME, COST WHERE PART.DESCRIP EQ NUT<CR>**

The two possible relevant entry sets are INVENTRY and VENDORS

```
error_status = dms ( fmsk, VENDORS, 12345, null );
if error_status ~= 0 then exit;
error_status = dms ( som, SUPPLIES, VENDORS, null );
error_status = dms ( ffm, null, SUPPLIES, null );
while error_status = 0 do
  error_status = dms ( smm, FROM, SUPPLIES, null );
  error_status = dms ( soo, ONORDER, FROM, null );
  error_status = dms ( ffm, null, ONORDER, null );
  while error_status = 0 do
    error_status = dms ( smm, LINES, ONORDER, null );
    error_status = dms ( smo, ORDERS, LINES, null );
    error_status = dms ( geto, ORDERS, null, null );
    LIST CUSTOMER.NAME;
    error_status = dms ( fnm, null, ONORDER, null );
  end
  error_status = dms ( fnm, null, SUPPLIES, null );
end
```

Figure 11.6. The code generated and executed for example 1 (see text).

```
error_status = dms ( fmsk, POSTAL, 7700, null );
continue = ( error_status = 0 );
while continue do
  error_status = dms ( som, ARLEDGER, POSTAL, null );
  error_status = dms ( ffm, null, ARLEDGER, null );
  while error_status = 0 do
    error_status = dms ( cmt, ARLEDGER, DEBIT, null );
    if error_status = 0
      then
        error_status = dms ( getm, ARLEDGER, null, null );
        LIST CUSTOMER.NAME, DEBIT.DATE, DEBIT.AMOUNT;
    error_status = dms ( fnm, null, ARLEDGER, null );
  end
  continue = find_next_duplicate ( POSTAL );
end
```

Figure 11.7. The code generated and executed for example 2 (see text).

```
error_status = dms ( ffm, null, INVENTRY, null );
while error_status = 0 do
  if PART.DESCRIP EQ NUT
    then
      error_status = dms ( som, FROM, INVENTRY, null );
      error_status = dms ( ffm, null, FROM, null );
      while error_status = 0 do
        error_status = dms ( smm, SUPPLIES, FROM, null );
        error_status = dms ( som, PARTCOST, FROM, null );
        error_status = dms ( geto, SUPPLIES, null, null );
        error_status = dms ( getm, PARTCOST, null, null );
        LIST SUPPLIER.NAME, COST;
        error_status = dms ( fnm, null, FROM, null );
      end
  error_status = dms ( fnm, null, INVENTRY, null );
end
```

Figure 11.8. The code generated and executed for example 3 (see text).

(see Figure 11.1). Although the key item of set INVENTRY (namely PARTNO) is not being tested, a condition has been placed on another item of the PART record type (namely DESCRIP). Since no conditions have been placed on any items of the SUPPLIER record type which is the member record of VENDORS (the other possible entry set), entry will be via the INVENTRY set.

The path which connects the record types in the above query is non-linear: all three sets in which the PARTSUPP record type participates must be traversed in order to satisfy the query. Since VIADUCT linearises the path so that the set SUPPLIES is traversed before the set PARTCOST, the code of Figure 11.8 is executed in order to process the above command.

The three preceding examples were chosen to illustrate both the power and ease-of-use of the VIADUCT commands in a practical application.

CHAPTER 12: **CONCLUSIONS**

This chapter reviews the features of VIADUCT with reference to the objectives presented in Chapter 1. Directions for possible extensions and further study are discussed.

12.1 **Conclusions**

It has been shown that it is possible to develop a path-free database manipulation language for a network database system. The removal of the need for path specification together with the interactive prompting and on-line help facilities of VIADUCT free the user from having to have detailed knowledge of the structure of the database being accessed and indeed of databases in general. Thus VIADUCT provides the features of a powerful database system and yet is easy for the inexperienced microcomputer user to use.

The security restrictions available in VIADUCT provide a means for defining a number of views of the database, each with a strict set of processing capabilities. By using these it should be possible to avoid any unauthorised access to or manipulation of data in the database.

It has been shown that the integrity constraints that can be imposed by the VIADUCT subschemata can reduce the amount of inaccurate or inconsistent data that could be entered into the database. These integrity constraints would otherwise have to be

specially written into every conventional MDBS application program accessing the database.

Although a number of additional features could be included in VIADUCT, these would introduce inconsistency and consequently complexity to the command syntax. This would have the undesirable effect of confusing the inexperienced user.

The inability to traverse cyclic and hence recursive database structures is not seen as a serious limitation of VIADUCT. The introduction of such a capability would certainly result in more alternate paths between record types being found and so would require more interaction from the user. Furthermore, recursive structures are not readily understood by the computer novice and so should perhaps rather be included only in schemata accessed exclusively by experienced database users.

12.2 Future study

When discussing the areas for further investigation it is important to distinguish between those which are of research interest and those which would merely make VIADUCT a more marketable product. Examples of work in the latter category include improved report writing facilities, improved output formats, provision for statistical analyses, column totals, and data transformation. The areas of research interest are discussed in the following two sections.

12.2.1 **Performance considerations**

The performance of VIADUCT could be enhanced in a number of areas. These include improved path selection and the availability of more memory space.

In situations when a number of entry sets appear equally suitable for selection (for example, where there are no conditions placed on any of their member records), VIADUCT could make use of the DMS function which returns the number of set member occurrences ('gmc') for each of the entry sets. The set with the fewest occurrences could then be chosen as the entry set.

Analysis of the time complexity of the path finding and traversal algorithms may reveal unacceptable orders of complexity. This would then make it desirable to optimize these algorithms.

As a result of the extensive use of overlays, VIADUCT itself occupies only about 24K bytes of memory when executing. However, this compaction is achieved at the cost of additional time for loading an overlay both before a command is entered (the parser) and before it is executed (one of the function modules). With the DMS routines included VIADUCT requires about 42K bytes of memory leaving about 14K bytes for database buffer storage. (The operating system uses the remaining memory.) By implementing memory bank switching as used in multi-user operating systems [Digi81], the performance of VIADUCT could be greatly enhanced. If an additional 48K byte bank of memory were installed in the machine the 18K bytes of DMS routines could reside in this additional bank leaving 30K bytes for buffer

space. VIADUCT could then utilize 48K bytes of memory in the original bank of memory thus removing the need for any overlays. The database interface (Section 10.2) would be responsible for switching between the two banks of memory on every database call.

The amount of storage wasted as a result of using static data structures could be reduced. A saving might be made by storing the incidence matrix using techniques for storing sparse matrices and by allocating path storage dynamically. The wastage resulting from the fact that every record type in the pseudo-subschema has enough storage allocated for it to accomodate the maximum number of item types in any one record could be reduced by using a single static structure to store items, with the items contained in each record type linked by array subscript. However, the resulting code would be far more complex and thus more difficult to maintain.

12.2.2 **Additional features**

A number of additional capabilities for increasing the power of the VIADUCT commands could be added.

For example, the update function could allow the user to update item values by specifying an expression containing the original value. An example of this might be:

UPDATE Q-ONHAND EQ Q-ONHAND + 5 WHERE PART.PARTNO EQ LS173<CR>

Comparisons between the values of item types within the same record type could be permitted in the selection clause of a

command. For example,

WHERE Q-ONHAND LT RE-ORDER

The selection clause might also permit the specification of recursive queries such as:

LIST EMPLOYEE.NAME WHERE SALARY LT ;

(SALARY WHERE EMPLOYEE.NAME EQ JONES)<CR>

However, the inclusion of such capabilities would increase the command syntax complexity; worse still, the consistency of command syntax would be lost.

From the above discussion it is evident that a number of improvements could be made to VIADUCT. However, it is felt that even in its present form VIADUCT constitutes a interesting piece of research as well as a useful product.

Reference

[Digi81] MP/M Operating System - System Implementor's Guide. Digital Research Inc., Pacific Grove, California, 1981.

APPENDIX A

The VIADUCT command syntax is given below.

```
<command>            ::= <list_command> | <update_command> |
                         <create_command> | <add_command> |
                         <remove_command> | <delete_command> |
                         <exit_command>
<list_command>       ::= LIST <item_specification> { , <item_
                         specification>}* {<selection_clause>}
<update_command>     ::= UPDATE <item_specification>
                         {<selection_clause>}
<create_command>     ::= CREATE <record_name> {<selection_clause>}
<add_command>        ::= ADD <record_name> {<selection_clause>}
<remove_command>     ::= REMOVE <record_name> {<selection_clause>}
<delete_command>     ::= DELETE <record_name> {<selection_clause>}
<exit_command>       ::= EXIT
<item_specification> ::= {<record_name> . } <item_name>
<selection_clause>   ::= WHERE <item_specification> <operator>
                         <value> { , <item_specification> <operator>
                         <value>}*
<record_name>        ::= <character_string>
<item_name>          ::= <character_string>
<operator>           ::= EQ | NE | LT |  GT
<value>              ::= <character_string> | <number> |
                         ' {<character_string> | blank}* '
<character_string>   ::= <letter> {<letter> | <digit> | - }*
<letter>             ::= A | B | ... | Z | a | b | ... | z
<digit>              ::= 0 | 1 | ... | 9
<number>             ::= <integer> | <real_number>
```

<sign> ::= + ! -

<integer> ::= {<sign>} <digit> {<digit>}*

<real_number> ::= {<sign>} {<digit>}* { . {<digit>}*}
{ E {<sign>} {<digit>}*}

APPENDIX B

The PL/I-80 declaration of the MDBS DMS functions, their associated integer values, and the source of each MDBS acronym follow.

```
%REPLACE
ams      BY  1,  /* Add Member to Set */
closedb  BY  3,  /* CLOSE the DataBase */
cmt      BY  4,  /* Check Member Type */
cot      BY  5,  /* Check Owner Type */
crs      BY  7,  /* Create Record and Store data */
drm      BY  9,  /* Delete Record that is current Member */
ffm      BY  12, /* Find First Member */
ffo      BY  13, /* Find First Owner */
flm      BY  16, /* Find Last Member */
flo      BY  17, /* Find Last Owner */
fmsk     BY  18, /* Find Member based on Sort Key */
fnm      BY  19, /* Find Next Member */
fno      BY  20, /* Find Next Owner */
fosk     BY  21, /* Find Owner based on Sort Key */
fpm      BY  22, /* Find Previous Member */
fpo      BY  23, /* Find Previous Owner */
getm     BY  25, /* GET data from current Member */
geto     BY  26, /* GET data from current Owner */
getr     BY  27, /* GET data from current Record */
opendb   BY  37, /* OPEN the DataBase */
rms      BY  42, /* Remove current Member from Set */
sfr      BY  50, /* Set Field in current Record */
smm      BY  52, /* Set current Member based on current Member */
```

```
smo     BY  53, /* Set current Member based on current Owner */
som     BY  56, /* Set current Owner based on current Member */
soo     BY  57, /* Set current Owner based on current Owner */
sor     BY  58, /* Set Owner based on Record */
src     BY  59; /* Set Record based on Current of run unit */
```

APPENDIX C

The MDBS DMS functions and their associated parameter requirements follow. The underlined parameters are returned by the DMS. The parameters are in fact the addresses of the various types and data rather than the type names or data themselves.

Function	Parameters		
ams	record type,	set type,	null
closedb	null,	null,	null
cmt	set type,	record type,	null
cot	set type,	record type,	null
crs	record type,	record data,	null
drc	null,	null,	null
ffm	null,	set type	null
ffo	null,	set type	null
flm	null,	set type,	null
flo	null,	set type,	null
fmsk	set type,	item value,	null
fnm	null,	set type,	null
fno	null,	set type,	null
fosk	set type,	item value,	null
fpm	null,	set type,	null
fpo	null,	set type,	null
getm	set type,	record data,	null
geto	set type,	record data,	null
getr	record type,	record data,	null
opendb	file name,	userid/passwd	status
rms	set type,	null,	null

sfr	record type,	item value,	item type
smm	set type,	set type,	null
smo	set type,	set type,	null
som	set type,	set type,	null
soo	set type,	set type,	null
sor	set type,	record type,	null
src	null,	null,	null

BIBLIOGRAPHY

Included in the bibliography are all references made in the text of the thesis as well as other books and journal articles consulted during the course of the research.

ANSI/X3/SPARC Study Group on Data Base Management Systems, Interim Report FDT, **ACM SIGMOD bulletin**, Vol 7, No 2, 1975.

Bachman, C.W. Data Structure Diagrams. **Data Base** (journal of **ACM SIGBDP**), Vol 1, No 2 (Summer 1969).

Bachman, C.W. Why restrict the modelling capability of CODASYL data structure sets, **AFIPS Conference Proceedings**, Vol 46, 1977, pp 69-75.

Barly K.S. and Driscoll J.R. A Survey of Data-Base Management Systems for Microcomputers, **Byte**, Vol 6, No 11 (November 1981), pp 208-234.

Chamberlin, D.D. and Boyce, R.F. SEQUEL: A Structured English Query Language, IBM Technical Report RJ1394, 1974.

Chamberlin, D.D. A survey of user experience with the SQL data sublanguage, IBM Research Report R92769(35322), San Jose, California, May 1980.

Christensen, M.A. QUEASY: The design and implementation of a Managerial Information System for Casual Users, **Proceedings ACM Annual Conference**, 1978, pp 230-233.

Data Base Task Group of CODASYL Programming Language Committee, Report (April 1971).

CODASYL Committee. CODASYL Data Description Journal of Development, Ottawa, Canada, 1981.

Codd, E.F. A Relational Model of Data for Large Shared Data Banks. **Commun. ACM**, Vol 13, No 6 (June 1970).

Date, C.J. **An Introduction to Database Systems**, 2nd Edition. Addison-Wesley, Reading, Massachusetts, 1977.

Date C.J. **An Introduction to Database Systems**, 3rd Edition. Addison-Wesley, Reading, Massachusetts, 1981.

Denning, D.E., Denning, P.J., and Schwartz, M.D. The tracker: A threat to statistical database security. **ACM Trans. Database Syst.**, Vol 4, No 1 (March 1979), pp 76-96.

Derfler, F.J. and Greene, W.H. PL/I-80 two-pass compiler from Digital Research. **Infoworld**, Vol 3, No 28 (November 30, 1981), pp 56-57.

An Introduction to CP/M Features and Facilities. Digital Research Inc., Pacific Grove, California, 1978.

ED: A Context Editor for the CP/M Disk System - User's Guide. Digital Research Inc., Pacific Grove, California, 1978.

CP/M Assembler (ASM) User's Guide. Digital Research Inc., Pacific Grove, California, 1978.

CP/M Dynamic Debugging Tool (DDT) User's Guide. Digital Research Inc., Pacific Grove, California, 1978.

PL/I-80 Reference Manual. Digital Research Inc., Pacific Grove, California, 1980.

PL/I-80 Applications Guide. Digital Research Inc., Pacific Grove, California, 1980.

Link-80 Operator's Guide. Digital Research Inc., Pacific Grove, California, 1980.

MP/M Operating System - System Implementor's Guide. Digital Research Inc., Pacific Grove, California, 1981.

XLT86 User's Guide. Digital Research Inc., Pacific Grove, California, 1981.

Everest G.C. and Lawrence C.T. Comparative Study of Database Management Systems on Microcomputers, **Proceedings ACM SIGMOD Workshop on Small Database Systems**, Orlando, Florida, October 13-15, 1981, pp 77-89.

Gagle M., Koehler G.J., and Whinston A. Data-Base Management Systems: Powerful Newcomers to Microcomputers, **Byte**, Vol 6, No 11 (November 1981), pp 97-122.

Gerritson, R. A Preliminary System for the Design of DBTG Data Structures. **Commun. ACM**, Vol 18, No 10 (October 1975), pp 551-557.

Gilbreath, J. A High Level Language Benchmark. **Byte**, Vol 6, No 9 (September 1981), pp 180-198.

Greuner, G. Simulating CP/M on a mini pays off in available software. **Electronic Design**, (July 8, 1982).

Harris, L.R. The Robot System: Natural language processing applied to data base query, **Proceedings ACM Annual Conference**, 1978, pp 165-172.

Hogan, T. **CP/M User's Guide.** Osborne/McGraw-Hill, Berkeley, California, 1981.

Holsapple C. **A Primer on Data Base Management Systems.** Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

Jensen, K. and Wirth, N. **Pascal User Manual and Report**, 2nd Edition. Springer-Verlag, New York, 1975.

Jerold-Kaplan, S. On the difference between natural language and high level query languages, **Proceedings ACM Annual Conference**, 1978, pp 27-38.

Kernighan, B.W. and Ritchie, D.M. **The C Programming Language.** Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.

Knuth, D.E. **The Art of Computer Programming,** Vol 3, Sorting and Searching. Addison-Wesley, Reading, Massachusetts, 1973.

Landgarten, H. MDBS, Part I: The Database Manager. **Lifelines,** Vol 2, No 2 (July 1981), pp 10-15.

Landgarten, H. MDBS, Part II: QRS. **Lifelines,** Vol 2, No 4 (September 1981), pp 9-14.

Lehman, P.L. and Yao, S.B. Efficient Locking for Concurrent Operations on B-Trees. **ACM Trans. Database Syst.,** Vol 6, No 4 (December 1981), pp 650-670.

Ling, T., Tompa, F.W., and Kaneda, T. An improved third normal form for relational databases. **ACM Trans. Database Syst.,** Vol 6, No 2 (June 1981), pp 329-346.

Machrane, B. A Review of MDBS. **Microsystems,** Vol 3, No 3 (May/June 1982), pp 28-33.

Martin J. **Computer Data-Base Organization,** 2nd Edition, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.

Martin, W.A. Some comments on EQS, a near term natural language database query system, **Proceedings ACM Annual Conference,** 1978, pp 156-164.

MDBS.DDL User's Manual. Micro Data Base Systems Inc., Lafayette, Indiana, 1979.

MDBS.DMS User's Manual. Micro Data Base Systems Inc., Lafayette, Indiana, 1979.

MDBS.RTL - Recovery/Transaction Logging System. Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

MDBS.DRS - Dynamic Restructuring System. Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

MDBS - CP/M Machine Language Interface Notes. Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

MDBS - PL/I Interface Notes. Micro Data Base Systems Inc., Lafayette, Indiana, 1980.

MDBS.QRS - Query System/Report Writer, Version 1.02. Micro Data Base Systems Inc., Lafayette, Indiana, 1981.

MDBS III User's Manual. Micro Data Base Systems Inc., Lafayette, Indiana, 1981.

Metz, S.J. Correspondence. **ACM SIGPLAN Notices**, Vol 17, No 2 (February 1982).

Micro-64 System Documentation. Microcom (Pty) Ltd, Cape Town, 1981.

Pfaltz, J.L. **Computer Data Structures.** McGraw-Hill, New York, 1977.

Reisner, P. Human Factors Studies of Database Query Languages: A Survey and Assessment, **ACM Computing Surveys**, Vol 13, No 1 (March 1980).

Ritchie, D.M. and Thompson, K. The UNIX Timesharing System. **Commun. ACM**, Vol 17, No 7 (July 1974), pp 365-375.

Schach, S.R. A portable trace for the Pascal heap. **Software - Practice and Experience**, Vol 10, No 6 (June 1980), pp 421-426.

Schneidermann, B. **Software Psychology: Human Factors in Computer and Information Systems**, Winthrop, Cambridge, 1980.

Sperry Univac. Data Management System (DMS 1100), Level 8R1, Data Adminstrator Reference. Sperry Univac, St. Pauls, Minnesota, 1978.

Sperry Univac. Query Language Processor (QLP 1100), User Reference Manual UP-8615. Sperry Univac, St. Pauls, Minnesota, 1979.

Stachour, P.D. Correspondence. **ACM SIGPLAN Notices**, Vol 17, No 3 (March 1982).

Tsou, D.M. and Fischer, P.C. Decomposition of a relational scheme into Boyce-Codd normal form. **Proceedings ACM Annual Conference**, 1980, pp 411-417.

Ullman, J.D. **Principles of Database Systems**, Computer Science Press, Rockville, Maryland, 1980.

Welty, C. and Stemple, D.W. Human Factors Comparison of a Procedural and a Nonprocedural Query Language. **ACM Trans. Database Syst.**, Vol 6, No 4 (December 1981), pp 626-649.

Zaks, R. **The CP/M Handbook With MP/M.** Sybex, Berkeley, 1980.

Zloof, M.M. Query-by-Example, **AFIPS Conference Proceedings**, Vol 44, 1975, pp 431-438.

Zloof, M.M. Design Aspects of the Query-By-Example Data Base Management Language. In **Databases: Improving Usability and Responsiveness** (ed., B. Schneidermann). Academic Press, New York, 1978.