DESIGN AND IMPLEMENTATION OF A RELATIONAL DATA BASE SYSTEM FOR A MINICOMPUTER

by

Sue Marie Thompson Dintelman

UUCS-77-108

A thesis submitted to the faculty of the
University of Utah in partial fulfillment of the requirements
for the degree of

Master of Science

Department of Computer Science

University of Utah

Summer 1977

ABSTRACT

A data base system provides the advantages of centralized control of data including increased data independence. Design specifications for a low level relational data base interface are given in the form of a formal description which separates the implementation details from the description of the functions making up the interface. The formal description may be used by both the user and implementer of the system.

The minicomputer implementation of the data base system described is intended to be used directly by application programmers or as a base on which to build a higher level interface. The functions making up the system are FORTRAN callable subroutines making it convenient for minicomputer application programs to use the system. A method of utilizing the low level interface to implement a higher level interface based on the relational algebra is outlined.

ACKNOWLEDGMENTS

I wish to thank my advisor and committee chairman, Dr. John M. Smith, for his invaluable comments, criticisms, encouragement and patience throughout this project. Also, I wish to acknowledge the Computer Aided Design Laboratory of the Graduate School of Architecture which provided machine support. In addition I express my appreciation to Professor Steven Gregory of the Graduate School of Architecture who suggested the project and whose comments greatly improved the final version of the implementation.

I also wish to thank the other members of my committee, Dr. Martin Newell and Professor Roy Keir for their help in preparing the final thesis draft. And finally, a special thank you to my husband, Bob, for his constant understanding and support.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

This thesis describes the design and implementation of a data base system for a minicomputer. A _data base system_ provides centralized control of data by handling access to the physical data through a common _data base interface_. Most minicomputer systems have no data management facilities beyond a simple file system. In file systems, application programs allocate and maintain their own set of files with little sharing of data and there is usually no centralized control of data. The difference between a file system and a data base system is illustrated in Figure 1. The centralized control in a data base system offers several advantages including i) increased data independence, ii) reduced redundancy and iii) more easily maintained consistency. Each advantage is discussed separately below.

i) _Data independence_. If a program uses data stored on secondary storage and if changes in the format of the stored data require program changes then the program is said to be _data dependent_. The common interface allows programs to be written which are more _data independent_. Programs may access data without regard to the physical properties of the data and therefore are unaffected by changes in the data format, location or storage structure.

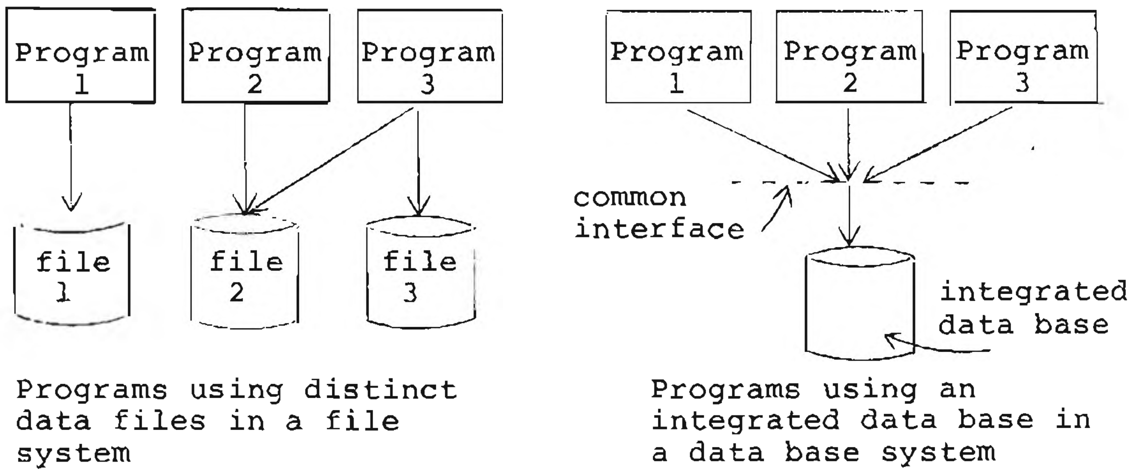

Program 1
Program 2
Program 3
file 1
file 2
file 3
Programs using distinct data files in a file system
Program 1
Program 2
Program 3
common interface
integrated data base
Programs using an integrated data base in a data base system

Figure 1. File System vs. Data Base System

ii) Reduced redundancy. In many file systems each application program has its own set of stored data. These data sets often contain some of the same information. This common data is stored with each program using it, causing wasted storage space. With a data base system redundancy is reduced because programs can share the common data and eliminate the need for duplicate copies.

iii) More easily maintained consistency. When a piece of information is stored in more than one location and a change is required in the information, there is a period of time during which the separate entries will be inconsistent. If redundancy is eliminated (as in a data base) and changes have to be made only once, the problems of inconsistency are reduced.

1.1 The Relational Model

The data in the data base must be structured so the user has some way to work with it. A _data model_ provides this structure. A data model is the information content of the data base as it is represented by the common data base interface. It is the use of a high level data model which gives a data base system most data independence.

The relational data model was developed by E. F. Codd (1970) and is a relatively concise, high level way in which to represent a user's interpretation of his data. The relational model was chosen for this implementation because i) it allows a high degree of data independence, ii) the tabular structure used to represent the data is easy to understand and iii) the model allows simple updating of data. The following discussion briefly introduces the relational model. More detailed information about the relational model and its advantages may be found in Chamberlin (1976).

A _relation_ may be represented by a table in which information is organized into rows and columns. An example of a relation is given in Figure 2. The rows are called _tuples_. The columns of a relation are given names and are called _domains_. The domains of the relation PARTS are P#, PNAME, COLOR and WEIGHT. Two additional properties of relations are that i) the ordering of the rows is not significant and ii) there are no duplicate rows. The second property implies that in each relation there is some domain (or combination of domains) which uniquely determines a tuple. This domain(s) is called the primary key. In the relation of Figure 2 the primary key is P#. Because the primary key uniquely determines a tuple the primary key may be used to determine if a new tuple is a duplicate. That is, a tuple may not be inserted into a relation if its primary key has the same value as the primary key of an existing tuple. The relational model consists of a collection of relations containing the information needed by a user of the data base.

PARTS

P#	PNAME	COLOR	WEIGHT
P1	Nut	Red	12
P2	Bolt	Green	17
P3	Nut	Blue	17
P4	Screw	Red	14

Figure 2. The Relation PARTS

Consider the relational model in Figure 3. This is a data model representing a warehouse inventory. The S# and P# domains of the SUPPLIER-PART relation correspond to the primary key domains of the SUPPLIER and PART relations. The SUPPLIER-PART relation represents an association between the two entities - suppliers and the parts which they supply. It is a property of the relational model that there is no difference between the representation of the entities in a data base and the relationships between these entities.

SUPPLIER

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Blake	30	Paris
S3	Jones	10	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SUPPLIER-PART

S#	P#	QTY
S1	P1	3
S1	P2	2
S1	P3	4
S1	P4	2
S2	P1	3
S2	P2	4
S2	P4	6
S3	P3	4
S4	P2	3
S4	P4	2
S5	P2	5

PARTS

P#	PNAME	COLOR	WEIGHT
P1	Nut	Red	12
P2	Bolt	Green	17
P3	Nut	Blue	17
P4	Screw	Red	14

Figure 3. A Warehouse Inventory Data Model

The development of suitable relational models is a topic of current research and will not be dealt with here.

1.2 The Data Base Interface

The user of a data base requests information based on the data model. However, the data base must be accessed using storage details of the data. It is the job of the data base interface to translate between the data model and the storage structure of the data.

As shown in Figure 4, a data base interface may consist of more than one level. At the level closest to the data base is the low level interface which actually accesses and changes information in the data base. This interface deals mainly with one tuple of one relation at a time. Although this type of interface is useful for some applications by itself, it also provides a foundation on which to build higher levels. Higher level interfaces can accept more complex queries and enforce integrity constraints spanning more than one relation, and allow alternate views and concurrent users. XRM (Lorie 1972, 1974) is an example of a low level interface which has been used to implement higher level interfaces (Astrahan 1975). The leveled approach is also used in other documented systems such as INGRES (Stonebraker 1976), ZETA (Czarnik 1975) and RISS (McLeod 1975).

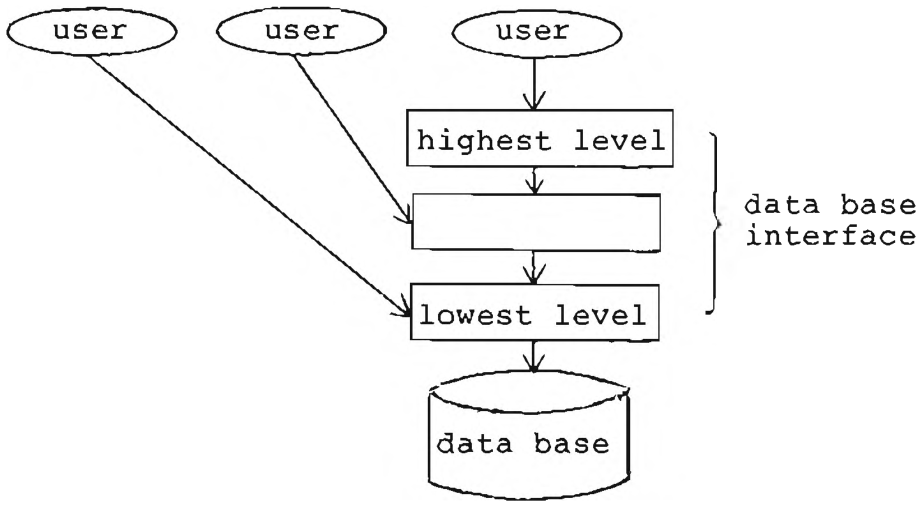


Figure 4. The Data Base Interface

The interface implemented as part of this thesis is a low level interface and was designed to provide a convenient way to access the data base and to provide a basis for developing higher level modules. The interface allows the user to create and destroy relations; to retrieve, insert, update and delete tuples from relations and to check stored information about the relations. All information about the physical location of the data is isolated at this level so that storage details of relations may be changed without affecting higher levels.

1.3 Overview of the Thesis

The remainder of the thesis describes the design, implementation and use of the data base system. Chapter 2 contains a more detailed explanation of the structure of the data base and the data base interface. This chapter also describes a formal description of the data base interface. This formal description contains as few of the implementation details as possible and makes few assumptions about the type of usage that will be made of the interface. The formal description is designed to serve as a common ground for discussion between the user and implementer of the data base system as illustrated in Figure 5. It is essentially the design specifications for the interface.

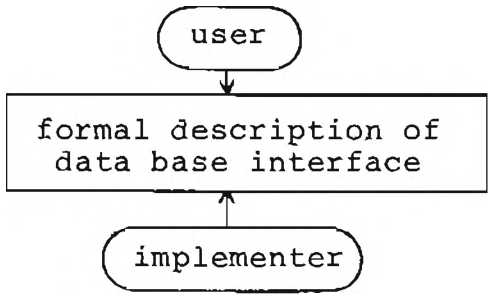


Figure 5. Role of the Formal Description

Chapter 3 concentrates on an implementation of the data base system described in Chapter 2. The details are of an implementation done as part of an ongoing project of the Computer Aided Design Laboratory, Graduate School of Architecture, University of Utah. Various design decisions are analyzed with the reasons for particular choices discussed.

The use of the data base system is emphasized in Chapter 4. The data base system is currently being used to implement an architectural design application, Building Design with an Integrated Relational Data Base (BIRD) (Gregory, 1977). This chapter describes the use of the interface directly by application programs and as a tool to implement some higher level constructs. Chapter 4 also presents notes on extensions to the system planned as part of the BIRD project.

The complete formal specification described in Chapter 2 is included as Appendix A. Appendix B contains the data base system Users' Manual.

CHAPTER II

DESIGN OF THE DATA BASE SYSTEM

This chapter describes the data base and its low level interface. In order to facilitate the development of similar systems on other machines using other host languages, this chapter contains as few of the implementation details as possible. The subroutines making up the interface are called the access method (AM) functions. A formal specification for the AM functions is given to further separate the implementation details from the description of the AM functions.

2.1 The Data Base

The data base consists of <u>user relations</u> created and maintained by the user. If programs using these relations are to be data independent the information about the storage details of the relations must be kept with the relations rather than in the programs. If this information is stored in relational form all the data base information will be in the form of relations and may be treated in a uniform way. In this implementation there are <u>Master relations</u> stored as part of the data base containing information about the individual user relations. To insure that the Master relations accurately reflect the contents of the data base they are maintained by the system and may not be changed directly by the user. They are changed automatically as a result of calling the AM functions. The information in the Master relations is available directly on a read-only basis.

2.2 The AM Functions

The set of AM functions may be divided into three groups as shown in Figure 6: i) the functions which involve the data base as a whole, ii) those involving single relations and iii) those involving individual tuples of a relation. Within each group there are functions which do not alter the contents of the data base, these are labeled read-only functions in Figure 6. The other functions when called, change the contents of the data base, either the user relations or the Master relations, in some way and are labeled as update functions.

Data Base Functions	Relation Functions	Tuple Functions	
BEGIN	SETSCAN	GET	
END	RTRV_REL_INFO	GETNEXT	Read-only
RTRV_REL	RTRV_DOM_INFO	FINDTID	Functions
-------	-------	-------	
	CREATE	INSERT	
	DESTROY	DELETE	Update
		REPLACE	Functions

Figure 6. AM Functions

BEGIN and END are used to start and finish a session. A session consists of all data base activities between and including the BEGIN and END function calls. Higher level signon/signoff procedures incorporating security checks may be developed using these functions. CREATE and DESTROY define and delete relations. GET, DELETE, REPLACE and INSERT are used to manipulate the tuples of a relation. Each tuple in the data base is identified by a unique number. This tuple identifier, the TID, is determined by the system at the time a tuple is inserted into a relation. GET, DELETE, REPLACE and INSERT access individual tuples by their TID's.

There are many instances when a tuple (or set of tuples) must be retrieved based on the value of a particular domain and the TID is not known. In this case a systematic search of the relation must be made in order to locate the desired tuples. Associated with a stored relation are one or more traversal schemes. Each traversal scheme provides a linear ordering of all the tuples in a relation so that, given the TID of a tuple and a traversal scheme, there is a unique next tuple.

The SETSCAN and GETNEXT functions provide the means to search relations for specific tuples. Using a property of the desired tuple or tuples, SETSCAN determines the traversal scheme which will limit the number of tuples to be searched as much as possible. The selected traversal scheme and other information depending on the storage structure is encoded into a traversal code as will be explained in Section 3.3. SETSCAN returns this initial traversal code as well as the TID's of the first and last tuples in the selected linear ordering which may fit the search criteria.

GETNEXT is used with the first tuple's TID and the traversal code to retrieve the first tuple, the TID of the next tuple in the relation and an updated traversal code. The tuple returned by GETNEXT must always be checked to see if it satisfies the search criteria because the scanning returns possible matches, not guaranteed matches. The TID and traversal code returned by GETNEXT are then used for another call to GETNEXT. The user program continues to call GETNEXT in this manner until either a desired tuple or set of tuples is found or until the last TID returned by SETSCAN is used to retrieve a tuple. These two AM functions provide the means to answer general queries about the tuples in a relation rather than retrieving a specific tuple based on its TID.

For the special case where the value of a tuple's primary key is known, the function FINDTID may be used to determine the tuple's TID. FINDTID uses the same procedure described above to limit the search by choosing a traversal scheme and checking successive tuples. FINDTID handles the search details internally and returns the TID of the desired tuple if it exists. This function is not required as SETSCAN and GETNEXT could be used to find the required

TID, but it is included because it provides a more convenient way to map from the primary key to the TID. FINDTID cannot replace the SETSCAN and GETNEXT functions because it handles only the specific case where the search is for a primary key value and it is known in advance that the search will always result in at most one match.

The other AM functions are used to interrogate the Master relations. RTRV_REL returns a list of all user relations in the data base. RTRV_REL_INFO returns all of the information contained in the Master relations for a specific relation and RTRV_DOM_INFO returns all of the information about the domains of a specific relation.

2.3 The Master Relations

The Master relations contain all the information necessary to access the user relations. To support the existing interface there are two relations, the Relation Information relation and the Domain Information relation.

Although the relation name could be used to identify a relation and to cross reference from the Domain Information relation to the appropriate entry in the Relation Information relation, it is more efficient to use a numeric identifier. The relation name is chosen by the user to make it easier to remember the contents of the relation. The corresponding relation identifier, the RID, is assigned by the system. The RID is then used with the AM functions to more efficiently reference the relations of the data base. The RID does not change during the lifetime of the relation, but when a relation is destroyed its RID may be reassigned to another relation.

The Relation Information relation contains the relation name, the RID, the number of domains in the relation, the storage structure of the relation, its physical location and the length of tuples and other information depending on the storage structure of the relation. Figure 7 summarizes this list of information. The AM function RTRV_REL retrieves from the Relation Information relation a directory consisting of all the relation names and their corresponding RID's. RTRV_REL_INFO may be called to retrieve the tuple of the Relation Information relation corresponding to a particular RID.

RELNAME	Relation name
RID	Internal relation identifier
TUPLE_LENGTH	Length of the tuple
STORAGE_STRUCT	Storage structure of the relation
PLOC	Physical location of the relation
STRUCT_INFO	Other necessary information depending on storage structure

A. Relation Information Relation Domains

DOM_NAME	Domain name
DID	Internal domain identifier
RID	Relation identifier of relation containing this domain
DOM_TYPE	Data type (numeric, character)
DOM_LENGTH	Length of the domain
DOM_LOC	Location of domain in tuple
DOM_KEY	Coded primary key information

B. Domain Information Relation Domains

Figure 7. Master Relations

The Domain Information relation contains a tuple for each domain of each relation. As shown in Figure 7, each tuple contains the domain name for the convenience of the user, a numeric domain identifier for use by the system and the RID of the relation to which the domain belongs. The tuples of the Domain Information relation also contain domains to identify the data type, length and location of each domain in the user relation tuples. It is also necessary to identify the domains that are part of the primary key.

The REPLACE and INSERT functions both use the primary key information in the Domain Information relation. As explained in Section 1.1 each primary key value is unique and before inserting a new tuple a check must be made to insure that there is not an existing tuple with the same primary key value. The REPLACE function will not allow altering of the primary key domains because such alteration is in fact creating a new tuple. Other information in the Domain Information relation is used by the SETSCAN function to determine an efficient traversal scheme and to determine if the storage structure of a relation is one in which a scan might be limited for some particular domain value, i.e., a relation that is sorted on a particular domain.

This is all the information about relations and their domains necessary for this interface. Since higher level interfaces may need additional information, such as directory information and protection data, the use of a relational structure for the Master relations makes it easy to add either additional domains or additional relations to contain this information.

2.4 The Tuple Identifiers

The tuple identifier (TID) of a tuple is related to the physical address of a tuple and is therefore determined by the system and not the user. Since the TID is related to a physical location, if a tuple is moved its TID is changed. In this system there is no tuple reorganization during a session so the TID of a tuple remains unchanged for the duration of a session. This allows user programs to repeatedly reference tuples by their TID's and therefore take advantage of the efficient GET, REPLACE, INSERT and DELETE functions without having to relocate the tuples using the SETSCAN and GETNEXT functions or the FINDTID function.

It would be impractical to maintain the same TID's indefinitely however, because eventually some type of garbage collection will need to be done and tuples will need to be moved. Therefore no guarantee is made for the constancy of TID's from session to session. This assumption allows relations to be reorganized between sessions not only to do necessary garbage collection, but also to optimize different application programs.

Because the TID is not necessarily the same from session to session it does not replace the primary key as the unique identifier for a tuple. The TID is a convenience for use within a session and must be re-established each session using the SETSCAN and GETNEXT functions or the

FINDTID function.

2.5 The Formal Description

The formal description of the AM functions is based on a specification technique suggested by D. L. Parnas (1972). The purpose of this technique is to provide complete information to both the user and the implementer of the interface without giving unnecessary details. This requirement that the specification contain as little extraneous information as possible is the minimality requirement for specification techniques described in (Liskov 1975). Because this technique does produce fairly minimal specifications it is applicable to the AM function package. An important design goal was to isolate all knowledge of the storage structure of relations to the AM function level. This means that user programs need not be aware of changes made in the actual storage of data. Also, since several different programs will be calling the AM functions, it is important not to assume anything about the calling programs when implementing the functions. The specification, therefore contains information on what the interface does and not on how it is achieved nor on how it should be used.

The complete formal specification is included as Appendix A. The specification of each function consists of a description of the input and output parameters, a description of the effect of the function and a list of exceptions (error conditions) that may occur. The effect of a function, (i.e. the changes a function makes to the data base) is given in terms of the other functions in the AM function package. The data base may be accessed only by calling other AM functions. Some of the functions have no effect. This means there is no way to tell that the function has been called and it may be called repeatedly with no visible effect except the passage of time. This type of function will most likely be interrogating the data base without making changes.

If a function has an effect on the data base the relationship between the input and output parameters is contained in the description of the effect of the function. For example the effect of CREATE is to add relation and domain information to the data base Master relations. This effect on the data base is expressed by using the output parameter, RID, to retrieve the stored information. If a function has no effect on the data base the relationship between the input and output parameters is included as part of the description of the output parameters. Calling the function SETSCAN has no effect on the data base and the only reason for calling SETSCAN is to determine values for the output parameters FIRST_TID, LAST_TID and TC (traversal code). In order for the user to know what to expect from this and other functions which have no effect and in order for the implementer to know how to determine the output values, the relationship between the input and output parameters must be explicitly stated as part of the specification. The style used to express these relationships is adapted from that used in (Guttag 1976) for the axioms describing the semantics of operations defined for data types.

No error handling is included in any of the functions. Rather, the exceptions are "trapped" and reported to the calling program. This allows each calling program to handle exceptions in a manner suitable for the type of application. If an exception occurs control is returned to the user and the user may assume the function has had no effect. The only type of exception for which this assumption may not hold is an unrecoverable I/O error. In the event that this type of error occurs when reading information from the data base, a "Type 1 I/O error condition" is returned. The user may assume there has been no effect on the data base but some data transfer from secondary to primary storage may have occured. If this type of error occurs when writing it is possible that information from primary storage has only been partially written in the data base and a "Type 2 I/O error condition" is returned. Recovery from I/O errors is discussed as part of the implementation in Chapter 3. In all other cases (including Type 0 I/O errors, where there has been no data transfer) the function may be called again after the reason for the exception has been corrected and it will be as if there had been no previous call.

In addition to the formal specification a natural language explanation of each function is included to aid both the user and implementer in the use of the function and to serve as documentation for the formal description.

The notation uses PASCAL-like specifications for the array and record data types used. Capitalized names are used for the function names, parameters and global variables. Lower case letters, such as "i", are used as free variables. Although names were chosen to have as much intuitive meaning as possible to make the specification easier to read, names are explained when they first occur or where it seemed necessary to prevent confusion. Symbols used in the notation include "$\leftarrow$" which is used as the assignment operator and "=" which indicates the equality relation. The symbol "$\exists$" is read "there exists", "$\ni$" is read "such that" and "$\forall$" is read "for all".

Because all the functions are specified in the same style only the INSERT and GET functions from Appendix A are discussed in detail here. The function INSERT is called to insert a tuple into a specified relation. The input parameters are the relation identifier, RID, and the tuple. The tuple identifier, TID, of the newly inserted tuple is returned as an output parameter. FINDTID is used in the expression of the effect of INSERT because the TID is only guaranteed to remain the same during a session but the tuple will still be in the data base even if the TID changes. So although the TID may be used to reference a tuple during the session in which it is inserted, the function FINDTID is used to represent the current TID of the tuple so the effect of INSERT is expressed independently of the session. The effect of inserting a tuple is to allow the TID of the tuple to be referenced without error (VAL_TID(RID,FINDTID(RID,TUPLE)) = true) and the tuple may be retrieved using the current tuple identifier (GET(RID,FINDTID(RID,TUPLE)) = TUPLE).

Calling INSERT before the session has been started by calling BEGIN (DBINIT = false) is one exception which may occur. Other exceptions include using an invalid RID, inserting a tuple with a blank in a primary key domain and trying to insert a duplicate tuple. And because the INSERT function causes information to be written to the data base as well as read from it all three types of I/O errors may occur.

The function GET is used to retrieve tuples from the data base using the current TID. The values of the relation identifier and tuple identifier are input to the function and the output is the desired tuple. The fact that the tuple returned is the one with the given TID is expressed using the FINDTID function, that is, calling FINDTID with the returned tuple will yield the input TID. The exceptions that may occur when using the GET function are calling GET before calling BEGIN (DBINIT = false), using an invalid RID or TID, and the I/O exceptions associated with reading information from the data base.

CHAPTER III

IMPLEMENTATION OF THE DATA BASE SYSTEM

The previous chapter presented a formal description of the AM functions to be used by both the user and the implementer. The chapter contains a description of one possible implementation of the data base and the AM functions.

3.1 System Architecture

The data base system has been implemented on an INTERDATA 70 minicomputer. As shown in Figure 8, the present hardware configuration includes 64K bytes of memory, a teletype, a paper tape reader/punch, line printer, a single disk drive and a Computek graphics terminal with data tablet.

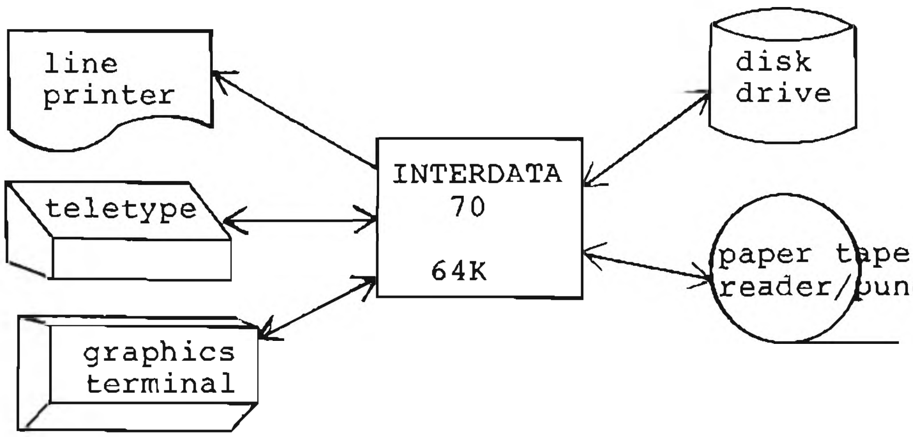


Figure 8. System Architecture

Provided with the machine is a Disk Operating System (DOS) and the system programs necessary to run assembly language and FORTRAN language programs.

3.2 Implementation of the AM Functions

The AM functions have been designed as a set of FORTRAN callable subroutines. FORTRAN was used as the host language because it is the programming language most readily available to minicomputer users. To use the data base system a programmer does not need to learn a new language, but only how to use a package of FORTRAN callable subroutines. Also, by using FORTRAN as the host language and as the language to implement many of the subroutines the system becomes more portable because of the standardization and wide availability of FORTRAN.

Figure 9 shows the set of subroutines making up the data base interface. LEVEL 1 subroutines are the AM functions that may be called directly by the user and are used to access and manipulate the data base. LEVEL 2 subroutines are those necessary for a complete interface specification but are not user callable. These subroutines are used mainly to check for exceptions. In this implementation many of the functions used in the specification to check for exceptions were not implemented as subprograms but as in-line code and are therefore not individual LEVEL 2 subroutines. LEVEL 3 subroutines are the service routines which perform specific tasks for some of the LEVEL 1 subroutines. The descriptions of the LEVEL 3 subroutines are found in Appendix B as part of the Users' Manual. The I/O routine is used to actually read and write data base <u>pages</u> (physical records). The I/O routine is an assembly language routine and is the only subroutine not coded in FORTRAN.

	BEGIN	INSERT
	END	GET
	CREATE	REPLACE
LEVEL 1	DESTROY	DELETE
	SETSCAN	RTRV_REL
	GETNEXT	RTRV_REL_INFO
	FINDTID	RTRV_DOM_INFO
LEVEL 2	VAL_RID	VAL_TID
	LOCATE	GET_SPACE
	GET_BUF	FREE_SPACE
	CHECK_BUF	PRIMARY_KEY_FINDER
LEVEL 3	SEARCH_INDEX	SAVE_REL_DATA
	SAVE_BUF	WRITE_REL_DATA
	WRITE_BUF	DUMP_MASTER_REL
	DUMP_BUF	
I/O	IOSVC1	

Figure 9. The AM Function Package

<u>3.3 Implementation of the Data Base</u>

Space for a data base is allocated, using DOS, on a disk pack as one large direct access file with physical records of the maximum size of 256 bytes. Once this <u>data base file</u> has been created there is no need to use the operating system to reserve and release space and FORTRAN subroutines may be used to manage the space within the file. The maximum record size is used as this gives the most efficient data transfer rate. The first <u>pages</u> (physical records) of a data base file are reserved for the Master relations. The remaining pages are available for storing user relations.

3.3.1 Implementation of the Master Relations

The Relation Information relation has been implemented as two relations. The relation name is not used by the low level interface, with the exception of RTRV_REL, and is included only for the convenience of the user. The relation names and corresponding RID's are maintained in one relation, the Relation Index. The remaining information is maintained as the Relation Data relation. In Figure 10 the Master relation domains utilized by the AM functions are denoted by an "X". All of the AM functions require information from the Relation Data relation. This relation is kept in primary storage during a data base session to eliminate at least one disk access per AM function call. When information in the relation is changed, however, it is written immediately to secondary storage so there is no discrepancy between the copy in the data base and the copy in main memory.

The relation identifier, RID, in this implementation is the offset of the relation's tuple in the two relations discussed above. The RID may be used to index into the relations to locate any of the desired information. This is especially efficient since the relations when in main memory are stored as arrays and the RID may be used directly as a subscript.

As shown in Figure 10 the Domain Information relation

		GET	DELETE	REPLACE	INSERT	SETSCAN	GETNEXT
Relation Index	RELNAME						
	RID						
Relation Information	TUPLE_LENGTH	X		X	X		X
	STORAGE_STRUCT	X	X	X	X	X	X
	PLOC	X	X	X	X		X
	STRUCT_INFO	X	X	X	X	X	X
Domain Information	DOM_NAME						
	DID					X	
	RID						
	DOM_TYPE					X	
	DOM_LENGTH			X	X	X	
	DOM_LOC			X	X	X	
	DOM_KEY			X	X	X	

Figure 10. Use of the Master Relations

is not used by every AM function. Partially for this reason, but mainly because it is a much larger relation than the Relation Data relation, it is not maintained in main memory. Information for the desired domains is read as necessary.

3.3.2 Storing User Relations

By using the AM functions the users of the data base are not aware of the physical storage structure of relations. There must, however, be a data base administrator who is concerned with the details of storing the information. The data base administrator (this may be a committee) must consider the usage of relations and determine a storage structure for each relation which will provide efficient access to the information. The storage structure of a relation may be changed by restructuring the relation using the utility functions to be discussed in Section 3.7. Storage structures of relations might be changed to improve response time for an application or they might be changed from application to application as the use of the information changes.

There are several alternatives available for storing the tuples of a relation, not all of which have been implemented. Among these alternatives are sequential, hashed and linked storage structures. When a hashing scheme is not used, access to tuples may be improved with the addition of indexes. A primary index entry contains a primary key value and the location of the tuple associated with the value. _Dense_ primary indexes contain an entry for each tuple of the relation. _Nondense_ primary indexes do not contain an entry for each tuple, but take advantage of the physical sequence of stored tuples. For example, this type of index may be used to provide access to sorted sequential structures. The index would contain only the entries for the tuples with the highest valued key on each track, cylinder, or other partition of a relation.

Secondary indexes are indexes for domains other than the primary key domains and provide alternate access paths which can improve response for some queries. For a large index additional levels of index may be created forming a _multilevel_ index which will decrease the number of comparisons necessary to find a specific entry. The benefits and disadvantages of these and other storage techniques are discussed in (Knuth 1968, Held 1975, London 1973) and only the specific structures currently available with this system are discussed below. Other structures may be added to the system by the data base administrator. Because of the data independence provided by the AM functions, these additions will not require changes in the programs using the AM functions. The AM functions and the data base have been designed to make the addition of storage structures as simple as possible.

Currently with this system, user relations may be stored using one of two storage structures. These structures are based on those used in the INGRES relational data base system (Stonebraker 1976). One of the available structures is an unordered sequential structure in which tuples appear sequentially in no particular order. This structure was used mainly to provide a simple structure in which to store relations during the development of the AM functions. This structure may still be used for small temporary relations or relations which must always be exhaustively searched. The other structure is an indexed sequential one in which tuples are stored sorted on a specified key. Figure 11 shows an example of a relation stored using this structure where the key is the supplier name. An index is provided giving the highest key value of a tuple on each physical record, or page. For large relations spanning several pages an index to the index is added to decrease the number of disk accesses necessary to reach a desired tuple. This type of storage structure is useful for storing relations for which anticipated queries will specify the key value as falling within a range of values.

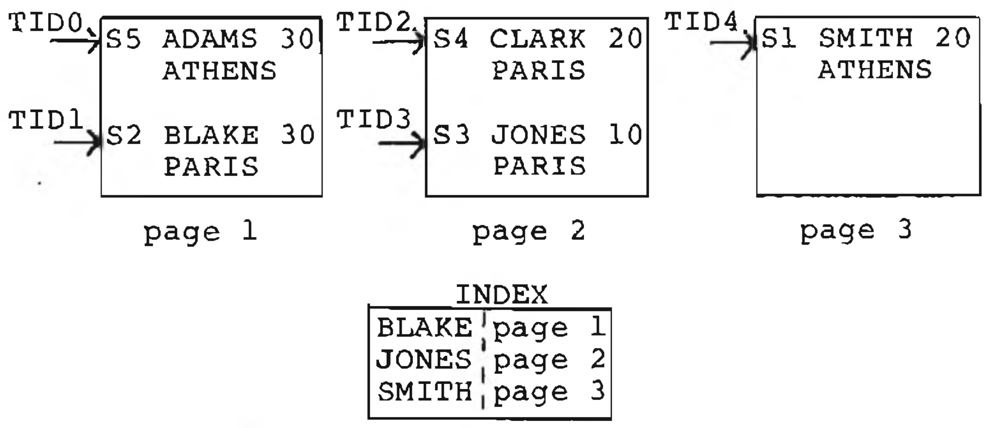


Figure 11. SUPPLIER Relation Stored Using the Indexed Sequential Structure

3.4 Accessing Tuples

Tuples may be accessed directly by their TID's because a TID is interpreted as a physical address. Tuples may also be accessed when their TID's are not known but when they are to be selected based on some property of their domains.

3.4.1. Accessing Tuples Using a Search Criterion

The search criterion is the property which specifies the desired tuples. Figure 12 shows the four forms of the search criterion allowed by the AM functions where RELATION may be any of the operators =, <, >, ≤, ≥ or ≠. The first case is the only form of the search criterion allowed by FINDTID. All four cases are handled by the SETSCAN operator.

```
(1) primary key = value
(2) primary key RELATION value
(3) single domain RELATION value
(4) single domain RELATION single domain
```

Figure 12. Allowable Forms for the Search Criterion

The operands of the search criterion are limited to a single domain unless the primary key consists of more than one domain, in which case the domains are concatenated and treated as a single operand for the purpose of searching. The use of the concatenation of non-key domains was not included in the SETSCAN function because multicriteria searches may be done by first calling SETSCAN with a single criterion and then checking the candidate tuples as they are retrieved by GETNEXT for compliance with all criteria. This allows for the development of a higher level module which may analyze multicriteria searches and then, using information about the stored relation, may choose the single criterion which will limit the number of tuples to check as much as possible.

The search criterion and the storage structure of a relation are used to determine (1) a traversal scheme for scanning the relation and (2) the TID's of the first and last tuples which need to be checked. In the case of FINDTID this information is used internally to locate the desired tuple and in the case of SETSCAN is returned to the calling program. Associated with each stored relation is at least one traversal scheme so that a complete scan of the relation may be made without checking any tuple twice.

One traversal scheme, which may be used with any stored relation, orders tuples corresponding to their physical position in the relation. Figure 13 is a representation of the SUPPLIER relation stored using the unordered sequential storage structure. Using the traversal scheme based on physical position the tuples would be scanned in the following order: S5, S2, S1, S3 and S4. Figure 14 is the same relation stored using the indexed sequential structure where the primary key is SNAME. (S# could also have been selected as the primary key in this relation.) The tuples are now ordered by the traversal scheme as S5, S2, S4, S3 and S1.

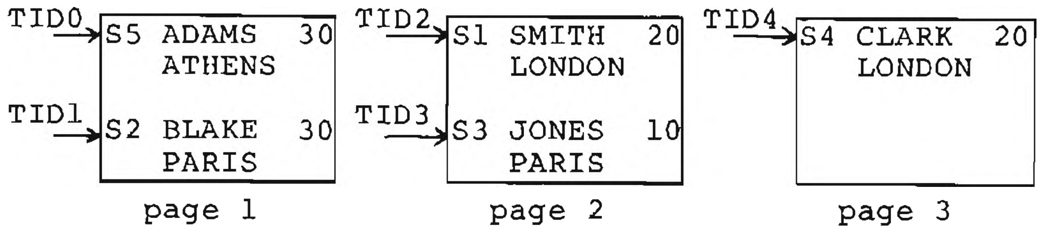
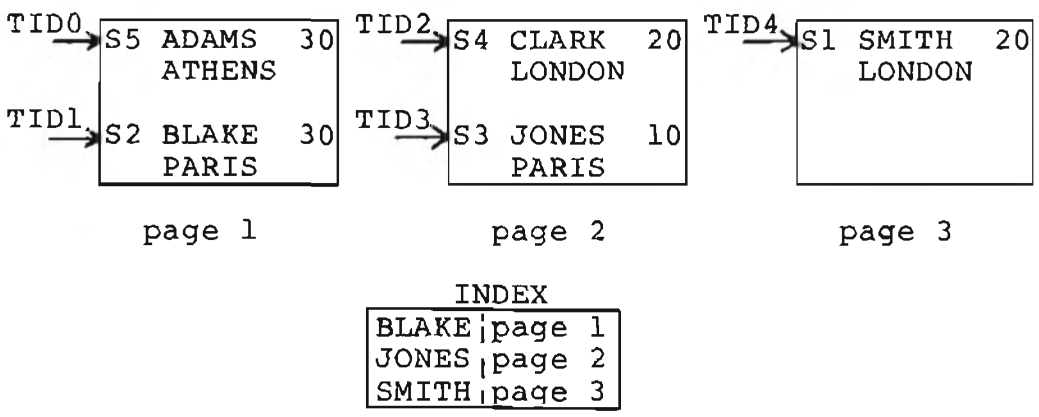


Figure 13. The SUPPLIER Relation Stored Using the Unordered Sequential Structure

Additional traversal schemes may be added to either version of the stored relation by creating additional indexes. These additional indexes will be called directories to prevent confusion with the index associated with the indexed sequential file structure. Directories are not implemented in the current version of the data base system but, because they provide the efficiency of alternate access paths, their inclusion in the system has been anticipated in the current implementation and their use in accessing tuples is discussed in this section. Figure 14B shows a secondary directory for the domain CITY for the stored relation in Figure 14A. The ordering of the tuples indicated by the traversal scheme using the CITY directory is: S5, S4, S1, S2 and S3. Directories may be created for the primary key or any single domain. One type of directory is the type shown in Figure 14B where each directory entry consists of the domain value and the TID's of the tuples corresponding to those values. This type of directory is maintained sorted on the domain value to facilitate lookup.

The algorithm used by FINDTID and SETSCAN to select a traversal scheme will first check Master Relation information to see if a directory is maintained for the operand specified in the search criterion. If there is no directory the traversal scheme based on physical ordering is used. This traversal scheme is also used if the search criterion is a comparison between two domains, since this requires a full scan of all tuples of a relation.



14A. The Relation Stored Using the Indexed Sequential Storage Structure

ATHENS	TID0
LONDON	TID2, TID4
PARIS	TID1, TID3

14B. The CITY Directory

Figure 14. A Stored Version of the SUPPLIER Relation

Once the traversal scheme is chosen the search criterion may be used to limit the number of tuples which must be checked. Consider the SUPPLIER relation stored as in Figure 14 and the following examples.

Example 1. If FINDTID is called using the search criterion "SNAME = JONES" the traversal scheme based on physical ordering is chosen because there is no directory for

SNAME. Because SNAME is the primary key and the storage structure is indexed sequential the index may be used to limit the search to tuples stored on page 2 of the relation. FINDTID will check the tuple with TID2 and will then obtain a match for the tuple with TID3. TID3 is the TID returned as the result of the call.

Example 2. SETSCAN is called using the search criterion "CITY = PARIS". The traversal scheme based on the CITY directory is chosen and the directory is used to set the FIRST_TID as TID1 and the LAST_TID as TID3. The user program calls GETNEXT with the FIRST_TID, TID1, and issues calls to GETNEXT using the NEXT_TID returned by GETNEXT until it has been called using the LAST_TID, TID3. In this example GETNEXT is called twice, first with TID1 and then with TID3. Both the returned tuples need to be checked to see if CITY does equal PARIS and in this case they both do.

Example 3. SETSCAN is called using the search criterion "STATUS = 20". Because there is no directory for STATUS the traversal scheme based on physical location is chosen. STATUS is not the primary key so the entire relation must be scanned. The FIRST_TID is set as TID0 and the LAST_TID as TID4. GETNEXT is called five times by the user program and each returned tuple is checked for STATUS = 20.

The traversal code returned by SETSCAN and GETNEXT contains an encoding of the traversal scheme selected. If the traversal scheme is based on a directory, additional information about the location of the current TID in the directory will be included in the traversal code. Although the TID and traversal scheme uniquely determine the next TID, the directory location will enable GETNEXT to determine the next TID much more efficiently as the current TID will not need to be relocated in the directory for each call to GETNEXT. The traversal code returned by GETNEXT will be updated in this case and will consist of the traversal scheme and the new current location in the directory.

3.4.2 Accessing Tuples Using the Tuple Identifier

The tuple identifier (TID) is used to uniquely identify a tuple. It may be used to retrieve a tuple and must be used when deleting or replacing a tuple. The TID of a tuple is determined by its physical placement in the stored relation.

There are two TID schemes used in the current implementation. The first is used with the unordered sequential storage structure. The TID decodes into a <u>relative page number</u> and a <u>tuple offset</u>. The relative page number is added to the starting address of the relation obtained from the Relation Data relation. This sum indicates the actual page number of the page in the data base which contains the tuple. After the correct page is retrieved from the data base, the tuple offset is used to locate the starting word of the tuple on the page. Given an offset, S, and the tuple length, TL, obtained from the Relation Data relation the starting word, SW, is

$$SW = S * TL + 1.$$

For example, an offset of S = 0 indicates that the tuple is the first on the page and the starting word is

$$0 * TL + 1 = 1.$$

Similar computations will locate the starting word for other tuples. Using the starting word of the tuple and the tuple length the tuple may be retrieved or replaced. A deleted tuple is not immediately removed from the data base but is marked as being deleted and may not be accessed. There is no automatic garbage collection done by the implementation, however, Section 3.7 describes the procedure which may be used by the data base administrator to free space occupied by deleted tuples. Also, Section 3.5 discusses the use of space occupied by deleted tuples when inserting new tuples into a relation.

The second method of using a TID to locate a tuple is used by the indexed sequential storage structure. The TID again decodes into a relative page number and offset. The correct page is determined as above. The offset, however, is not a tuple offset, but the offset in the page directory found on each page. The entry in the page directory gives the starting word address of the tuple on the page. The starting word and length of the tuple are used when retrieving or replacing a tuple. When deleting a tuple the page directory entry as well as the tuple are marked deleted.

Both TID schemes were implemented as each has its advantages. The advantage of the first scheme is that it requires no space overhead. The second method requires the overhead of one half word per tuple, but allows tuples to be reorganized on a page without changing TID's. Because the storage structure of a relation determines which scheme is used for interpreting the TID there is no problem with having more than one scheme.

The current implementation encodes the relative page number and offset of a TID into a single half word (16 bit) integer. Allowing for a maximum of 64 tuples per (256 byte) page requires using 6 bits. The remaining 10 bits are used to denote relative page number. This allows a maximum of 1024 pages per relation which is sufficient for planned application programs. The encoding and decoding of TID's is isolated in all coding, however, so a switch to another scheme for handling the page number and offset could be made.

3.5 Inserting Tuples into User Relations

Inserting a tuple into a relation using the INSERT function consists of three operations. First a check is made to insure that there is no duplicate tuple. Then if there is no duplicate, the position in the stored relation where the new tuple is to be placed must be determined. Finally, based on the position of the new tuple the TID of the tuple is computed to be returned by INSERT to the calling program.

Inserting a tuple into a relation stored using the unordered sequential structure requires a scan of all the tuples in the relation to determine if there is a duplicate.

If a deleted tuple is found on this scan its position is noted. The new tuple is placed in the first available position, that is, if there is a deleted tuple its position is used as the position of the new tuple, otherwise the tuple is placed at the end of the stored relation. The TID of the tuple is computed using the relative page number and the offset of the position chosen for the new tuple.

A complete scan of a relation stored using the indexed sequential structure is not required because the index may be used to locate the page on which a duplicate might be stored. The tuples on the page are checked to determine if there is a duplicate. The position of a deleted tuple (if one exists) is noted during this check. If there is no duplicate the new tuple may be inserted in the position occupied by a deleted tuple. Otherwise it is inserted at the end of the page. In the event the page is full an overflow page is allocated and linked to the primary page as illustrated in Figure 15. The tuple being inserted is placed on the overflow page.

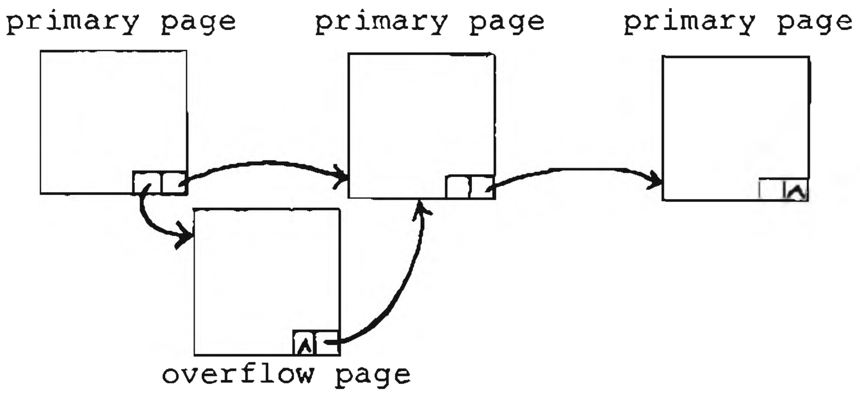


Figure 15. The Chaining of Overflow Pages

Note that a tuple inserted into a relation stored using the indexed sequential structure is probably not in the correct order based on the key of the structure. When a tuple is inserted no reorganization takes place because in some cases this would involve moving a tuple from one page to another, thereby changing its TID. In order to allow constant TID's a primary page and its overflow pages are not maintained in sorted order. This decision implies that the primary page and all of its overflow pages must be included in a scan for a particular tuple.

The TID of the inserted tuple is computed using the relative page number and offset of the page directory entry.

3.6 I/O Considerations

All input and output is performed by the subroutine IOSVC1. This is an assembly language subroutine which issues a request to transfer the contents of a page from memory to secondary storage or vice versa. Buffers are used to hold up to five user relation pages in memory at one time. Before a user relation page is read, the contents of the buffers are checked by the subroutine CHECK_BUF to see if the desired page is present. If the page is not in a buffer CHECK_BUF determines if there is an available buffer. If the page is not in memory and there is no available buffer then the subroutine GET_BUF is called to determine which buffer should be used. In the present implementation GET_BUF arbitrarily chooses a buffer but when a more desirable scheme is developed it can be more easily implemented as the selection part of this task has been isolated in the GET_BUF subroutine.

When doing I/O there are three types of errors which may occur. Type 0 I/O errors are those which result in no data transfer either into or out of the data base. Type 1 errors are those which occur during a read operation when an undetermined amount of data is transfered from the data base to main memory before the error condition occurs. Type 2 errors are the same as Type 1 errors except they occur during write operations, that is, an undetermined amount of information has been transfered from main memory to the data base. These errors can occur when reading or writing either Master Relation or user relation pages. The recovery procedures for these I/O errors are outlined in the following paragraphs.

Because the Relation Index and Relation Data Master relations are small enough to fit entirely in main memory, they are read and written in one I/O operation whether or not they occupy more than one page. In the following discussion block refers to the unit of information read or written by one IOSVC1 call, whether it is one page as in the case of user relations or the Domain Information relation or multiple pages in the case of the other two Master Relations.

Before the contents of a block are changed, the entire block is saved in a save area. The save area is presently in memory, but could be on an external device. The save area provides the means for backing out of an AM function if it cannot be completed due to the occurence of an I/O error. As an example consider the function CREATE which requires reading and writing the Relation Index and Relation Data relations and reading and writing perhaps multiple pages of the Domain Information relation. If the updated Relation Index and Relation Data relations are successfully written in the data base and an I/O error occurs when reading or writing the first Domain Information page, the new relation is only partially created. To back out of the CREATE function the saved versions of the Relation Index and Relation Data relations are written back into the data base thereby returning the data base to its previous state.

Each update function has a similar recovery procedure. Copies of the original version of all updated blocks are maintained. If an error does occur the function first attempts to return the data base to its state prior to the function call by writing the saved blocks back into the data base. If there are saved blocks which cannot be re-written due to further I/O errors they are dumped to an external device. The data base administrator may make further attempts to restore the data base using these dumped blocks. If the saved blocks are lost before they can be used to restore the data base there is no way to recover completely from the I/O error and action must be taken by the data base administrator to restore the integrity of the effected relation.

If a function returns an I/O error exception code the following actions should be taken. Termination due to a Type 0 I/O error requires no special action beyond fixing the cause of the error, for example readying an off-line device. If a Type 1 error has occurred some main memory block may have questionable information. For this reason the session should be terminated and a new session started so all main memory blocks may be reinitialized. If a Type 2 error occurs the session should be terminated and the dumped blocks written into the data base using a utility function, WRTBLK, which is described in Section 3.7.

The recovery procedure outlined above provides some protection in the event of hardware errors, but as in all data processing environments the entire data base should be periodically backed up. In the event some unrecoverable error occurs, the backup prevents loss of the entire data base, although all operations done since the most recent backup are lost. As the data base system receives more use the addition of some type of system log to record the output operations of a session will allow even better recovery procedures. With the addition of a log if the data base is lost a previous copy may be recovered to the point of the error using the operations recorded in the log.

3.7 Utility Functions

In addition to the AM functions already described the following routines have been developed for use by the data base administrator to maintain the data base. INIT is used to initialize a new data base, COPY allows entire relations to be transfered into or out of the data base and WRTBLK will write a specified block into the data base.

When creating a new data base the subroutine INIT is used to initialize the data base file. In particular the global variable NUM_REL (the number of relations) must be set to zero because it is used in a variety of places to terminate loops and must accurately reflect the total number of relations. The space allocated for the Relation Index and Relation Data relations is also initialized because the entire space is read and written by the AM functions, not just the space occupied by tuples. The space available for storing user relations is not initialized but the total number of pages available for the user relations is used to initialize the free space list.

The function COPY provides the means for transfering entire relations into or out of the data base. Input parameters for COPY are the relation identifier, the mode of operation (input or output) and the logical unit number assigned to the external device to be used for input or output.

In the output mode COPY scans all tuples of the specified relation using the traversal scheme corresponding to physical position. As discussed in Section 3.4 using this traversal scheme means that relations stored using the unordered sequential structure are scanned from beginning to end. Relations stored using the indexed sequential structure are also scanned from beginning to end but any pointers to overflow pages are followed so overflow pages are scanned in the correct order. Any tuples marked deleted are ignored by the scan. As a tuple is scanned it is written to the device indicated by the user. If this device produces machine readable copy (i.e. paper punch or disk drive) the copy may be used as a backup for the relation in the event the data base copy is destroyed due to hardware or software error. If the output is the line printer the copy is a formatted list of the relation.

The input mode of COPY uses the Master relation information to determine the storage structure of the relation to be copied. COPY reads tuples from the input device indicated by the user and places them in the data base. If the storage structure is unordered sequential the tuples are positioned one after another as they are read. If the storage structure is indexed sequential the tuples are assumed to be ordered on the primary key and are placed in the data base in the order in which they are read. When a data base page is filled a first level index entry is created using the key of the last tuple on the page. When a first level index page is filled a second level index entry is created. The input mode of COPY provides a convenient way to populate a relation.

The COPY function may be used by the data base administrator to restructure relations and to manage space allocated to relations. The procedure used to restructure a relation involves using COPY to copy the tuples of a relation out of the data base then destroying the relation using the DESTROY function. The relation is then recreated using the CREATE function with the desired storage structure as one of the input parameters. If the new structure requires tuples to be ordered on a particular field the tuples in the external copy of the relation must be sorted on this field. The input mode of COPY is then used to copy the tuples into the data base where they are placed according to the new storage structure. This procedure requires each step to be initiated by the data base administrator. A higher level restructuring function could be implemented to perform the necessary steps automatically given the relation identifier and the new storage structure desired.

Because deleted tuples are ignored by COPY, garbage collection of deleted tuples in a relation is done any time a relation is copied out of the data base and then back into the data base using the COPY function.

A procedure similar to that used for restructuring may be followed if an adjustment needs to be made to the amount of space allocated to a relation. When a relation is created one of the input parameters for the CREATE function is the projected maximum number of tuples. This number is used to determine the number of contiguous pages to allocate to a relation. If the amount of space initially allocated proves to be too little or too much the data base administrator may follow the restructuring procedure to make the necessary adjustment. When the relation is recreated a new figure for the maximum number of tuples may be used. This space management scheme is sufficient for the planned application, because the sizes of relations are predictable and fairly static. This scheme would not be effective however, if relations change size often. A more dynamic space allocation scheme could involve the use of a free page list. Pages could be allocated to relations as necessary and returned to the free list when they were no longer needed. This type of scheme would require more overhead than the scheme presently implemented but would result in a savings of data base space if accurate estimates of relation size could not be made.

WRTBLK allows a particular data base block to be written into the data base. As explained in Section 3.6 a block is either the entire Relation Index or Relation Data relation or a single page of either the Domain Information relation or a user relation. The block to be copied is in the internal format (binary) of the machine and IOSVC1 is used to write the block into the data base. WRTBLK is part of the recovery procedure described in Section 3.6 which is used when a hardware error has caused an AM function to terminate with a Type 2 I/O error.

CHAPTER IV

USE OF THE DATA BASE SYSTEM

The AM functions were developed to be used in two ways. Application programs may use the AM functions directly to access information in the data base, or they may be used to implement a higher level interface. This chapter discusses both of these uses of the AM functions. Also included in this chapter are suggestions for further development of the current implementation.

4.1 Using the AM Functions

The system requirements and details involved in the use of the data base system are described in the Users' Manual included as Appendix B. The Manual is organized into two parts. Part I is primarily for use by the data base administrator when creating new data bases or when making additions or changes to the system. It contains information about initializing system variables and adding new storage structures. It also contains a detailed description of the implementation of each subroutine in the system including the utility functions. Part II of the Manual is directed to the user of the AM functions and contains only a description of the calling sequence of each user callable AM function and the details of other programming requirments necessary to use the data base system.

4.2 Using Information from the Data Base

When using the low level interface provided by the AM functions the basic unit of data is the tuple. In practice user programs will probably be interested in a particular domain or domains of a tuple. To maintain the maximum degree of data independence the system information contained in the Master relations about the number and location of domains in the tuple should be used rather than coding this information directly. If the order of domains within a tuple is changed or if a domain is added to a relation it will have no effect on existing programs if the system information is used.

Figure 16 is an example of an application program which retrieves from the SUPPLIER-PART relation the tuples with S# = 2. An array is declared for each domain of interest, in this case SPP for the P# domain and SPQTY for the QTY domain. Although the RID of a relation does not change and may be used directly by an application program, the example illustrates the use of the RTRV_REL function (RTR) to determine the RID. The SUPPLIER-PART relation name (SUPPRT) is compared to each relation name in the Relation Index until a match is made. The corresponding RID is assigned to the variable SPRID for use in the program. Next, the domain identifier (DID) and location in the tuple of each of the three domains is found by the

```
      SPP(I) = TUPLE(SPPLOC)
      SPQTY(I) = TUPLE(SPQLOC)
25    CONTINUE
      IF (TID.EQ.LASTID) GO TO 30
      TID = NXTTID
      TC = NXTTC
      GO TO 20
30    CONTINUE
C
C        NUMSP IS THE NUMBER OF SELECTED TUPLES
C
      NUMSP = I
C
C
C        REMAINDER OF PROGRAM HERE
C
      WRITE (7,300) SPP(I),(SPQTY(I),I=1,NUMSP)
300   FORMAT ('RELATION',(2I5))
      CALL ENDIT (STATUS)
      STOP
C
C        ERROR EXIT
C
10    CONTINUE
      STOP
      END


      SUBROUTINE FNDDOM(RID,DOMNAM,DOMNUM,DOMLOC,DOMDID)
      INTEGER*2 RID,DOMNAM(3,8),DOMNUM,DOMLOC(2,8),DOMDID(8),
     *   TOTDOM,DOMINF(12,8),STATUS,TST
      TST = 0
      DO 10 I = 1,8
      DOMLOC(1,I) = 0
      DOMLOC(2,I) = 0
      DOMDID(I) = 0
10    CONTINUE
      CALL RTDOM(RID,TOTDOM,DOMINF,STATUS)
      IF (STATUS.NE.0) GO TO 60
      DO 20 I = 1,TOTDOM
      DO 30 J = 1,DOMNUM
      IF (DOMLOC(1,J).EQ.0.AND.DOMNAM(1,J).EQ.DOMINF(2,I)
     *   AND.DOMNAM(2,J).EQ.DOMINF(3,I).AND.
     *   DOMNAM(3,J).EQ.DOMINF(4,I)) GO TO 40
30    CONTINUE
      GO TO 20
40    CONTINUE
      DOMDID(J) = DOMINF(9,I)
      DOMLOC(1,J) = DOMINF(6,I)
      DOMLOC(2,J) = DOMINF(7,I)
      TST = TST + 1
      IF (TST.EQ.DOMNUM) GO TO 60
20    CONTINUE
60    CONTINUE
      RETURN
      END
```

Figure 16. Example of the Use of the
Low Level Interface

```
C
C        DECLARE VARIABLES NECESSARY FOR SUPPLIER-PART RELATION INFO
C
      DIMENSION SPP(200),SPQTY(200)
      INTEGER*2 SPNAM(3),SPRID,SPNAMS(3,8),SPSLOC,SPPLOC,SPQLOC
C
C        OTHER VARIABLES
C
      DIMENSION TUPLE (64),VAL(32)
      INTEGER*2 I,NREL,RIDS(10),RELNAM(3,10),STATUS,NUMDOM,
     *   DOMINF(12,8),DOMLOC(2,8),DOMDID(8),TID,FRSTID,LASTID,TC,
     *   NXTTID,NXTTC,NUMSP,J
C
C        INITIALIZE
C
      DATA SPNAM,((SPNAMS(I,J),I=1,3),J=1,3)
     *   /'SUPPRT','SPLIER','PARTNO','QTY    '/
      DATA SPP,SPQTY,SPRID,SPSLOC,SPPLOC,SPQLOC
     *   /200*0.,200*0.,0,0,0,0/
      CALL BEGIN (STATUS)
C
C        DETERMINE RID FOR SUPPLIER-PART RELATION
C
      CALL RTR (NREL,RIDS,RELNAM,STATUS)
      IF (STATUS.NE.0) GO TO 10
      DO 12 I = 1,NREL
      IF (SPNAM(1).EQ.RELNAM(1,I).AND.SPNAM(2).EQ.RELNAM(2,I)
     *   .AND.SPNAM(3).EQ.RELNAM(3,I)) GO TO 15
12    CONTINUE
      GO TO 10
15    CONTINUE
      SPRID = RIDS(I)
C
C        FIND LOCATIONS OF DOMAINS OF INTEREST
C
      CALL FNDDOM(SPRID,SPNAMS,3,DOMLOC,DOMDID)
      SPSDID = DOMDID(1)
      SPSLOC = DOMLOC(1,1)
      SPPLOC = DOMLOC(1,2)
      SPQLOC = DOMLOC(1,3)
      WRITE (7,200) SPSLOC,SPPLOC,SPQLOC
200   FORMAT ('SPSLOC,SPPLOC,SPQLOC',3I5)
      IF (SPSLOC.EQ.0.OR.SPPLOC.EQ.0.OR.SPQLOC.EQ.0) GO TO 10
C
C        RETRIEVE SUPPLIER-PART RELATION
C
      VAL(1) = 2.
      CALL SETSCN (SPRID,1,SPSDID,0,VAL,FRSTID,LASTID,TC,STATUS)
      IF (STATUS.NE.0) GO TO 10
      I = 0
      TID = FRSTID
20    CONTINUE
      CALL GETNXT (SPRID,TID,TC,TUPLE,NXTTID,NXTTC,STATUS)
      IF (STATUS.NE.0) GO TO 10
      IF (TUPLE(SPSLOC).NE.2.) GO TO 25
      I = I + 1
```

Figure 16. (continued)

FNDDOM subroutine. This subroutine is an example of how the RTRV_DOM_INFO (RTDOM) function may be used to retrieve system information from the Domain Information relation. The domain location is given by a starting and ending offset but in this example because all three domains are numeric (single word) domains only the starting word of each domain is necessary. The offsets in the tuple of the three domains S#, P# and QTY are SPSLOC, SPPLOC and SPQLOC respectively.

Once the RID and domain locations are determined the relation is retrieved from the data base using the SETSCAN and GETNEXT functions. The domain identifier of the S# domain, SPSDID, and the value 2 are passed to SETSCAN with the code for the equality relation which is 1. (The parameter set to 0 is the second DID used in some forms of the search criterion.) SETSCAN returns the first TID (FRSTID), last TID (LASTID) and the traversal code (TC) to be used in the scan. Each tuple retrieved by GETNEXT is tested against the criterion "S# = 2". Notice that the tuple is returned from GETNEXT in an array of the maximum tuple size (TUPLE(64)). When a tuple meets the criterion the P# and QTY domains are mapped to the next available rows in the SPP and SPQTY arrays. GETNEXT is called until the last tuple indicated by SETSCAN (TID = LASTID) has been retrieved.

4.3 Use of the AM Functions to Implement a Higher Level Interface

The example application program shown in Figure 16 retrieves tuples from the SUPPLIER-PART relation which satisfy the criterion S# = 2. If the tuples from the PART relation with WEIGHT > 15 are desired another program would be necessary. A higher level interface allows queries to be made without requiring new coding for each new query. For a discussion of the variety of relational languages currently being developed as higher level interfaces see (Chamberlin 1976). One higher level language is based on the relational algebra. The following discussion defines three relational algebra operators and illustrates how the AM functions could be used to implement them.

Relational algebra operators take relations as operands and always return a single relation as a result. The restrict operator selects tuples from a relation that satisfy a given condition. The condition may be a comparison between the value of a particular domain and a constant or between the values of two similarly typed domains in a tuple. For example, Figure 17 shows the result of restricting the PARTS relation to those tuples satisfying the condition WEIGHT > 15. The RESULT relation contains only the tuples from the PARTS relation satisfying the given condition.

Figure 18 is a flowchart illustrating the use of the AM functions to implement the restrict operator. The relation A is restricted using the condition "DOMAIN RELATION VALUE", where RELATION is $=, <, >, \leq, \geq$ or $\neq$.

PARTS

P#	PNAME	COLOR	WEIGHT
P1	Nut	Red	12
P2	Bolt	Green	17
P3	Screw	Blue	17
P4	Screw	Red	14
P5	Cam	Blue	12
P6	Cog	Red	19

RESULT = RESTRICT(PARTS,WEIGHT >15)

P#	PNAME	COLOR	WEIGHT
P2	Bolt	Green	17
P3	Screw	Blue	17
P6	Cog	Red	19

Figure 17. The Restrict Operator

The RESULT relation will contain the selected tuples. At the relational alebra level the relations and domains are referenced by name but the AM functions require the use of the internal identifiers. As shown in Figure 18 the RTRV_REL function is used to establish the RID of relation A and the RID of the RESULT relation if it exists. The RTRV_DOM_INFO function is then used to determine the DID and location within the tuple of the DOMAIN domain. If the RESULT relation has not been created the CREATE AM function will be used to create it at this time.*

SETSCAN and GETNEXT are then used to scan the relation A checking each returned tuple to see if it satisfies the given condition. If a tuple does meet the criterion it is added to the RESULT relation using INSERT. A similar procedure would be followed if the condition is a comparison

*If the relational algebra operators are to be responsible for creating result relations then default values for some of the CREATE parameters such as the maximum number of tuples in the relation will need to be used.

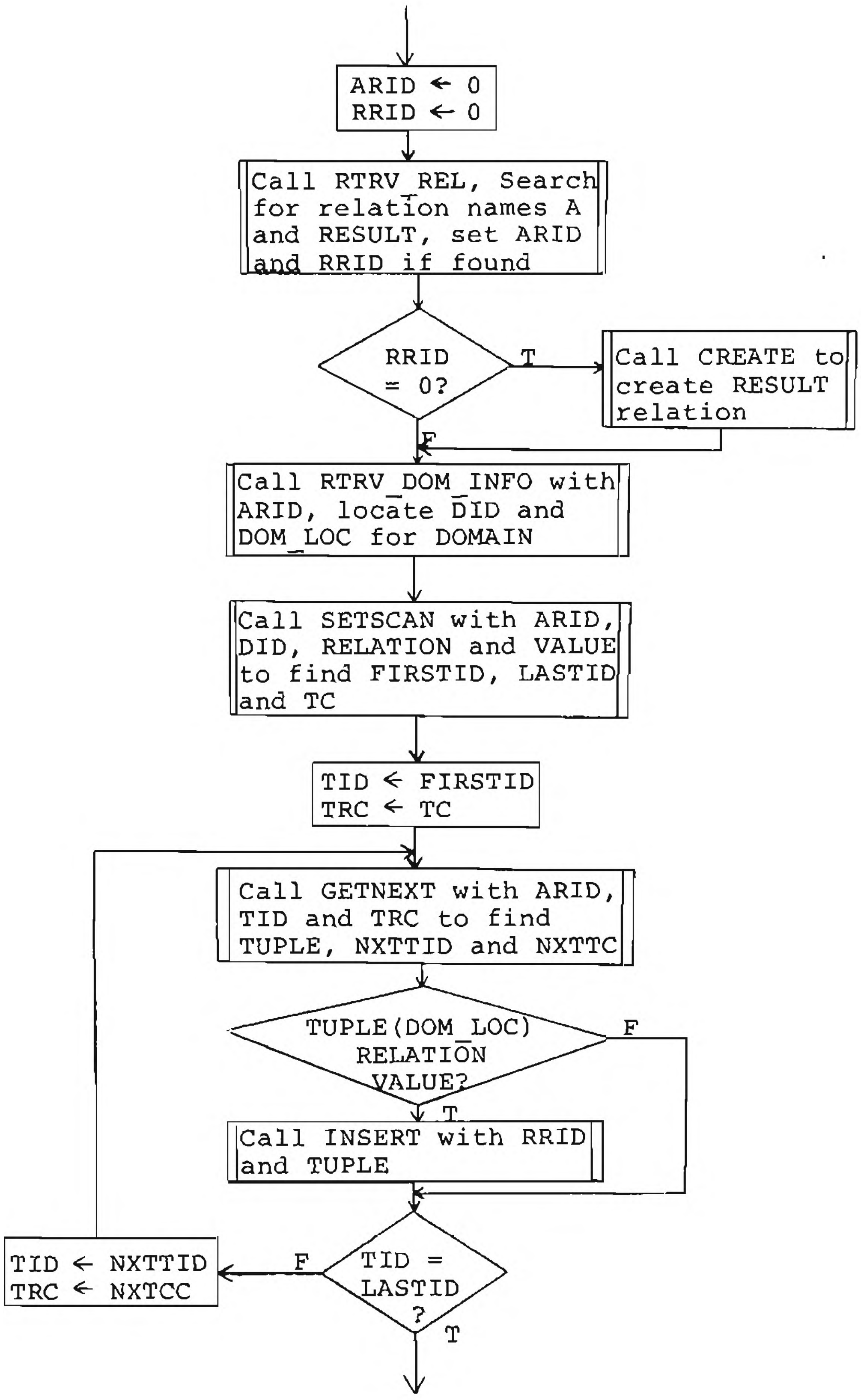

ARID ← 0
RRID ← 0
Call RTRV_REL, Search for relation names A and RESULT, set ARID and RRID if found
RRID = 0?
T
Call CREATE to create RESULT relation
F
Call RTRV_DOM_INFO with ARID, locate DID and DOM_LOC for DOMAIN
Call SETSCAN with ARID, DID, RELATION and VALUE to find FIRSTID, LASTID and TC
TID ← FIRSTID
TRC ← TC
Call GETNEXT with ARID, TID and TRC to find TUPLE, NXTTID and NXTTC
TUPLE(DOM_LOC) RELATION VALUE?
F
T
Call INSERT with RRID and TUPLE
TID = LASTID ?
F
T
TID ← NXTTID
TRC ← NXTCC

Figure 18. Implementation of the RESTRICT Operator

between two domains rather than a domain and a value.

The operator project forms a new relation by selecting specified domains from tuples of a relation and eliminating any duplicate tuples from the result. For example Figure 19 shows the result of the projection of the PARTS relation over the domain COLOR. Project would make use of the RTRV_REL and RTRV_DOM_INFO functions to retrieve Master Relation information. SETSCAN and GETNEXT would be used as in the restrict operator to scan the entire relation. The RESULT tuples are formed using only the selected domains from the operand relation and are then inserted using INSERT into RESULT. The INSERT function does not allow the insertion of duplicate tuples and will return an appropriate error code if an attempt is made to insert a duplicate. The project operator therefore needs no special mechanism for eliminating duplicates as it may check for and ignore the error code indicating a duplicate returned by INSERT.

PARTS

P#	PNAME	COLOR	WEIGHT
P1	Nut	Red	12
P2	Bolt	Green	17
P3	Screw	Blue	17
P4	Screw	Red	14
P5	Cam	Blue	12
P6	Cog	Red	19

RESULT = PROJECT(PARTS,(COLOR))

COLOR
Blue
Green
Red

Figure 19. The Project Operator

The join relational operator is a binary operator which forms one relation by concatenating a tuple from one of the original relations with a tuple of the other relation whenever a given condition holds. For example Figure 20 shows the result of joining the PARTS relation and the SUPPLIER-PARTS relation using the condition PARTS.P# = SUPPLIER-PARTS.P#. Information about the relations and domains of interest are retrieved as in the previous operators. Two scans will be in operation at once to build the RESULT relation. A straight-forward implementation of this operator would involve a complete scan of one relation for each tuple in the other.

Much can be done to improve the implementations outlined above for the relational algebra operators. For example, using temporary relations sorted on specific domains will improve the efficiency of both the project and join operators. No matter what type of improvements are made, however, the implementation of relational operators could still be based on the SETSCAN, GETNEXT and INSERT functions.

4.4 Planned Extensions to the Current System

The design of the data base system and the current implementation were done to allow further development of the system without requiring extensive changes in the current implementation. As an example, the Master relations were designed to simplify the addition of extra domains to existing relations and the addition of extra relations. The needs of the relational algebra interface discussed in Section 4.3 were considered as much as possible in the

PARTS

P#	PNAME	COLOR	WEIGHT
P1	Nut	Red	12
P2	Bolt	Green	17
P3	Screw	Blue	17
P4	Screw	Red	14
P5	Cam	Blue	12
P6	Cog	Red	19

SUPPLIER-PARTS

S#	P#	QTY
S1	P1	3
S1	P2	2
S1	P3	4
S1	P4	2
S1	P5	1
S1	P6	1
S2	P1	3
S2	P2	4
S3	P3	4
S3	P5	2
S4	P2	2
S4	P4	3
S4	P5	4
S5	P5	5

RESULT = JOIN(PARTS,SUPPLIER-PARTS,P#=P#)

P#	PNAME	COLOR	WEIGHT	P#	S#	QTY
P1	Nut	Red	12	P1	S1	3
P1	Nut	Red	12	P1	S2	3
P2	Bolt	Green	17	P2	S1	2
P2	Bolt	Green	17	P2	S2	4
P2	Bolt	Green	17	P2	S4	2
P3	Screw	Blue	17	P3	S1	4
P3	Screw	Blue	17	P3	S3	4
P4	Screw	Red	14	P4	S1	2
P4	Screw	Red	14	P4	S4	3
P5	Cam	Blue	12	P5	S1	1
P5	Cam	Blue	12	P5	S3	2
P5	Cam	Blue	12	P5	S4	4
P5	Cam	Blue	12	P5	S5	5
P6	Cog	Red	19	P6	S1	1

Figure 20. The Join Operator

current implementation so that extending the system to include a higher level interface would be possible without changing the low level interface. The following features of the low level interface were also considered as possible extensions in the design and implementation of the current system. Their inclusion in the interface would provide additional flexibility and efficiency.

Additional storage structures. One of the purposes of the low level interface is to provide access to data independent of the way in which it is physically stored. This feature allows the storage structure of data to vary as the need arises. In order that the data base may be organized to increase the efficiency of applications it is desirable to have a variety of storage structures. The indexed sequential structure provides efficient access to tuples when the key domains fall in a specified range. A hashed structure would provide efficient access to a tuple based on the specific value of chosen key domains. In a storage stuucture using hashing the TID would be determined by a computation on the key fields. This TID would then be used to locate a specific tuple in the same manner as is used for the indexed sequential structure. Utilizing an existing TID scheme (the two current schemes are discussed in Section 3.4) limits the coding changes mainly to the SETSCAN, GETNEXT and INSERT functions because the GET, DELETE and REPLACE functions utilize the TID to access tuples.

Additional structures required by application programs or higher level modules may be implemented by the data base administrator. The Users' Manual contains guidelines for implementing additional structures.

Directories. As discussed in Section 3.4 directories can be used to increase the efficiency of tuple retrieval when the search criterion is based on properties of non-key domains. The creation of directories as separate relations is possible with the current implementation. Directory entries may be inserted, deleted, replaced, etc. as are tuples in any other relation. However, a directory, to be meaningful must reflect the contents of the relation it modifies. This implies the maintenance of interrelational information because insertions, deletions and modifications to a relation must be reflected in any directories for the relation. One possible way to include this information would be to add a new Master relation containing entries for existing directories with information about the relation and the domain for which the directory is maintained and the location and type of the directory. This information would be used to maintain directories and to use them as alternate traversal paths for scanning relations.

Variable Length Domains. The present implementation allows only tuples containing fixed length domains. Some applications may require domains of variable length, especially applications using graphical data. For example a

COMPONENT relation in a building design application may consist of information for standard doors, windows, beams, etc. Although tuples of this type of relation may have many domains with a similar length, such as NAME, MANUFACTURER, UNIT-COST, etc., the graphical information may vary in length significantly from one component to another. Rather than having a domain sized to accommodate the largest amount of graphical data, a scheme for allowing variable size tuples would eliminate wasted space.

The TID scheme used for the indexed sequential storage structure may be adapted to handle tuples with variable length domains. The page directory as implemented contains the address of the first word of the tuple. In a relation declared as having variable length tuples the first word of a tuple would be the length of the tuple. This length could be used when retrieving the tuple and to determine the starting address of the next tuple on the page. The starting address of the next tuple would be the address of the current tuple plus its length. Deleting, inserting and replacing tuples with variable length domains would make use of the fact that tuples may be reorganized on a page without changing the tuples' TID's. For example, instead of simply marking a tuple deleted as is presently done the page would be reorganized so that the remaining tuples on the page would fill the space previously occupied by the deleted tuple. This would leave all available space at the end of the page to be used for new tuples. If replacing a tuple involved a change in tuple size the update would be viewed as a deletion followed by an insertion. With any page reorganization the page directory would be updated to include the current values of the starting addresses for tuples.

An alternate method for implementing variable length domains could involve maintaining the variable length information in a special single-domain relation and using a pointer in the original tuple. This pointer may then be used to retrieve the variable length domain as necessary.

4.5 In Conclusion

The data base system designed and implemented meets the desired objectives of allowing access to a data base and providing the advantages of centralized control over the data including increased data independence. Because the storage structures of the relations are an implementation detail not visible to the user of the data base interface the storage structures of relations may be changed without affecting programs using them. This means that structures may be implemented to maximize the efficiency of specific applications without requiring changes in other application programs.

The formal description of the AM functions making up the low level interface provides detailed design specifications which may be used by both user and implementer. In addition, by separating the implementation details from the description of the AM functions the formal description allows other implementations to be done without redesigning the functions.

The implementation was designed to be used either directly by application programmers or as a base on which to build a higher level interface. It is possible for an application programmer to make use of the implementation of the low level interface because (1) the functions are FORTRAN callable subroutines and (2) the relational model, because of the tabular structure used, is not difficult to use. The low level interface has been successfully used to implement a higher level interface based on the relational algebra.

APPENDIX A

THE FORMAL SPECIFICATION

This appendix contains the formal specification of the AM functions. Chapter 2 contains the description of the specification technique and discusses the INSERT and GET functions in detail.

```
BEGIN  This function initializes system variables.  It must
     be called before issuing any other AM function calls.

     Input - none
     Output - none
     Effect - DBINIT ← true
     Exceptions - IO_ERROR = type0, type1

END  This function terminates a session.

     Input - none
     Output - none
     Effect - DBINIT ← false
     Exceptions - DBINIT = false

CREATE  This function is used to define new relations.

     Input - RELINF = record of
               RELATION_NAME
               NUM_DOMAINS   The order of the relation
               MAX_TUPLES    Number of tuples expected
               TUPLE_LENGTH
               DOMAIN_INFO array of DOM_DESC

             DOM_DESC = record of
               DOM_NAME
               DOM_TYPE      Character or numeric
               DOM_KEY       Primary key information
               DOM_LENGTH
               DOM_LOC       Location of domain in tuple
     Output - RID
     Effect - VAL_RID(RID) = true
          RTRV_DOM_INFO(RID) = DOMAIN_INFO, DOMAIN_ID array
```

of DID, where DOMAIN_ID is the array of domain identifiers corresponding to the domains of the relation. The domain identifiers returned by RTRV_DOM_INFO are assigned by the system.

RTRV_REL_INFO(RID) = REL_INFO, where REL_INFO contains the same information as RELINF excluding the domain information, DOM_DESC.

NUM_REL ← NUM_REL + 1

$\exists$ i $\ni$ 1 $\leq$ i $\leq$ NUMREL and RELID(i) = RID and RELNAME(i) = RELATION_NAME

Exceptions - DBINIT = false

NUMREL = MAX_NUM_REL

$\exists$ i $\ni$ 1 $\leq$ i $\leq$ NUMREL and RELATION_NAME = RELNAME(i)

NUM_DOMAINS MAX_NUM_DOMAINS

CHECK_SPACE = false

TUPLE_LENGTH > MAX_TUP_LENGTH

VAL_DOM_TYPE(DOM_TYPE(i)) = false

DOM_KEY(i) = 0 $\forall$ i $\ni$ 1 $\leq$ i $\leq$ NUM_DOMAINS

DOM_KEY(i) = DOM_KEY(j) $\neq$ 0, 1 $\leq$ i,j $\leq$ NUM_DOMAINS, i $\neq$ j

$$\sum_{i=1}^{\text{NUMDOM}} \text{DOM\_LENGTH}(i) \neq \text{TUPLE\_LENGTH}$$

IO_ERROR = type0, type1, type2

DESTROY This function eliminates information about a relation from the MASTER relations.

Input - RID

Output - none

Effect - The sequence CREATE $\{\}$ DESTROY(RID) for the same relation has no effect, where $\{\}$ indicates any sequence of AM functions.

Exceptions - DBINIT = false

VAL_RID(RID) = false

IO_ERROR = type0, type1, type2

INSERT This function is used to insert tuples into existing relations

Input - RID

TUPLE

Output - TID

Effect - VAL_TID(RID,TID) = true

GET(RID,TID) = TUPLE, where TID = FINDTID(RID,TUPLE)

Exceptions - DBINIT = false

VAL_RID(RID) = false

BLANK_PK(RID,TUPLE) = true

DUP_PK(RID,TUPLE) = true

IO_ERROR = type0, type1, type2

DELETE This function deletes the specified tuple from the specified relation.

Input - RID
TID
Output - none
Effect - INSERT(RID,TUPLE) { } DELETE(RID,FINDTID(RID, TUPLE)) has no net effect
Exceptions - DBINIT = false
VAL_TID(RID,TID) = false
IO_ERROR = type0, type1, type2

REPLACE This function is used to replace an existing tuple with another. Note that REPLACE will not allow a change in the primary key domain.

Input - RID
TID
TUPLE
Output - none
Effect - GET(RID,FINDTID(RID,TUPLE)) = TUPLE
Exceptions - DBINIT = false
VAL_RID(RID) = false
VAL_TID(RID,TID) = false
CHANGE_PK(RID,TUPLE) = true
IO_ERROR = type0, type1, type2

GET This function is used to retrieve a tuple from a relation.

Input - RID
TID
Output - TUPLE, where FINDTID(RID,TUPLE) = TID within the same session.
Effect - none
Exceptions - DBINIT = false
VAL_RID(RID) = false
VAL_TID(RID,TID) = false
IO_ERROR = type0, type1

SETSCAN This function is used to initiate a traversal of a relation. Using information about the relation and the search criteria a traversal path is chosen. An initial traversal code and the TID's of the first and last tuples on the chosen path are returned.

Input - RID
DID1
RELATION $=, <, >, \leq, \geq, \neq$
VALUE or DID2 Comparison value of domain identifier or domain to be used in comparison

```
     Output - TC   Initial traversal code
          FIRST_TID
          LAST_TID, where TC indicates a linear ordering of
               all tuples of the relation and if there exist
               tuples in the relation satisfying the search
               criteria then they lie between the tuples
               indicated by FIRST_TID and LAST_TID inclusive.

     Effect - none
     Exceptions - DBINIT = false
          VAL_RID(RID) = false
          VAL_DID(RID,DID1) = false
          VAL_REL(RELATION) = false
          VAL_DID(RID,DID2) = false
          IO_ERROR = type0, type1
```

GETNEXT This function is used to traverse a relation. A tuple of a relation is retrieved and the TID of the next tuple according to the traversal scheme is determined. The traversal code is updated if necessary and returned.

```
     Input - RID
          TID
          TC
     Output - TUPLE
          NEXT_TID
          NEXT_TC where TUPLE is the tuple with identifier
               TID, NEXT_TID is the tuple identifier of the
               tuple following TUPLE in the linear ordering
               indicated by TC and NEXT_TC indicates the
               same linear ordering of the relation as TC.
     Effect - none
     Exceptions - DBINIT = false
          VAL_RID(RID) = false
          VAL_TID(RID,TID) = false
          VAL_TC(RID,TC) = false
          IO_ERROR = type0, type1
```

FINDTID This function is used to retrieve the TID of a tuple when the value of the primary key is known.

```
     Input - RID
          TUPLE(PK)  This is the tuple with at least the
               value of the primary key. The values of the
               other domains are ignored.
     Output - TID where GET(RID,TID)(PK) = TUPLE(PK)
     Effect - none
     Exceptions - DBINIT = false
          VAL_RID(RID) = false
          BLANK_PK(RID,TUPLE(PK)) = true
          TUPLE(PK) ≠ T(PK) ∀ tuples T in the relation RID
          IO_ERROR = type0, type1
```

```
RTRV_REL_INFO  This function returns the information con-
     tained in the Relation Information Master relation for
     a given relation.

     Input - RID
     Output - RELINFO where RTRV_REL_INFO(CREATE(...RELINFO
          ...)) = RELINFO
     Effect - none
     Exceptions - VAL_RID(RID) = false
          IO_ERROR = type0, type1
          DBINIT = false

RTRV_DOM_INFO  This function returns the domain information
     for all the domains in the given relation.

     Input - RID
     Output - DOMAIN_INFO
          DOMAIN_ID where RTRV_DOM_INFO(CREATE(...DOMAIN_
               INFO...)) = DOMAIN_INFO, DOMAIN_ID
     Effect - none
     Exceptions - DBINIT = false
          VAL_RID(RID) = false
          IO_ERROR = type0, type1

RTRV_REL  This function returns a subset of the Relation
     Information Master relation.  It returns the RID's and
     relation names of all the relations.

     Input - none
     Output - RELINDX array of RINDX
          RINDX = record RELATION_NAME
                         RID
          where CREATE(...RELNAME...) = RINDX.RID(i) for
               some i, 1≤ i ≤NUMREL
     Effect - none
     Exceptions - DBINIT = false
          IO_ERROR = type0, type1
```

The following functions are necessary for the complete description of the data base interface, but are not accessible by the user of the interface. They are mainly used by the other functions to check for exceptions. Their functional specification is implicit in the previous specifications. For reference their input and output types are listed below.

```
VAL_RID
     Input - RID
     Output - boolean

VAL_TID
     Input - RID
          TID
     Output - boolean

VAL_DOM_TYPE
     Input - DOM_TYPE
     Output - boolean

CHECK_SPACE
     Input - TUPLE_LENGTH
          MAX_TUPLES
     Output - boolean

BLANK_PK
     Input - RID
          TUPLE
     Output - boolean

DUP_PK
     Input - RID
          TUPLE
     Output - boolean

CHANGE_PK
     Input - RID
          TUPLE
     Output - boolean

VAL_REL
     Input - RELATION
     Output - boolean

VAL_TC
     Input - RID
          TC
     Output - boolean
```

The following are global variables used in the specification.

```
DBINIT boolean
MAX_NUM_REL integer
MAX_NUM_DOMAINS integer
MAX_TUP_LENGTH integer
NUM_REL integer
IO_ERROR  {type0, type1, type2}
```

APPENDIX B

USERS' MANUAL

The Users' Manual consists of two parts. The first part contains detailed information about the data base and the subroutines making up the data base system for the use of the data base administrator. The second part is directed to the user of the AM functions and contains only the information necessary to use the system. Except for explanatory comments in the code itself this Manual is the documentation for the current implementation.

B.1 Users' Manual Part I

This part of the Users' Manual contains a description of the system variables and how to initialize them, a description of each subroutine in the system and guidelines for creating new data bases and adding additional storage structures.

B.1.2 System Variables

The system variables are organized into labeled COMMON areas. Following is a description of the variables contained in each COMMON area. Array dimensions are given enclosed in parentheses following the array name. Variable names are used for array dimensions where appropriate.

<u>SYS1 COMMON</u> (all INTEGER*2 variables unless noted)

WCODE	- Write code for IOSVC1
RCODE	- Read code for IOSVC1
SYSLU	- System logical unit, used to read and write Master relations
FDBBLK	- First page (physical record) used for user relations
LDBBLK	- Last page used for user relations
FDOMBK	- First page used for Domain Information relation
LDOMBK	- Last page used for Domain Information relation
DBINIT	- (Logical variable) Data base initialization flag, initialized to false, set to true in BEGIN
MAXREL	- Maximum number of relations allowed
MAXDOM	- Maximum number of domains allowed per relation
MAXTL	- Maximum tuple length allowed
NRIDXE	- Number of Relation Index relation domains
NRINFE	- Number of Relation Data relation domains
NDOMSE	- Number of Domain Information relation domains
NRXBLK	- Number of pages needed for Relation Index relation
NRIBLK	- Number of pages needed for Relation Data relation
NDIBLK	- Number of pages needed for Domain Information relation

<u>SYS2 COMMON</u> (all INTEGER*2 variables)

NUMREL - Actual number of relations
RELINF(NRINFE,MAXREL) - Area for in-core copy of Relation Data relation

The following are constants used to reference the domains of the Relation Data relation which are stored in RELINF. For example, RELINF(STADDR,RID) is the first physical record in the relation RID.

STADDR	- First page of relation
LSTBLK	- Last page assigned to relation
TUPLTH	- Tuple length
NDOM	- Number of domains
STRUCT	- Storage structure

For the unordered sequential file structure:

LSTTID	- Last used tuple identifier

For the indexed sequential file structure:

KLNTH	- Key length
NXTOV	- Next available overflow page
INDX1	- First page of first level index
INDX2	- First page of second level index

<u>BUF COMMON</u> (all INTEGER*2 variables unless noted)

NBUF	- Number of buffers
BRID(NBUF)	- RID of relation to which page in buffer belongs
BBLKNO(NBUF)	- Relative page number in relation of page in buffer
BLU(NBUF)	- Logical unit assigned to the data base used when doing I/O for this buffer
BUFFER(64,NBUF)	- (REAL array) Each buffer holds one page of a user relation

<u>SAV COMMON</u> (all INTEGER*2 variables)

SRELIX(NRXBLK*128)	- Save area for Relation Index relation
SRELD(128,3)	- Save area for Domain Information relation pages
SDOM(128)	- Work area for saving Domain Information relation pages
SDOMBK(3)	- Page number of Domain Information page in associated save area
SDN	- Index of next available slot in SRELD

<u>SAV2 COMMON</u> (INTEGER*2 variable)

SREL(NRIBLK*128)	- Save area for Relation Data relation

<u>SAV3 COMMON</u> (REAL variable)

SBUF(64) - Save area for user relation page

B.1.2 Initializing System Variables

Most system variables are initialized by the DATA statements contained in the BLOCK DATA subroutine INITD. To change any of the system variables the initial value should be changed in the INITD subroutine. Because FORTRAN does not allow dynamically declared array dimensions changing some variables also involves changing any array dimensions dependent on them, for example if MAXREL is changed the dimension for RELINF(NRINFE,MAXREL) must also be changed.

The subroutine BEGIN initializes the NUMREL and RELINF variables of the SYS2 COMMON area by reading the Relation Data relation from the data base.

B.1.3 The Subroutines

The description of each subroutine contains the COMMON areas required, the number of lines of code (excluding comments), the compiled size in bytes, a list of called subroutines and additional comments on local variables and computations.

```
BEGIN
     COMMON areas - SYS1, SYS2, BUF
     Lines of code - 38
     Compiled size - 512
     Called subroutines - IOSVC1

ENDIT
     COMMON areas - SYS1
     Lines of code - 14
     Compiled size - 130

CREATE
     COMMON areas - SYS1, SYS2, SAV, SAV2
     Lines of code - 273
     Compiled size - 4920
     Called subroutines - GETSPC, IOSVC1, SAVREL,
          WRTREL, DMPMR

DSTROY
     COMMON areas - SYS1, SYS2, SAV, SAV2
     Lines of code - 182
     Compiled size - 4326
     Called subroutines - VALRID, FRESPC, SAVREL,
          WRTREL, DMPMR, IOSVC1

RTR
     COMMON areas - SYS1, SYS2
     Lines of code - 53
     Compiled size - 1336
     Called subroutines - IOSVC1
```

```
RTRIN
     COMMON areas - SYS1, SYS2
     Lines of code - 36
     Compiled size - 360
     Called subroutines - VALRID

RTDOM
     COMMON areas - SYS1, SYS2
     Lines of code - 62
     Compiled size - 1012
     Called subroutines - VALRID, IOSVC1

GET
     COMMON areas - SYS1, SYS2, BUF
     Lines of code - 67
     Compiled size - 1380
     Called subroutines - VALRID, VALTID, LOCATE

DELETE
     COMMON areas - SYS1, SYS2, BUF
     Lines of code - 95
     Compiled size - 2034
     Called subroutines - VALRID, VALTID, LOCATE,
          SAVBUF, WRTBUF, SAVREL, WRTREL, IOSVC1,
          DMPBUF

REPLACE
     COMMON areas - SYS1, SYS2, BUF
     Lines of code - 84
     Compiled size - 1798
     Called subroutines - VALRID, VALTID, LOCATE,
          PKFNDR, SAVBUF, WRTBUF

INSERT
     COMMON areas - SYS1, SYS2, BUF
     Lines of code - 255
     Compiled size - 4970
     Called subroutines - VALRID, PKFNDR, LOCATE,
          SAVBUF, WRTBUF, SAVREL, WRTREL, IOSVC1,
          DMPBUF, SRCHIX

SETSCN
     COMMON areas - SYS1, SYS2
     Lines of code - 46
     Compiled size - 520
     Called subroutines - VALRID, GET, SRCHIX

GETNXT
     COMMON areas - SYS1, SYS2, BUF
     Lines of code - 129
     Compiled size - 2532
     Called subroutines - VALRID, GET, LOCATE
```

FNDTID
    COMMON areas - SYS1, SYS2
    Lines of code - 68
    Compiled size - 1418
    Called subroutines - PKFNDR, SETSCAN, GETNEXT

VALRID
    COMMON areas - SYS2
    Lines of code - 9
    Compiled size - 138
    Comments - The criterion for a valid relation identifier is a positive value for the starting page address.

VALTID
    COMMON areas - SYS2
    Lines of code - 18
    Compiled size - 338
    Comments - A TID is valid if the relative page number is within the range of the pages allocated to the relation and the offset is less than or equal to the maximum possible offset for the relation.

LOCATE
    COMMON areas - SYS1, SYS2, BUF
    Lines of code - 40
    Compiled size - 552
    Called subroutines - CHKBUF, GETBUF
    Comments - After LOCATE is called the user relation page specified is in the buffer indicated by the output buffer number.

PKFNDR
    COMMON areas - SYS1, SYS2
    Lines of code - 66
    Compiled size - 1412
    Called subroutines - IOSVC1
    Comments - PKFNDR is used to find the location of the primary key in a tuple of a specified relation.

SRCHIX
    COMMON areas - SYS1, SYS2
    Lines of code - 72
    Compiled size - 1492
    Comments - SRCHIX is called by INSERT and SETSCN to search the index of a relation stored using the indexed sequential structure. SRCHIX determines the page or range of pages where a specific key value may be found.

GETSPC
COMMON areas - SYS1
Lines of code - 48
Compiled size - 541
Comments - GETSPC scans the list of free blocks of pages until a block of sufficient size for a new relation is found.

FRESPC
COMMON areas - SYS1
Lines of code - 59
Compiled size - 754
Comments - This subroutine returns the pages which were allocated to a destroyed relation to the list of free pages.

GETBUF
COMMON areas - SYS1, SYS2, BUF
Lines of code - 22
Compiled size - 256
Comments - GETBUF chooses the buffer to be over-written by a new user relation page.

CHKBUF
COMMON areas - BUF
Lines of code - 24
Compiled size - 510
Comments - CHKBUF determines if a desired user relation page is already in a buffer

SAVBUF
COMMON areas - SAV3
Lines of code - 13
Compiled size - 164
Comments - This subroutine saves the indicated buffer in the save area.

WRTBUF
COMMON areas - SYS1, SYS2, BUF, SAV3
Lines of code - 34
Compiled size - 124
Called subroutines - DMPBUF
Comments - This subroutine writes the indicated buffer to the data base.

SAVREL
COMMON areas - SYS1, SYS2, SAV2
Lines of code - 25
Compiled size - 240
Comments - SAVREL saves the Relation Data relation in the save area.

WRTREL
COMMON areas - SYS1, SYS2, SAV2
Lines of code - 29
Compiled size - 294
Called subroutines - DMPMR, IOSVC1
Comments - This subroutine writes an updated version of the Relation Data relation into the data base.

DMPBUF
COMMON areas - SAV3
Lines of code - 6
Compiled size - 110
Comments - DMPBUF dumps a user relation page to an external device

DMPMR
Lines of code - 6
Compiled size - 168
Comments - DMPMR dumps a Master relation block to an external device.

IOSVC1
Lines of code - 29 (assembly language)
Compiled size 108
Comments - IOSVC1 issues the I/O supervisor call, SVC1.

COPY
COMMON areas - SYS1, SYS2
Lines of code - 316
Compiled size - 6792
Called subroutines - PKFNDR, IOSVC1
Comments - Input parameters to COPY are the mode (input or output), the RID and the logical unit of the external device to be used.

INIT
Lines of code - 86
Compiled size - 1516
Called subroutines - IOSVC1
Comments - INIT initializes all Relation Index and Relation Data pages and sets the beginning of the free list for user relation pages.

WRTBLK
Lines of code - 10
Compiled size - 148
Called subroutines - IOSVC1

B.1.4 Creating a New Data Base File

To create a new data base the following four things should be done:

(1) The size of the data base must be determined. The values for the maximum number of relations and the maximum number of domains per relation are used to compute the number of pages necessary for the Master relations according to the following computations:

```
          NRXBLK = (NRIDXE * MAXREL) / 128 + 1
          NRIBLK = (NRINFE * MAXREL) / 128 + 1
          NDIBLK = (MAXREL * MAXDOM)/ (128 / NDOMSE) + 1
          Total pages for MASTER relations =
                     NRXBLK + NRIBLK + NDIBLK
```

The number of pages to be allowed for the user relations must be decided based on the anticipated use of the data base.

(2) The system variable array dimensions and values must be changed to agree with the sizes chosen in step 1 in the subroutines making up the AM function package and in the utility functions.

(3) The data base file must be allocated using the DOS operating system.

(4) The data base file must be initialized using the utility function INIT.

B.1.5 Guidelines For Adding a New Storage Structure

Adding a new storage structure, although involving few changes to existing code, does require major additions to several of the subroutines in the AM function package and in the utility function COPY. The following are general comments on what decisions need to be made and what subroutines are affected by a storage structure addition. Specific details depend, of course, on the structure to be added.

(1) Decide on the TID to physical location mapping scheme to be used. Using an existing scheme reduces the number of additions necessary as the subroutines utilizing TID's to reference tuples, such as GET, DELETE and REPLACE, will need few changes.

(2) Determine the traversal schemes to be associated with the new storage structure.

(3) Determine the need for new Master relation entries. Additional domains may be added to any of the Master relations as necessary to record information needed by the AM functions. If the addition of domains causes a change in the number of physical records required to store the Master relations a new data base file must be created.

(4) To the CREATE function add the algorithm for computing the amount of space required by a user relation stored using the new structure and the code for initializing any new Master relation entries.

(5) Add the code necessary to process the new storage structure to the INSERT, SETSCAN, GETNEXT, FINDTID and COPY functions. These functions depend directly on the storage structure of a relation and will require the most changes.

(6) Add the necessary code to GET, DELETE and REPLACE. These functions will require major changes only if a new TID scheme is used.

B.2 Users' Manual Part II

This section of the Users' Manual contains the information necessary to use the current implementation of the data base system. This information does not replace the formal specification which describes the relationships between the different AM functions and what the functions do, but contains only the implementation specific details necessary to use the system. The first section contains the calling sequences of all user callable subroutines and the second section is a complete listing of the exception codes.

B.2.1 The AM functions

The input and output parameters are described for each user callable AM function. The data types of the variables are either half word integer (I) or real (R). Dimensions for arrays are enclosed in parentheses following the array name. The system variables described in Section B.1.2 are used where appropriate. For each variable the name and type are given followed by any additional comments necessary to explain the restrictions or use of the variable. The following four variables are used for many of the calls and are explained only here.

```
TID, I, Tuple identifier
RID, I, Relation identifier
STATUS, I, Status code returned by a function
     STATUS = 0 implies correct termination
     STATUS ≠ 0 implies error termination.  The specific
     errors are listed in Section B.2.2.
     TUPLE(MAXTL), R, Relation tuple.  When used as an input
          variable only the declared length of the tuple is
          used.  When used as an output variable the tuple
          information is left justified and any unused
          portion is zero filled.

BEGIN
     Input - none
     Output - STATUS

ENDIT
     Input - none
     Output - STATUS

CREATE
     Input - RELNAM(3), I, Relation name, may be any
                  alphanumeric characters.
          NUMDOM, I, Number of domains where 1≤ NUMDOM ≤
              MAXDOM.
          LTH, I, Tuple length where 1 ≤ LTH ≤ MAXTL
          DOMINF(NDOMSE,MAXDOM), I, where
              DOMINF(1,i) - RID
              DOMINF(2-4,i) - Domain name
              DOMINF(5,i) - 0 if domain is not part of
                               primary key
                           -1 if domain is a single domain
                               primary key
                            1, 2, ...  if domain is part of
                               of multiple domain key and 1,
                               2, etc. indicates position in
                               key
              DOMINF(6,i) - Starting word of tuple in relation
              DOMINF(7,i) - Ending word of tuple in relation
              DOMINF(8,i) - 0 if domain is alphanumeric
                            1 if domain is real
          MAXTUP, I, Maximum number of tuples anticipated
          STORST, I, Code for desired storage structure
              0 - system default (unordered sequential)
              1 - unordered sequential
              2 - indexed sequential
     Output - RID

DSTROY
     Input - RID
     Output - STATUS

RTR
     Input - none
     Output - NREL, I, Number of relations
          RIDS(MAXREL), I, RID's of relations
          INDEX(3,MAXREL), I, Names of relations where
              RID(i) is the RID of the relation with the
              name in INDEX(1-3,I)
```

```
RTRIN
     Input - RID
     Output - RELIN(NRINFE), I, Relation information where
              RELIN(1) - Location of first page of relation
              RELIN(2) - Storage structure
              RELIN(3) - Location of last page of relation
              RELIN(4) - Tuple length
              RELIN(5) - Number of domains
            For unordered sequential structure
              RELIN(6) - Last used tuple identifier
            For indexed sequential structure
              RELIN(6) - Next available overflow page
              RELIN(7) - First page of first level index
              RELIN(8) - First page of second level index
              RELIN(9) - Key length
          STATUS

RTDOM
     Input - RID
     Output - NUMDOM, I, Number of domains
          DOMINF(NDOMSE,MAXDOM), I, where for each domain i,
              DOMINF(1-8,i) - as in CREATE
              DOMINF(9,i) - Domain identifier (DID)

          STATUS

GET
     Input - RID
          TID
     Output - TUPLE
          STATUS

REPLAC
     Input - RID
          TID
          TUPLE, Complete tuple to replace existing one
     Output - STATUS

DELETE
     Input - RID
          TID
     Output - STATUS

INSRT
     Input - RID
          TUPLE
     Output - TID
          STATUS

SETSCN
     Input - RID
          TYPE, I, 1 - =, 2 - <, 3 - >, 4 - ≤, 5 - ≥, 6 - ≠,
                   7 - all
          DID1, I, DID of domain of interest where 1 ≤ DID1 ≤
              NUMDOM or -1 indicating the primary key
          DID2, I, DID of second domain of interest where
              1 ≤ DID2 ≤ NUMDOM or 0 if VAL is used
          VAL(32), R, Value for comparison or unused if DID2
              > 0
     Output - FRSTID, I, First tuple identifier on traversal
              path
          LASTID, I, Last tuple identifier on traversal path
          TC, I, Code for chosen traversal path
          STATUS

FNDTID
     Input - RID
          TUPLE, Only primary key domains are used
     Output - TID
          STATUS

GETNXT
     Input - RID
          TID, I, The FRSTID returned by SETSCN or NXTTID
              returned by a previous call to GETNXT
          TC, I, TC returned by SETSCN or NXTTC returned by
              a previous call to GETNXT
     Output - TUPLE
          NXTTID, I, Tuple identifier of next tuple on
              traversal path
          NXTTC, I, Traversal code, updated if necessary
          STATUS
```

B.2.2 Exception Codes

The following is a list of all exception codes returned by the AM functions through the STATUS variable. The value of STATUS should always be checked before using the values of other output variables. The exception code is a five digit integer. The first two digits indicate the subroutine and the final three digits indicate the specific error.

```
BEGIN 00
     060, 092 - Type 0 I/O error
     032 - Type 1 I/O error

END 01
     000 - DBINIT = false

CREATE 02
     000 - DBINIT = false
```

100 - NUMREL = MAXREL
101 - Relation name already exists
102 - NUMDOM > MAXDOM
103 - Insufficient space
104 - Tuple length greater than maximum allowed
105 - Invalid storage structure
106 - Invalid domain type
107 - No domain is primary key
108 - Two domains have same position in primary key
109 - Sum of domain lengths is greater than specified tuple length
060, 092 - Type 0 I/O error
032 - Type 1 I/O error
132 - Type 2 I/O error

DSTROY 03
000 - DBINIT = false
001 - Invalid RID
060, 092 - Type 0 I/O error
032 - Type 1 I/O error
132 - Type 2 I/O error

INSERT 04
000 - DBINIT = false
001 - Invalid RID
100 - Blanks in primary key
101 - Duplicate primary key
102 - Relation area overflow
060, 092 - Type 0 I/O error
032 - Type 1 I/O error
132 - Type 2 I/O error

DELETE 05
000 - DBINIT = false
001 - Invalid RID
002 - Invalid TID
003 - Deleted TID
060, 092 - Type 0 I/O error
032 - Type 1 I/O error
132 - Type 2 I/O error

REPLAC 06
000 - DBINIT = false
001 - Invalid RID
002 - Invalid TID
003 - Deleted TID
100 - Change in primary key domain
060, 092 - Type 0 I/O error
032 - Type 1 I/O error
132 - Type 2 I/O error

```
GET 07
     000 - DBINIT = false
     001 - Invalid RID
     002 - Invalid TID
     003 - Deleted TID
     060, 092 - Type 0 I/O error
     032 - Type 1 I/O error

SETSCN 08
     000 - DBINIT = false
     101 - Invalid RID
     100 - Invalid DID1
     101 - Invalid relation code
     102 - Invalid DID2
     060, 092 - Type 0 I/O error
     032 - Type 1 I/O error


GETNXT 09
     000 - DBINIT = false
     001 - Invalid RID
     002 - Invalid TID
     003 - Deleted TID
     100 - Invalid TC
     060, 092 - Type 0 I/O error
     032 - Type 1 I/O error

FNDTID 14
     000 - DBINIT = false
     001 - Invalid RID
     100 - Blanks in primary key field
     101 - Tuple not found
     060, 092 - Type 0 I/O error
     032 - Type 1 I/O error

RTRIN 10
     000 - DBINIT = false
     001 - Invalid RID
     060, 092 - Type 0 I/O error
     032 - Type 1 I/O error

RTDOM 11
     000 - DBINIT = false
     001 - Invalid RID
     060, 092 - Type 0 I/O error
     032 - Type 1 I/O error

RTR 12
     000 - DBINIT = false
     001 - Invalid RID
     060, 092 - Type 0 I/O error
     032 - Type 1 I/O error
```

REFERENCES

Astrahan, M. and Lorie, R., "Sequel-XRM: a relational system," Proceedings ACM Pacific 75 Regional Conference, April 1975, ACM, New York, 1975, pp. 34-38.

Bjorner, D.; Codd, E.; Deckert, K. and Traiger, I., "The GAMMA Zero n-ary relational data base interface: Specification of objects and operators, "IBM Research Report RJ 1200, San Jose, California, April 1973.

Chamberlin, D., "Relational data-base management systems," ACM Computing Surveys 8, 1 (March 1976), pp. 43-66.

Codd, E., "A relational model of data for large shared data banks," Communications of the ACM 13, 6 (June 1970), pp. 377-397.

Cyarnik, B.; Schuster, S. and Tsichritzis, D. "ZETA: A relational data base management system," Proceedings ACM Pacific 75 Regional Conference, April 1975, ACM, New York, 1975, pp. 21-25.

Gregory, S., "Application of the Relational Data Base Model to Computer-Aided Building Design", Phd Disertation (to be published), University of Utah, Department of Computer Science, 1977.

Guttag, J.; Horowitz, E. and Musser, D., "The design of data type specifications," 2nd Software Engineering Conference, October, 1976, San Francisco, California, pp. 214-220.

Held, G., "Storage structures for relational data base management systems," Memorandum No. ERL-M533, University of California, Berkeley, California, 1975.

Knuth, D., The Art of Computer Programming, Vol. 3, Addison-Wesley Pub. Co., Reading, Mass., 1968.

Liskov, B. and Zilles, S., "Specification techniques for data abstractions," IEEE Transactions on Software Engineering, SE-1, 1 (March 1975), pp. 7-19.

London, K., Techniques for Direct Access, Auerback, Philadelphia, Pa., 1973.

Lorie, R. and Symonds, A., "A relational access method for interactive applications," Courant Computer Science Symposia, 6: Data Base Systems, Prentice-Hall, New York, 1972, pp. 99-124.

Lorie, R., "XRM - an extended (n-ary) relational memory," IBM Scientific Center Report G320-2096, Cambridge, Mass., January 1974.

McLeod, D. and Meldman, M., "RISS: a generalized mini computer relational data base management system," Proceedings AFIPS National Computer Conference, 44, May 1975, AFIPS Press, Montvale, N. J., 1975, pp. 397-402.

Parnas, D., "A technique for software module specification with examples," Communications of the ACM, 15, 5 (May 1972), pp. 330-334.

Stonebraker, M.; Wong, E.; Kreps, P. and Held, G., "The design and implementation of INGRES," Memorandum No. ERL-M577, University of California, Berkeley, California, 1976.