May 23, 1973

To the Graduate Council:

I am submitting herewith a thesis written by Stuart Todd Byrom entitled "A Data Management System for Tree Structured Data Bases." I recommend that it be accepted for nine quarter hours of credit in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Sara P. Jordan
Major Professor

We have read this thesis
and recommend its acceptance:

Ronald P. Leinius

Robert M. Aiken

Accepted for the Council:

Hilton A. Smith
Vice Chancellor for
Graduate Studies and Research

A DATA MANAGEMENT SYSTEM FOR TREE STRUCTURED DATA BASES

A Thesis

Presented to

the Graduate Council of

The University of Tennessee

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

---

by

Stuart Todd Byrom

August 1973

ABSTRACT

The current trend towards generalized data management systems is a result of efforts to design systems that perform functions for a wide range of user needs. While these systems save the user the time and expense of program development, they may sometimes utilize compromise techniques that may jeopordize the efficiency of some individual applications. A solution to this problem is the ability to construct specialized systems from a range of modules each of which optimize a given feature of the system.

To illustrate this solution a specialized system is constructed using modules that could be used in a variety of other systems. Specifically, this system provides the efficient retrieval of information from a tree structured data base. An important feature of this system is the ability to create, retrieve, and manipulate subsets of the data base. To represent a set of information, the author proposes a technique which conserves storage requirements for large sets while maintaining the ability to perform set operations. In addition, it can be shown that this technique may be used with any data management system since it is independent of internal data structures.

1100951

TABLE OF CONTENTS

CHAPTER		PAGE
I.	INTRODUCTION	1
	Organization of Data	1
	Existing Data Management Systems	3
II.	SYSTEM GOALS	7
	The Design of a New System	9
	Goals of the System	10
	Tasks of the System	13
III.	DATA STRUCTURES AND THE RETRIEVAL FUNCTION	17
	Data Structures	17
	File Structure	22
	Retrieval Algorithms	29
IV.	SET REPRESENTATION AND MANIPULATION	36
	Set Operations	36
	Set Representation	38
	Definition of a Bit Stream Set Representation	39
	Set Operations with BSSR's	45
	System Commands	49
	Summary	50
V.	SYSTEM DESIGN	52
	System Tables	56
	The Retrieval Function	59
	The Set Manipulation Function	65
VI.	SUMMARY	69

PAGE

BIBLIOGRAPHY 73

APPENDIX 75

VITA 80

LIST OF FIGURES

FIGURE		PAGE
1.	Definition Tree for a Sample Tree Structured Data Base	11
2.	Example Data Base of State Information	12
3.	Tree Structure for Data Base of State Information	19
4.	Example of an Inverted File Structure	20
5.	Index to the Inverted List	21
6.	Use of the PAGES Table	25
7.	Use of the MULTIPLE Table	27
8.	Relationship of Tables FIND and PRINT	28
9.	System Tables Used in SELECT Command	31
10.	System Overview	53
11.	Schematic Diagram of Retrieval Tables	57
12.	Example of Set Tables	58
13.	Schematic Diagram of Set Tables	60
14.	Example CONTROL Table	61
15.	The Retrieval Function	63
16.	Page Frame Table for the Buffer Area	66
17.	The Set Manipulation Function	68

CHAPTER I

INTRODUCTION

The technology of today's high speed computers makes possible large collections of information for a wide variety of applications in business and government. Centralizing separate, though logically related, collections of data into a single data base can make more information available to more users and at the same time reduce the storage of redundant information. However, the failure to provide easy access to needed information (unless it is for security reasons) defeats the purpose of centralization. It is also important that there is a logical structuring of the data so that the relationships among information are apparent to the user community. Therefore, a data management system must maintain a fully integrated data base which is adequately generalized for all user demands and which provides for file maintenance, control, and security. (14)

Organization of Data

Initial attempts to collect and maintain large volumes of data were organized in a relatively uncomplicated manner due to hardware and software restrictions. Over the years, organizational techniques have changed rapidly to take advantage of new developments in storage devices. Early data management systems were constrained to serial processing on magnetic tapes. Today, the modern disk and drum direct access devices along with greater software capabilities permit larger data bases, faster processing times, and generally greater flexibility in designing generalized and specialized systems. Every data management system provides the functions of storing, retrieving, and updating information in the data base; however,

each system must be designed to optimize that function most critical to efficient operation of the system. For example, an information retrieval system for a library would be organized with different criteria from those for an airline reservation system; the former would be geared for rapid retrieval, the latter for rapid updating. (7) (14)

Since no one data orgainization technique can optimize all types of user demands, various organizations are available to the system designer to evaluate within the existing physical and financial constraints. Typically, the designer must decide among the time and space trade-offs which are characteristic of different organizations.

Although there are many complex data structures from which they all are built; these are the sequential, random, and list structures. The sequential structure is advantageous for retrieval of records that are logically (and physically) adjacent to one another in a file. Retrieving records not in sequence, however, requires relatively greater amounts of time. Updating of sequential files is difficult since additions or deletions usually require reorganization of many records. Random organizations permit relatively rapid retrieval of a record with a known key compared to the sequential organization, but are not efficient for a larger number of retrievals due to mechanical delays on direct access devices. Updating random records is rapid, since it does not interfere with other records. Simple list orgainizations allow rapid update of records, while more sophisticated forms such as inverted lists allow rapid retrieval on keys at the expense of storage requirements. Consequently, it is not uncommon for the overhead required to be as large or larger than the raw data itself in order to reduce access time. (5) (7)

Each of the functions of storing, retrieving, and updating of information can be adapted to time or storage requirements by utilizing one form (which may be quite complex) of the three basic structures outlined above. Rarely, though, can a single structure simultaneously optimize two opposing functions such as searching and updating. Consequently, it becomes clear that many factors complicate the choice of structure for a system. For instance, more complex data structures can reduce search time by a significant increase in storage overhead; however, availability and expense of external mass storage devices and core storage become critical in the design of large data bases. Typically, for generalized systems some compromise structure is chosen in order to insure efficient overall operation. Detailed discussions and formal definitions of data structures and retrieval techniques are given in (4), (5), (7), and (10).

Existing Data Management Systems

The needs of users influence the range of functions that are available in addition to storing, retrieving, and updating of information. Some systems allow access to the data base through an extension of existing programming languages such as COBOL and PL/1. Others allow retrieval of information by non-programmers via an English-like command language. Report generators on some systems can automatically provide summary reports in user-defined formats. Statistical analysis functions may also be available to apply directly to sets of retrieved items. These and other features are typical of those found in existing data management systems as described in the CODASYL survey of generalized systems in 1969 (1) and are summarized in the following discussion.

One of the pioneering systems of the 1960's, the Advanced Data

Management System (ADAM) was designed by the MITRE Corporation for the United States Air Force. This highly complex system was intended to be a research tool in the evaluation and design of information processing systems. Its breadth of capabilities had significant effect on the design of the later large scale systems. ADAM allowed the definition, creation, updating, and interrogation of files in both the batch and on-line modes. Its capabilities included the ability to accept user provided application languages, to describe new languages, to use multiple I/O stations, and to provide report generators. Most significantly, the ADAM system demonstrated that the same data structure could be used by all levels of users and that these structures could be of virtually any length.

By the time IBM released its Generalized Information System (GIS) in 1969, the functions of definition, creation, update, and interrogation of data files had become standard on generalized data management systems. In addition to these functions, GIS incorporated file access codes to protect system files from unauthorized interrogation or updating of information. Users utilized a procedual language for each of two GIS versions: one for batch processing only and one for either batch or on-line processing.

General Electric's Integrated Data Store (IDS) is not a data management system as are the ADAM and GIS systems, but is actually a procedural language that may be incorporated into a host language, usually COBOL. With IDS a user may define his own system for the maintenance of data stored in a network structure. While IDS allows a system to designed specifically for each user's needs, it therefore requires the user to be an experienced programmer. Unlike IDS, User Language/1 (UL/1) may be used by non-programmers to build their own systems. UL/1 was designed by the

Information Systems Division of RCA in such a way that the internal storage structure is not apparent to the user.

System Control-1 (SC-1) developed by the Finance Division of Western Electric Company, offers the usual functions of other systems for use internal to the company. Routines of this system act as system primitives from which new systems, services, or user functions may be constructed.

Another system developed by IBM is the NMCS (National Military Command System) Information Processing System which has the function of selecting sets of information which may be named, stored, and processed with other data on later runs. In addition, this system offers the standard information processing features along with report generating and sorting capabilities.

The goal of Systems Development Corporation's Time-Shared Data Management System (TDMS) is to provide general data management functions for the non-programmer. It is designed to provide rapid response to unpredictable user queries in on-line operation. Commands in the non-procedural language extract data from tree structured data bases.

Perhaps the most ambitious system yet is the ISL-1 Processor designed by the Information Systems Leasing Corporation. The actual organization of data in the system is unknown to the user and is chosen by a complex algorithm which selects among alternative configurations. The processor bases its selection on response requirements and on proportions of interrogation and updating. Ideally, the structures chosen optimize the mix of user commands indicated by historical statistics maintained by the processor. A significant change in this mix could automatically initiate a structural reorganization completely unknown to the user.

Each generalized data management system attempts to provide a variety of functions to satisfy the demands of users in a range of applications. Often, however, there exists no system whose philosophy of operation exactly coincides with the needs for a specialized system. Overall efficiency of a planned system may rely on the efficiency of a particular function that has been compromised in the design of the generalized system. In fact, certain specialized functions may not exist at all on current systems. In Chapter Two a discussion of this problem will be presented including a possible solution. To illustrate this solution, some new as well as existing techniques will be incorporated into a system that offers features not found on any generalized system today.

CHAPTER II

SYSTEM GOALS

The functions provided by any data management system should be those which satisfy the needs of its users. Since the development of complex systems is very costly, the trend has been towards generalized systems which attempt to handle a wide range of services that would be desirable in most data management situations. Similar to operating systems, data management systems may be tailored to each installation's primary needs. Unfortunately, compromises in the generalized system design may lead to inefficient operation of a specific function vital to a particular user's problem. Consequently, the need exists for many highly efficient, specialized systems whose processing characteristics may run counter to those of the generalized systems.

The solution to this problem exists in the design of a wide variety of specialized system modules, each capable of joining with selected modules to assemble the desired system. This idea is not a completely new one--System Control-1 among others has used the concept of "system primitives" for users to construct their own systems. However, this construction is limited to a fixed set of modules that may or may not satisfy the problem at hand.

While it is inconceivable that the developer of generalized system building blocks, or modules, could visualize or finance every possible module, it is possible that it could design a working set of those most commonly needed for an ever-growing library of specialized modules.

Design of the modules would involve careful consideration of their interrelationships and of the system environment itself. In this discussion a system unit is considered to be a physical or logical entity that may be involved in any number of system functions. For example, a system unit could be a disk file, a buffer area in core, or a data structure (list, network, tree, etc.). Whenever possible, the maintenance of a system unit should be confined to one system module (which could itself be constructed of primitive sub-modules). In this way, the most efficient management techniques for each unit may be employed. Further, should changes occur in the environment, the effects should be isolated to specific system modules. Modification or even substitution of system modules would then not disturb others. The advantages of this technique are clear. First, the addition (or removal) of storage devices to expand (reduce) the capabilities of the system would not mean interruption of the overall system design. Second, additional functions would be possible by integrating existing modules with new ones. The management duties of each module, as well as its input and output parameters, would have to be well defined for linkage, but the way in which its duties are performed should be unknown to other modules. Third, the virtual evolution of the system is possible by installing and pruning modules and functions to produce an entirely new system with new goals and philosophies of operation.

The system to be described here is designed using the idea of system building blocks or modules to conceal the maintenance and operation of each system unit. It may be shown that this technique does not require impractical overhead or restrictions on the types of functions allowed. (13) The specific functions of the system illustrate a relatively specialized

system that could probably not be implemented on the previously mentioned generalized systems. However, by utilizing elementary system modules, this system has the potential to be modified in any number of ways. Many of the modules could be used in a wide variety of dissimilar systems that have common system tasks.

The Design of a New System

As previously discussed, each data base has its own unique set of user demands made upon it. The design of the data management system is influenced by the relative proportions of storing, retrieving, and updating performed on the data. Unlike an inventory control system which must perform rapid updating, a collection of geographical or historical data must concentrate on retrieval. To show how a system could be designed for the latter type of situation, the following observations and assumptions are made:

1. After initial loading of the data base, most accesses will be for retrieval as opposed to updating. This implies a relatively stable data base.

2. From the first assumption it follows that the system should be geared for efficient retrieval, possibly at the expense of updating and initial loading efficiency.

3. Since geographical or historical data generally lend themselves well to a hierarchical organization, a logical choice for a data structure is the tree.

4. Since the use of such a data base is often analytical in nature, it is desirable for system integrity that retrievals be consistent with mathematical properties and laws.

5. Access to the data base should be available to non-programmers.

In order to make the discussion of system design more meaningful, an example data base of information collected in various states will be used. In this way, the advantages of the tree organization may be shown for this and other data bases with similar characteristics.

The tree structure is a multi-leveled logical organization that allows a variable number of records to be related to a single parent record which, in turn, may be one of several sibling records of another parent. The definition tree, such as the example in Figure 1, indicates the logical relationships among data in the data base. Each node in the tree represents a record of information that may repeat any number of times under a parent node. The definition also indicates the data items or attributes that comprise each record. This data base then is a collection of one or more entries representing state information, each having the structure of the definition tree. A data base with two state entries is shown in Figure 2. The ability to allow sibling nodes of different types, as shown in the example, is a concept that stretches the formal definition of a tree (such as found in Knuth (11)) but is a common feature on TDMS-type systems.

Goals of the System

By analyzing the physical (and financial) limitations of the system and by isolating the assumptions to be made about the system environment, the system designer should be able to determine the goals and constraints of the desired data management system. Consistent with the assumptions enumerated above about a data base similar to the given example, a realistic set of system goals are outlined below.

1. The system should be constructed with modules that conceal the

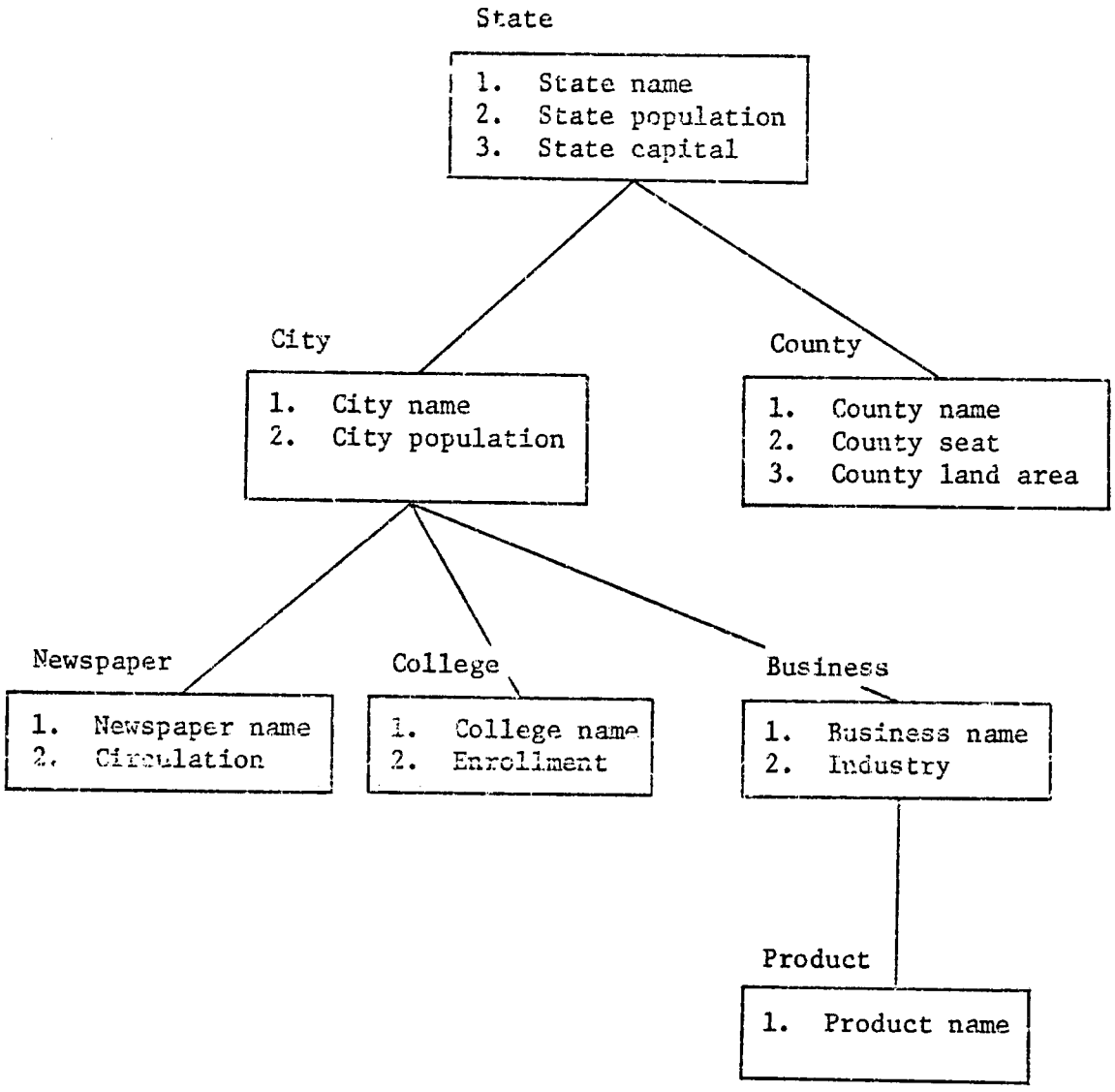


Figure 1

Definition tree for a sample tree structured data base.

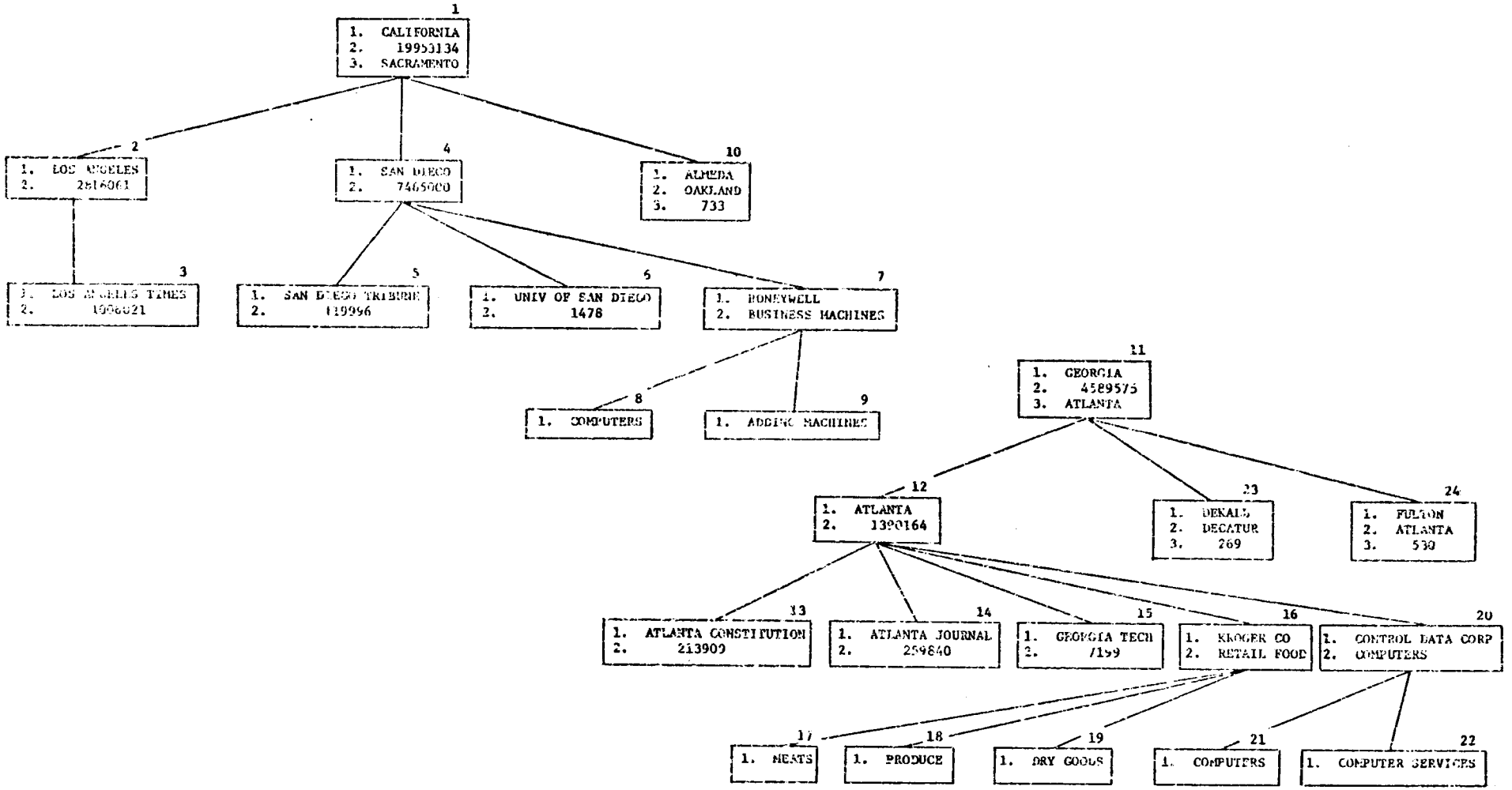


Figure 2

Example data base of state information

operation of system units to allow easy system modification and expansion.

2. The system should provide efficient retrieval of information from a relatively static tree-structured data base.

3. The system should allow the representation, naming, and storage of subsets of the data base. The sets may then be retrieved or combined with others.

4. The system is to function under an English-like command language for the efficient retrieval of information and set manipulation.

Tasks of the System

With the above goals in mind, the next step is to clearly outline those individual tasks the system should provide in order to meet these goals. For the sake of simplicity, only the essential functions will be included, although provision for modification and expansion will be maintained. In particular, the task of updating is not to be implemented initially into the system, but may be added later. It is assumed in this paper that the need for this task is sufficiently small enough not to change the conclusions drawn about the system. Therefore, the two functions to be dealt with in this paper are retrieval and set manipulation. A minimal set of commands are necessary to perform these functions adequately.

Retrieval is accomplished by a set of tasks which permit the user to isolate occurrences of items in the data base as a named representation of that set. Specifically, the SELECT command causes a search of the tree structure for nodes (i.e., attributes within those nodes) that satisfy an attribute-relation-value (a-r-v) triple. An "attribute" is one of the data items in the definition tree, "relation" is a logical operator ("less than", "greater than", "equal to", etc.) and "value" is a literal or numerical

constant. The records, or nodes, in the data base which contain data items satisfying the a-r-v triple constitute a selection set. Using the example data base in Figure 2 on page 12, the command

SELECT SET1 WHERE PRODUCT EQUALS COMPUTERS

would create and store a set named SET1 representing the nodes with identification numbers 8 and 21. Once a set is stored in this way, the user may request a listing of data items references by the set, combine it with other stored sets via one of the set manipulation functions, or request a search for the logical ancestors and descendents of this set.

Another important task to the user is the ability to retrieve data that is logically related to a selection set of nodes. That is, by retrieving all descendents and direct ancestors of each node in a set, a new set may be created which is known as a qualification set. In figure 2, the descendents of node number 16 are the nodes numbered 17, 18, and 19 (including all of their descendents if there were any); its direct ancestors are nodes numbered 11 and 12. The QUALIFY command creates and stores the new qualification set in the same manner as sets resulting from a SELECT command. To qualify the set created in the last example, the command

QUALIFY SET1 GIVING SET2

stores a representation of the set of node identification numbers 1, 4, 7, 8, 11, 12, 20, and 21 under the name SET2.

A user may request all or part of the information referenced by a set of node identification numbers to be printed for his inspection via the PRINT command. For example, the command

PRINT STATE NAME, STATE POPULATION OF SET2

would produce the printed results

CALIFORNIA

19953134

GEORGIA

4589575

While these retrieval commands by no means exhaust the possible tasks that could aid the system user, they do meet the minimum requirements of the goals of the system. With these commands, it is possible to retrieve virtually any information from the data base. Next, it will be seen that by performing set operations on selection and/or qualification sets it is possible to form complex retrieval requests.

The ability to create, store, and retrieve a representation of a set of information is a very important aspect of the management of relatively stable data bases. (This feature allows the batch system to avoid searching the tree structure for the same set of specifications (a-r-v) on each program "run".) Since updates are assumed minimal, the storage of these sets over short periods of time should not cause great concern for system integrity. In addition, the ability to manipulate these sets is critical to the performance of detailed studies and analysis of information in the data base.

New sets of node identification numbers may be created and stored by operations on existing sets. For example, a new set could be formed by including all the node identification numbers in both of two sets or by including only those that are common to both. These operations correspond to the concepts of set union and set intersection which may be specified in the user language by the operators OR and AND respectively. The operator NOT specifies the complement of a given set; that is, the set of

all node identification numbers which are not contained in that given set. Complex requests may be specified by expressions involving existing (stored) sets, the set operators, and parentheses. The FORM command allows the user to isolate data by using these expressions to describe the construction of new sets. For example, the two sets previously described could be logically combined via the command

FORM: SET3 = SET1 OR SET2 .

The resulting SET3 contains every node identification number that occurs in either of the existing sets SET1 and SET2. While storage of sets is not critical for small data bases, it will be seen that large data bases can present substantial storage problems. The command ERASE removes sets from storage that are no longer needed to help conserve storage requirements. However, for large data bases, the storage of only one set may impede efficiency. A new solution to this problem is to be presented which reduces the storage requirements for large sets.

CHAPTER III

DATA STRUCTURES AND THE RETRIEVAL FUNCTION

Under the assumptions presented in Chapter Two about the nature of the data base to be constructed, attention is now directed towards the selection of appropriate data structures. Only the node identification numbers are stored in the representation of an information set, implying that logical structure is irrelevant to set manipulation as long as the number of each record (node) is unique in the data base. Since the function of set manipulation does not depend upon the data structure, only the function of retrieval need be considered here. The assumption that updating will be minimal limits the problem of structure selection to consideration of those structures that optimize the retrieval function.

Data Structures

Various random and list data structures make it possible to significantly reduce search times--a critical element of most retrieval tasks. Typically, though, these structures tend to require relatively large amounts of overhead for indexes and pointers. Since optimization of retrieval generally has come to imply rapid response times, this system also will sacrifice storage requirements to meet this end. In addition, sacrifices in infrequent update and initial storage times for the sake of an efficient retrieval organization are acceptable.

A type of list structure, the tree, is used to represent the logical structure of the records in the data base. Each node in the tree is a record whose type (a particular set of attributes) is represented in the definition tree described previously. Pointers associated with each record

link the record to its immediate parent (one hierarchial level up), to its first descendent (which is the first of a simple list of siblings one level down), and to a sibling record (on the same level). Figure 3 illustrates this structure for the information of the example data base of state entries. Note that the pointers are the node identification numbers and that the end of a linked list is indicated by the null pointer, zero. The tree may be traversed by following the UP, RIGHT, or DOWN pointers from node to node. Although the UP pointer is not necessary to describe the tree structure, it does facilitate "climbing" the tree to locate the ancestors of a given node (an important aspect of the task of qualifying a node).

To facilitate the search for data values that are stored in the tree structure, a completely inverted list structure is maintained. The inverted list is an index of each unique data value, or key, in the data base along with its associated pointers to the nodes in the tree structure containing that key. An inverted list structure to the example data base is shown in Figure 4. This technique allows the retrieval of all nodes containing a given key without an exhaustive traversal (i.e., search) of the tree itself. Note that the large storage requirement of the inverted list is the price paid for the increased speed of access.

Another structure to aid in the retrieval of information by key is an index to the inverted structure. As noted above, the inverted list tends to be as large or larger than the data items themselves; thus, to prevent a time consuming sequential search of this table for a given key, an index can point to a location in the inverted list from which to start the search. The index in Figure 5 demonstrates how an index of data items,

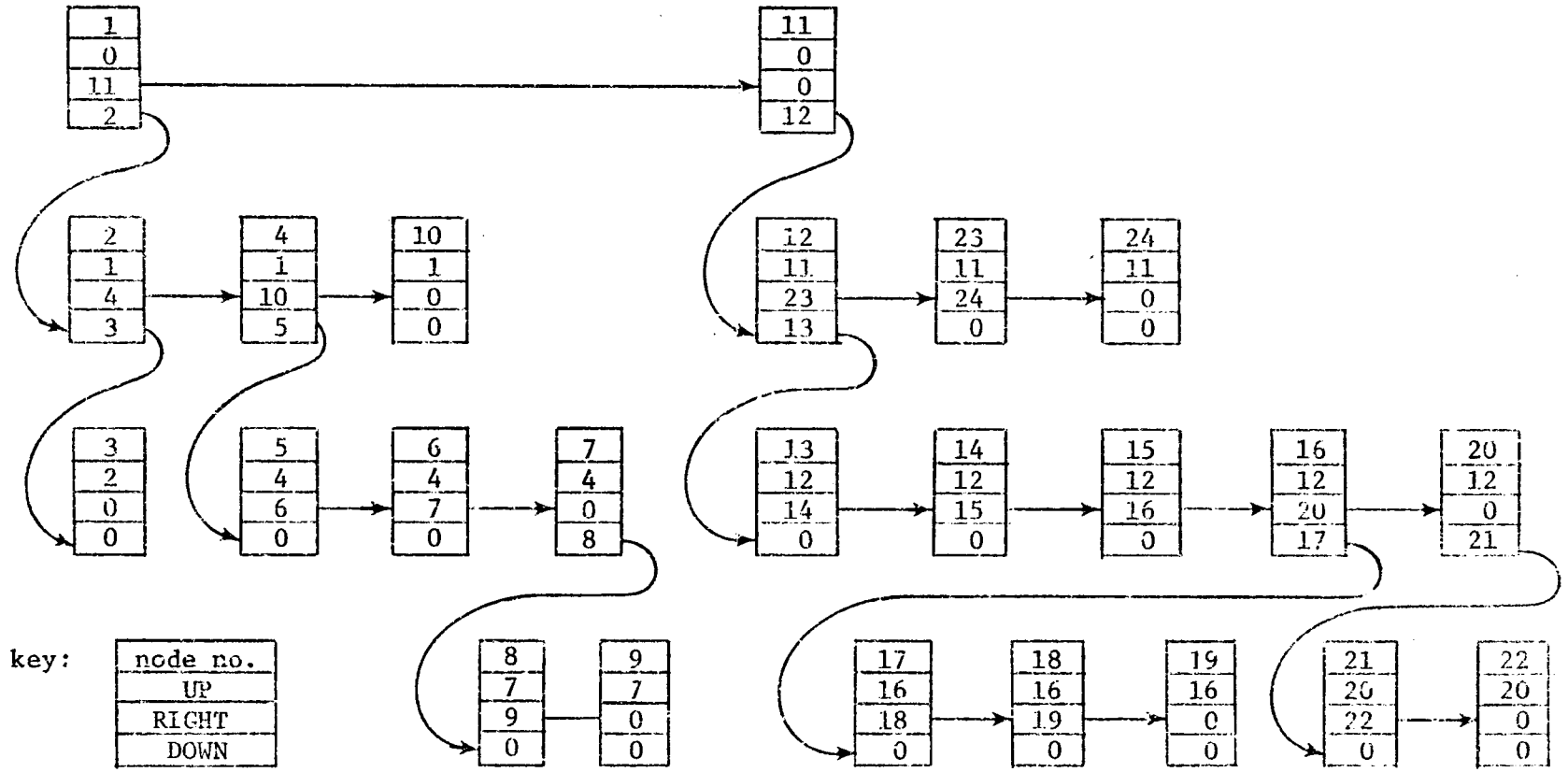


Figure 3

Tree structure for data base of state information.

entry	data values	node identification number	attribute
1	CALIFORNIA	1	State name
2	GEORGIA	11	
3	4589575	11	State population
4	19953134	1	
5	ATLANTA	11	State capital
6	SACRAMENTO	1	
7	ATLANTA	12	City name
8	LOS ANGELES	2	
9	SAN DIEGO	4	
10	746500	4	City population
11	1390164	12	
12	2816061	2	
.			
.			
.			

Figure 4

Example of an inverted file structure.

attribute	beginning entry in inverted file
STATE NAME	1
STATE POPULATION	3
STATE CAPITAL	5
CITY NAME	7
CITY POPULATION	10
NEWSPAPER	31
CIRCULATION	35
COLLEGE	39
ENROLLMENT	41
COMPANY	22
INDUSTRY	25
PRODUCT	28
COUNTY NAME	13
COUNTY SEAT	16
COUNTY AREA	19

Figure 5

Index to the inverted list.

or attributes, associates pointers to the beginning entry number (in the inverted list) of those data values which are occurrences of the corresponding attributes. Even though this index adds more storage overhead in order to reduce search time, it does remain the same size (for a given number of attributes) regardless of the magnitude of the inverted list. Consequently, for large data bases this index is relatively smaller in size and greater in its usefulness in reducing access time to the desired information.

The three structures described above all are important to the retrieval function to meet the goals of the system. Specifically, the tree structure defines logical relationships among the records in the data base, and the inverted list, along with its index, permit the rapid access of information by key. Next, the construction of the actual system files to represent these and other necessary structures is discussed.

File Structure

Due to the often tremendous size of data bases, it is necessary to modify somewhat the use of the data structures described above, and to add several additional structures. Each of these structures and its representation as a system file will be discussed. First, however, it is important to point out that the size of several of these files will be too great to reside in core storage. Since most of the data structures are based on list organizations, it does not appear to be efficient to utilize sequential devices such as magnetic tape units. Instead, magnetic disk or drum direct access devices are used to store the larger structures. This approach also has its drawbacks, however; the cost of direct access can be expensive if an access must be made for each record of a file, for

example, to traverse a tree structure. The inverted list structure requires essentially a sequential search once the indexed starting point is located. Therefore, a logical compromise to this problem is to access a block of records from external storage. This block of records, or page, is more efficient when several records to be used are found on one page, thereby reducing the number of accesses required. The choice of page size itself has a significant effect on retrieval efficiency. For now, the assumption will simply be made that the physical page size will be constant for all files regardless of the logical record size.

Returning now to the design of the system files, it seems that a logical approach to storing the lengthy inverted list is to first partition it into sublists. Furthermore, if each sublist, or partition, is the range of values that occur for an attribute in the data base, then the index to the inverted list is simply an index to the partitions of attribute values. For identification, the file of inverted values is assigned the system name VALUES; and the index to VALUES is given the name DEFINE since its entries correspond to the attributes of the definition tree. The physical files of the system are logically referred to as system tables.

Partitioning the VALUES table by attribute as described, unfortunately, does not completely solve the storage problems. For large data bases, each partition may span more than one physical page of external storage. Rather than adding a pointer for each page to the corresponding entries of the DEFINE table, an additional structure, or table, is used as an intermediate link between the DEFINE and VALUES tables. This table, system name PAGES, has an entry for each of the physical pages in the

VALUES table. Each entry (corresponding to an attribute) in DEFINE points to the beginning of a (linked) list of PAGES entries, each of which points to a page in VALUES. An example of this organization is illustrated in Figure 6. To illustrate, a search for a given state name would begin with entry number 1 of DEFINE which indicates that the first of a list of PAGES entries is entry number 2; and, by following this linked list to entries 3 and 1, it is found that pages 2, 1, and 4 respectively contain the "state name" partition of VALUES.

As indicated in the discussion above, a pointer is simply the entry number of a record relative to the beginning of its own table regardless of partitions or physical page size. In the case of the tree structure, which is the system file FIND, each entry represents a node of the logical structure. Consequently, a pointer to a FIND table entry is logically and numerically the same as the identification number of its corresponding node in the logical structure of the data base. Because of this convention, it is possible to represent a set of node identification numbers (and thereby, the set of information stored in those nodes) by storing the pointers to the corresponding entries of the FIND table. Thus, the selection of information by key terminates with the storage of the pointers to FIND as located in the VALUES table. External to the system files, these stored pointers are interpreted as the node identification numbers of nodes containing the given key. In this way, the representation of sets of node identification numbers is kept independent of internal data structures.

In an attempt to conserve storage, an additional table, the MULTIPLE table, is used in conjunction with the VALUES table. For each entry in

DEFINE table

attribute	pointer to PAGES table
STATE NAME	2
STATE POPULATION	6
STATE CAPITAL	5
CITY NAME	7
CITY POPULATION	4
NEWSPAPER	8
CIRCULATION	9

PAGES table

entry	link to next entry	pointer to page in VALUES
1	0	4
2	3	2
3	1	1
4	0	3
5	6	7
6	0	6

VALUES table

page	data values
1	FLORIDA GEORGIA HAWAII
2	ALABAMA ALASKA CALIFORNIA
3	85416 127901 865462
4	IDAHO KANSAS TENNESSEE

Figure 6

Use of the PACES table.

VALUES that would otherwise contain more than one pointer to the FIND table, there is a single pointer to a MULTIPLE entry that contains the pointers to FIND. This technique prevents unused locations in the VALUES table (since entries are fixed length and the number of pointers to FIND is variable). The use of the MULTIPLE table is illustrated by the example in Figure 7. If the number of pointers exceeds the entry length of the MULTIPLE table, a link to another entry is made as demonstrated by entries 1 and 2 in the example.

The FIND table is somewhat more complex than is implied by the data tree structure of Figure 3 on page 19. Omitted from that diagram is any reference to the actual data values stored in each of the nodes of the tree. Each entry of the system table FIND includes storage locations for these references in addition to the pointers UP, RIGHT, and DOWN as previously described. For each numerical data item, the corresponding storage location contains the machine coded representation of the data value. For each alphabetic item, the storage location contains a pointer (entry number) to a system table of display coded alphabetic values. This table is simply called the PRINT table. Figure 8 demonstrates its use. As with the MULTIPLE table, PRINT conserves storage in its associated table (FIND) due to the variable length of numeric and alphabetic data values.

The system tables described above are modified versions of the tree, inverted list, and indexed sequential data structures. Modifications include the addition of two tables (MULTIPLE and PRINT) that help to minimize storage requirements (of VALUES and FIND respectively) and of another (PAGES) to reduce access time to a table (VALUES). Each of these tables may be very large compared to the physical page size chosen for

VALUES table

data values	pointer
	1
	3
	.
	.
	.

MULTIPLE table

entry	link	pointers to FIND			
1	2	6	7	11	16
2	0	23	26	0	0
3	0	1	3	4	0
4	5	8	9	13	14
.					
.					
.					

FIND table

entry	
1	
2	
3	
4	
5	
6	
.	
.	
.	

Figure 7

Use of the MULTIPLE table.

FIND table

entry	UP	RIGHT	DOWN	first data value	second data value	third data value
1	0	11	2	1	19953134	2
2	1	4	3	3	2816061	0
3	2	0	0	4	1006021	0
4	1	10	5	5	746500	0
5	4	6	0	6	119996	0
.						
.						
.						

PRINT table

entry	data value
1	CALIFORNIA
2	SACRAMENTO
3	LOS ANGELES
4	LOS ANGELES TIMES
5	SAN DIEGO
6	SAN DIEGO TRIBUNE
.	
.	
.	

Figure 8

Relationship of tables FIND and PRINT.

external storage devices. Therefore, any reference to a table entry by a pointer must first be translated to a page number and page entry number. In this way, only the page containing the needed entry is transferred from external storage. Later in this paper there will be a discussion of the control mechanism that performs the task of reading pages into core storage.

Retrieval Algorithms

Having discussed the data structures and the system tables implementing them, it is now possible to describe the way in which the overall organization is used to perform the function of retrieval. In meeting the system goals, the following tasks have been described as being adequate:

1. The selection of those nodes (records) in the data base containing a given key.

2. The qualification of one or more nodes by locating the descendents and direct ancestors of each node.

3. The display of those data values referenced by a set of nodes.

The system names for these tasks are SELECT, QUALIFY, and PRINT respectively. A brief description of each task follows which includes the relationships of the tasks to the system tables.

The most involved retrieval task, or command as seen by the user, is SELECT which involves four of the six system tables already described. To issue this command causes a representation of the selection set to be named and stored for later use. The form of the SELECT command is as follows:

SELECT set-name WHERE attribute-relation-value

such that "set-name" is a unique user-supplied name by which the stored

representation is referenced, and the "attribute-relation-value" triple defines a range of keys that provide the selection criterion.

The retrieval algorithm for processing a SELECT command begins with a search for the existence of "set-name" as a name already in use. If this name is unique, then a search of the DEFINE table is initiated for the entry corresponding to the given attribute. A pointer in this entry indicates the first of a linked list of PAGES entries that reference physical pages in the VALUES table. This sequence of pages make up the partition of VALUES entries all of which are occurrences in the data base of the data values for the given attribute. At this point an example may be helpful. Suppose that the command

SELECT SETA WHERE CITY POPULATION IS LESS THAN 1000000

is issued for the previously described example data base (as shown in Figure 2 on page 12). If SETA is an unused set name, then a search of DEFINE is begun for the CITY POPULATION entry. This may be a sequential search since the DEFINE table is relatively small. As seen in Figure 9, this entry contains a pointer to the fourth entry in PAGES. The relation IS LESS THAN means that each data value algebraically less than 1000000 in the CITY POPULATION partition is to be found. To simplify this search, the VALUES entries are sorted into ascending order for each partition. Thus, each entry is scanned sequentially until a value greater than or equal to 1000000 is found. For partitions spanning more than one page of storage, the scan is interrupted whenever the end of a page is encountered in order to locate the next page (by linking to the next PAGES entry). Furthermore, to reduce the search time for this and other relations, the high value of each page is stored in its corresponding PAGES entry. Since

DEFINE table

attribute	pointer to PAGES table
STATE NAME	3
STATE POPULATION	1
STATE CAPITAL	2
CITY NAME	5
CITY POPULATION	4
NEWSPAPER	6
CIRCULATION	7
.	
.	
.	

PAGES table

entry	link	high value	pointer to VALUES
1	0	19953134	1
2	0	SACRAMENTO	3
3	0	GEORGIA	5
4	0	2816061	2
5	0	SAN DIEGO	7
.			
.			
.			

VALUES table

page	entry	data values	POINTER to FIND
1	1	4589575	11
	2	19953134	1
	3		0
	4		0
	5		0
2	1	745600	4
	2	1390164	12
	3	2816061	2
	4		0
	5		0
3	1	ATLANTA	11
	2	SACRAMENTO	1
.			
.			
.			

Figure 9

System tables used in SELECT command.

1000000 is less than the high value of entry 4, it is known that the VALUES entries satisfying this a-r-v triple are all located on page 2. For other a-r-v triples, substantial search time may be saved by searching only those pages of the partition known to contain the desired values. In this case, only the value of 746500 in entry 1 is less than 1000000; therefore, the associated pointer to FIND is the sole member of the stored set SETA.

The QUALIFY command causes the creation, naming, and storage of a set of node identification numbers representing the qualification set of the nodes in a previously stored set. That is, this new set is the result of adding the node identification numbers of the descendents and direct ancestors of each node represented in the old set. The form of the command is

QUALIFY set-name$_1$ GIVING set-name$_2$

where set-name$_1$ is the name of a stored set and set-name$_2$ is the name under which the qualification set is to be stored.

The first action to be taken as a result of this command is to locate the stored set to be qualified. Next, one by one each node identification number in this set is added to the new set along with its ancestors and descendents. The process of qualifying is accomplished by traversing the tree structure represented by the system table FIND via the pointers UP, RIGHT, and DOWN. To locate the ancestors, the tree is traversed upwards by simply following the (linked) list of UP pointers until the null pointer is encountered in the highest level ancestor. Somewhat more complicated is the traversal for descendents summarized by the following algorithm (beginning with the node to be qualified):

Step 1. If the DOWN pointer is not null, then advance to that node and repeat this step, otherwise (DOWN is null) go to Step 2.

Step 2. If the RIGHT pointer is not null, then advance to that node and go to Step 1, otherwise go to Step 3.

Step 3. If the UP pointer is to the original node being qualified then terminate, otherwise advance to that node and go to Step 2.

To illustrate, this algorithm is applied to node number 4 in Figure 3. The UP pointer indicates that node number 1 is the only ancestor since the list of UP pointers ends with this node. By applying the algorithm above to locate descendents, the following is the sequence of node identification numbers encountered in traversing the tree: 4, 5, 6, 7, 8, and 9.

The result of the two commands

SELECT SETA WHERE CITY POPULATION IS LESS THAN 1000000

QUALIFY SETA GIVING SETB

is, then, two representations of sets of node identification numbers stored under the names SETA and SETB representing numbers (4) and (1, 4, 5, 6, 7, 8, 9) respectively. Having formed these sets a user may want to inspect the data values represented by them. The command PRINT instructs the system to display those values desired. The form of the command is

$$\text{PRINT}\left\{\begin{array}{l}\text{attribute (,attribute)}\\ \text{ALL}\end{array}\right\}\text{OF}\left\{\begin{array}{l}\text{SET set-name}\\ \text{NODE node-identification-number}\end{array}\right\}$$

where either of the options in brackets may be specified. The option "attribute (,attribute)" implies that one or more specific attributes (of the definition tree) in each node are to be displayed, while the option "ALL" indicates that all attributes are to be displayed. The choice of

displaying information referenced by a single or set of node identification numbers is available. In the former case the "node-identification-number" must be supplied, and in the latter, the "set-name" must reference an existing stored set. To illustrate,

PRINT STATE NAME, COLLEGE, ENROLLMENT OF SETA

produces the results

NODE NUMBER 1

CALIFORNIA

NODE NUMBER 5

UNIV OF SAN DIEGO

1478

and the command

PRINT ALL OF NODE 10

the results

NODE NUMBER 10

ALMEDA

OAKLAND

733

If the "SET set-name" option is specified, the system locates the set and repeats the following process for each node identification number. First, the corresponding entry in the FIND table is located. Within that FIND entry there is a pointer to the DEFINE table at the beginning of the (sequential) list of entries corresponding to the attributes represented in the node. Then, by comparing the specified attributes to those in DEFINE, only those data values referenced by the FIND entry are displayed (unless the "ALL" option is chosen, in which case all data values are displayed). As previously described, numeric values are stored in the FIND

itself; otherwise, there are pointers to the alphanumeric values in the PRINT table (as shown in Figure 8 on page 28).

It is seen that the organization of data into system tables allows the system to perform a range of user commands and provide the retrieval function specified by the system goals. Trade-offs are made to best meet these goals at the expense of functions not considered critical to the overall efficiency of the system. The use of the tree, the inverted list, and other indexing structures permit efficient retrieval, while the actual system tables are constructed to conserve storage for these high overhead data structures. Retrieval commands reference system tables to permit the desired tasks of selection of information by key values, the logical association of data to other data in the tree (qualification), and the display of specified information for user inspection.

CHAPTER IV

SET REPRESENTATION AND MANIPULATION

The function of set manipulation is essential to the user whose retrieval requests demand the satisfaction of two or more logical conditions. It has been shown that the proposed data management system provides for the selection of occurrences of data values based on the a-r-v triple. By representing these occurrences as a selection set of node identification numbers, it is then possible to perform operations on two of these sets--such as extracting common node numbers or combining the node numbers of both to create a third set of information. This discussion will describe the way in which sets may be represented and manipulated. It is also to be demonstrated that these techniques may be applied to nearly any data management system since they are independent of data structure.

Set Operations

The term "set" has been used up to now to simply mean a collection of node identification numbers (NIN). However, by considering the range of numbers in the data base to be elements of the "universe" of NIN's, it is possible to apply set theory to the data base. (12) In this way, a "set" of NIN's takes on a more formal definition, and it becomes possible to speak of the operations of union, intersection, and complement of sets of NIN's. These operations allow the user to create new sets whose NIN's represent information satisfying complex conditions. Suppose, for example, it is desired to isolate all occurrences (nodes) in the example data base

where the city population is greater than one million but less than two million. Two selection sets could first be formed using the retrieval commands

SELECT SETL WHERE CITY POPULATION IS GREATER THAN 1000000

SELECT SETG WHERE CITY POPULATION IS LESS THAN 2000000 .

The resulting sets named SETL and SETG would contain the NIN's (2, 12) and (4, 12) respectively. In terms of set theory, it is seen that to meet the requirements of both conditions a NIN must be an element of both sets. Thus, the intersection of these two sets is the set of NIN's satisfying the desired request, in this case the set with the single element 12. The intersection of sets is indicated in the user language by the operator AND, such that in the example the new set is formed by the expression SETL AND SETG. Similarly, the operator OR indicates the union of sets. For example, SETL OR SETG is the set of elements belonging to either set, that is 2, 4, and 12. The complement of a set includes every element of the universe except the elements of that set; this operation is indicated by the operator NOT--such as NOT SETL which is the set 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 22, 23, and 24.

One of the advantages of being able to apply set theory to sets of NIN's is that the user is assured of the mathematical integrity of the system. The results of set manipulation tasks are guaranteed to be consistent with the properties of set operations on sets of positive integers. Being aware of the set theoretical implications of the system, a user may unambiguously describe the conditions under which information is to be isolated and stored.

The retrieval system described is based on the node (of the tree

structure) as the basic retrieval element of the system. That is, the set of all nodes in the data base represents the universe from which distinct subsets of elements may be collected and stored. Sets resulting from set operations are subsets of the same universe as the operand sets. Other systems employing the tree structure, such as TDMS (1), are based on the qualification set of a given node as the basic retrieval element. In other words, these systems automatically qualify each member of a selection set to include all logically related data (ancestor and descendent nodes). Therefore, the universe of such systems is the set of all qualification sets in the tree. Sets resulting from set operations, then, are also qualification sets. Consequently, retrieval requests generally will not be the same for these two types of systems. The TDMS type of system will always include ancestor and descendent node relatives so that these retrievals typically are more broad in scope. However, it may be to the user's advantage to be able to obtain the more specific information from nodes retrieved by the former system; later, the selection set may then be qualified for related nodes via the QUALIFY command. A detailed discussion of the differences in these two systems is given by Hardgrave (7).

Set Representation

It has been seen that entries (nodes) in the data base are assigned identification numbers in order that the representation of subsets of the data base may be in the form of a set of positive identification numbers. Therefore, the problem of manipulating sets of information in the data base reduces to one of efficiently representing subsets of positive integers. For relatively small data bases the requirements for set storage are not great. An adequate method of representing these sets is a simple

sequential list (array) of storage locations, each containing a node identification number (NIN). However, for large data bases, as many tend to be, this scheme is too inefficient. It is important to use a technique that permits the efficient storage of a set representation while maintaining the ability to perform set operations on these representations.

One method of storing sets of positive integers in order to conserve storage space is the Bit Stream Set Representation (BSSR) which assigns a single binary digit (bit) in storage to each integer in the universe of NIN's in the data base. While this method is similar to that suggested by Bloom (2), the author has proposed a method in which further savings in storage may be attained. The techniques herein described were first presented by the author at a session on trends in data base management at the First National Computer Conference and Exposition (3).

The discussion to follow includes the definitions describing the BSSR method as will as the algorithms required to perform set operations. To demonstrate the important concepts, examples are supplied from a data base that is assumed to contain no more than $2^{31} - 1$ nodes (a limit so chosen because it is the maximum value allowed on a computer with thirty-two bit storage locations for integers.

Definition of a Bit Stream Set Representation

The range of NIN's in the data base may be considered to be the set P defined as follows:

$$P = \{ i \mid i \text{ is an integer such that } 1 \leq i \leq k \} .$$

Let S be a subset of P, and let R be a Bit Stream Set Representation of S, defined as follows:

$$R = (b_0, b_1, \ldots , b_k)$$

such that

1. If $b_0 = 0$, then $b_i = 1$ for all $i \in S$, and
   $b_i = 0$ for all $i \notin S$;
2. if $b_0 = 1$, then $b_i = 0$ for all $i \in S$, and
   $b_i = 1$ for all $i \notin S$.

Thus, the BSSR is a one to one mapping from the set of all subsets of integers 1 through k to the ordered k+1 bits. Note that the bit b_0 acts as an indicator as to which of the two sets of rules defined above are in effect. However, the remainder of the discussion will be under the assumption that the second set of rules (for $b_0 = 1$) will not be used except in special cases which are to be discussed later. Consequently, under the first set of rules, the inclusion of the integer i in the set S is represented by the ith bit having the value "1". Likewise, the integer i not in the set S is represented by its corresponding ith bit having the value "0". Each subset of P has a unique bit stream representation. Furthermore, every subset of the integers 1 through k may be represented by a bit stream of length k+1.

For example, the set {2, 5, 100000, 2000000000} is a subset of the integers 1 through $2^{31} - 1$ (that is, 2,137,483,647) and may be represented by the BSSR

$$(0010010 \ldots 010 \ldots 010 \ldots 0)$$

such that

$$b_i = 0 \quad \text{for} \quad 0 \leq i \leq 2^{31} - 1, \text{ except where}$$

$$b_2 = b_5 = b_{100000} = b_{2000000000} = 1 \ .$$

In this manner, any subset of the integers 1 through $2^{31} - 1$ may be represented by a BSSR of length 2^{31}.

When a set is represented in the bit stream form, a substream of zero bits may be eliminated in order to save space with no loss of information. By dividing the BSSR into equal length substreams called blocks, it is possible to omit those which contain all zero bits. However, in order to maintain the contiguous numbering scheme of the bits, it is necessary to indicate any omissions with a directory. The definition of a block and the way in which it may be used follows.

Let R be the BSSR of the set S as previously defined. Let R be divided into k/m distinct substreams $B_{0,j}$ of length m. Then each $B_{0,j}$ is a block of R and is defined as follows:

$$B_{0,j} = (b_{mj}, b_{mj+1}, \cdots, b_{m(j+1)-1}) \quad \text{for} \quad 0 \leq j \leq k/m - 1 .$$

This definition implies that each bit of R is contained in one, and only one, of the k/m blocks. In addition, to indicate a block of all zero bits, the notation

$$B_{0,j} = \overline{0}$$

implies that $b_i = 0$ for $mj \leq i \leq m(j+1)-1$.

In the previous example, the BSSR of the set S may be divided into 2^{20} blocks of length $m = 2^{11}$ bits as follows:

$$B_{0,0} = (0010010 \ldots 0)$$
$$B_{0,1} = (0 \ldots 0)$$
$$\vdots$$
$$B_{0,48} = (0 \ldots 010 \ldots 0)$$
$$\vdots$$
$$B_{0,976562} = (0 \ldots 010 \ldots 0)$$
$$\vdots$$
$$B_{0,1048575} = (0 \ldots 0)$$

such that

$$B_{0,0} \neq \overline{0} \ , \quad B_{0,48} \neq \overline{0} \ , \ B_{0,976562} \neq \overline{0} \ ,$$
$$\text{otherwise} \quad B_{0,j} = \overline{0} \quad \text{for} \quad 1 \leq j \leq 2^{20} - 1 \ .$$

Having described the blocks of a BSSR, it may now be seen that blocks of all zeros may be omitted when a directory as defined below is used.

Let R be a BSSR of the set S and let $B_{0,0}, B_{0,1}, \cdots, B_{0,k/m-1}$ be the blocks of R as defined above. Then the bit stream directory D_1 may be defined as follows:

$$D_1 = (d_{1,0}, d_{1,1}, \cdots, d_{1,k/m-1})$$

such that

$$\text{if} \quad B_{0,j} \neq \overline{0} \ , \quad \text{then} \quad d_{1,j} = 1 \ , \text{ and}$$
$$\text{if} \quad B_{0,j} = \overline{0} \ , \quad \text{then} \quad d_{1,j} = 0 \quad \text{for} \quad 0 \leq j \leq k/m - 1 \ .$$

In the example, the directory for the BSSR of S is

$$D_1 = (10 \ldots 010 \ldots 010 \ldots 0)$$

where

$$d_{1,0} = d_{1,48} = d_{1,976562} = 1 \ ,$$
$$\text{otherwise} \quad d_{1,j} = 0 \quad \text{for} \quad 0 \leq j \leq 2^{20} - 1 \ .$$

By prefixing the directory to the three non-zero blocks of the BSSR of S, the following unambiguous representation results:

$$\underbrace{(10\ldots010\ldots010\ldots0}_{D_1} \ \underbrace{0010010\ldots0}_{B_{0,0}} \ \underbrace{0\ldots010\ldots0}_{B_{0,48}} \ \underbrace{0\ldots010\ldots0)}_{B_{0,976562}}$$

The length of the resulting BSSR is $2^{20} + 3(2^{11}) = 1{,}054{,}720$ which is substantially less than the approximately two billion bits required by the BSSR without a directory.

As described above, the directory D_1 indicates which blocks of the BSSR are omitted in order to conserve storage space. However, the length

of D_1 (possibly very large itself) may also contain relatively long substreams of zero bits. Thus, by dividing this directory into blocks of length n, a higher level directory D_2 may be constructed to indicate omissions in D_1. In fact, the idea of a "directory to a directory" may be applied recursively as many times as desired to eliminate all blocks containing all zero bits.

A directory block is defined in the same manner as is a block of R (that is, $B_{0,j}$). Therefore, first let the level L directory D_L be divided into substreams of length n_L; then let $B_{L,j}$ be a directory block of D_L defined as follows:

$$B_{L,j} = (d_{L,0}, d_{L,1}, \cdots, d_{L,n_L})$$

such that

$B_{L,j}$ is a substream of D_L for $L \geq 1$, and for

$$0 \leq j \leq \frac{k}{mn_1n_2\cdots n_L} - 1 = \max_L .$$

The directory D_1 is defined to be the lowest level directory (L = 1) to R in a BSSR. Using the above definition of directory blocks, a higher level directory D_L may be defined as follows:

$$D_L = (d_{L,0}, d_{L,1}, \cdots, d_{L,\max_L})$$

such that

$$\text{if } B_{L-1,j} \neq \overline{0}, \text{ then } d_{L,j} = 1,$$

$$\text{if } B_{L-1,j} = \overline{0}, \text{ then } d_{L,j} = 0 \text{ for } 0 \leq j \leq \max_L .$$

The level 1 directory D_1 of the example BSSR may be divided into blocks of length $n_1 = 2^{11}$ as follows:

$$B_{1,0} = (10 \ldots 010 \ldots 0)$$

$$B_{1,1} = (0 \ldots 0)$$

$$\vdots$$

$$B_{1,428} = (0 \ldots 010 \ldots 0)$$

$$\vdots$$

$$B_{1,511} = (0 \ldots 010 \ldots 0)$$

where all blocks except $B_{1,0}$ and $B_{1,428}$ contain all zero bits. Thus, the level 2 directory for the BSSR is

$$D_2 = (10 \ldots 010 \ldots 0)$$

such that

$$d_{2,0} = d_{2,428} = 1 \text{ , and}$$

$$\text{otherwise} \quad d_{2,j} = 0 \quad \text{for} \quad 0 \leq j \leq 2^9 - 1 \text{ .}$$

Each D_1 directory block that contains all zeros may be omitted when prefixed by D_2. In the example BSSR, the representation would then be

$$\underbrace{(10\ldots010\ldots0}_{D_2} \ \underbrace{10\ldots010\ldots0}_{B_{1,0}} \ \underbrace{0\ldots010\ldots0}_{B_{1,428}} \ \underbrace{0010010\ldots0}_{B_{0,0}} \ \underbrace{0\ldots010\ldots0}_{B_{0,48}} \ \underbrace{0\ldots010\ldots0)}_{B_{0,976562}}$$

The length of this stream is $k/(mn_1) + 2n_1 + 3m = 2^9 + 2(2^{11}) + 3(2^{11}) = 10752$. Although this length is small in comparison with a BSSR without directories, it is somewhat long for a representation of only four NIN's. However, it is clear that a much larger number of values could be included in the representation without increasing its length. For example, if each bit of the three blocks of R were "1" (D_1 and D_2 remaining unchanged), then the number of values represented would be $3(2^{11}) = 6144$. While it is unlikely that any subset chosen would "fit" this well, it is not unlikely for a set of values to be somewhat grouped depending upon the numbering scheme of the nodes in the data base. When the sequence of NIN's are assigned by attempting to descend at each node (called a depth first

traversal), then the NIN's of the ancestors and descendents of a given node tend to be relatively close together. Consequently, sets that include qualification sets will typically show this grouped characteristic. A very important aspect affecting the efficiency of a BSSR is the block size. In general, it is expected that some block size is optimum for a given data base. The numbering scheme, the number of levels in the tree, and the number of nodes in the data base all influence the efficiency of a given block size.

Set Operations with BSSR's

One of the advantages of representing sets with bit streams is the ease with which set operations may be performed. For sets in the BSSR form, these operations are simply the Boolean operations AND, OR, and NOT. The operations AND and OR are possible by pairing the bits of both BSSR's corresponding to each integer value in the universe. For example, to perform the Boolean AND on two subsets

$$S_1 = \{2, 5, 100000, 2000000000\}$$

$$S_2 = \{1, 2, 100000\}$$

(from the universe 1 through $2^{31} - 1$), each pair of bits in the BSSR's

$$BSSR_1 = (0010010 \ldots 010 \ldots 010 \ldots 0)$$

$$BSSR_2 = (0110000 \ldots 010 \ldots 000 \ldots 0)$$

may be operated on by the Boolean AND. The resulting bit stream will be called BSSR*, such that

$$BSSR^* = (0010000 \ldots 010 \ldots 000 \ldots 0) .$$

The Boolean NOT may be performed by simply reversing each binary value (except b_0). For BSSR's without directories, the complement will

require exactly the same amount of storage. However, in large universes it is impractical not to use directories to eliminate long streams of zeros. In this case, the BSSR for the complement of a small subset may be quite large. Therefore, it may be more efficient to simply reverse bit b_0 to "1" to reverse the meaning of the bit stream according to the second set of rules given in the definition of a BSSR. By reversing the meaning of each bit, the complement of the BSSR is obtained while not requiring any additional storage. Since complements may often be used for intermediate, or temporary, results it is more efficient to convert a BSSR to its complement (via a Boolean NOT so that $b_0 = 0$) only when necessary.

For BSSR's without directories, Boolean operations are straightforward. However, for BSSR's with directories these operations require algorithms to produce the resultant BSSR*. These algorithms assume that the BSSR's used in an operation will have common block lengths and levels of directories. If necessary, higher level directories of "all ones" may be generated to meet this requirement. The algorithms permit operations on BSSR's using either set of definition rules ($b_0 = 0$ or $b_0 = 1$) so that conversion to the longer form via a Boolean NOT need not be made.

The algorithm for the Boolean AND is as follows:

Step 1. Let L be equal to the number of the highest directory level of the two BSSR's.

Step 2. If neither of the b_0 bits of the two BSSR's are "0", then the resultant level L blocks of BSSR* are equal to the Boolean AND of the two BSSR's level L blocks.

If one of the b_0 bits is "1", then the level L blocks of BSSR* are equal to those of the <u>other</u> BSSR.

Step 3. If the b_0 bit of a BSSR is not "1", then for each $d^*_{L,j} = 0$

in D_L^* such that $d_{L,j} = 1$ in that BSSR, omit the block $B_{L-1,j}$ from D_{L-1}. Furthermore, omit all lower level blocks (through level zero) within the range of $B_{L-1,j}$.

Step 4. If L > 1 then subtract "1" from L and go to Step 2.

Step 5. If neither b_0 bit is equal to "1" then go to Step 6; otherwise, replace the level zero blocks of the BSSR with $b_0 = 1$ with their complement (Boolean NOT). Then, for each $d^*_{1,j} = 1$ in D_1^* such that $d_{1,j}$ in D_1 of that BSSR, insert the block $B_{0,j} = \overline{1}$ (which is a stream of all ones) in the proper relative position in D_0.

Step 6. The level zero blocks of BSSR* are equal to the Boolean AND of the level zero blocks of the two BSSR's.

If both BSSR's have $b_0 = 1$, then the algorithm for the Boolean OR should be used to obtain a BSSR* also with $b_0 = 1$; since by theorem NOT R_1 AND NOT R_2 is equivalent to NOT (R_1 OR R_2). In this case, both of the BSSR's should be treated <u>as if</u> $b_0 = 0$. The algorithm for the Boolean OR is as follows:

Step 1. Let L be equal to the number of the highest directory level of the two BSSR's.

Step 2. The level L blocks of BSSR* are equal to the Boolean OR of each corresponding pair of bits of the level L blocks of the two BSSR's.

Step 3. For each $d^*_{L,j} = 1$ in D_L^* such that $d_{L,j} = 0$ in a BSSR, insert the block $B_{L-1,j} = \overline{0}$ in the proper relative position in D_{L-1} in that BSSR.

Step 4. If L > 1, then subtract "1" from L and go to Step 2.

Step 5. If neither $b_0 = 1$, then go to Step 6; otherwise replace the level zero blocks of the BSSR with $b_0 = 0$ with their complements.

Finally, perform the Boolean AND on the level zero blocks of the two BSSR's to obtain the resultant level zero blocks of BSSR* with $b_0 = 1$. Then terminate.

Step 6. The level zero blocks of BSSR* are equal to the Boolean OR of the level zero blocks of the two BSSR's.

Similar to the previous descussion, if both BSSR's have $b_0 = 1$ then the algorithm for the Boolean AND should be used to obtain a BSSR* also with $b_0 = 1$. This is possible because of the theorem NOT R_1 OR NOT R_2 is equivalent to NOT (R_1 AND R_2). As before, both of the BSSR's should then be treated as if $b_0 = 0$.

As discussed above, the algorithm for the Boolean NOT should not be used when it is more efficient to simply reverse the bit b_0 to "1". For very large BSSR's, it may be advantageous to use the algorithm to actually reduce the size of the representation. The Boolean NOT algorithm is as follows:

Step 1. Let H be equal to the number of the highest level directory of the BSSR, and let L = 0. (Note that R = D_0).

Step 2. Perform the Boolean NOT on each bit in the level L blocks by reversing its vinary value (with the exception of b_0).

Step 3. For each $B_{L,j} \neq \overline{0}$, change its corresponding L + 1 directory bit to "0".

Step 4. Insert all omitted blocks from D_L with blocks of "all ones".

Step 5. Omit all level L blocks where $B_{L,j} = \overline{0}$.

Step 6. Add "1" to L. If L < H then go to Step 2; otherwise terminate.

Although it would not be instructive to continue the discussion of

these Boolean operations here, some examples given in the Appendix illustrate the use of the above algorithms.

The advantages of using the BSSR technique in this system for large data bases are clear; the speed of Boolean operations and savings in storage requirements make possible the efficient operation of a set-oriented retrieval system. In addition, it may be seen that since the set manipulation function is performed external to the data structure, this function could be used with any system. In particular, the BSSR method may be of great value to the proposed relational data bases in which massive amounts of information may be stored for a large number of users. (6) By storing representations of sets, a user avoids repeated retrievals and their time-consuming searches. The system itself may find the BSSR useful in internal table representations (for example, to replace lists of pointers) or to act as "masks" that filter user-created sets for security purposes.

System Commands

The creation of sets of node identification numbers by the commands SELECT and QUALIFY cause the representation of the numbers as a BSSR and its storage on a page (or pages) of external storage. The user-supplied set name is placed in an entry of the system table SAVED with a reference to the page on which the BSSR is stored. In this way, access to a stored set is possible by searching the SAVED table for the given set name.

The FORM command allows a user to perform Boolean operations on stored sets by creating a new stored set from any valid Boolean expression. This command is specified in the following way:

FORM: set-name = Boolean-expression

where "set-name" is the name under which the resulting set is to be referenced in the SAVED table, and "Boolean-expression" may be constructed by using the re-write rules below in any order (where, here, "set-name" is the name of a previously stored set).

Boolean-expression → (Boolean-expression)

→ Boolean-expression AND Boolean-expression

→ Boolean-expression OR Boolean-expression

→ NOT Boolean-expression

→ set-name .

For example, a new set may be created from the stored sets SET1, SET2, and SET3 which specifies that the result of the OR operation on SET1 and SET2 is to be used in an AND operation with SET3. The new set SET4 in BSSR form is then stored and referenced by an entry in the SAVED table. This command would be

FORM: SET4 = (SET1 OR SET2) AND SET3 .

Since storage space is not unlimited, it is necessary to be able to remove stored sets that are no longer needed. The command

ERASE set-name

causes the set stored under "set-name" and its corresponding entry in the SAVED table to be removed (that is, made available for other use).

Summary

It has been shown that a data management system based on set theory permits a user to form retrieval requests consistent with the properties of set operations. In this way, the user may expect mathematically and logically sound results to his queries. Reference to collections of information satisfying his queries is made possible by a representation of the

of the set of node identification numbers of those nodes (that is, records in the data base) containing that information. These sets are given a user-supplied name and stored for future reference. Boolean expressions enable the user to create new sets from those previously stored via complex operations. For large data bases, the problem of set representation may be solved by the Bit Stream Set Representation which reduces storage requirements for long lists of node identification numbers. This bit stream technique is independent of internal data structure and may be useful for any large data base application.

CHAPTER V

SYSTEM DESIGN

One of the goals of the data management system described is to design the program modules in such a way that changes and additions to the system may be easily accomplished. The resulting advantages of system flexibility are obvious. The ability to construct specialized systems as well as to allow the evolution of new systems is not limited by design constraints imposed by many existing generalized systems. The philosophy is that specific system characteristics such as storage devices, table formats, and language syntax, should be confined to as few system modules as possible. That is, each module should be designed to supervise a particular system feature, or unit, so as to hide its operation from other modules. In this way, changes to a system unit are unnoticed by external modules (11).

An overview of the components making up the system may be seen in Figure 10. The system functions represented in this diagram indicate how the major features of the system are isolated into separate program components. Each performs one of the tasks vital to achieving the system goals as defined in Chapter Two. The descriptions of many of these tasks independent of the system as a whole have already been given; next it is to be shown how each fits into the overall plan.

The Language Translator provides the interface between the English-like command language and the control program responsible for the performance of the commands. Since the number of commands now available to the user is small, it is doubtful that there need be any substantial modifica-

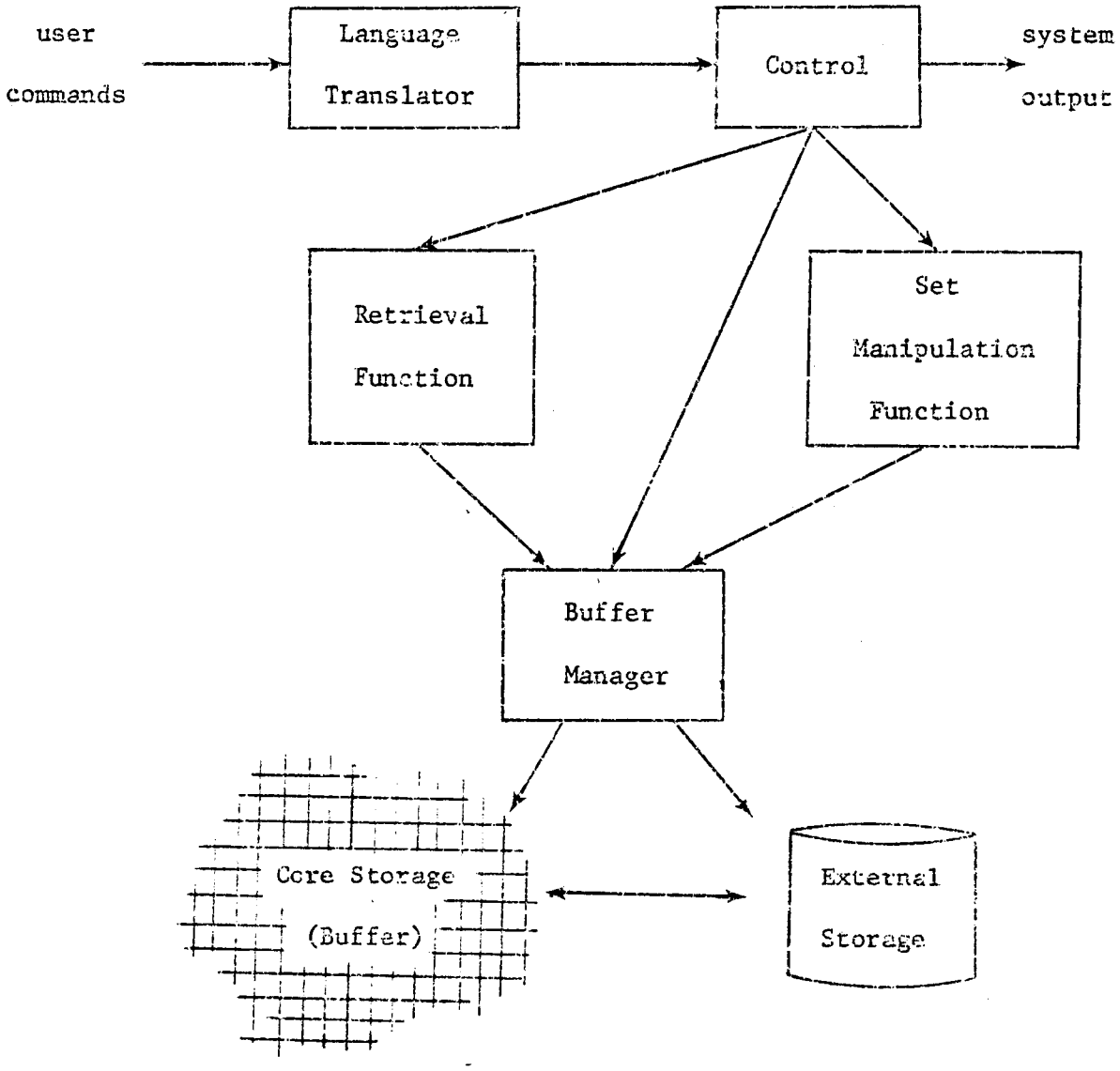


Figure 10

System Overview.

tions to the language until more tasks are available. However, it is possible to change the syntax without disturbing system operation. The simplicity of the current set of commands is such that the translator does little more than act as a preprocessor by arranging codes and data in a condensed form for their non-procedural execution. The evolution of a more sophisticated system at a later time, however, may require the substitution of a more powerful translator (possibly for a procedural language).

Upon receiving the queue of commands from the translator, the Control module proceeds to call upon those system components needed to process each command in sequence. The way in which the commands are processed is, of course, unknown to Control. Typically, only the success or failure of a task is reported to the control program, so that the appropriate messages may be returned to the user. In some cases, Control serves as an interface between functions when parameters or information need to be exchanged. Some communication also exists between Control and the Buffer Manager concerning the location and priorities of tables in the buffer area. In general, the Control module is simply a clearing house for command processing and involves very few operations of its own. However, the capability exists of modifying its range of tasks to include new system components or even of modifying its overall function in the system. For example, it could conceivably be converted to a subroutine enterable from a user main program, or perhaps coud be modified to handle a procedural set of instructions.

The retrieval function controls the use and maintenance of the system tables (that is, the data structures) which permit the tasks of SELECT,

QUALIFY, and PRINT as previously described. This table structure is completely independent of the other functions. Therefore, it is possible to change table formats or even the data structures themselves when more efficient methods are found or a change in system goals requires new techniques. In fact, the individual tables themselves amy be modified without jeopordizing the others.

The Buffer Manager is responsible for the transmission of data between external storage devices and an internal storage area or buffer. Specifically, it locates a requested page from a system table and makes it available to the given routine. In this way, the Buffer Manager serves as an interface between system modules and physical storage. Thus, the problems of managing external storage devices as well as those of allocating internal core storage are removed from the modules that must actually use the system tables.

The function of set manipulation is strictly independent of the retrieval and storage of sets via the buffer area. The way in which sets are represented and the method by which set operations are performed are unknown even to Control and the Buffer Manager itself. Consequently, changes in the procedures of this module are undetectable to other operations and the user.

The above discussion illustrates some of the ways in which the system is designed for modification and expansion. A more detailed description of the interaction among system tables and among the routines that make up the various system functions follows in the next section. The technique of isolation tasks is also applied within these functions so that they too may be modified without disruption of other tasks.

System Tables

The data structures needed to perform the retrieval tasks are implemented via the system tables DEFINE, PAGES, VALUES, MULTIPLE, FIND, and PRINT which were described in Chapter Three. A definition of the fields making up a record (or entry) of each table and the way in which the tables interact via pointers is shown in Figure 11. A pointer to a table entry is simply the entry number relative to the beginning of that table. Since each table is divided into fixed length pages for transmission from external mass storage devices, the entry number must always be translated into a page number and page entry number. Thus, access to a given entry is preceded by the Buffer Manager's reading in the needed page to the buffer area.

Each of the retrieval tables is assigned a program supervisor that is responsible for the access to, or search for, any desired entry. These supervisors, of course, operate independently of one another under the control of a module that organizes the performance of a given task. Therefore, the logical data fields and record length may be changed even though this table may be used in any number of different tasks. In addition, the supervisors are designed to function using any given physical page size, so that a change in this parameter is apparent only to the Buffer Manager.

The tables needed for the set manipulation function are much less involved than those for retrieval. The table SETS is simply the stored set representations, such that each set is an entry in the table. The table SAVED is an index to SETS, such that each SAVED entry contains the set name and a pointer to its corresponding set. Figure 12 illustrates this organization. Since the SAVED table is also used to store sets

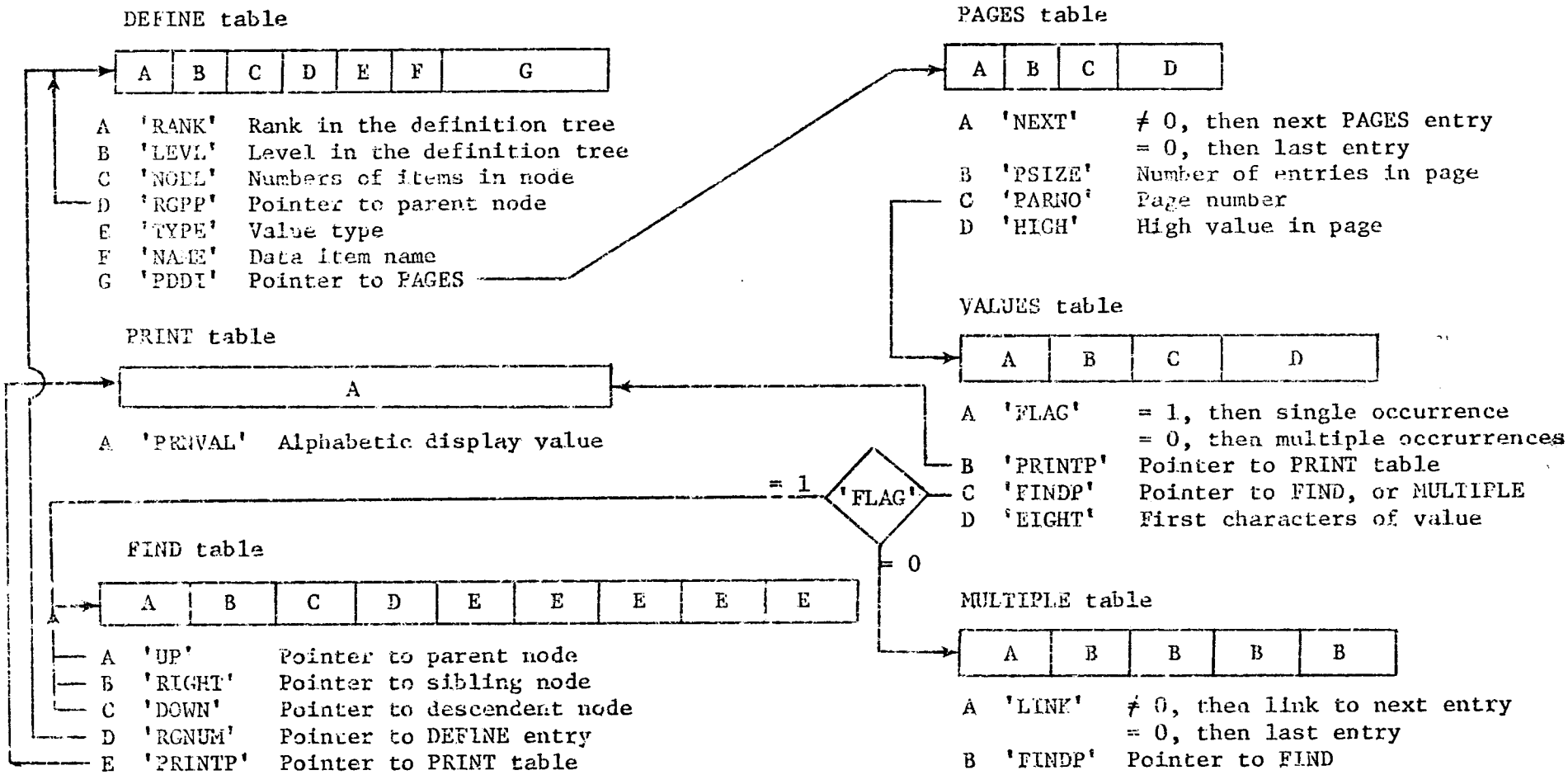


Figure 11

Schematic diagram of retrieval tables.

SAVED table

set name	pointer to SETS
SET1	2
SET2	3
SETB	1
SETA	4
SETL	7
SETG	6
.	
.	
.	

SETS table

entry	representation of set
1	
2	
3	
4	
5	
6	
7	
.	
.	
.	

Figure 12

Example of set tables.

formed by retrieval tasks, it is available only through Control. Both of these tables act under the control of supervisors that are generalized enough to allow changes in set representation, set size, or other parameters without effect on other modules. The schematic diagram of SAVED and SETS is shown in Figure 13. Note that provisions for security and other information concerning the sets have been made.

One last system table to be discussed is the CONTROL table which acts as an index to every page of external storage. This table enables the Buffer Manager to locate the physical address of a requested page given a table name and the page number relative to that table. CONTROL itself is always located in physical page one, so that it may be initially read into the buffer area. Each entry of this table specifies a range of physical, or actual, page numbers that spans a given system table. Later, if a table acquires more pages, then a link from its original CONTROL entry is provided. This linking procedure may then be repeated for subsequent entries. For example, Figure 14 illustrates the way in which the CONTROL table relates each system table to its actual location. Entry four of CONTROL indicates the first three logical pages of the VALUES table are stored in actual pages four, five, and six respectively, and that a link to entry nine is made to reference more pages; entry nine indicates that relative pages four and five of VALUES are stored in actual pages thirteen and fourteen. The use of the CONTROL table in relation to the various system functions will be shown next.

The Retrieval Function

The tasks of selecting nodes of information based on an a-r-v triple, qualifying references to data in order to retrieve logically related

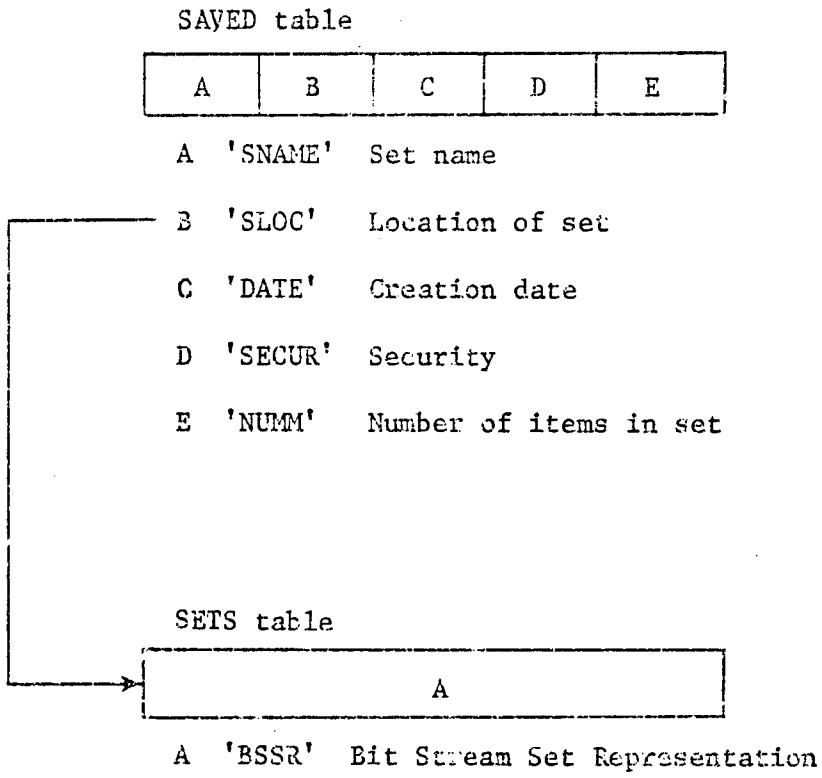


Figure 13

Schematic diagram of set tables.

CONTROL table

entry	table name	beginning page	ending page	link
1	CONTROL	1	1	0
2	DEFINE	2	2	0
3	PAGES	3	3	0
4	VALUES	4	6	9
5	MULTIPLE	7	7	0
6	FIND	8	9	8
7	PRINT	10	11	0
8	FIND	12	12	0
9	VALUES	13	14	0
⋮				

external storage

actual page	table name	relative page
1	CONTROL	1
2	DEFINE	1
3	PAGES	1
4	VALUES	1
5	VALUES	2
6	VALUES	3
7	MULTIPLE	1
8	FIND	1
9	FIND	2
10	PRINT	1
11	PRINT	2
12	FIND	3
14	VALUES	4
15	VALUES	5
⋮		

Figure 14

Example CONTROL table.

information, and displaying information for a user's inspection are assigned to the retrieval function. The commands SELECT, QUALIFY, and PRINT are specified by the user to initiate these tasks. Program subroutines are designed to process commands independently of one another although there are routines common to them all. To obtain a better overall understanding of the interactions among program modules and physical storage, an overview is given in Figure 15. Note that under the direction of CONTROL, a command is forwarded to the appropriate moduel (indicated by the command name itself in Figure 15) for processing. The performance of a given task (command) is made possible by the data structures represented in the system tables. Access to a given table is always made through a program supervisor for that table (indicated by the table name itself in Figure 15). Consequently, CONTROL organizes those tasks constituting the retrieval function, and each command module organizes the table supervisors for the performance of each task.

Figure 15 also indicates a heirarchy of program responsibility with three levels. On the lowest level, table supervisors are responsible only for the access to and operations within their respective tables. In this way, table formats, search algorithms, and other features of a given table are concealed within a single program unit. Overlapping sets of the table supervisors are required by the next higher level modules to coordinate their use in performing the retrieval tasks. The highest level of responsibility belongs to Control which oversees the entire retrieval function. Spanning the lower levels of responsibility is the Buffer Manager which acts directly or indirectly as a service module to all other modules.

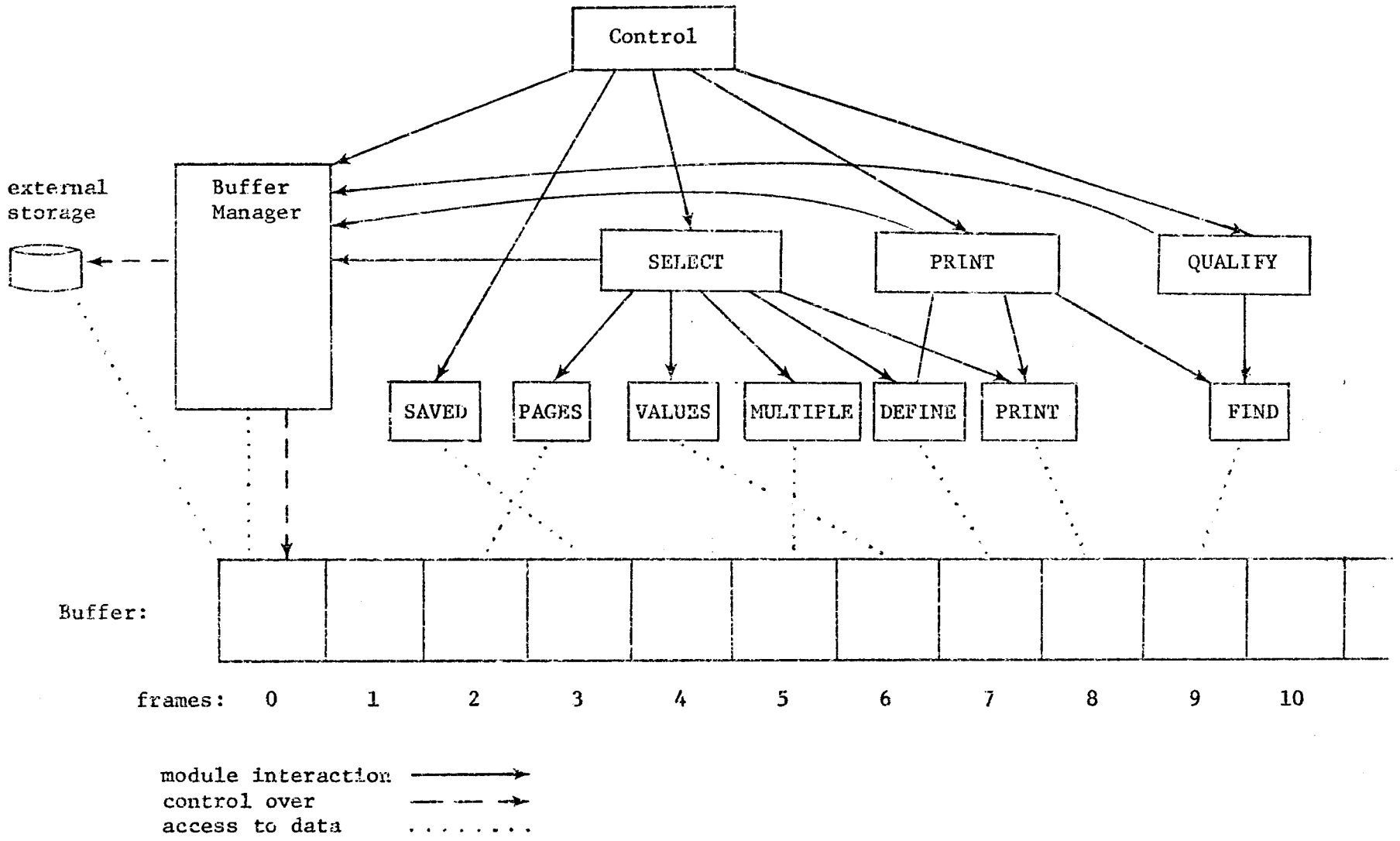


Figure 15

The Retrieval Function.

The Buffer Manager in Figure 15 shows how external storage devices are screened from direct interaction with other modules. Before requesting an operation by a given table supervisor, the command processor requests the Buffer Manager to retrieve the needed page of that table from external storage. This page is then placed in one of the available page frames in the buffer area. The location of this frame is then returned to the command processor so that it may make the needed page available to the table supervisor. This process is repeated each time a new table, or new page in a table, is needed by the command processor.

Always occupying buffer frame zero is the system table CONTROL which is needed by the Buffer Manager so that the actual location of a given page in external storage is known. This table is read in automatically prior to initiating command processing from actual page one of external storage. Since the buffer area has only a finite number of pages, it is important that the Buffer Manager keep track of available page frames. The page frame table is maintained internally to the Buffer Manager so that the content of each frame in the buffer is known. In this way, it is possible to search for available frames via the page frame table. In addition, the table may be searched for a requested page to determine whether it may already be in the buffer.

Since it is often necessary to use more pages than there are page frames, it is necessary to determine a procedure for managing the buffer once no more empty frames are available. When the buffer becomes full and a request is made for another page it is obvious that a page in one of the frames must be "lost" (that is, no longer available in the buffer). Ideally, the frame chosen contains the page least likely to be needed again so that as few pages have to be read in as possible. To facilitate

this choice, a priority is assigned to each page and stored in its corresponding entry in the page frame table. Then, the frame is chosen which contains the page with the lowest priority. Of course, frame zero which contains the CONTROL table should never be chosen, so it is removed from consideration; that is, it implicitly has the highest priority of any page. For example, the page frame table shown in Figure 16 indicates that the buffer is full (no empty frames) and that the next frame to be chosen will be frame number four since page five has the lowest priority number. Of course, the assignment of priorities is a critical factor in the efficiency of the system. Thus, it is important that the algorithm for assigning priorities be changed when a better technique is found. Typically, however, it is expected that the so-called optimum algorithm would consider factors such as the number of references to (requests for) a given page, the time since the last reference, and possibly even the probability that a page will be needed by looking ahead to future commands. The problems involved here are similar to those encountered by computer systems operating under demand paged allocation of memory. (8)

The Set Manipulation Function

The ability to apply set theory to the records of a data base is dependent upon an efficient method of representing sets and of performing set operations. These methods should be independent of the retrieval function and others so that a selection of any of the available techniques is possible. To satisfy the requirements of this system, the set manipulation function must provide for conversions to and from its internal representation and for the operations AND, OR, and NOT in Boolean expressions involving stored sets. The modules needed regardless of the repre-

frame number:	1	2	3	4	5	6	7	8	9	...
actual page number:	6	3	4	5	8	7	2	9	11	
priority:	8	10	10	3	7	5	8	12	7	

Figure 16

Page frame table for the buffer area.

sentation technique are shown in Figure 17. The management of these routines is similar in principal to that of the retrieval function. The Control module is responsible for keeping track of stored sets via the SAVED table which indexes these sets in external storage. Control also coordinates the FORM command in the creation of new sets from Boolean expressions.

The highest level of responsibility belongs to Control and the lowest belongs to the set conversion and operation modules. These lowest level modules operate on sets in the buffer area similar to the way system tables are accessed by their supervising programs. The transfer of sets to and from storage is performed by the Buffer Manager at the request of the FORM command processor.

The technique chosen to perform the set manipulation function is the BSSR. The BSSR is efficient for both large and small data bases, so that there probably would be little need to modify the modules described for future changes in data base size. In addition, the modules are general enough to be used with any data management system in which information may be represented by sets of positive identification numbers.

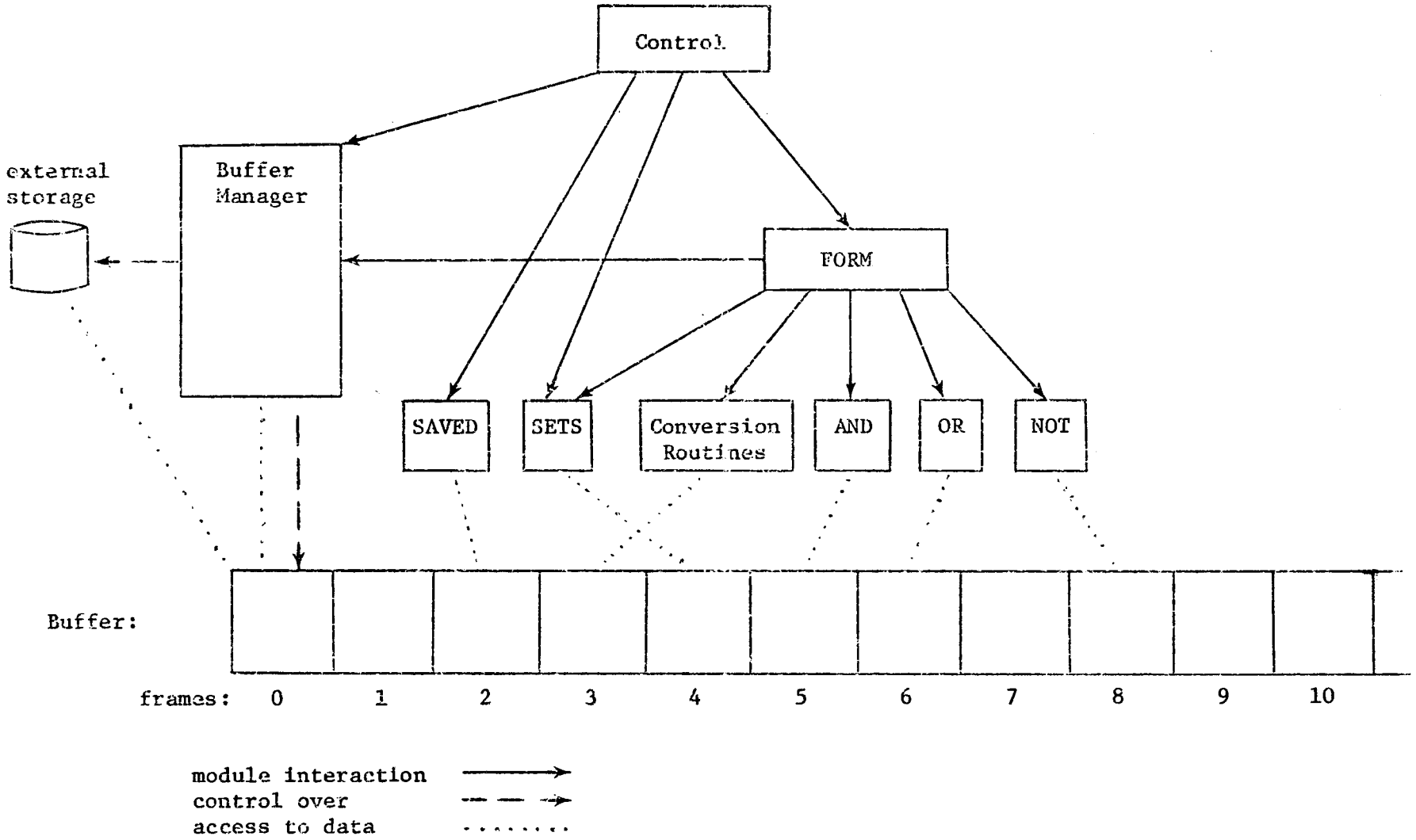


Figure 17

The Set Manipulation function.

CHAPTER VI

SUMMARY

A data base of factual information presents a situation which requires careful consideration of the types of requests that are to be made on the data management system. As previously discussed, it is expected that after initial loading, most accesses to this type of data base will be for retrieval rather than updating. Although no effective data management system may operate without some updates, it has been assumed that updating is proportionately small compared to other tasks in this system. However, the ability to include the updating task into the retrieval function does exist. Routines in the various table supervisory modules could permit entries to be added, deleted, or changed in order that information may be directly controlled without reloading the entire data base. Another assumption made in this discussion is that entries in the data base should be structured to permit access to logically related data. An important function to the user of such a system is that of manipulating retrieved items in order to extract all pertinent information. In addition, the system should be available to those users who are non-programmers.

To illustrate the way in which system functions may be designed for independent operation, a data management system has been constructed which meets the requirements listed above. The functions necessary for this system are performed by the following system components:

1. a language translator for communication between the user and the system,

2. a retrieval module that searches and maintains the data struc-

tures that define the logical relationships among the data,

3. a module that permits the manipulation of data to isolate the desired information,

4. a module that supervises the transmission of data between internal and external storage, and

5. a control mechanism that coordinates the operation of the system as a whole.

In Chapter Five, it is shown that each of these components is designed to perform its special function without being aware of the nature of the others. Consequently, modification of a system function is confined to one module. Furthermore, substitution of different components within a system or the sharing of components among systems is possible. In this way it is feasible to build any number of individualized systems, each with its own specialized control program, for a single computer installation or data base.

With a relatively stable data base, an effective management system should be geared for efficient retrieval. In addition, the relationships among data items should be clear and logical to the user conducting an investigation of the stored information. Data structures that permit both rapid access and logical relationships are therefore necessary to the retrieval module. The tree is a multi-leveled hierarchical data structure that groups information in an ancestor-descendent type of organization. In this manner, related data may be easily obtained for a given set of data items. The tree is implemented with two system tables designed to most efficiently conserve storage. For rapid access to information represented in the tree, an inverted list is maintained that permits the

retrieval of data items satisfying a given condition without an exhaustive search of the tree structure itself. In addition, an index to the inverted list decreases the required access time even more. Besides the two system tables needed to implement the inverted list and its index, two other tables are included; one serves as a page table to the lengthy inverted list, and the other helps to conserve storage by indicating multiple entries. The cost of the tree and the inverted list is paid in high storage overhead as well as in time-consuming initial loading and updating time. The stability of the data base must be able to justify this cost in order for this organization to be effective.

To extract information that satisfies specified conditions from the data base, the user must be able to combine various collections of data in a logical manner. By permitting the retrieval of sets of nodes in the tree (or simply records if the tree structure is not used), the system can be used to apply the properties of set theory. Thus, the operations of union, intersection, and complement may be performed on sets of nodes to obtain the desired information. For large data bases, the problem of representing the possibly very large sets is critical to the efficiency of the system. To solve this problem, the author has introduced the Bit Stream Set Representation (BSSR) technique which permits a substantial reduction in the storage requirements for the representation of sets. This method also allows the performance of the set operations on sets represented in this form. Furthermore, the BSSR technique may be applied to any data base in which the records have been assigned positive identification numbers (integers). The importance of this fact may be seen in view of the increasing trends towards large

centralized data banks.

In order for a collection of data to be useful, it must be possible for all authorized users to be able to easily access the information required. Therefore, an English-like command language permits even non-programmers to specify the tasks to be carried out by the system. These commands invoke the retrieval and set manipulation functions in creating, storing, and retrieving sets of information pertinent to the user's needs. However, it may be noted that the language is designed to be independent of the other system components. That is, it is possible for the system to allow an experienced programmer to program his way around the language translator to directly interact with the system components.

The data management system described in this paper is designed to offer a combination of features not currently found on any generalized system. It may be seen, however, that each feature may be useful in other systems designed towards another set of goals. Therefore, by designing modules that work independently from one another, it is possible to provide functions that may be interchangable among dissimilar systems. In this way, it is possible to solve problems not efficiently handled by many of the generalized systems by constructing more highly specialized systems. In particular, this system also may be noted for the technique of efficiently representing large sets of information that has been described. The usefulness of this method extends beyond the scope of the proposed system here; the implication for use with large data bases makes this method valuable for nearly any data management system.

BIBLIOGRAPHY

BIBLIOGRAPHY

1. A Survey of Generalized Data Base Management Systems, CODASYL Systems Committee Technical Report, The Association of Computing Machinery, New York, May 1969.

2. Bloom, B. H., "Some Techniques and Trade-offs Affecting Large Data Base Retrieval Times," Proceedings of the 24th National Conference, ACM, 1969, pp. 83-95.

3. Byrom, S. T., and Hardgrave, W. T., "Representation of Sets on Mass Storage Devices for Information Retrieval Systems," Proceedings of the First National Computer Conference and Exposition, AFIPS, 1973.

4. Chapin, N., "A Comparison of File Organization Techniques," Proceedings of the 24th National Conference, ACM, 1969, pp. 273-283.

5. Chapin, N., Computers - A Systems Approach, Van Nostrand Reinhold Co., New York, 1971.

6. Codd, E. F., "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

7. Dodd, G. G., "Elements of Data Management Systems," Computing Surveys, Vol. 1, No. 2, June 1969, pp. 117-133.

8. Donovan, J. J., Systems Programming, McGraw-Hill Inc., New York, 1972.

9. Hardgrave, W. T., "Theoretical Aspects of Boolean Operations on Tree Structures and Implications for Generalized Data Management," Ph.D. Dissertation, The University of Texas at Austin, Austin, Texas, August 1972.

10. Hsiao, D., and Harary, F., "A Formal System for Information Retrieval from Files," Communications of the ACM, Vol. 13, No. 2, February 1970, pp. 67-73.

11. Knuth, D. E., The Art of Computer Programming, Vol. 1 and 3, Addison-Weseley, Reading, Mass., 1968.

12. Lipschutz, S., Set Theory and Related Topics, Schaum Publishing Co., New York, 1964.

13. Parnas, D. L., "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM, Vol. 15, No. 12, December 1972, pp. 1053-1058.

14. Shubert, R. F., "Basic Concepts in Data Base Management Systems," Datamation, Vol. 18, No. 7, July 1972, pp. 42-47.

APPENDIX

EXAMPLE BOOLEAN OPERATIONS

The sets S_1 and S_2 and their complements will be used in the examples to follow where

$$S_1 = \{2, 5, 100000, 200000000\}$$

$$S_2 = \{1, 2, 100000\} .$$

Example One

S_1 AND S_2

$BSSR_1$:

$$\underbrace{(10...010...0}_{D_2}\ \underbrace{10...010...0}_{B_{1,0}}\ \underbrace{0...010...0}_{B_{1,428}}\ \underbrace{0010010...0}_{B_{0,0}}\ \underbrace{0...010...0}_{B_{0,48}}\ \underbrace{0...010...0)}_{B_{0,976562}}$$

$BSSR_2$:

$$\underbrace{(10...0}_{D_2}\ \underbrace{10...010...0}_{B_{1,0}}\ \underbrace{0110...0}_{B_{0,0}}\ \underbrace{0...010...0)}_{B_{0,48}}$$

Step 1. L = 2

Step 2. Find Boolean AND of level 2 directories:

D_2 of $BSSR_1$ = (10...010...0)

D_2 of BSSR2 = (10... 0)

D_2^* of BSSR* = (10... 0)

Step 3. Since $d^*_{2,428} = 0$ in D_2^* and $d_{2,428} = 1$ in D_2 of $BSSR_1$, omit block $B_{1,428}$ from $BSSR_1$. Next omit all blocks within the range of $B_{1,428}$ in $BSSR_1$; that is, block $B_{0,976562}$.

Step 4. L = 2 - 1 = 1

Step 2. Find Boolean AND of level 1 directories:

D_1 of $BSSR_1$ = (10...010...0)

D_1 of $BSSR_2$ = (10...010...0)

D_1^* of BSSR* = (10...010...0)

Step 3. (No occurrences)

Step 4. L = 1

Step 5. Neither $b_0 = 0$, so continue to Step 6.

Step 6. Find Boolean AND of level 0 blocks:

R of $BSSR_1$ = (0010010...0 0...010...0)

R of $BSSR_2$ = (0110000...0 0...010...0)

R of BSSR* = (0010000...0 0...010...0)

Since $b_2 = b_{100000} = 1$, the resulting BSSR* represents the set {2, 100000}.

Example Two

S_1 OR NOT S_2

The BSSR for S_1 is given in Example One, and the BSSR for NOT S_2 is

$BSSR_2$ = (10...0 10...010...0 1110...0 0...010...0) .

Step 1. L = 2

Step 2. Find Boolean OR of level 2 directories:

D_2 of $BSSR_1$ = (10...010...0)

D_2 of $BSSR_2$ = (10... 0)

D_2^* of BSSR* = (10...010...0)

Step 3. Since $d^*_{2,428} = 1$ and $d_{2,428}$ in $BSSR_2$, insert the block $B_{1,428} = \overline{0}$ into $BSSR_2$.

Step 4. L = 2 - 1 = 1

Step 2. Find Boolean OR of level 1 directories:

D_1 of $BSSR_1$ = (10...010...0 0...010...0)

D_1 of $BSSR_2$ = (10...010...0 0... 0)

$$D_1^* = (\underbrace{10\ldots010\ldots0}_{B_{1,0}}\ \underbrace{0\ldots010\ldots0}_{B_{1,428}})$$

Step 3. Since $d^*_{1,976562} = \overline{0}$ in $BSSR_2$, insert the block $B_{0,976562}$ which is equal to $\overline{0}$ into $BSSR_2$.

Step 4. L = 1

Step 5. Replace the level zero blocks of the BSSR with $b_0 = 0$ with their complement (reverse each binary value); then find the Boolean AND of the level zero blocks:

R of $BSSR_1$ = (1101101...1 1...101...1 1...101...1)

R of $BSSR_2$ = (1110000...0 0...010...0 0... 0)

R of BSSR* = (1100000...0 0... 0 0... 0)

Since $b_1 = 1$ and $b_0 = 1$, then the resulting BSSR* is the complement of the set {1} .

Example Three

NOT S_1

The BSSR for S_1 is shown in Example One.

Step 1. H = 2 and L = 0

Step 2. Find the Boolean NOT of R:

$$D_0 = R = (0010010\ldots0\ 0\ldots010\ldots0\ 0\ldots010\ldots0)$$

$$D_0^* = R^* = (\underbrace{0101101\ldots1}_{B_{0,0}}\ \underbrace{1\ldots101\ldots1}_{B_{0,48}}\ \underbrace{1\ldots101\ldots1}_{B_{0,976562}})$$

Step 3. Since $B_{0,0} \neq \overline{0}$, $B_{0,48} \neq \overline{0}$, and $B_{0,976562} \neq \overline{0}$, then $d_{1,0} = d_{1,48} = d_{1,976562} = 0$.

Step 4. Insert all omitted blocks (with blocks of "all zeros") into R.

Step 5. (Each block $B_{0,j} \neq \overline{0}$)

Step 6. $L = 0 + 1 = 1$

Step 2. Find Boolean NOT of D_1^*:

$$D_1^* = (\underbrace{1\ldots\ 1}_{B_{1,0}}\ \underbrace{1\ldots\ 1}_{B_{1,428}})$$

Step 3. Since $B_{1,0} \neq \overline{0}$ and $B_{1,428} \neq \overline{0}$, then $d_{2,0} = d_{2,428} = 0$.

Step 4. Insert all omitted blocks (with blocks of "all ones") into D_1.

Step 5. (Each block $B_{1,j} \neq \overline{0}$)

Step 6. $L = 1 + 1 = 2$, therefore terminate.

VITA

Stuart Todd Byrom was born in Shelbyville, Tennessee on June 24, 1949. He attended elementary school in Nashville, Tennessee and in June of 1967 graduated from Isaac Littion High School in that city. In September, 1967, he entered the University of Tennessee and received the Bachelor of Science degree in Industrial Engineering in August of 1971. As an undergraduate he became a member of Tau Beta Pi, Alpha Pi Mu, and Phi Kappa Phi honor societies. He began a teaching assistantship that September while studying for a Master's degree. He received that degree in Computer Science in August of 1973.