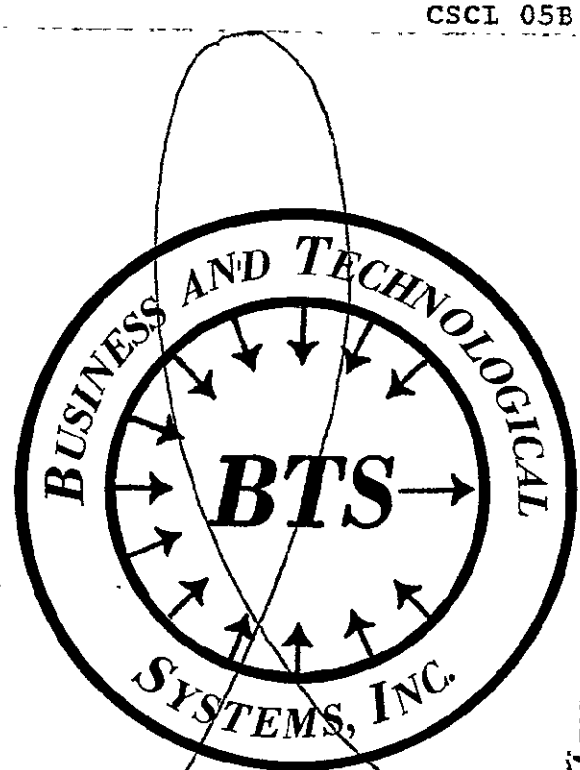
NASA CR- 159926

[redacted]

(NASA-CR-159926) A CONCEPTUAL DESIGN FOR AN INTEGRATED DATA BASE MANAGEMENT SYSTEM FOR REMOTE SENSING DATA Final Report (Business and Technological Systems, Inc.) 408 p HC A18/MF A01 CSCL 05B G3/82

N79-24892

Unclas 22958



BTS-FR-78-65

A CONCEPTUAL DESIGN
FOR AN
INTEGRATED
DATA BASE MANAGEMENT SYSTEM
FOR
REMOTE SENSING DATA

Paul A. Maresca
R. Michael Lefler

BUSINESS AND TECHNOLOGICAL SYSTEMS, INC.
Aerospace Building, Suite 440
10210 Greenbelt Road
Seabrook, Maryland 20801

Final Report
September 1978

Prepared under Contract No. NAS 5-24360
for
GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland 20771

FOREWORD

This report was prepared by Business and Technological Systems, Inc. under Contract No. NAS 5-24360 for the NASA/Goddard Space Flight Center. The report describes a conceptual design for a data base management system to support a wide variety of scientific applications and research activities. Because of the ever increasing volume of data from science and applications satellites, both launched and proposed for the 1980's, Goddard Space Flight Center is entering an era in its data analysis activities when it becomes necessary to locate, integrate and process various remotely sensed data in a timely fashion to achieve their maximum utilization and obtain their maximum benefit. It is imperative that state-of-the-art techniques in data management be applied to the problem of providing these data to the end user as quickly and as easily as possible. To that end, this study was initiated to design an integrated data base management system which addresses these problems and others.

ACKNOWLEDGEMENTS

The authors wish to acknowledge the contributions provided to the work reported herein by Dr. Johannes G. Moik, Technical Monitor, Mr. J. P. Gary, Ms. Karen Posey and Ms. Rita Jamros, all of NASA Goddard Space Flight Center. The discussion and ideas that were interchanged during weekly meetings between the authors and these members of the GSFC/Goddard technical staff have influenced all phases of this design effort. Additionally, the authors would like to thank Ms. Patricia McKeever, Ms. Francine Knox and Ms. Elizabeth Hammond for their able and patient assistance in typing and editing this report.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS (Continued)

LIST OF TABLES

SECTION 1 - INTRODUCTION

1.1 Study Requirements

This report presents the results of a study to produce a conceptual design of an Integrated Data Base Management System to support a variety of applications research activities. The design was developed so as to be independent of any specific computer or operating system. Initially, the system would be required to support applications investigations in weather and climate. Ultimately, it is anticipated that the technology developed during this study and presented herein will be used to support applications investigations in hydrology, agriculture and related earth resources disciplines. Prior to entering into the actual design phase, a sample of the requirements of potential users of this system were analyzed. These users included scientists performing studies in the above named areas as well as programmers developing applications software in those areas.

Several factors affecting the design of the system were brought to light during the evaluation of the user requirements. Foremost among these was the need for the system to maintain information about a large quantity of data stored mostly on magnetic tape in a variety of formats. This information must be maintained by the system in a form such that users can easily determine what data is available and locate and retrieve subsets of that data without regard for format or physical location. Because of the research environment in which this system would operate, users must have the ability, not only to locate a desired subset of data, but to be able to retrieve that subset and structure it into a data base to meet their needs. Once a user has located and retrieved the required data, the system must provide capabilities such that the data can be manipulated in numerous ways to obtain the desired results.

These data manipulation operations must provide users with the ability to access and update data in on-line data bases organized for random access as well as the ability to perform meaningful operations on subsets of the sequentially organized data in the off-line data base on magnetic tape. Thus, it became evident that significant flexibility in accessing, structuring, relating and processing data within the system was necessary to support the user requirements.

1.2 The Applicability of Existing Data Base Management Systems

Since there are currently several data base management systems available for a number of computers, the possibility that one of these systems might satisfy the requirements of the user community must be considered. The existing systems represent what might be referred to as the first generation of data base management systems. Some of these systems have been available for several years with relatively little change. Others have appeared in the past few years. However, all of these systems share the first generation characteristics of a centralized data base with centralized definition and control of that data base. While these characteristics are desirable under certain circumstances and can be realized with the system proposed herein, they would severely hinder users working in the environment described in the previous subsection. Thus, the system described in this report, which we feel is representative of the new generation of data base management systems, stresses a more flexible approach to data base management wherein the user has considerably more control over the organization and processing of his information.

As an example of the operational restrictions of the current generation of data base management systems, consider a typical scenario for developing a data base. Initially,

the data base itself must be designed. This is, very often, a lengthy process in which an individual with extensive knowledge of the data base management system, often referred to as the Data Base Administrator, and several of the potential users are involved. An attempt is made by this group to define all of the current requirements for system usage that might affect data base content and structure and to foresee future requirements so that they can be included in the data base when it is initially designed. The reason for this is that in most systems it is difficult to restructure or extend a data base once it has been constructed.

Once the data base has been designed, the Data Base Administrator must code a description of the data base, referred to in some systems as a schema. This description is coded in a language, usually unique to the data base management system being used, referred to as the Data Description Language. After coding the data base description, it is compiled by a program which is a part of the data base management system, referred to as the Data Description Language Compiler or Schema Compiler. Often, subschemas must be defined which describe various subsets of the entire data base for use by specific users or applications. These too must be compiled. Thus, in most of the current generation of data base management systems, users access the same central data base via different subschemas.

Very often, the addition of files, data fields or new relationships within an existing data base requires the intervention of the Data Base Administrator. Usually, the Data Base Administrator must modify the data base description or an existing description of a subset of the data base or create a new description of a subset of the data base; all of which will require some sort of recompilation. Additionally, the Data Base Administrator might be required to run a utility program to perform the necessary restructuring or to unload the data base and then reload it to make the required modifications. All of this is often a time-consuming and error-prone process.

It was felt that the procedures outlined in the scenario above were intolerable in the environment in which this system would operate and that some new concepts must be applied to the design of a system which would be responsive to the needs of the user community. It should also be noted that none of the existing data base management systems satisfactorily address the problem of supporting a large off-line data base consisting of sequential data files.

1.3 The Conceptual Basis for a New System

Because of the need for flexibility and ease of use, the relational data model was chosen over the network and hierarchical data models to represent data stored in on-line random access data bases. A data model is simply the way in which a user logically views data. A more thorough discussion of the relational model as well as the other data models is provided in Appendix A of this document. Briefly, the relational data model permits users to logically view a data base as if it contained one or more flat, two-dimensional tables. The rows of a table are analogous to records in a file while the columns are analogous to data fields in those records. Additionally, the user does not explicitly define relationships within a relational data base since the system maintains these relationships based on the contents of the tables. Thus, the relational data model provides considerable data independence between the way in which a user logically views data and the way in which that data is actually stored and manipulated by the system. Therefore, the relational data model can provide the basis for a much more user friendly interface to the system and can also be the basis for future research and development into a near-natural language interface to the system. For these reasons, it was felt that the relational data model would provide the flexibility necessary to support the way in which a scientific user community would use the system.

In addition to the use of the relational data model, the design employs the concept of dynamic data definition. That is, relational data bases and the tables that constitute them can be created and destroyed dynamically via interactive commands. Additionally, existing tables can be dynamically expanded by the addition of data fields and special indices can be created which facilitate access to data stored in tables.

While the relational data model provides a basis for the description of random access data bases stored on-line, it does not address the problem of managing a large number of sequential files on magnetic tape and, perhaps, direct access devices. To accomplish this, it was felt that a file management system was needed. Thus, a dual system concept was evolved with a relational data base management system as a "front end" to provide the necessary flexibility and ease of access to on-line data bases and a file management system as a "back end" to provide access to sequential files in the large non-relational data base. Additionally, techniques are defined whereby information can be transferred between the front and back end of the dual system. It should be noted that although the approach defined herein is somewhat unique, each concept is based upon work currently being done in the field of data base management or on techniques that have been applied successfully in data processing for many years.

1.4 Advantages of the Design

By employing the relational data model to describe on-line data, the user is freed from the necessity of defining relationships through which data can be accessed. In essence, the user need not know how the data is physically stored to access it. The concept of dynamic data definition permits a user to create on-line random access relational data bases

as necessary to support his requirements. Thus, the time-consuming processes of data base definition and, perhaps, restructuring are all but eliminated.

By employing a file management system to support access to sequential data files in the off-line, non-relational data base, the advantages of sequential processing can be realized when necessary.

Two concepts within the design provide the logical interface between the relational data base management system and the file management system. These are the concepts of a data file catalog and a data file directory. The Data File Catalog is an on-line random access table maintained by the relational data base management system. It provides a one-to-one correspondence between a unique file identifier assigned by the system and the physical location of a data file. Each time a new sequential data file is added to the off-line non-relational data base, a record is inserted in the Data File Catalog which contains the unique identifier assigned to the new data file and its physical location. Thus, at any time, given the unique identifier, the system can locate the corresponding sequential file in the off-line data base.

While the Data File Catalog provides the system, and the users, with knowledge of the physical location of a data file, it does not provide an indication of the contents of the file. This information is provided by the Data File Directory. The Data File Directory consists of one or more random access on-line tables maintained by the relational data base management system. Each record in a directory table contains the data file identifier of a single sequential file in the off-line non-relational data base. Additionally, the record will contain values for attributes which are

descriptive of the type of data contained in the data file. Thus, a record in a directory table describes the contents of a file but not its physical location. Rather than attempt to define a static directory structure which might be applicable to all types of data maintained in the off-line data base, a dynamic structure was chosen whereby additional directory tables can be added to the Data File Directory as a function of new data types entered into the off-line data base. By interactively querying the Data File Directory, a user can locate the data in the off-line non-relational data base which might be required to perform a particular study.

One other advantage of providing file management capabilities is that data from existing systems such as AOIPS and Smips/VICAR can be processed by users of this system. Naturally, this is important because of the investment already made in software systems development and data processing.

At this point, it may be helpful to provide some scenarios, similar to the one above, which illustrate the use of the proposed system. The first scenario illustrates the use of the relational data base management system while the second illustrates the interactive processing capabilities of the file management system.

As in the previous scenario, a data base must be designed before it is created. However, the design can be performed by the user who will create the data base since it will be tailored to the user's requirements. Since the relational data base management system supports the dynamic expansion of a data base by adding tables and the expansion of tables by adding data fields, the user need not try to foresee future requirements; thus, reducing the time required in the design phase. After the design of the data base has been completed, the user interactively defines his new data base and the tables contained

therein. At this point, data may be entered into the tables in the new data base. If, at some later time, new tables must be defined or existing tables must be expanded or removed, this can be accomplished very simply with interactive commands.

If the data to be placed in the newly defined data base is located in the off-line data base, the user could query the Data File Directory to locate the required data. This data could then be extracted from the off-line data base using the interactive commands and placed into the new relational data base. As indicated previously, this can be done by the user interactively with no intervention by the Data Base Administrator. If data are required that are contained in other on-line relational data bases, the user can extract that data from those data bases and transfer it to his own data base using a powerful set of interactive data manipulation commands.

An alternate scenario can be envisioned if the data to be processed by the user is contained in the off-line data base and is not in a form which can be easily manipulated interactively in a tabular form via the relational data base management system (e.g., image data). To process such data, the user could still locate the data required using the Data File Directory. However, it would not be placed on-line in tabular form to be accessed randomly but would be loaded on a direct access device in sequential form and, in the process, converted to one of several system standard formats. These formats are discussed briefly below and in more detail in Section 7 of this document. The data can then be processed via interactive commands in its sequential form. For example, functions which can be performed include the regridding of a gridded data file, the removal of a two-dimensional slice from a multi-dimensional gridded data file, the overlaying

of two or more gridded data files, the extraction of a subset of parameters from a gridded data file or the extraction of a rectangular window from an image or gridded data file. Additional functions can be added to those defined above since this facility is implemented via library subroutines. The result of processing a sequential data file with any of these functions is the creation of a new sequential data file for which an entry is made in the Data File Catalog.

The preceding scenarios illustrate a powerful interactive capability for processing on-line, random access data in tables as well as data in sequential files. However, the system description also includes an extensive Application Program Command Language which provides facilities for manipulating data in tables as well as data in sequential files by application programs. Extended file manipulation commands for sequential files are included in the Application Program Command Language which permit the searching of data files, the standard reading and writing of data files as well as a re-read capability and a re-write capability. Thus, the Application Program Command Language extends the capability of an application program to process sequential files.

1.5 Data Formats

As part of the study, the use of a standard format or formats for sequential data files was investigated. Since existing data to be included in the off-line data base were already in such diverse formats and since new data in unknown formats must be supported in the future, it was determined that it would not be feasible to define a single format to encompass all data. Additionally, it was felt that it would not be practical to require that all data entered into the off-line non-relational data base be reformatted prior to inclusion. However, it was

felt that the use of standard formats for internal processing of sequential data by the system would simplify that processing. Thus, several types of standard formats have been defined; one for each type of data managed by the system (e.g., gridded, image, etc.). New standard formats can be defined as new data types are introduced into the system. Methods have been defined within this document by which sequential data files in their original format can be converted into the proper system standard format.

1.6 Contents of the Design Document

The remainder of this document contains the conceptual description of the system introduced in this section. The description is detailed and somewhat technical in nature. It was intended to provide a working basis for the development of a data base management system. Sections 2 and 3 provide an overview of the system and its capabilities. Section 2 provides a system overview from the internals standpoint, while Section 3 discusses the use of the system. Section 4 describes, in some detail, the proposed Interactive Command Language. Both the relational calculus based language and a Query-By-Example type language are discussed. It is intended that whatever interactive command language would be implemented for the system, it would be user friendly in that it would carry on a dialogue with the user to assist him in entering commands. Section 5 contains a description of the Application Program Command Language. This includes a proposed calling sequence for each command and a brief description of each of the arguments. Section 6 describes an approach to storing tabular data maintained by the relational data base management system. Section 7 discusses the handling of data files and the use of system standard formats. While this document does not attempt to define in detail all system standard

formats, it does include examples of some possible standard formats. Section 8 discusses the system internals which include the various control structures required to support the internal architecture of the system. These control structures consist of control blocks, control block extensions, dictionaries, lists and queues. Section 9 describes the actual system architecture including the various modules needed to implement a system as conceived of in this document.

Two appendices are included to provide additional information, mostly of a theoretical nature, to the reader. Appendix A describes the concepts on which the relational data model is based. Appendix B covers additional topics which are associated with the design and development of such a system. These include data integrity, consistency and quality, as well as a discussion of backup and recovery techniques and provisions for supporting concurrent access to data within the system.

SECTION 2 - SYSTEM OVERVIEW

2.1 Conceptual Description

This section describes the architecture of the Integrated Data Base Management System. This system is designed to provide multi-user access to structured and unstructured data. Structured data is stored in tabular form while unstructured data is stored in the standard sequential form. Data stored in tabular form resides on direct access devices and can have various types of indices associated with it to facilitate retrieval. Data stored in sequential form are treated as standard sequential data files and can reside on any device which supports the sequential organization of information. The indices which are associated with tables are constructed as a function of the data contained in the tables and are referred to generically as "superstructures". Superstructures provide rapid access to data in tables and a logical ordering to records in tables.

2.1.1 The Dual System Concept

To support this dichotomy of data structure, a dual system concept has been employed. The dual system is comprised of a "front-end" relational data base management system which manages information in tabular form and a "back-end" data file processor which manages sequentially organized files. This design philosophy not only provides the capability of processing tabular and sequential data but forms the basis for the development of a distributed data base system. Naturally, the entire system can be implemented on a single computer. However, the back-end data file processor could be implemented on one or more physically separate computers from the one on which the front end relational data base management system is implemented. This would allow the user to locate and access data which are stored at installations that are remote from

Integrated Data Base Management System

Dual System Architecture

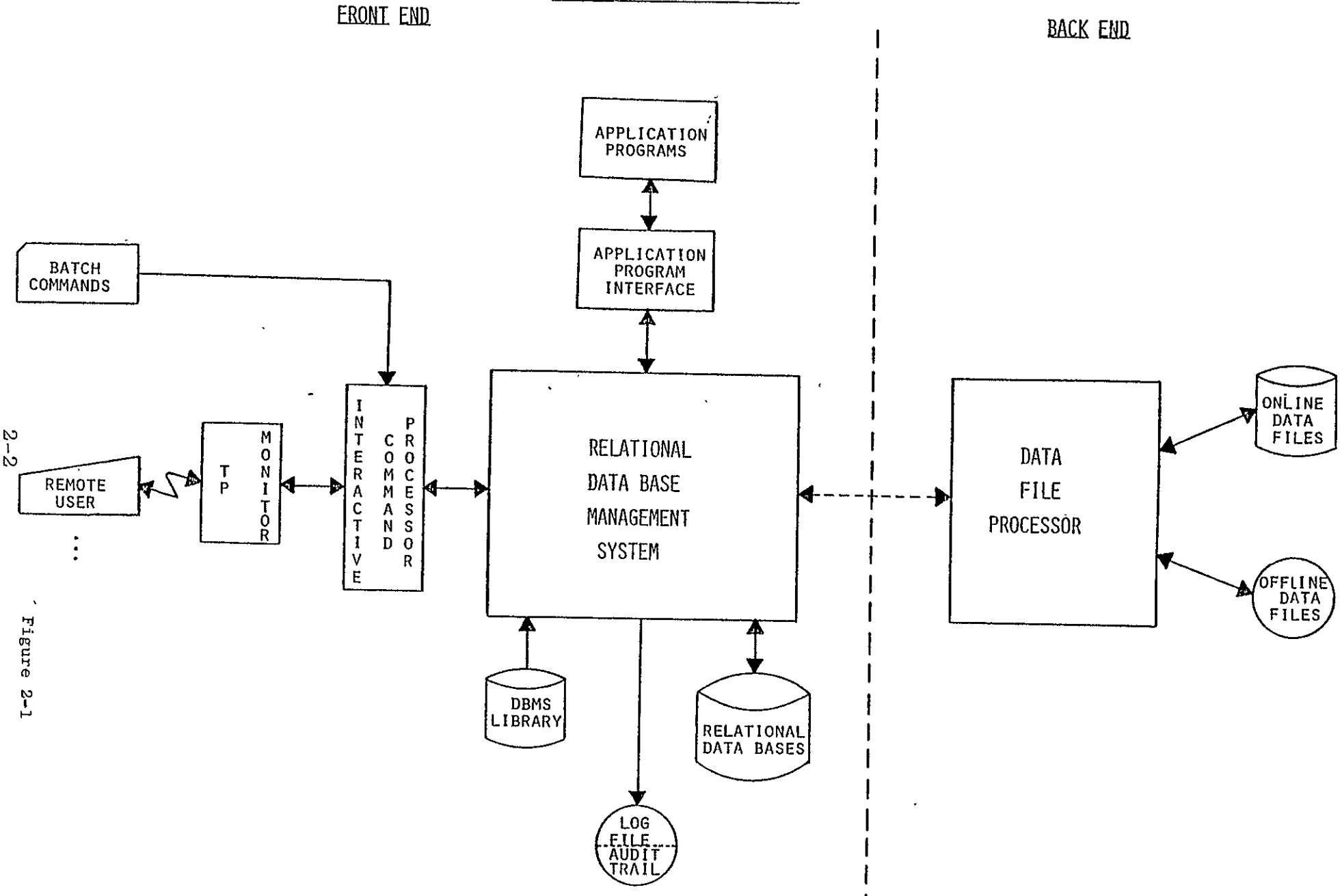


Figure 2-1

the central computing facility on which the front-end relational system is implemented. Figure 2-1 depicts graphically the dual system architecture of the Integrated Data Base Management System.

2.1.1.1 The Front End

The front-end will consist of a relational data base management system with interfaces which support concurrent access by multiple interactive users and multiple application programs. The relational system supports a tabular representation of data. Logically, data can be viewed as one or more tables with the data fields as columns and the records as rows. New rows may be added to the table and existing rows may be deleted. Likewise, new columns may be added to the table and existing columns may be updated. In the relational system, one or more tables can be organized into a data base. Each data base maintained by the relational system is independent; however, data may be transferred between data bases. The definition and removal of data bases and tables is a dynamic process and is under complete control of the users. Relationships among data in the tables of a data base are based entirely on data values. No predefinition of data base structure or access paths is necessary. Thus, data bases can be created and new tables added dynamically as a function of the users' requirements.

The data required to control the processing of the relational system is, itself, stored in tables. Thus, tables exist which contain information about the users of the system, the data bases currently maintained by the system, the tables contained in each of the data bases, the data fields within each of the data bases and the rights to perform certain operations on the tables. The system tables are contained in a system data base referred to as the Global Data Base. The Global Data Base also contains one or more tables which

constitute the Data File Directory. The Data File Directory is the vehicle by which users can locate sequential data files, maintained by the back-end data file processor, as a function of their data content. Any number of tables can be created and included as part of the Data File Directory. The format of a directory table is not predefined. Normally, a directory table will contain data fields which represent attributes of the data files which it cross references.

Another concept of importance is that of ownership. Ownership of data bases and tables is of primary importance in determining who is allowed to remove data bases or tables and who can grant specific access rights to tables. The Global Data Base is owned by the Data Base Administrator. Thus, the Data Base Administrator has complete access to all information in the Global Data Base. All other data bases maintained by the relational system are owned by users within the user community. Tables within data bases may be owned by the owner of the data base or other users. Only the owner of a table can grant access rights to that table. Thus, the owner of a table can grant operational rights to read, update, insert, and delete records within a table to the entire user community or various subsets thereof. Likewise, the owner of a table may revoke any of those granted rights.

2.1.1.2 The Back-End

The back-end will consist of software, referred to as the Data File Processor, which manages a large non-relational data base consisting of sequential data files. Each data file is assigned a unique data identifier by the system when it is created. All references to data files in the front-end relational system are via a data identifier.

When a new data file is entered into the Non-Relational

Data Base, an entry is inserted into a system table, referred to as the Data File Catalog, which contains, among other things, the data identifier and the physical location of the data file. The data file may be physically stored on magnetic tape or a direct access device which supports the sequential data organization.

There are several methods which might be employed to generate the data identifier. This document does not attempt to select the best approach, however, two techniques are mentioned briefly below. The first technique involves the generation of a random number containing some fixed number of decimal digits. The number would be converted into the internal alphanumeric code of the machine on which the system is implemented (e.g., ASCII, EBCDIC). The converted string of digits would become the data identifier for the new data file. This technique would probably require that an attempt be made to retrieve a record from the Data File Catalog to verify that the new data identifier is, indeed, unique. Another technique would be to use the last two digits of the current year and the three digit day number as the first five characters of the data identifier. Two or three other digits, generated using a counter, could be appended to obtain the entire data identifier. Thus, each day the counter would be reset to zero and would be incremented each time a new data file were added to the Non-Relational Data Base. While this technique does not require access to the Data File Catalog, it does require that the system maintain a counter which is not destroyed should the system terminate abnormally and it places an upper limit on the number of data files that could be added in a one day period. A variation of this technique would simply use an n-digit counter without including date information.

A data file may exist in two different formats: its

original data file format and a system standard format. It is anticipated that the Integrated Data Base Management System will support several system standard formats. For example, system standard formats might be defined for gridded data, image data and other data. New system standard formats can be defined as required by adding new-format conversion routines to the system library to convert data files from their original data file format into the new system standard format. Data file formats are discussed in a subsequent subsection, but no attempt is made in the document to define in detail a working set of system standard formats.

To facilitate the manipulation of data files, several interactive commands are provided which permit users to control the content and format of the Non-Relational Data Base and to transform data files into relational tables and relational tables into data files. Thus, a user could transform a data file into a table, manipulate the data in the table, combine the data with that of other tables and transform the resulting table into a new data file. Also, a user could load an off-line copy of a data file on magnetic tape onto a direct access device or unload an on-line copy to an off-line magnetic tape. Additionally, a user can invoke library procedures to perform operations, such as regridding, on a data file or display or plot the contents of a data file.

Additional facilities exist whereby data files can be accessed or created by application programs. Using application program commands, an application program can open and close data files, read all or a portion of a data file record, write records into new data files, search a data file record-by-record for a particular value, rewrite records in a newly created data file and process any header and history records which may be associated with a data file.

2.1.2 The System Environment

The structure of the Integrated Data Base Management System Software described herein can be divided into several independent tasks that can be performed in an asynchronous manner. Implementation of this software structure requires a multi-programming operating system that can support this form of subtasking. If the system is implemented on a computer whose operating system does not support these features, some modifications must be made to the internal software structure. The asynchronous tasks communicate with each other through information queues. Each task will be in a run state only while it has information to process. At all other times, it will be in a wait state. Thus, several commands in various states of completion can be within the system at any one time. Two other features which facilitate this software structure are the dynamic allocation of main memory and operating system facilities which support reentrant coding. Neither of these features is mandatory to the implementation of the system, as described in this document, since both can be implemented as part of the software system.

2.2 The Organization of Information

Information managed by the Integrated Data Base Management System can be viewed at two levels: the logical level and the physical level. At the logical level, information is viewed as data bases, tables and data files. At the physical level, information is viewed as physical pages containing tabular data, keys and pointers and sequential files containing data records.

2.2.1 The Logical View of Data

The logical view of data is content oriented. That is,

it is not concerned with storage structures, access methods or access paths but represents the way in which the users view data. It is important that the user have the ability to logically structure information in a natural manner. Certainly, it is desirable that the system impose as few restrictions on the logical view of data as possible. To that end, the Integrated Data Base Management System permits users to logically structure data into one or more tables of user defined format and to organize tables into one or more data bases to satisfy the user's requirements. Additionally, large volumes of data can be stored in data files.

2.2.1.1 Data Bases

A data base is a collection of tables. New data bases can be created at any time. Each data base has a name associated with it which must be unique among data base names known to the system. The creator of a data base becomes its owner. Within a data base, tables can be created and removed as necessary. Each data base has a Data Dictionary associated with it. The Data Dictionary contains the description of each data item in the data base. New data items can be defined as needed to support the creation of new tables in the data base. Each data base maintained by the system is independent from all other data bases. Every interactive user or application program can be logically attached for processing purposes to one and only one data base at any given time. Data can be transferred between data bases via a Workspace Table; however, a user attached to one data base cannot access the contents of another data base.

The system will support a classification scheme for data bases. When a data base is created, the system classifies it as a working data base. The classification of a data base can be changed at any time by the owner of the data base

or the Data Base Administrator. Besides the working data base class, the Data Base Administrator can define any data base classification scheme that is meaningful to the user community.

A special data base, referred to as the Global Data Base, is maintained by the system. The Global Data Base is owned by the Data Base Administrator. It contains the system tables, all directory tables and any other tables that the Data Base Administrator determines to be of use to the user community.

2.2.1.2 Tables

A table is a logical view of stored data. New tables can be created at any time. A new table belongs to the data base to which the user is attached when the table is created. Each table has a name associated with it which must be unique among table names in the data base to which it belongs. The creator of a table becomes its owner. Tables contain zero or more records which can be visualized as rows in the table. Each record contains one or more data fields which can be visualized as columns in the table.

Columns or combinations of columns can have superstructures defined on them. As currently defined, the system supports hierarchical indices, referred to as B-trees, and inverted indices. Superstructures can be defined when a table is created or after data values have been loaded into it. Superstructures may be dropped at any time. Rows or records within a table can be added or deleted. Columns or data fields can be updated or added. A column can not be physically deleted but can be set to a null value in all rows. The retrieval of data from a table is based entirely on data values in the table. Retrieval can be restricted to specific rows or columns. Data from several tables can be retrieved jointly into a special table referred to as the Workspace Table.

A Workspace Table is associated with each user connected to the system. The Workspace Table is used to contain the rows and columns that are retrieved from one or more tables as a result of a data base query operation. The Workspace Table associated with each user is not contained in any data base. Since it is not contained in any data base, the Workspace Table can be used to transport data from one data base to another. The contents of a Workspace Table can be accessed only by the user with whom it is associated.

When a table is created, access to it is limited to its owner and the Data Base Administrator. Either the owner or the Data Base Administrator can grant rights to perform the following operations on the table: read, update, insert and delete. The operational rights can be granted to individual users, to groups or to the entire user community. Likewise, only the Data Base Administrator or the owner of a table can revoke rights that have been granted on the table. Certain other functions, such as the addition of columns to a table or the removal of the table from a data base, are limited to the Data Base Administrator, the owner of the data base containing the table or the owner of the table.

2.2.1.3 Data Files

A data file is a collection of records which is treated as an entity by the system. Each data file has a unique identifier, referred to as the data identifier, assigned to it when it is entered into the system. The data identifier is used to reference the data file as an entity. Data files can be stored on any on-line device which supports the sequential organization of data. However, the primary function of data files is the storing of large quantities of data in an off-line mode on magnetic tape.

The collection of all data files known to the system is referred to as the Non-Relational Data Base. The system

maintains a catalog of all data files in the Non-Relational Data Base. Existing data files may be added to the Non-Relational Data Base by simply inserting a new entry into the Data File Catalog. Likewise, a data file in the Non-Relational Data Base can be removed by deleting its corresponding entry in the Data File Catalog. When a data file is entered into the Non-Relational Data Base, it is assigned a read-only status so that the data file can not be overwritten.

Data files in the Non-Relational Data Base can be accessed on a record by record basis by application programs. Application program facilities exist to read or search data files and to write new data files. Using the Interactive Command language, a data file can be copied from an off-line device to an on-line device and from an on-line device to an off-line device. Also, a data file can be transformed into a table and a table can be transformed into a data file.

2.2.2 The Physical View of Data

The physical view of data involves the actual storage mechanisms employed in the system to support the logical view. Because of the dual system concept on which the logical design is based, the physical storage facilities must support both tabular data stored in relational data bases and sequential data files stored in the Non-Relational Data Base. Thus, the information space managed by the Integrated Data Base Management System is partitioned into an area of on-line storage where tabular data are stored and an area of storage consisting, for the most part, of magnetic tapes on which sequential data files are stored.

2.2.2.1 The Storage of Tables

The area in which tabular data are stored must be on-line and can span multiple packs and multiple direct access devices.

The mapping of the tabular data storage area to the physical storage media occurs at system generation time. At any other time, a utility program can be used to extend this area.

The area in which tables are stored is subdivided into pages. A page is the basic unit of storage for tabular data and consists of a fixed size block of data which is transferred between peripheral storage and main memory by a single I/O operation. The size of a page is defined at system generation time and cannot be changed. A page may contain data records from a table or superstructure records associated with a table or the page may be part of the free pool of unused pages. Some portion of the tabular storage area will contain "before" images of pages that have been modified by commands in progress. These "before" images facilitate dynamic restoration of data bases when a command that was performing an update operation is aborted by the user or terminated prematurely due to an I/O error. A more detailed discussion of "before" images and dynamic recovery techniques is contained in the subsection entitled Backup and Recovery in this section.

Some area of main memory is allocated for page buffers. The size of a single page buffer is the same size as a page of data in the tabular data storage area. A default value is specified for the number of page buffers at system generation time. However, a different value can be specified each time the system is started. The transfer of pages between peripheral storage and the page buffers in main memory is controlled by the Integrated Data Base Management System on an as needed basis. An algorithm for buffer usage control is defined in Section 6.

The actual content and format of a page in the tabular data storage area depends upon the table to which the page has been assigned and the function of the page. Each page assigned to a table will contain either data records, records

from a hierarchical B-tree index or records from an inverted index. Each page will contain a prologue which specifies the characteristics of the information stored on that page. Each data record on a data page will, itself, have a prologue containing one bit for each data field in the record. A bit will have a particular setting to indicate that the corresponding data field contains a data value and the opposite setting to indicate that the data field contains a null value. A data field will contain a null value if no value was specified for it when the data record was inserted into the table. No pointers to other records are stored within pages containing data records from a table. Therefore, superstructures can be added to or dropped from a table without affecting the data pages of a table.

2.2.2.1.1 Sequential Tables

A table which has no superstructures associated with it is referred to as a sequential table. Data records in a sequential table have the same physical storage structure as those in tables for which superstructures exist; however, there are some differences in their processing. Since no superstructures exist to facilitate access to a sequential table, the retrieval, update and deletion of records requires the sequential searching of the data records and, in some cases, requires the accessing of every record in the table. Also, the holes in data pages caused by the deletion of data records are not reused. All new records inserted in a sequential table are stored at the logical end of the table.

At any time, a user can create hierarchical B-tree or inverted indices on a sequential table. When this occurs, the table ceases to be processed as a sequential table. Whenever possible, the superstructures are used to facilitate access to data records in the table and any existing holes in data pages become available for the insertion of new data records. If

all superstructures are dropped from a table, the table becomes a sequential table and is processed as such. Any existing holes in the data pages of the table become unavailable for the insertion of new records.

Sequential tables are useful in cases where the contents of a table are static, all or almost all data records are retrieved whenever the table is read and the ordering of records is immaterial or the data records must be retrieved in the order in which they were inserted. Also, if the number of data records in a sequential table is such that they can all be stored on one or two data pages, it may be more efficient to treat the table as a sequential table rather than create superstructures for it.

2.2.2.1.2 Superstructures on Tables

The term superstructure is used generically to refer to any type of indexing scheme for tabular data which is supported by the Integrated Data Base Management System. Superstructures are used to reduce the time required to access data records in a table or to provide a logical ordering of data records in a table. The system, as described in this document, supports two types of indexing for tabular data: the hierarchical B-tree index and the inverted index.

Superstructures can be created on single data fields or multiple data fields in a table. Any data field in a table can have either a B-tree or inverted index created on it, but not both. Both B-tree indices and inverted indices can be created on multiple data fields. A data field or combination of data fields on which a superstructure has been created is referred to as a key field.

Superstructures created for tables are stored on separate pages from the data records in the table. Superstructures

can be created for tables and dropped from tables dynamically under user control. The creation and dropping of superstructures does not affect the data pages of a table. Whenever a table for which one or more superstructures have been created has a data record inserted, deleted or updated, all of its superstructures are modified to reflect the new contents of the table.

2.2.2.1.2.1 B-Tree Indices

A B-tree index can exist for any data field or combination of data fields in a table. A single data field for which a B-tree index exists can not have an inverted index created for it. However, a data field which forms part of a combination B-tree key field can, itself, have either a B-tree or inverted index created for it. Thus, by specifying a single data field as a combination B-tree key field, the data field can have, in effect, both a B-tree and an inverted index created for it.

When a B-tree index is defined for a data field or combination of data fields, a uniqueness condition can be specified indicating that no duplicate key values are permitted. If the insertion or modification of a data record in a table would cause a duplicate key value to be added to a B-tree index for which the uniqueness condition has been specified, the operation will be aborted. If the uniqueness condition is not specified for a B-tree index, duplicate key values will be permitted.

Each B-tree index consists of one or more index pages organized in an hierarchical structure sometimes referred to as a tree structure. Each index page contains one or more key field values and their associated pointers to lower level pages in the tree. Pages in the lowest level of the index contain key values in ascending order and associated pointers to the data records containing the key values.

A B-tree index should be created for data fields that will contain unique or nearly unique values since there is a one-to-one correspondence between a key value/pointer pair in the lowest level of the index and a data record in the table. The advantage of a B-tree index is that one record or a group of records can be located in a large table with very few I/O operations. Also, a B-tree index, as described in this document, causes the data records in a table to be logically ordered on a data field or combination of data fields that have been specified as a B-tree key field. Thus, data records can be retrieved from a table in the ascending sequence of data values in a B-tree key field.

2.2.2.1.2.2 Inverted Indices

An inverted index can exist for any data field or combination of data fields in a table. A data field for which an inverted index exists can not have a B-tree index created for it; however, it can be part of a combination B-tree key field

Each inverted index consists of two parts: a domain directory and a set of pointer lists. The domain directory contains one entry for each distinct value found in the data field on which the inverted index was created. Each entry in the domain directory consists of a data value and a pointer to the corresponding pointer list. There is one pointer list associated with each domain directory entry in an inverted index. Each pointer list contains one or more pointers to data records in the table which contain the data value in the associated domain directory entry.

An inverted index should be created for data fields where the same value will be repeated in several records so that there is a one-to-many correspondence between a data value in a domain directory entry and the pointers in the associated pointer list. The advantage of an inverted index is that a set of data records that contain a specific value can be

located rapidly without accessing the data records themselves. Also, boolean operations can be performed easily on data fields for which an inverted index exists. After locating the domain directory entries containing the data values of data fields specified in a boolean expression, the boolean operations are performed on the associated pointer lists yielding a resulting pointer list containing pointers to all data records satisfying the boolean expression.

2.2.2.2 The Storage of Data Files

A data file is a collection of records organized for sequential access and terminated by an end-of-file mark. A data file can contain either data in its internal binary representation or data which has been converted to some external code such as ASCII or EBCDIC. Up to three copies of a data file can exist simultaneously and be referenced by the same data identifier. These include an off-line copy on magnetic tape in its original data file format, an on-line copy on a direct access device in one of the system standard formats and an off-line copy on magnetic tape in the same system standard format. Any one or a combination of these forms of a data file can exist and their physical location be maintained by the system in the Data File Catalog.

While it will not be a requirement of the system that data files be put into a system standard format prior to being entered into the Non-Relational Data Base, the use of system standard formats will be encouraged so as to facilitate the sharing of data among the Integrated Data Base Management System and other information processing systems. All data files created by application programs using the facilities of the Integrated Data Base Management System will be in one of the system standard formats. Also, the loading of a data file to a direct access device by the system will cause it to be converted to one of the system standard formats unless it is

already in such a format. It is anticipated that all data file processing procedures invoked through the Integrated Data Base Management System will read and write data files in a system standard format. The general structure of a data file in a system standard format includes a header record which has a fixed format and describes the format of the data records in the data file, zero or more processing history records in a free format and one or more data records whose format is a function of the type of data contained therein (e.g., gridded, image, text, etc.).

An indication of the format of each copy of a data file is stored along with its physical location in the Data File Catalog. During the loading of a data file from magnetic tape to a direct access device, the format indicator is used to locate a module, residing in the Integrated Data Base Management System library, that can be loaded and used to access the data file. If the off-line copy of a data file to be loaded is already in one of the system standard formats, the corresponding input module will perform no format conversion but may perform windowing on the data file causing a data file with a different data identifier to be created. If the off-line copy is in its original data file format, it can be loaded on-line only if an input module corresponding to the data file format has been placed in the library. If an input module exists for the data file format, any data files in that format that are loaded on-line will be converted, by the input module, to a predefined system standard format. Thus, existing data files can be entered into the Non-Relational Data Base without first being put into a system standard format.

Data files can reside on magnetic tapes or direct access devices. One magnetic tape can contain more than one data file and a data file can span more than one magnetic tape. Data files created by application programs or internal

procedures will be stored, initially, on a direct access device in one of the system standard formats. If required, the data file can be unloaded to magnetic tape in the same system standard format by an application program or interactively. Data files residing on a direct access device may be stored in a non-contiguous manner if this facility is supported by the operating system of the computer on which the Integrated Data Base Management System is implemented. That is, the data file may be physically fragmented on the direct access device but will be treated logically by the system as an entity. If this facility is not supported by the operating system, an alternative approach would be to write new data files created by application programs or internal procedures directly to magnetic tape since it would be difficult for the system to anticipate the amount of direct access space required to store a new data file so that contiguous space could be preallocated. Another option is to have the user specify the amount of direct access space required for storage of a new data file. While this might be feasible for certain types of data, such as image data, the requirement of preallocating contiguous direct access space for a data file would lead, in general, to inefficient use of the space available for the storage of data files and should be avoided, if possible.

2.3 The Global Data Base

The Global Data Base is a relational data base which is automatically defined at system generation time by the System Generation Program. The Data Base Administrator is the owner of the Global Data Base. The Global Data Base contains the system tables which control much of the processing within the system. The tables which constitute the system Data File Directory also reside in the Global Data Base. At any time, the Data Base Administrator can create new tables in the Global Data Base which are neither system tables

nor directory tables. Presumably, these tables would contain information of general interest to the user community. The Global Data Base is structured in the same way as any other data base within the system. Any operations that can be performed on user defined data bases can be performed on the Global Data Base. However, access rights which would allow the tables in the Global Data Base to be modified will be restricted or controlled by special commands. Also, access rights which would allow retrieval of certain information from the system tables may be restricted or controlled by special commands. Thus, the purpose of the Global Data Base is to contain system information in a form that is consistent with that of other information, to provide a repository for information that is of interest to the entire user community and to permit the Data Base Administrator to control access to this information.

2.3.1 System Tables

All system tables are contained within the Global Data Base. They are automatically defined at system generation time by the System Generation Program. The system tables are used to store system control blocks and other system related information. Special commands are available to the user community to define and remove data bases, data fields, and tables and to grant and revoke access rights to tables. Additional privilaged commands are available to the Data Base Administrator to define and remove users from the system, define and remove user groups and to catalog and uncatalog data files. These commands ultimately cause one or more system tables to be modified. Only the Data Base Administrator is permitted to use the full complement of data manipulation commands on the system tables. System tables are stored and accessed in the same way as all other tabular data. Superstructures are defined for the system tables by the System

Generation Program to facilitate the storage and retrieval of data records by the Integrated Data Base Management System software. As the owner of the system tables, the Data Base Administrator can create additional superstructures on them to support any additional processing requirements. However, the Data Base Administrator can not drop any superstructure defined on a system table by the System Generation Program. The following subsections describe briefly the contents and structure of each of the system tables.

2.3.1.1 SYSUSER Table

The SYSUSER table contains one record for each valid user of the system, including the Data Base Administrator. Each record in the SYSUSER table contains a User Control Block. The User Control Block contains user descriptive information and is described in Section 8.

When a new user is defined to the Integrated Data Base Management System by the Data Base Administrator, a User Control Block is created for the user and is inserted, as a record, into the SYSUSER table. If an attempt is made to add a new user to the system whose user-id will duplicate that of an existing user, the new user will be rejected because a unique B-tree index exists on the user-id field of the SYSUSER table. The B-tree index on the user-id field also provides a logical ordering by user-id of the records in the SYSUSER table.

When a user connects to the Integrated Data Base Management System, the record containing the user's User Control Block is retrieved from the SYSUSER table. The record is located using the unique B-tree index created on the user-id field. When a user is removed from the Integrated Data Base Management System by the Data Base Administrator, the record containing the User Control Block for the user is

located via the B-tree index on the user-id field and the record is deleted from the SYSUSER table.

2.3.1.2 SYSGROUP Table

The Data Base Administrator can define a group for the purpose of granting common access rights to all users belonging to the group. The concept of group access rights is discussed in Section 3. The SYSGROUP table contains one record for each group defined within the Integrated Data Base Management System and one record for each user in each group. Thus, the SYSGROUP table contains two types of records: one record which defines the existence of a group and zero or more records which specify the users who belong to that group. The collection of all records that specify to which groups a user belongs constitute the Group Extension for that user. The Group Extension is described in Section 8.

When a new group is defined to the Integrated Data Base Management System by the Data Base Administrator, a record containing the name of the group and a blank user-id field is inserted into the SYSGROUP table. This record indicates the existence of the group and is used for verification purposes whenever a user is included in the group. When a user is included in a group by the Data Base Administrator, a Group Extension entry is created for the user and is inserted, as a record, into the SYSGROUP table. If an attempt is made to add a new group to the system whose group name will duplicate that of an existing group or an attempt is made to include a user in a group to which he already belongs, the request will be rejected because a unique B-tree index exists on a combination of the user-id and group-name fields in the SYSGROUP table. The B-tree index on the combination of user-id and group-name also provides both a logical grouping by user-id and a logical ordering by user-id and group-name to the records in the SYSGROUP table.

When a user connects to the Integrated Data Base Management System, any Group Extension records associated with the user are retrieved from the SYSGROUP table to form the user's Group Extension to the User Control Block. The records are located using an inverted index created on the user-id field. Each entry in the Group Extension will point to an Authorization Extension in main storage which specifies the access rights granted to the group represented by the entry. For the purpose of determining the user's right to access tables, these group access rights will be treated as if they had been granted to the individual user.

When the Data Base Administrator removes a user from a group, the record corresponding to the specified user and group is deleted from the SYSGROUP table. The record to be deleted is located via the unique B-tree index on the combination of user-id and group-name fields. When the Data Base Administrator removes a group from the system, all records containing the specified group-name are deleted. This includes the record containing a blank user-id field which defines the existence of the group and any other records which contain the user-id of users belonging to the group. The records to be deleted are located via an inverted index on the group name field.

2.3.1.3 SYSDB Table

The SYSDB table contains one record for each data base defined within the Integrated Data Base Management System, including the Global Data Base. Each record in the SYSDB table contains a Data Base Control Block. The Data Base Control Block contains information pertaining to the data base and is described in Section 8.

When a new data base is defined to the Integrated Data Base Management System by a user, a Data Base Control Block

is created for the data base and is inserted, as a record, into the SYSDB table. If an attempt is made to define a new data base with a data base name which duplicates that of an existing data base, the new data base will be rejected because a unique B-tree index exists on the data-base-name field of the SYSDB table. The B-tree index on the data-base-name field also provides a logical ordering by data base name of the records in the SYSDB table.

If a user connected to the Integrated Data Base Management System indicates an intent to process information in a data base whose Data Base Control Block is not resident in main storage, the record containing the Data Base Control Block for the data base is retrieved from the SYSDB table. The record is located using the unique B-tree index created on the data-base-name field. Additional records associated with the data base may be loaded from other system tables at that time. When a data base is removed from the Integrated Data Base Management System by a user, the record containing the Data Base Control Block for the data base is located via the B-tree index on the data-base-name field and the record is deleted from the SYSDB table.

Additional superstructures are created on the SYSDB table by the System Generation Program to facilitate the retrieval of information about data bases using the DESCRIBE command which is available to the user community. Inverted indices are created on the data base classification field, the date created field and the field containing the user-id of the owner of the data base. Thus, the DESCRIBE command can retrieve information about data bases maintained by the Integrated Data Base Management System as a function of the data base classification, date created or owner of the data base.

2.3.1.4 SYSDD Table

The SYSDD table contains one record for each data field in each data base defined within the Integrated Data Base Management System, including the Global Data Base. Each record in the SYSDD table contains a Data Dictionary entry which describes the attributes of the data field which it defines. The collection of all records that define data fields in a data base constitutes the Data Dictionary for that data base. The Data Dictionary is described in Section 8.

When a new data field is defined by a user for an existing data base, a Data Dictionary entry is created for the data field and is inserted, as a record, into the SYSDD table. If an attempt is made to define a new data field with a field name which will duplicate that of an existing data field in the same data base, the new data field will be rejected because a unique B-tree index exists on a combination of the data-base-name and field-name fields in the SYSDD table. The B-tree index on the combination of data-base-name and field-name also provides both a logical grouping by data base and a logical ordering by data base name and field name to the records in the SYSDD table.

When a user indicates an intent to process information in a data base whose Data Base Control Block is not resident in main storage, all Data Dictionary entry records associated with the data base are retrieved from the SYSDD table to form the Data Dictionary for the data base. The records are located using an inverted index created on the data-base-name field. Additional records are loaded from other system tables at that time. When a data field is removed from a data base by a user, the record containing the corresponding Data Dictionary entry is located via the B-tree index on the combination of data-base-name and field-name and the record is deleted from the SYSDD table.

2.3.1.5 SYSREL Table

The SYSREL table contains one record for each table in each data base defined within the Integrated Data Base Management System including the system tables in the Global Data Base. Each record in the SYSREL table contains a Relational Control Block. The Relation Control Block contains information pertaining to the table and is described in Section 8.

When a new table is defined by a user for an existing data base, a Relation Control Block is created for the table and is inserted, as a record, into the SYSREL table. If an attempt is made to define a new table with a table name that will duplicate that of an existing table in the same data base, the new table will be rejected because a unique B-tree index exists on a combination of the data-base-name and table-name fields in the SYSREL table. The B-tree index on the combination of data-base-name and table-name fields also provides both a logical grouping by data base and a logical ordering by data-base-name and table-name of the records in the SYSREL table.

When a user indicates an intent to process information in a data base whose Data Base Control Block is not resident in main storage, all records containing Relation Control Blocks for tables in the data base are read from the SYSREL table. The records are located using the unique B-tree index created on the data-base-name and table-name fields. When a table is removed from a data base by a user, the record containing the Relation Control Block for the table is located via the B-tree index on the combination of data-base-name and table-name fields and the record is deleted from the SYSREL table.

2.3.1.6 SYSDOM Table

The SYSDOM table contains one record for each data field in each table in each data base defined within the Integrated Data Base Management System, including system tables in the Global Data Base. Each record in the SYSDOM table contains a Domain Extension entry. Whereas, a Data Dictionary entry describes the general attributes of a data field, a Domain Extension entry contains information pertaining to a data field as it is used in a particular table. The collection of all records in the SYSDOM table that describe data fields in a particular table constitutes the Domain Extension for that table. The Domain Extension for a table is described in Section 8.

When a new table is defined by a user, the data fields which make up the table are specified. A Domain Extension entry is created for each of the data fields in the table and is inserted, as a record, into the SYSDOM table. A unique B-tree index exists on a combination of the data-base-name, table-name and field-name fields in the SYSDOM table. The B-tree index provides both a logical grouping by data base and table and a logical ordering by data-base-name, table-name and field-name to the records in the SYSDOM table.

When a user indicates an intent to process information in a data base whose Data Base Control Block is not resident in main storage, all Domain Extension entry records associated with tables in the data base are retrieved from the SYSDOM table to form the Domain Extensions for each of the tables in the data base. The records are located using the unique B-tree index created on the combination of data-base-name, table-name and field-name fields. Additional records are loaded from other system tables at that time. When a table is removed from a data base, the Domain Extension entry records associated with the table are removed from the SYSDOM table.

The records containing the Domain Extension entries are located via the B-tree index on the combination of data-base-name, table-name and field-name fields and the records are deleted from the SYSDOM table.

2.3.1.7 SYSAUTH Table

The SYSAUTH table contains one record for each user or group who has been explicitly authorized by the owner of a table to perform one or more data manipulation operations on the table. Records in the SYSAUTH table are used to control access to tables maintained by the Integrated Data Base Management System. Each record in the SYSAUTH table contains an Authorization Extension entry that indicates which of the operational rights (READ, INSERT, UPDATE, DELETE) have been explicitly granted to the user or group on the table identified in the record. The collection of all records that define explicit operational rights granted to an individual user or group constitutes the Authorization Extension for that user or group. The Authorization Extension is described in Section 8.

When the owner of a table grants one or more operational rights to an individual user or group, the Authorization Extension associated with that user or group is checked to determine if the user or group has been granted rights previously on the same table. If so, the existing Authorization Extension entry is modified and the corresponding authorization record in the SYSAUTH table is updated to reflect the new authorizations. If no authorizations exist for the specified user or group on the table, an Authorization Extension entry is created and is inserted, as a record, into the SYSAUTH table. A unique B-tree index exists on a combination of the user-id of the user or group-name of the group, the data-base-name of the data base containing the table and the table-

name fields in the SYSAUTH table. The B-tree index provides both a logical grouping by data base and table and a logical ordering by user-id or group-name, data-base-name and table-name of the records in the SYSAUTH table.

When a user connects to the Integrated Data Base Management System, any authorization records associated with the user are retrieved from the SYSAUTH table to form the Authorization Extension to the User Control Block. The records are located using an inverted index created on the user-id field. Additional authorization records might be retrieved from the SYSAUTH table to form Authorization Extensions for groups to which the user belongs; if such Authorization Extensions are not already resident in main storage. When the owner of a table revokes one or more operational rights from a user or group, the authorization record corresponding to the user or group and the table is retrieved and checked to determine if the user or group will retain any operational rights on the table. If so, the authorization record is updated to reflect the reduced authorizations. If no authorizations remain for the user or group on the table, the corresponding authorization record is deleted from the SYSAUTH table.

Additional superstructures are created on the SYSAUTH table by the System Generation Program to facilitate the deletion of authorization records. Inverted indices are created on the user-id/group-name field, the data-base-name field and the table-name field. Thus, all authorization records associated with an individual user or group can be located and deleted if the user or group is removed from the system by the Data Base Administrator. If a data base is removed, all authorization records associated with tables in the data base can be located and deleted. Likewise, if a table is removed from a data base, all authorization records for the table can be located and deleted.

2.3.1.8 SYSCATL Table

The SYSCATL table contains one record for each data file maintained by the Integrated Data Base Management System in the Non-Relational Data Base. The physical location and format of up to three copies of each data file can be contained in a single record. Each copy is referenced using the same data identifier and, while they may not be in the same format, their data content will be exactly the same. The three copies which can exist for a data file include: an off-line copy on magnetic tape in the original data file format, an on-line copy on a direct access device in one of the system standard formats and an off-line copy on magnetic tape in the same system standard format. Each record in the SYSCATL table will contain a unique data identifier and the physical location and format of each existing copy of the data file. The form in which the physical location of each copy is specified may depend upon the operating system requirements for sequential file handling. Additionally, each record in the SYSCATL table will contain the date on which each copy was created or entered into the Non-Relational Data Base, the date on which each copy was last accessed, the user-id of the user who created the on-line copy and a temporary/permanent indicator associated with the on-line copy. The SYSCATL table is referred to as the Data File Catalog.

When a new data file is entered into the Non-Relational Data Base, a catalog entry is created for the data file and is inserted, as a record, into the SYSCATL table. The record will contain the physical location and format of the initial copy of the data file being cataloged, as well as any other pertinent information. The initial copy of the data file being cataloged may be on-line if it was created by an application program or internal procedure or off-line if it is an existing data file on magnetic tape. A unique data identifier is created by the Integrated Data Base Management System for each new data file and a unique B-tree index is

created on the data identifier field in the SYSCATL table.

When a data file is to be processed, the data identifier must be specified. The record containing the catalog entry corresponding to the data identifier is retrieved from the SYSCATL table using the unique B-tree index created on the data identifier field. The physical location of the data file is obtained from the retrieved catalog entry so that the data file can be accessed. When commands are issued which cause a new copy of an existing data file to be created, the appropriate record in the SYSCATL table is updated with the physical location and format of the newly created copy of the data file. When an on-line copy of a data file is created, it is given a temporary status in its catalog entry and the user-id of the user creating it is retained. That user or the Data Base Administrator may issue a command to change the status of the on-line copy to permanent, in which case the appropriate record is updated in the SYSCATL table to reflect the change in status. If the user who created the on-line copy of a data file or the Data Base Administrator issues a command to scratch the on-line copy from the system, the appropriate record is updated in SYSCATL table to reflect the removal of the on-line copy. When a data file is removed from the Non-Relational Data Base, the record containing the catalog entry for the data file is located via the B-tree index on the data identifier field and the record is deleted from the SYSCATL table.

Additional superstructures are created on the SYSCATL table by the System Generation Program to facilitate the retrieval of information about data files. Inverted indices are created on several of the data fields in the SYSCATL table. Thus, information pertaining to data files in the Non-Relational Data Base can be retrieved as a function of one or more of the physical attributes of the data file.

2.3.2 The Data File Directory

The Data File Directory consists of one or more tables in the Global Data Base which provide the user community with a content based directory to data files in the Non-Relational Data Base. Unlike the Data File Catalog which maintains a one-to-one relationship with the data files and contains the physical attributes of each data file, the Data File Directory supports a many-to-one relationship with the data files and contains attribute values of data contained within each data file. Thus, using the Data File Directory, a data file can be located as a function of its information content. The addition of new directory tables to the Data File Directory is under complete control of the Data Base Aministrator, as is the format of each directory table. Thus, with no predefined structure or format, the Data File Directory can evolve to meet the changing requirements of the user community and the varying contents of the Non-Relational Data Base.

The tables which constitute the Data File Directory can be referred to collectively as if they were a single table using the name SYSDIR. Therefore, the retrieving, updating and deleting of records in the Data File Directory can be performed on an individual directory table by specifying the directory table name in the appropriate command or on the collection of all directory tables specifying the table name SYSDIR. All data records must be inserted into a specific directory table.

Since all directory tables are contained in the Global Data Base, each new directory table must be defined by the Data Base Administrator, thereby making the Data Base Administrator the owner of the directory tables. Presumably, a new directory table would be defined to support each type of data maintained in the Non-Relational Data Base. Since it is likely that specific groups of users may be more cognizant than the Data

Base Administrator of the handling required for different types of data files, the Data Base Administrator might wish to designate an individual user from each such group as a Data File Administrator. The Data File Administrator, as well as the Data Base Administrator, would have the power to grant and revoke access rights to the directory table. To accomplish this, the Data Base Administrator would transfer ownership of the directory table for a particular type of data to the user designated as the Data File Administrator for that data. It should be noted that the access rights to the collection of all directory tables, referred to as SYSDIR, are the same as the access rights to the individual directory tables. For example, if a user were granted the right to delete records in one directory table but no others, the deletion of records by that user via SYSDIR could cause the deletion of records only from the directory table for which the user had been granted the right to delete records.

When a new directory table is defined, one of the data fields in the table must be the data identifier. The other data fields should represent the attributes of the information contained in the data files for which the directory table is being created. When a record is inserted into a directory table, it must contain a non-null value in the data identifier data field. Other data fields can contain the null value. The data identifier is verified using the B-tree index on the data identifier field in the SYSCATL table, which is the Data File Catalog. Thus, an entry can be made in a directory table only for a data file for which an entry already exists in the Data File Catalog.

Because some data files in the Non-Relational Data Base can contain several types of data, a directory table can contain more than one record with the same data identifier. In fact, different directory tables can contain records with the

same data identifier; thereby allowing multiple descriptions of the same data file to exist in the Data File Directory simultaneously. When a record is deleted from the SYSCATL table, indicating that the corresponding data file is being removed from the Non-Relational Data Base, any records in the Data File Directory containing the data identifier of the data file are deleted from the directory tables. Thus, no references to data files that no longer reside in the Non-Relational Data Base will exist in the Data File Directory.

2.4 System Design Concepts

The system design described in this document relies on a set of control structures for intrasystem communication and the management of system processes. The term "system internals" is used herein to refer generically to control blocks, control block extensions, dictionaries, lists and queues and their relationship to one another. The various control structures have been divided into categories as a function of their usage within the system and are described in detail in Section 8.

All control structures are transient in nature. Transient control structures may exist only while a command is being processed or while a user is connected to the system or while a particular data base is being accessed. The control structures resident in main storage are dependent, for the most part, on interactive user and application program activity, thus reducing the main storage requirements of the system. The main storage required for the transient control structures is allocated dynamically, as required, and freed when no longer needed.

The Integrated Data Base Management System software, as described in Section 9, consists of several asynchronous processes or tasks. Thus, the software design assumes the

availability of a multitasking operating system with subtasking for implementation. The software processes are essentially event driven. That is, each software process is in a non-executing or "wait" state until one or more events on which it is waiting occurs. At that time, the process beings executing and continues in the executing or "run" state until no more work remains for it; at which time, it places itself into the wait state again. Communication among the various asynchronous processes is via the queues which were mentioned previously.

The division of the software into separate asynchronous processes is based on the various functions which must be performed on a command as it proceeds through the system. Thus, several commands can be in different stages of processing at any one time without delaying each other. When a delay does occur, the commands are held in queues to await further processing.

2.5 Backup and Recovery

The backup and recovery features of the Integrated Data Base Management System provide facilities for "backing out" the effects of a command which terminates prematurely or is aborted and for recovering the system to a consistent state after a malfunction. The mechanism for providing both types of backup and recovery is considerably different. The backup facilities and the recovery technique for each are described below.

2.5.1 Command Recovery Facilities

The command recovery facility provides the capability of removing the effects of a command which has not completed successfully. Only commands which cause the tabular data

storage area to be modified will activate the command recovery facilities. These include commands that modify superstructures and system tables as well as those that modify user defined tables. Commands which only retrieve information from the tabular data storage area do not invoke the command recovery facilities since no recovery is required should these commands fail. The command recovery facilities are designed to provide protection from intermitent command failure, not from a malfunction which causes the Integrated Data Base Management System or the operating system to terminate. Thus, this type of backup and recovery procedure is controlled by the Integrated Data Base Management System and is transparent to the user and to the computer operator. The command recovery facilities consist of two phases. The first phase is the backup phase and occurs during command processing. The second phase is the recovery phase and occurs during command termination.

During command processing, an image is written of each page in the tabular data storage area that is modified. The image, referred to as a "before" image, is simply a copy of the page prior to modification. When, during the processing of a command, a page is about to be modified, a before image is written into the tabular data storage area on the first free page on the chain of free pages. The page on which the before image was written is removed from the free page chain and is placed on a chain of backup pages associated with the command causing the modification. A page pointer to the first page on the backup chain for a command is contained in its Command Control Block. Each new before image page is placed at the beginning of the backup chain so that the backup chain will be in inverse chronological order.

When a command which has modified the tabular data storage area is terminated abnormally or is aborted, the effects of that command on the tabular data storage area must be removed.

A command can be terminated abnormally due to an Input/Output error or a hardware or software error. During command termination, the Application Program Command Terminator or the Interactive Command Terminator determines whether or not the command being terminated completed successfully. If not and the command has modified the tabular data storage area, the recovery procedure of the command recovery facilities are invoked. The recovery procedure uses the page pointer in the Command Control Block to locate the first before image page in the backup chain for the command. Each before image page contains the page number of the page of which it is an image. Using that page number, the recovery procedure can replace the existing page with its before image. If the same page were modified more than once by the same command, it may be replaced more than once during the recovery procedure; however, the recovery procedure follows the backup chain from the Command Control Block, replacing pages with their before images as they are encountered on the chain. Since the chain is in inverse chronological order, all pages will be recovered in the reverse order to that in which the modifications were made; thus, leaving all modified pages with their contents prior to execution of the command. Since tables which are being modified by a command cannot be accessed by any other command until the table is released by the terminator, no other command can be affected by the recovery procedure.

2.5.2 System Recovery Facilities

The system recovery facility provides the capability of restoring the information in the system to a consistent state prior to restarting the Integrated Data Base Management System following a malfunction which terminates execution. As in the command recovery facility, the system recovery facility consists of two phases. The first phase involves the generation of a log file. The second phase involves the recovery of the system to a consistent state using the Log File.

The generation of a Log File is a continuous process which occurs during the execution of the Integrated Data Base Management System. The log file will be written as a sequential data set. Normally, the Log File will be written on a magnetic tape but it can be written on any device which supports the sequential organization of data. The Log File contains before images of all records in the tabular data storage area that have been added, deleted, or updated; begin-command and end-command records; both application program and system checkpoint records and any other information which might provide a useful audit trail of system activity. Whenever an interactive or application program command enters the system, a begin-command record is written on the Log File. If, during processing, a command modifies the tabular data storage area, a before image of the record involved is written on the Log File. Whenever a new data file is created by the Data File Processor, a data file creation record is written on the Log File. When a command terminates, an end-command record is written on the Log File.

When a malfunction causes the execution of the Integrated Data Base Management System to be terminated, the Log File must be used to recover the system. During the restart procedure, the operator must identify the Log File to be used. The Integrated Data Base Management System will position the Log File at the end of the data set and read it in a backward mode during the recovery procedure. If a backward read feature is not supported by the operating system, the Log File will be read in a forward mode, sorted in descending order by time or a counter and written to a new data set prior to entering the recovery phase. The recovery procedure will retrieve before images of records from the Log File and restore the tabular data storage area using these before images. Using the begin-command and end-command records, a list will be generated of all commands that have been backed out during the

recovery procedure and must be reissued. When the recovery procedure encounters a system checkpoint record on the Log File, the operator is notified. The operator may terminate the recovery of the system at that point or he may specify that the recovery procedure should continue to the next system checkpoint record. When the system has been recovered, it can be restarted and users may then access the system.

SECTION 3 - USING THE INTEGRATED DATA BASE MANAGEMENT SYSTEM

3.1 Operator Control

The Integrated Data Base Management System is envisioned as being a single copy, multi-user system which operates continuously in its own region of main storage. The starting and stopping of the system will be under the control of the computer operator, using a set of special operator commands which can be entered only through the operator's console. Besides being able to start and stop the Integrated Data Base Management System, the operator can monitor the system activity and perform a system recovery operation using other operator commands.

Under normal operating conditions, the Integrated Data Base Management System will be started at the beginning of each operational day by the operator. Parameters may be entered when the system is started to control page buffer allocation, the system checkpoint interval, and other such functions. Occasionally, the operator may request information on the Integrated Data Base Management System activity. At the end of each operational day, the system will be stopped by the operator. Normally, the stopping of the system will be preceded by a message to all interactive users to disconnect from the Integrated Data Base Management System. When the operator issues the command which stops the system, no more users will be allowed to connect to the Integrated Data Base Management System. All active users will be allowed to disconnect from the system before execution is stopped. Under unusual circumstances, the operator has the capability of aborting execution of the Integrated Data Base Management System. When system execution is aborted, no new commands are accepted and all commands in progress are aborted, thus removing the effects of the executing commands from the system. The appropriate records are written to the Log File and the Log File is closed.

When the Integrated Data Base Management System terminates during execution due to a hardware or software failure, the operator must perform a system recovery operation. To do so, the operator issues a command to initiate the recovery operation and identifies the Log File to be used. The Integrated Data Base Management System uses the Log File to restore the system, stopping at each system checkpoint to allow the operator to terminate the recovery operation. When the recovery operation has been completed, the Integrated Data Base Management System is restarted.

3.1.1 Operator Commands

Operator commands are processed by the System Control Program and are not available to the Data Base Administrator or the user community. These commands can be issued only by the computer operator and allow the computer operator to control the execution of the Integrated Data Base Management System as described above.

The operator commands include:

START	-	Start the Integrated Data Base Management System
STOP	-	Stop the Integrated Data Base Management System. Allow all active users to disconnect from the system. Do not allow any new users to connect to the system.
ABORT	-	Stop the Integrated Data Base Management System immediately. Write back modified buffers to the data bases. Write necessary checkpoint records to the log file. Do not accept any new commands.
USERS	-	Display the user-id and processing status of all active users of the Integrated Data Base Management System.

STATS - Display a predefined set of usage statistics representing current activity of the Integrated Data Base Management System.

RECOVER - Perform a system recovery operation using a specified Log File and restart the Integrated Data Base Management System.

3.2 Accessing the System

The Integrated Data Base Management System is designed to be a multi-user, multi-access system. Thus, the system can support concurrent access by several users and provide multiple modes of access to the users. The system is capable of accepting and processing, concurrently, interactive commands from several remote terminals, interactive commands from a system card reader and application program commands from several application programs. The following subsections describe the concept of a Workspace Table to support retrieval and the three modes of access available to a user of the Integrated Data Base Management System.

3.2.1 The Workspace Table

One Workspace Table is associated with each interactive user and application program connected to the Integrated Data Base Management System. A Workspace Table is not contained within any data base, but is associated directly with the user or application program. A Relation Control Block for the Workspace Table is created in main storage when a user or application program connects to the system and a pointer to the Relation Control Block is stored in the User Control Block. No Domain Extension is created for the Workspace Table at the time that the user or application program connects to the system. The Workspace Table ceases to exist when the user or application program with which it is associated disconnects from the Integrated Data Base Management System.

At any given time, a Workspace Table is in one of three states. The state of a Workspace Table is a function of the previous operations performed by the user or application program. The state indicator is contained in the User Control Block for the user with which the Workspace Table is associated. The Workspace Table states are as follows:

(1) The Workspace Table is empty.
(2) The Workspace Table is not empty and contains data from the data base to which the user is currently attached.
(3) The Workspace Table is not empty and contains data from the data base to which the user was previously attached.

Data are placed into the Workspace Table as a result of a SELECT command. The SELECT command can be issued by an interactive user or an application program and retrieves data from one or more tables. The SELECT command is discussed in subsequent sections. When a SELECT command is issued, a Domain Extension is created in main storage for the user's Workspace Table and is linked to the Relation Control Block. The entries in the Domain Extension correspond to the data fields specified in the SELECT command. Each data field must exist in the Data Dictionary associated with the data base to which the user or application program is attached when the SELECT command is issued.

As a result of the execution of a SELECT command, records are retrieved which meet the conditions stated in the WHERE clause, or its application program equivalent, and are stored in the Workspace Table. The records will contain only the data fields specified in the SELECT command. The resulting Workspace Table is a sequential table. Thus, there are no superstructures associated with it. The contents of the Workspace Table may be displayed or inserted in a permanent table or, in the case of an application program, retrieved sequentially.

The Workspace Table may be referenced in subsequent SELECT commands. Any Workspace Table which exists when a SELECT command is executed is always replaced by the new Workspace Table.

If a Workspace Table is created by a user or application program while attached to one data base, data base A, and the user or application program attaches to another data base, data base B, the Workspace Table created from data base A continues to exist and is accessible to the user or application program. Therefore, the Workspace Table can be used to transfer data from one data base to another.

3.2.2 Access from a Remote Terminal

A remote terminal can be any remote transmitting and receiving device which is supported by the telecommunications monitor that performs the message handling for the Integrated Data Base Management System. Remote terminal access to the system is via a simple, yet powerful, interactive command language. Commands in the interactive command language are logically grouped into five categories which reflect the functions performed by the commands therein. Each command is identified by a key word usually followed by one or more clauses.

When processing interactive commands, the system will treat each line entered from a remote terminal, as a message. If, during the syntactic analysis of a message, the system determines that the message does not contain a complete command, the system will suspend processing of the command and await a continuation in the next message received from the terminal. If the next message received from the terminal is not the expected

continuation but is a new command, the partially processed command will be aborted and the new command will not be processed. A diagnostic message will be returned to the originating terminal notifying the user of the action taken.

The five categories of commands are described briefly below. A brief description of the function performed by each command is included. A more extensive description of each command, including command syntax, is contained in Section 4.

3.2.2.1 Utility Commands

Utility commands provide the user with general support functions which include connecting to and disconnecting from the Integrated Data Base Management System, designating a data base for processing, browsing through data bases, specifying alias names for tables, changing passwords and using the menu feature. These commands are available to the user community without restriction.

The utility commands include:

ENTER	- Connect a user to the Integrated Data Base Management System.
EXIT	- Disconnect a user from the Integrated Data Base Management System.
DESCRIBE	- Display a textual description of the contents of data bases, tables and fields.
ATTACH	- Indicate a user's intent to process information in a particular data base.
USE	- Establish a one character alias name for a table.
PASSWORD	- Change a password.
MENU	- Display a list of the available interactive commands or specify the mode of interactive processing to be used.

3.2.2.2 Data Definition Commands

Data definition commands provide the user with the capability of dynamically defining and altering the structure of data bases, tables and fields managed by the Integrated Data Base Management System. Also, the user can dynamically control the superstructures imposed on tables within a data base that facilitate rapid access to information. Use of these commands is restricted to the owner of the data base, table or field being referenced.

The data definition commands include:

DEFINE	- Identify a new data base, table, field, user or group to the Integrated Data Base Management System.
REMOVE	- Remove a data base, table, field, user or group from the Integrated Data Base Management System.
EXPAND	- Add new data fields to an existing table.
INVERT	- Create the required indices such that an inverted superstructure is placed on specified fields in a table.
INDEX	- Create a hierarchical B-tree superstructure on specified fields in a table.
DROPINDEX	- Remove both hierarchical B-tree and inverted superstructures from specified fields in a table.

3.2.2.3 Administrative Commands

Administrative commands provide the user with control over the state and accessibility of data bases under his purview. Using administrative commands, users can be added to or removed from groups and authorization for users to perform certain operations on data can be granted or revoked. Use of these commands is restricted to the owner of the data referenced by the command or the Data Base Administrator.

The administrative commands include:

GRANT	- Authorize individual users, groups or the entire user community the right to perform data manipulation operations on tables.
REVOKE	- Cancel previously granted authorizations.
INCLUDE	- Add a user to a group
EXCLUDE	- Remove a user from a group

3.2.2.4 Data Manipulation Commands

Data manipulation commands provide the user with access to records within the relational tables managed by the Integrated Data Base Management System. Using data manipulation commands, data can be retrieved from tables based on field values within each record, new records can be inserted in tables, records can be deleted from tables, fields within records can be modified and data can be displayed or printed. Use of these commands can be restricted by the owner of each table to a specified subset of the user community.

The data manipulation commands include:

SELECT	- Locate records in one or more tables which satisfy a specified set of conditions.
INSERT	- Add one or more records to a table.
DELETE	- Remove one or more records from a table.
UPDATE	- Modify fields in one or more records in a table.
DISPLAY	- Display fields from selected records at a remote terminal.
PRINT	- Print fields from selected records on a hardcopy device.

3.2.2.5 Data File Commands

Data file commands provide the user with the capability

of controlling the Non-Relational Data Base. Using data file commands, new data files can be entered into the Non-Relational Data Base and existing data files can be removed from it. Data Files that reside off-line on magnetic tape can be copied to an on-line file on a direct access device. Likewise, data files residing on-line can be copied to an off-line file. Use of these commands may be restricted to the Data Base Administrator.

The data file commands include:

CATALOG	- Enter a data file into the Non-Relational Data Base.
UNCATALOG	- Remove a data file from the Non-Relational Data Base.
LOAD	- Copy an off-line data file to an on-line direct access device and convert the data file to a system standard format, if necessary.
UNLOAD	- Copy an on-line data file to an off-line magnetic tape.
COPY	- Transform non-relational data files to tabular form and relational tables to data file format.
PERFORM	- Invoke a procedure from the Integrated Data Base Management System library to process data files.
KEEP	- Mark a temporary on-line copy of a data file as permanent.
SCRATCH	- Purge an on-line copy of a data file.

3.2.3 Access Via the Batch Command Reader

The Batch Command Reader refers to a reader task which places card images read from a system card reader into a data set for retrieval by the Interactive Command Input Processor. Commands from the Batch Command Reader will be treated similarly to those from a remote terminal. The Remote Terminal Communications List will contain an entry for commands entered via the Batch Command Reader. All interactive commands, except the MENU command which is described in Section 4, are valid for entry via the Batch Command Reader. Syntax checking of commands from the Batch Command Reader is the same as that for commands entered

interactively from a remote terminal. Any output resulting from the processing of a command entered through the system card reader is directed to a line printer.

Although any interactive command can be entered on cards via the Batch Command Reader, the primary use of card input will, most probably, be the entry of commands whose processing time exceeds that which a user would want to spend at a remote terminal. These commands might include the COPY command where the data file or table being copied is large, the INVERT or INDEX command to create superstructures on existing tables containing a large number of records and the LOAD or UNLOAD commands where the data file being moved is also large. Using the Batch Command Reader facility, the user can punch the commands on cards in the same format as they would be entered on a remote terminal. The first card in the command input stream must be an ENTER command just as in an interactive session. The command input stream should be terminated with an EXIT command; however, an end-of-file card will cause an EXIT command to be placed on the data set being created if one was not present in the card deck.

The implementation of the Batch Command Reader facility will be operating system dependent. Some operating systems may provide this capability with little or no additional development. Others may require special software to be written. On some systems, it may be impossible to implement this feature. Thus, the operating system on which the Integrated Data Base Management System is implemented will determine the feasibility of including the Batch Command Reader facility.

3.2.4 Access from an Application Program

The Integrated Data Base Management System is accessed from an application program using the Application Program Command Language. The Application Program Command Language is a language

which the programmer uses to cause information to be transferred between an application program and the Integrated Data Base Management System. The command language is not a complete language by itself. It relies on a host language to provide a framework for it and to provide the procedural capabilities required to manipulate data.

The Application Program Command Language consists of a set of CALL statements or its equivalent which are incorporated into a procedural host language program. The command language may be used with any host language (e.g., FORTRAN, COBOL, PL-1, assembler language) that supports a CALL statement. The subroutine name used in each CALL statement which accesses the Integrated Data Base Management System will be the same. The command to be executed will be defined by the first argument in the CALL statement argument list. For example:

```
CALL IDBMS('DELETE',...)
```

where: IDBMS is the common subroutine name
DELETE is the command to be executed.

The execution of the CALL to the subroutine, IDBMS, will cause control to be transferred to the Application Program Communication Module, described in Section 9. The remainder of the argument list will contain parameters which are relevant to the command to be executed.

The Application Program Command Language contains a set of commands for performing operations on tables and a set of commands for performing operations on data files. The application program commands that reference tabular data can include those commands that were specified for interactive users. It may be desirable to omit certain interactive commands from the Application Program Command Language, such as those in

the data definition category and, perhaps, some others; thus preventing application programs from creating and removing data bases, tables and fields and, perhaps, granting and revoking access rights. However, nothing in the system design would prevent these commands from being included in the Application Program Command Language.

As in the interactive command language, a SELECT command can be issued by an application program. The SELECT command will cause records to be placed in the Workspace Table associated with the application program, but it will not cause records to be transferred to the application program. An additional command, FETCH, is available to retrieve records serially from the Workspace table. If the Workspace Table is empty when the workspace retrieval command is issued, a status code so indicating will be returned to the application program. Otherwise, the system maintains a logical pointer, referred to as a cursor, which moves through the Workspace Table as records are accessed. The execution of a SELECT command causes the cursor to be set to the first record in the Workspace Table. The initial occurrence of the workspace retrieval command causes data to be retrieved from the first record in the Workspace Table and the cursor to be moved to the next record in the table. Each subsequent occurrence of the workspace retrieval command causes data to be retrieved from the record to which the cursor points and the cursor to be moved to the next record. When the last record in the Workspace Table is accessed, the cursor is set to indicate that the end of the table has been encountered. When the next workspace retrieval command is issued, a status code will be returned indicating an end-of-table condition has occurred.

3.2.4.1 Data Independence Within An Application Program

One of the important concepts for application programs is that of data independence. That is, the separation of the description of data maintained by the Integrated Data Base Management System from the application programs that process the data. This allows an application program to be insulated to a certain extent from changes to the data structure. Data independence within an application program is established at the data field level for tabular data. A data field represents a column within a table. Each command in an application program which initiates data transfer must specify, by name, the data fields to be transferred. The order of transfer is inferred from the order of the data field names in the argument list in the application program command. During retrieval, the data field names are used to extract data field values from records in tables. During update, the data field names are used to place data into records.

When accessing tabular data, an application program does not concern itself with record formats. The system uses information from the appropriate control structures to determine where the data fields specified in the data field name list are located in a record. Thus, the order of data fields in a table is immaterial to an application program. Therefore, data fields can be repositioned in a table or data fields not used by an application program can be added or deleted from a table without modifying or recompiling the application program. Also, data fields can be transferred between an application program and a table in any sequence without regard to their relative position within the actual table.

3.3 The Data Base Administrator

The administration of the Integrated Data Base Management System is an important function if the advantages of data base technology are to be fully exploited. The Data Base Administrator provides the coordination, perspective and administration of the system by exercising specific responsibilities. These responsibilities include the definition of system parameters, controlling the user community, controlling access to tables in the Global Data Base, the definition of directory tables, the definition of new tables in the Global Data Base and user education and assistance.

The Data Base Administrator will be responsible for specifying system parameters which are submitted to the System Generation Program during the initial system generation. Also, the Data Base Administrator would have the responsibility for providing guidelines to computer operators for the daily operation of the system. These would include any system parameters to be specified when the system is started each day and recovery procedures to be followed should the Integrated Data Base Management System malfunction in some manner or terminate processing altogether.

The responsibility for entering and removing users from the system rests solely with the Data Base Administrator. To control the user community, two privileged commands are available only to the Data Base Administrator. It is envisioned that a user wishing to gain access to the facilities of the Integrated Data Base Management System would submit a request to the Data Base Administrator who, upon approval of the request, would enter the new user into the system using the privileged command, DEFINE USER. A user-id and password would be specified for the new user in the command. To delete a user from the system, the Data Base Administrator would use the privileged command, REMOVE USER, specifying the appropriate user-id.

The Data Base Administrator, in conjunction with cognizant users, will define the format of new directory tables to be entered into the System Directory in the Global Data Base. Once the format has been defined, the Data Base Administrator will create the new directory table using the DEFINE DIRECTORY TABLE command. Since the DIRECTORY option of the DEFINE TABLE command is valid only for tables being added to the Global Data Base and since only the Data Base Administrator can add new tables to the Global Data Base, only the Data Base Administrator can add new directory tables to the System Directory. In addition to directory tables, other, non-directory, tables may be added to the Global Data Base by the Data Base Administrator at any time. These tables would probably contain information that is of general interest to the user community. Since the Date Base Administrator is the owner of the tables contained within the Global Data Base, access to these tables is controlled by the Data Base Administrator. As with tables in any data base maintained by the system, the GRANT and REVOKE commands must be used by the Data Base Administrator to control access to tables in the Global Data Base, including directory tables.

One of the primary responsibilities of the Data Base Administrator is to provide education and assistance to the users of the Integrated Data Base Management System. While it is difficult to specify, at this time, the manner in which these responsibilities should be carried out, some comments can be made. The Data Base Administrator or members of his staff could provide educational seminars for the user community. These could range from introductory seminars for new or potential users to seminars covering advanced concepts for more knowledeable users. The Data Base Administrator might distribute, periodically, a newsletter containing timely information of interest to the user community. Another area in which the Data Base Administrator must play a key role involves assisting users in entering new types of data into the Non-Relational Data Base. This requires the definition

of one or more directory tables for the new data type, as described above, and either writing or assisting a user in writing a data file input module to process the new data type.

3.4 The User Community

The term "user community" refers to all valid users of the Integrated Data Base Management System except the Data Base Administrator. A valid user of the system is one for which a record exists in the SYSUSER system table. A unique user-id and a password, which is not necessarily unique, are associated with each user. Additionally, a user may be included in a named group of users who share common access rights. Certain privileged commands, which are specified in the following subsections, allow the Data Base Administrator to control the user community by inserting and deleting records in the SYSUSER table.

3.4.1 Defining a New User to the System

A potential user cannot connect to the Integrated Data Base Management System until that user has been defined to the system by the Data Base Administrator. Prior to defining the new user to the system, a unique user-id must be assigned to the user by the Data Base Administrator. A password, which does not have to be unique, must be selected by the user or the Data Base Administrator. After assigning the user-id and password to the new user, the Data Base Administrator uses the DEFINE command with the USER option to identify the new user to the system. This command is a privileged command and will be accepted only from the Data Base Administrator. Additionally, the DEFINE USER command may specify one or more groups in which the new user is to be included for the purpose of sharing common access rights with other users. The concept of group access rights is described in the following subsection.

The execution of the DEFINE command with the USER option causes a record to be inserted into the SYSUSER system table. If one or more groups in which the user is to be included has been specified in the command, corresponding records will be inserted into the SYSGROUP system table. After the DEFINE USER command has been successfully processed, the new user can connect immediately to the Integrated Data Base Management System.

3.4.2 Defining a New Group to the System

Users can be grouped together for the purpose of sharing common access rights. Before a user can be included in a group, the group must be identified to the system. This is accomplished using a DEFINE command with the GROUP option. The DEFINE GROUP command specifies the name of the group, which must be unique among group names already known to the system. This command is a privileged command and can be issued only by the Data Base Administrator.

The execution of the DEFINE command with the GROUP option causes a record to be inserted into the SYSGROUP system table. At that time, the group will be empty; that is, the group will contain no users. Even though the group is empty, access rights can be granted to the group as described in a subsequent subsection. After successful completion of a DEFINE GROUP command, new or existing users can be included in the group as described in the following subsection.

3.4.3 Controlling Group Membership

Essentially, groups are formed to facilitate the granting and revoking of access rights to tables. As stated previously, all users in a group have common access rights to a specific set of tables. Users within a group may have additional access rights which have been granted to them individually.

Also, a user may not be a member of any group or may be a member of several groups.

A user can be included in a group in two ways. A new user can be included in one or more groups via the DEFINE USER command when he is initially defined to the system. An existing user can be included in one or more groups via the INCLUDE command. The INCLUDE command is a privileged command and can be issued only by the Data Base Administrator. When a user is included in a group, a record is inserted in the SYSGROUP system table.

A user can be removed from one or more groups of which he is a member via the EXCLUDE command. Like the INCLUDE command, the EXCLUDE command is a privileged command and can be issued only by the Data Base Administrator. When a user is removed from a group, a record is deleted from the SYSGROUP system table.

3.4.4 Removing a Group from the System

An existing group can be removed from the system using the REMOVE command with the GROUP option. The group to be removed must be named in the REMOVE GROUP command. This command does not remove users within the group from the system, but simply removes any access rights granted to users as a result of their membership in the group. The group may be empty or it may contain one or more users as members. Also, the group may have zero or more authorization records, representing access rights granted to the group, stored in the SYSAUTH system table. The granting and revoking of access rights to groups and to individual users is discussed in a subsequent subsection.

The execution of the REMOVE command with the GROUP option

causes all records corresponding to members of the group to be deleted from the SYSGROUP system table. Additionally, all authorization records that are associated with the group being removed are deleted from the SYSAUTH system table.

3.4.5 Removing a User from the System

An existing user can be removed from the system using the REMOVE command with the USER option. This command is a privileged command and can be issued only by the Data Base Administrator. The user-id of the user to be removed must be specified in the command. After removal, the user can no longer connect to the Integrated Data Base Management System. Also, the user will be removed from any groups of which he is a member and any authorizations granted to the user, as an individual, will be revoked. All data bases owned by the user will remain intact. They may be removed or their ownership transferred to another user by the Data Base Administrator.

The execution of the REMOVE command with the USER option causes the record corresponding to the user being removed to be deleted from the SYSUSER system table. All records containing the user-id of the user are deleted from the SYSGROUP system table. Also, all authorization records associated directly with the user are deleted from the SYSAUTH system table.

3.4.6 Connecting to and Disconnecting from the System

3.4.6.1 An Interactive User

The first action an interactive user of the Integrated Data Base Management System must take is to connect to the system using the ENTER command. A user-id and password must be

specified in the ENTER command. Only one user can be connected interactively to the system under a given user-id at any one time. Thus, if a user is already connected to the system under the same user-id as is specified in an ENTER command, the command will be rejected and the user will not be connected to the system. The same action is taken by the system if the password is not valid for the user-id specified in the ENTER command. No other commands can be issued by a user until a valid ENTER command has been processed for that user.

When an ENTER command is received, a check is made to determine whether or not a user is already connected interactively to the system under the user-id specified in the command. If not, the user-id and password are verified. To accomplish this, an attempt is made to retrieve a record, which is a User Control Block, from the SYSUSER system table that contains the user-id specified in the command. If such a record is located, it is read into main storage and the password contained in the User Control Block is compared with the password specified in the command. If they are the same, all authorization records associated with the individual user are retrieved from the SYSAUTH system table and are stored in the Authorization Extension to the User Control Block in main storage. All group records for groups of which the user is a member, if any, are retrieved from the SYSGROUP system table and are stored in the Group Extension to the User Control Block in main storage. Finally, all authorization records for groups to which the user belongs are retrieved from the SYSAUTH system table and are stored in the group Authorization Extension in main storage unless they are already resident therein. Then control is returned to the user and he is now connected to the system and attached to the Global Data Base. The concept of being attached to a data base for processing is discussed in a subsequent subsection.

When a user has completed an interactive session and wishes

to disconnect from the system, he must issue an EXIT command. During the execution of the EXIT command, all operations are performed to terminate processing for the user issuing the command. Main storage used for control structures which are associated only with the user being disconnected, such as the User Control Block, Authorization Extension and Group Extension, is freed. Additional control structures may be removed from main storage if they are not required to support other users connected to the system. Any data contained in the user's Workspace Table when he disconnects from the system will be lost.

3.4.6.2 An Application Program

Every application program that uses the services of the Integrated Data Base Management System must have the Application Program Communication Module linked to it. The Application Program Communication Module has a single entry point and is entered each time that entry point is referenced in a CALL statement in the application program. The command to be executed is identified by the first argument in the calling sequence. The Application Program Communication Module performs the initial processing of commands prior to invoking the Cross-Boundary System Routine to communicate with the Integrated Data Base Management System.

Prior to executing any other CALL statement referencing the Application Program Communication Module, a CALL statement must be issued to the Application Program Communication Module containing the ENTER command as its first argument. Additional arguments in the calling sequence must contain the user-id and password of the user running the application program. This causes the access rights associated with this execution of the application program to be those of the user running it. The system places no restrictions on the number

of application programs that can be run simultaneously by a single user and a user can be connected to the system interactively while one or more of his application programs are executing. Any commands issued by an application program to access the Integrated Data Base Management System that are issued prior to a successful ENTER command will be rejected with the appropriate status code returned to the application program.

When an ENTER command is received from an application program, an attempt is made to retrieve a record, which is a User Control Block, from the SYSUSER system table that contains the user-id specified in the calling sequence. If such a record is located, it is read into main storage and the password contained in the User Control Block is compared with the password specified in the calling sequence. If they are the same, the authorization records, group records, if any, and their associated authorization records are processed as described above for an interactive user. Finally, a character is appended to the user-id in the User Control Block such that the user-id is unique among those of both interactive users and application programs currently connected to the system, thereby permitting simultaneous access by the same user both interactively and via one or more application programs.

3.5 Relational Data Base Control

Using the Integrated Data Base Management System, a user can dynamically construct, extend, manipulate and destroy relational data bases to meet his changing requirements. Also, a user has complete control over which users in the user community can access his data bases and in what mode. The following subsections discuss the commands available to manage relational data bases. There is nothing inherent in the system design which would prevent all of these commands

from being issued by an interactive user or an application program. However, if desired, certain commands could be limited to interactive or application program usage, only.

3.5.1 Defining a Data Base

To define a new relational data base, a user simply issues a DEFINE command with the DATABASE option. The name of the new data base must be included in the command and must be unique among data base names already known to the system. After successful processing of the DEFINE DATABASE command, a new-relational data base will exist that is owned by the user issuing the command. However, the data base will be empty; that is, it will contain no tables. The user will be attached to the newly defined data base.

The execution of the DEFINE command with the DATABASE option causes a Data Base Control Block to be constructed for the new data base. The Data Base Control Block is inserted, as a record, in the SYSDB system table.

3.5.2 Specifying a Data Base for Processing

Every interactive user and application program connected to the Integrated Data Base Management System always has a relational data base to which any data manipulation command or other data related command will be directed. This data base is referred to as the user's primary data base and the user is said to be attached to his primary data base. More than one user can be attached to the same data base simultaneously.

As stated previously, when a user connects to the system, he is automatically attached to the Global Data Base. When an interactive user issues a DEFINE DATABASE command, he becomes attached to the newly created data base. During the course of an

interactive session or the execution of an application program, it may become necessary to access an existing data base other than the Global Data Base. To accomplish this, the user simply issues an ATTACH command specifying the name of an existing data base, which may be the Global Data Base. After processing of the ATTACH command, the data base named in the command becomes the user's primary data base. All subsequent data base related commands will reference that data base until the user issues another ATTACH command. If no other users were attached to the data base, the system will load all control blocks and extensions associated with the data base from the system tables.

At times, it is necessary for a user to transfer data from one data base to another. Two facilities within the system support this requirement. The first is the existance of a user's Workspace Table which has been discussed previously. The second is the concept of a secondary data base. A user's secondary data base is simply the data base to which the user was attached prior to attaching to his primary data base. The system always retains the identity of each user's secondary data base. If the user's Workspace Table contains data from his primary data base and he issues an ATTACH command, the contents of his Workspace Table remain intact. The Workspace Table will contain data from what is then the user's secondary data base. The contents of the Workspace Table can be placed in the primary data base, thus allowing the transfer of data from the secondary data base to the primary data base. If the user's Workspace Table contains data from his secondary data base and he issues an ATTACH command, the contents of the Workspace Table will be lost. Figure 3-1 illustrates the use of the ATTACH command, the concepts of primary and secondary data bases and the handling of the contents of a user's Workspace Table.

Command	Primary DB	Secondary DB	Comment
ENTER user-id, password	GLOBAL	None	User is attached to GLOBAL data base. Main storage is obtained for RCB for Workspace Table W.
USE S FOR SYSDIR			Establish an alias name for the System Directory, SYSDIR, in the GLOBAL data base.
SELECT (S.SCNAME,S.INST,S.DATE,S.DID) WHERE S.SCNAME='NIMBUS-G' AND S.INST='SBUV'			Workspace Table (W_G) is created from GLOBAL data base. Domain Extension is created for W_G containing SCNAME,INST,DATE and DID. Selected records are retrieved from the System Directory and are stored sequentially in W_G.
ATTACH OZONEPROFILE	OZONEPROFILE	GLOBAL	User is attached to OZONEPROFILE data base. W_G is still available to the user.
INSERT NIMBUSSBUV (W.SCNAME,W.INST,W.DATE,W.DID) WHERE *			All records in W_G are inserted into the NIMBUSSBUV table in the OZONEPROFILE data base.
SELECT (NIMBUSSBUV.DID) WHERE NIMBUSSBUV.DATE > 761110			New Workspace Table (W_O) is created from OZONEPROFILE data base. W_G ceases to exist. Domain Extension is created for W_O containing DID. Selected records are retrieved from the NIMBUSSBUV table and are stored sequentially in W_O.
DISPLAY			W_O records are displayed at the remote terminal.
ATTACH SOILMOISTURE	SOILMOISTURE	OZONEPROFILE	User is attached to SOILMOISTURE data base. W_O is still available to the user.
DELETE SEASATSMR WHERE DATE < 760101			Records are deleted from the SEASATSMR table in the SOILMOISTURE data base.
ATTACH GLOBAL	GLOBAL	SOILMOISTURE	User is attached to the GLOBAL data base. W_O ceases to exist since the user has been detached from the OZONEPROFILE data base from which W_O was created. No Workspace Table currently exists.

Figure 3-1: An Example of Workspace Table Handling and the Concept of Primary and Secondary Data Bases

3.5.3 Defining a Data Field

Each data field to be used within a table must be defined prior to its first appearance in a table definition or expansion command. A data field will be local to the data base to which the user is attached when the data field is defined. A data field can appear in zero or more tables in the data base with which it is associated.

To define a new data field, a user issues a DEFINE command with the FIELD option. The name of the new data field must be included in the command and must be unique among data field names already defined for the data base with which it is associated. Additionally, a description of the data field consisting of the data type, field length where not implied by the data type and units, where applicable, must be included in the command. After successful processing of the DEFINE FIELD command, a new data field will exist in the data base to which the user was attached when issuing the command. The new data field can now be used in the definition of new tables or expansion of existing tables in the data base.

The execution of the DEFINE command with the FIELD option causes a Data Dictionary entry to be constructed for the new data field. The Data Dictionary entry is stored in the Data Dictionary in main storage that is associated with the data base in which the new data field is contained. Also, the Data Dictionary entry is inserted, as a record, in the SYSDD system table.

3.5.4 Defining a Table

To define a new table, a user issues a DEFINE command with the TABLE option. The name of the table and the data fields which constitute the table must be included in the command.

The table name must be unique among table names already in the data base in which the table is contained. The data fields must have been previously defined within the context of the data base to which the table is being added. That is, the Data Dictionary associated with the data base must contain an entry corresponding to each of the data fields. After successful processing of the DEFINE TABLE command, a new table will be contained in the data base to which the user was attached when the command was issued. A new table can be added to an empty data base or one which already contains one or more tables. The user issuing the DEFINE TABLE command will be the owner of the new table. The table will be a sequential table in that no superstructures will exist for it. The creation of superstructures is discussed in a subsequent subsection. The new table will be empty; that is, it will contain no records.

The execution of the DEFINE command with the TABLE option causes a Relation Control Block to be constructed for the new table. Also, a Domain Extension containing one entry for each data field in the table is constructed and linked to the Relational Control Block. The Relation Control Block is placed on the chain of Relation Control Blocks for tables contained in the data base. The Relation Control Block is inserted, as a record, in the SYSREL system table. Each of the Domain Extension entries are inserted, as records, in the SYSDOM system table.

3.5.5 Expanding a Table

To append data fields to an existing table, a user simply issues an EXPAND command. The name of the table to be expanded and the new data fields to be added to the table must be included in the command. The table specified in the EXPAND command must already exist in the data base to which the user

is attached when the command is issued. The data fields must have been previously defined within the context of the data base containing the table. That is, the Data Dictionary associated with the data base must contain an entry corresponding to each of the data fields. After successful processing of the EXPAND command, the added data fields will be logically appended to the right side of the table in the order specified in the command. Superstructures can be created for the added data fields either individually or in combination with original data fields or other expansion data fields.

The execution of the EXPAND command causes a Domain Extension entry to be created for each of the added data fields. The Domain Extension entries are stored in the Domain Extension in main storage that is associated with the expanded table. Each of the new Domain Extension entries is stored, as a record, in the SYSDOM system table. Existing records, if any, in the expanded table are not modified. A null value will be supplied by the system whenever one of the added data fields is retrieved from a record that existed prior to the table expansion, unless an actual value has been stored in the added data field during an update operation. Added data fields will be physically present in records updated or inserted subsequent to the table expansion.

3.5.6 Creating and Dropping Superstructures for Tables

As stated previously, a newly defined table is considered to be a sequential table. It will remain as a sequential table until one or more indices, referred to as superstructures, are created for it. Superstructures may be created when a table is empty or after it contains records. The creation of superstructures is more efficient when the table is empty since the system need modify only the table's Relation Control Block and Domain Extension. The creation of superstructures after

records have been inserted into a table requires not only the modification of the Relation Control Block and Domain Extension, but the reading of each record in the table and the writing of records which constitute the specified superstructure.

Two types of superstructures can be created for a table: an inverted index and a B-tree index. Both types can exist for a single table. To create a superstructure for a table a user issues an INVERT or an INDEX command. An INVERT command creates an inverted index, while an INDEX command creates a B-tree index. Both commands require the name of the table for which the superstructure is being created and one or more key fields to be specified. The table named in the command must already exist in the data base to which the user is attached when the command is issued. Each key field must contain the name of one or more data fields from the table and represents an entity for which values will be maintained in the appropriate type of superstructure.

If a key field consists of a combination of two or more data fields, the key field must be given a unique name. A combination key field is specified in the form: key-name=(field-name-1,field-name-2[,field-name-n]...). The key name must not duplicate any data field name in the table for which the superstructure is being created nor any other key name already defined for that table. The data fields which constitute a combination key field need not be contiguous in the table nor do they have to be specified in the same order in the key field as in the table. A data field which has a superstructure created for it can be used in a combination key field. Also, a data field may be used in more than one combination key field. If a key field consists of only one data field, no key name is required and the data field name is used directly in the command. A single data field can not have both a B-tree index and an inverted index

created for it, but it can participate in a combination key field for both types of superstructures.

The execution of the INVERT and INDEX commands causes similar actions to take place. For any single data fields which are declared to be key fields, the corresponding Domain Extension entries are modified in main storage and are updated in the SYSDOM system table to reflect the existence of the specified superstructure. For any combination key fields, new entries are created in an auxiliary section of the Domain Extension in main storage and each new entry is inserted, as a record, in the SYSDOM system table. If the table for which the superstructures are being created is not empty, each record in the table is retrieved and the proper superstructure is created for each of the key fields.

To remove existing superstructures from a table, the user simply issues the DROPINDEX command. The name of the table with which the superstructures are associated and the names of the key fields which identify the particular superstructures to be dropped must be included in the command. The DROPINDEX command removes both B-tree and Inverted indices from a table for the key fields specified in the command. No ambiguities arrise since the key name assigned to a combination key field is unique within a table and is associated with either a B-tree or Inverted index and a single data field can have only one type of superstructure created on it.

The execution of a DROPINDEX command causes the following action to be taken. For any single data field specified in the command, the corresponding entry in the primary section of the Domain Extension is modified to reflect the removal of the superstructure from that data field and the associated record in the SYSDOM system table is updated. For any combination key fields specified in the command, the entries

associated with that key field in the auxiliary section of the Domain Extension are removed from main storage and the corresponding records are deleted from the SYSDOM system table. If the superstructures to be dropped are not empty, all pages containing records in those superstructures are returned to the free page list. A superstructure will be empty if the table with which it is associated is empty.

3.5.7 Controlling Access to a Table

3.5.7.1 Granting Access Rights

When a new table is created, the user who created it becomes its owner. Until the owner of a table grants access rights to other users, he is the only member of the user community who can access data in the table. To permit other members of the user community to access the table, the owner issues a GRANT command. The Grant command must contain three pieces of information: the access mode or modes for which rights are being granted, the name of the table on which the rights are being granted and the individual users or group to which the rights are being granted.

A table can be accessed in any one of four access modes. They are: READ, UPDATE, INSERT, DELETE. One or more of the previous key words denoting the mode of access being permitted must be included in the GRANT command. If all of the access modes are to be permitted, the access mode list in the GRANT command can be replaced by the key words ALL RIGHTS. Thus, the rights being granted can be restricted to a specific subset of the available access modes or can permit full access to a table.

The table name specified in the GRANT command identifies the table for which access rights are being granted. The

table must be contained within the data base to which the user is attached when the command is issued. As stated previously, the user issuing the GRANT command must be the owner of the table specified in the command, the owner of the data base containing the table or the Data Base Administrator.

The GRANT command must identify, either explicitly or implicitly, the users to whom the rights are being granted. Rights can be granted explicitly to individual users by including their user-ids in the command. Rights can be granted implicitly to a subset of the user community by specifying the key word GROUP followed by a previously defined group name in the GRANT command. This will have the effect of granting the specified access rights to all current members of the group. Rights can be granted to the entire user community by specifying the key word PUBLIC instead of a group name or a list of user-ids. This causes the specified access rights to be granted to every user of the Integrated Data Base Management System.

The execution of a GRANT command which includes either a group name or individual user-ids will cause one or more authorization records to be inserted or updated in the SYSAUTH system table. If the group, should a group name be specified, or an individual user, should user-ids be specified, already possess some access rights to the table named in the command, the existing authorization record associated with the group or user and the table is updated to reflect the newly granted access rights. If no access rights to the table exist for the group or individual users, an authorization entry is created and inserted, as a record, in the SYSAUTH system table.

The execution of a GRANT command which includes the key word PUBLIC rather than a group name or individual user-ids, will cause one or more flags to be set in the Relation Control

Block associated with the table named in the command. No authorization records will be inserted or updated in the SYSAUTH system table. The flags set in the Relation Control Block will permit any user to access the table in the modes that have been declared to be PUBLIC without checking the authorizations associated with that user.

3.5.7.2 Revoking Access Rights

The revocation of existing access rights to a table can be done only by the user who granted the rights or the Data Base Administrator. To revoke access rights granted on a table, the user issues a REVOKE command. The REVOKE command must contain the access modes for which rights are being revoked and the name of the table on which they are being revoked. Additionally, the command can identify the individual users or group from which the rights are being revoked.

One or more of the access modes can be included in the REVOKE command or, if rights to all access modes are to be revoked, the key words ALL RIGHTS can replace the access mode list. The table named in a command must be a table in the data base to which the user is attached when the command is issued. Also, the user issuing the REVOKE command must be the current owner of the table.

The specification of users, either explicitly or implicitly, from which access rights are to be revoked, is optional in the REVOKE command. If no users are identified in the command, all access rights both public and those granted to groups or individual users, will be revoked for the access modes specified in the command. Thus, the owner of the table will become the only member of the user community who can access the table in those modes. Optionally, the REVOKE command can include individual user-ids identifying users from which access rights

are to be revoked or can specify the key word GROUP followed by a group name to indicate a group from which access rights are to be revoked or can include the key word PUBLIC. If the key word PUBLIC is included in the REVOKE command, general access to the table by the user community in those modes specified in the command will be inhibited. The access rights associated with the table in those modes will revert to those rights that were previously granted to groups or to individual users and have not since been revoked.

The execution of a REVOKE command which includes either a group name or individual user-ids will cause one or more authorization records to be deleted from or updated in the SYSAUTH system table. If the group, should a group name be specified, or an individual user, should user-ids be specified, possess access rights to the table named in the command other than those being revoked, the existing authorization record associated with the group or user and the table is updated to reflect the loss of access rights. If no access rights to the table beyond those being revoked exist for the group or individual user, the authorization record is deleted from the SYSAUTH system table.

The execution of a REVOKE command which includes the key word PUBLIC rather than a group name or individual user-ids will cause one or more flags to be reset in the Relation Control Block associated with the table named in the command. No authorization records will be deleted from or updated in the SYSAUTH system table. Thus, access to the table in the modes for which public access has been revoked, will be denied to the user community as a whole, but will still be permitted for users to whom access rights have been granted either individually or as a member of a group.

The execution of a REVOKE command which does not contain

a clause identifying, either implicitly or explicitly, the users from which rights are to be revoked may cause one or more authorization records to be deleted from or updated in the SYSAUTH system table and one or more flags to be reset in the Relation Control Block associated with the table named in the command. Thus, the actions performed are a combination of those performed when either a group name or individual user-ids is specified or the key word PUBLIC is used in the REVOKE command. This permits the revocation of access rights to the table for the modes specified without requiring the knowledge of those users to whom access has been granted.

3.5.8 Manipulating Data in a Table

There are several commands available to a user of the Integrated Data Base Management System to manipulate and exhibit tabular data. These commands permit users to insert new records into a table, delete or update existing records in a table and retrieve data fields from one or more tables into a Workspace Table. Additionally, commands are available which transfer data from tables to a printer or to a remote terminal for display purposes.

Data manipulation commands can be performed without restriction by users upon tables of which they are the current owners. Use of these commands by users other than the owners is controlled by the owner as described in the previous subsection. The access modes, INSERT, UPDATE and DELETE, which can be specified in the GRANT and REVOKE commands control directly a non-owner's ability to perform insertions, modifications and deletions, respectively, on a table. The READ access mode controls a non-owner's ability to retrieve data from a table for the purpose of storing it in a Workspace Table or for printing or remote terminal display. Thus, for example, if a user who is not the owner

of a table were granted READ and UPDATE rights to a table, he could retrieve data from the table into his Workspace Table or print or display data from the table and, also, update existing records in the table, but he could not insert new records into the table or delete existing records from the table.

3.5.8.1 Inserting Records into a Table

To add one or more records to a table, a user issues an INSERT command. The name of the table to which the record or records are to be added must be included in the command. The table must be contained within the data base to which the user is attached when the command is issued. Additionally, the user issuing the command must be the owner of the table or must have been granted the right to insert records into the table.

The record or records to be inserted can be specified in one of two ways. Either the data values to be stored in each data field in a new record can be specified explicitly in the INSERT command or existing records can be retrieved from other tables in the data base and inserted into the table specified in the command. Using either form of the INSERT command, a null value is stored in each data field which is not specified in the command.

Each new record is stored in the tabular data storage area on a physical page that has been allocated to the table in which the record has been inserted. All superstructures associated with the table are updated to reflect the existance of the new record.

3.5.8.2 Updating Records in a Table

To modify one or more records in a table, a user issues

an UPDATE command. The name of the table containing the record or records to be modified must be included in the command. The table must be contained within the data base to which the user is attached when the command is issued. Additionally, the user issuing the command must be the owner of the table or must have been granted the right to update records in the table.

The UPDATE command selectively modifies data fields within existing records in a table. Each data field to be modified and its new value must be specified as an assignment statement. The new value may be a constant or a function which can be used to compute the new value (e.g., FREQ=42.7 or FREQ=1.1*FREQ). The records to be updated are identified in a WHERE clause which specifies the conditions that must be met by a record for it to be selected for modification. Any superstructures associated with the table that are affected by the modification of one or more records, are updated to reflect the changes in those records.

3.5.8.3 Deleting Records from a Table

To delete one or more records from a table, a user issues a DELETE command. The name of the table containing the record or records to be deleted must be included in the command. The table must be contained within the data base to which the user is attached when the command is issued. Additionally, the user issuing the command must be the owner of the table or must have been granted the right to delete records from the table.

The DELETE command removes entire records from a table. The records to be removed are identified in a WHERE clause which specifies the conditions that must be met by a record for it to be deleted. Any superstructures associated with

the table that are affected by the deletion of a record, are updated to reflect the removal of that record from the table.

3.5.8.4 Retrieving Records from a Table

Records can be retreived from a table for any one of three purposes: to create a Workspace Table, to display data fields from the records at a remote terminal or to print data fields from the records. No matter what the purpose, the user issuing the command must be the owner of any table from which data is to be retrieved or must have been granted the right to read each such table. A specific data manipulation command is associated with each type of retrieval. These commands are discussed in the following paragraphs.

The SELECT command is an exceptionally powerful retrieval command which provides the capability of retrieving data fields from one or more tables in a data base to create records in the user's Workspace Table. The SELECT command must include a list of data fields, referred to as the target list, which defines the record format for the Workspace Table. The data field names in the target list may have to be qualified by a table name if data fields from more than one table are to be joined in the resulting Workspace Table (e.g. (TAB1.SC,TAB2.INST,...)). The records to be retrieved are identified in a WHERE clause which specifies the conditions that must be met by a record for it to be selected for retrieval. Only those data fields identified in the target list are extracted from the records that satisfy the WHERE clause.

The results of a SELECT command will be zero or more records contained within a sequential table known as the Workspace table. The resulting records in the Workspace Table may contain all or a subset of the data fields from

a single table or from multiple tables. The records selected to create the Workspace Table may have been retrieved from a single table or from multiple tables. In addition, the contents of a data field in one table can be used to identify records to be retrieved from another table. For example, consider two tables, T1 and T2, both containing data fields, SC, whose values are drawn from the domain of all spacecraft names. The following SELECT command will cause records to be retrieved from table T1 as a function of spacecraft names contained in the data field SC in table T2.

```
SELECT (T1.SC,T1.INST,...)
   WHERE T1.SC=T2.SC...
```

The current contents of a user's Workspace Table can be referenced in a SELECT command in the same manner as any other table. The reserved table name, W , is used to refer to the Workspace Table. After successful execution of a SELECT command, the previous Workspace Table, if any, will be replaced by the new Workspace Table.

The DISPLAY command is used to return the contents of one or more data fields from a single table to a remote terminal. If no table is named in the command, the Workspace Table is assumed. If a table is named, it must be the Workspace Table or a table contained within the data base to which the user is attached when the command is issued. When the list of data fields to be displayed is omitted from the command, all data fields in the table are displayed. Otherwise, only those data fields named in the list are displayed. The data values will be displayed in a predefined format unless a format specification is included in the command.

The PRINT command is used to print the contents of one

or more data fields from a single table. The syntax of the PRINT command is exactly the same as that of the DISPLAY command except that a title can be specified in the PRINT command. The title will be printed at the top of the first page in the printed output.

3.5.9 Removing a Table

An existing table can be removed from the system using the REMOVE command with the TABLE option. The table to be removed must be named in the REMOVE TABLE command. The table must be contained within the data base to which the user is attached when the command is issued. Additionally, the user issuing the command must be the owner of the table being removed, the owner of the data base containing the table or the Data Base Administrator. After successful processing of the REMOVE TABLE command, the table and any associated superstructures will be removed from the system.

The execution of the REMOVE command with the TABLE option causes the data records in the table, if any, to be deleted and the pages in the tabular data storage area that contained them to be returned to the free page list. Also, any superstructure records associated with the table are deleted and the pages returned to the free page list. The Relation Control Block and Domain Extension associated with the table are removed from main storage and the corresponding records are deleted from the SYSREL and SYSDOM system tables, respectively. Finally, any authorization records corresponding to rights granted on the table being removed, are deleted from the SYSAUTH system table.

3.5.10 Removing a Data Field

An existing data field can be removed from the system using the REMOVE command with the FIELD option. The data

field to be removed must be named in the REMOVE FIELD command. The data field must be contained in the Data Dictionary associated with the data base to which the user is attached when the command is issued. Additionally, the user issuing the command must be the owner of the data base containing the data field or the Data Base Administrator. Also, a data field can not be removed from a data base if it is currently being used in a table contained in the data base. After successful processing of the REMOVE FIELD command, the description of the data field will be removed from the Data Dictionary and the data field can not be used in the definition of any new tables in the data base.

The execution of the REMOVE command with the FIELD option causes the entry for the specified data field to be removed from the appropriate Data Dictionary in main storage. The corresponding Data Dictionary entry record is deleted from the SYSDD system table.

3.5.11 Removing a Data Base

An existing data base can be removed from the system using the REMOVE command with the DATABASE option. The data base to be removed must be named in the REMOVE DATABASE command. The user issuing the command must be the owner of the data base being removed or the Data Base Administrator. After successful processing of the REMOVE DATABASE command, the data base and its Data Dictionary, all tables and their superstructures and all authorizations associated with tables in the data base will be removed from the system.

The execution of the REMOVE command with the DATABASE option causes the data records in all tables contained within the data base to be deleted and the pages that contained them to be returned to the free page list. Also, any superstructure records associated with the tables are deleted

and the pages returned to the free page list. The Data Base Control Block and the Data Dictionary associated with the data base as well as the Relation Control Blocks and Domain Extensions associated with the tables in the data base are removed from main storage. The corresponding records are deleted from the SYSDB, SYSDD, SYSREL and SYSDOM system tables. Finally, any authorization records corresponding to rights granted or any of the tables in the data base being removed, are deleted from the SYSAUTH system table.

3.6 Using the Data File Directory

The Data File Directory consists of one or more tables contained within the Global Data Base and provides the user community with the capability of locating data files in the Non-Relational Data Base as a function of their data content. The Data File Directory has no predefined structure. That is, it can contain as many tables as are required to reflect adequately the types of data contained in the Non-Relational Data Base and each of the tables can be defined so as to best describe the particular data files to which it refers. Thus, new directory tables can be defined as necessary, existing directory tables can be expanded and obsolete directory tables can be removed from the system.

Although each of the tables that constitute the Data File Directory are independent and can be accessed independently, the system maintains sufficient information to relate all directory tables in the Global Data Base such that they can be referred to collectively. Thus, each directory table has a name by which it can be accessed directly while the set of all directory tables can be referred to collectively using the table name SYSDIR.

Each record in a directory table contains a data identifier corresponding to the data file in the Non-Relational Data Base to which the values of the other data fields in the record pertain. One record in a directory table contains the attributes (such as spacecraft, date, time, latitude, longitude, etc.) that describe the data contained in the data file referenced by the data identifier in the record. Since a single data file may contain several logical subfiles whose attribute values differ (e.g., maps measuring different physical variables such as rainfall rate, cloud cover, etc. in a single data file), more than one record in a directory table can point to the same data file (i.e., contain the same data identifier). For example, consider a data file that contains measurements of several physical variables. If a directory table which references that type of data file has only a single data field to indicate physical variable type, multiple records pointing to the data file could be stored in the directory table to reflect the different physical variables measured in the data file.

3.6.1 Defining a Directory Table

The definition of a new directory table is essentially the same as the definition of any new table with some exceptions noted below. As in the definition of any table, a DEFINE command with the TABLE option is used. However, the key word, DIRECTORY, must precede the TABLE option when a new directory table is being defined. Therefore, the command to define a directory table is DEFINE DIRECTORY TABLE. The command must include the name of the directory table and the data fields in the table. The DEFINE DIRECTORY TABLE command is a privileged command and can be issued only by the Data Base Administrator. Additionally, this command will be accepted and processed by the system only if the Data Base Administrator is attached to the Global Data Base when it is issued. The table name must

be unique among table names already in the Global Data Base and the data fields must have been previously defined within the context of the Global Data Base. Unlike other tables, one of the data fields in every directory table must be the data identifier field, DID. Thus, every record in a directory table will point to a data file in the Non-Relational Data Base. Presumably, the other data fields would represent the attributes associated with the type of data (e.g., NIMBUS-G SMMR PARM-30 data or LANDSAT image data) for which the directory table is being created.

After successful processing of the DEFINE DIRECTORY TABLE command, a new table will be contained in the Global Data Base. The table will be logically treated by the system as part of the Data File Directory. It will be accessed, along with the other tables, whenever the table name SYSDIR is used in a command. Since the DEFINE DIRECTORY TABLE command can be issued only by the Data Base Administrator, he will be the owner of the new directory table. Subsequently, the Data Base Administrator may change the ownership of a directory table to a member of the user community, thus extending control over the authorization of access rights to that user. It is expected that the right to retrieve data from a directory table will be granted to the entire user community while the right to modify the table will be restricted to a small subset of the user community or to the Data Base Administrator alone.

As with any table, a new directory table will be, initially, a sequential table and will be empty. Superstructures can be defined for a directory table while it is empty or after records have been entered into the table in the same manner as any other table. A directory table can be expanded by its owner with the added data fields containing null values in any existing records until those data fields are updated.

3.6.2 Modifying the Data File Directory

The modification of a directory table is performed by the same data manipulation commands, INSERT, DELETE and UPDATE, which are used to modify any table. A directory table could be modified interactively, via the Batch Command Reader, from an application program or from any special purpose programs written to extract information from a data file header to create one or more directory records describing a new data file. A directory table can be modified by the Data Base Administrator, the owner of the directory table if other than the Data Base Administrator, and any user who has been granted the appropriate rights.

New records can be inserted into individual directory tables only. That is, SYSDIR cannot be specified as the table into which new records are to be stored by an INSERT command. However, the DELETE and UPDATE commands can be used to modify individual directory tables or the Data File Directory as a whole. If a DELETE or UPDATE operation is to be restricted to an individual directory table, the name of that table should be included in the command. If the operation is to be performed over the entire Data File Directory, SYSDIR should be used as the name of the table to be modified. The modification of the entire Data File Directory is carried out on a table by table basis. For each directory table to be modified, the system determines whether the user issuing the command has the right to perform the specified operation. If not, that directory table is not modified. When performing an UPDATE operation on the entire Data File Directory, the system determines, for each individual directory table, whether or not that table contains all of the data fields to be updated. If not, the system checks the next directory table until all directory tables have been processed. If a directory table is encountered which possesses all of the data fields to be

updated, any records in the table that satisfy the WHERE clause are updated appropriately. As with the UPDATE command, a DELETE command which specifies SYSDIR as the table to be modified causes each individual table in the Data File Directory to be processed. Any record in an individual directory table which satisfies the WHERE clause will be deleted.

During the modification of a directory table, the data field containing the data identifier is treated somewhat differently from the other data fields. The data field containing the data identifier must not contain the null value when a new record is being inserted into a directory table. Additionally, the data identifier specified in a new record must match an existing data identifier in the Data File Catalog which is the SYSCATL system table. During an update operation, a null value cannot be stored in the data field containing the data identifier. Also, any new value stored in the data field containing the data identifier during an UPDATE operation, will be checked against the Data File Catalog for validity. Finally, to ensure consistency between the Data File Directory and the Data File Catalog, whenever a data file is removed from the Data File Catalog via the UNCATALOG command, all records containing the corresponding data identifier will be deleted from the Data File Directory.

3.6.3 Retrieving Data from the Data File Directory

Retrieving data from a directory table is performed in the same manner as retrieving data from any other table in the system. The SELECT command, DISPLAY command and PRINT command can be used to retrieve data from individual directory tables or from the entire Data File Directory. As with other tables, the SELECT command will place the retrieved records in the user's Workspace Table, the DISPLAY command will exhibit the

retrieved data on the user's remote terminal and the PRINT command will print the retrieved data. Data can be retrieved from individual directory tables by specifying the table name explicitly in the command. Data can be retrieved from the entire Data File Directory by specifying SYSDIR in place of the individual directory table name. When the table name, SYSDIR, is specified in a retrieval command, each directory table is checked to determine if all data fields in the target list are contained within the table. If not, no data is retrieved from that table and the system checks the next directory table until all tables in the directory have been checked. If a directory table is encountered which contains all of the data fields in the target list, the data fields are retrieved from the records that satisfy the WHERE clause specified in the command.

For most users, the retrieval of data from the Data File Directory will be the initial step in obtaining data for study purposes. Using one of the retrieval commands, a user can locate the data files that possess spacial and temporal as well as other attributes required for his work. Several retrievals may be required from the Data File Directory to locate the required subset of data files in the Non-Relational Data Base. Once the required data files have been located, the user may wish to retrieve the records from the Data File Directory which point to these data files. Using a SELECT command, the user can retrieve those records and store them in a table in another data base. In this way, the user can create his own directory tables. It should be noted that these directory tables in user data bases are not maintained by the system as part of the Data File Directory.

3.7 The Non-Relational Data Base

The term "Non-Relational Data Base" refers to all data files for which an entry exists in the SYSCATL system table,

which is the Data File Catalog. As discussed in Section 2, up to three copies of a data file can exist within the system, simultaneously. The Data File Catalog can retain the location of an off-line copy of the data file in its original format, an on-line copy in one of the system standard formats and an off-line copy in the same system standard format. Using the facilities of the interactive command language, a user can manipulate data files in several ways. New data files can be cataloged, making them known to the system, and existing data files can be uncataloged, thus removing them from the system. Data files can be loaded on-line, placing them in system standard format or unloaded off-line in the same system standard format. Procedures, such as regridding or windowing, can be performed on data files under user control. Additionally, data files can be converted to tables for processing by the relational front-end and tables can be converted to data files. Also, application programs can read one or more data files and create new data files that become part of the Non-Relational Data Base. All data file commands, except the COPY command, can be issued while the user or Data Base Administrator is attached to any data base. The COPY command must reference a table in the data base to which the user issuing the command is attached. Thus, the ability to locate, manipulate and process this large, sequentially organized, data base provides the Integrated Data Base Management System with considerable power and flexibility. The use of all of the facilities for handling data files within the system is described in the following subsections.

3.7.1 Adding a Data File to the Non-Relational Data Base

A new data file can be added to the Non-Relational Data Base in two ways: by an application program or, interactively, using the CATALOG command. The creation and processing of data files by an application program is discussed in a subsequent

subsection. This subsection deals with the use of the CATALOG command. The CATALOG command logically enters a data file into the Non-Relational Data Base by creating a Data File Catalog entry and inserting it, as a record, into the SYSCATL system table. The CATALOG command is a privileged command and can be issued only by the Data Base Administrator and only while attached to the Global Data Base which contains the system tables.

Physically, the new data file must reside on a magnetic tape and should be placed into the tape library reserved for the Non-Relational Data Base. The CATALOG command must specify the physical location of the data file being added. The representation of the physical location may be system dependent but, most likely, will consist of a volume serial number, a file number and a format code. The format code indicates the format of the records in the data file and identifies the sub-routine, if one exists, in the system library which is used to load the data file on-line. If a duplicate volume serial number, file number and format code already exist in the SYSCATL system table, the new record is not inserted, but the data identifier of the matching entry is returned to the user. Otherwise, the system will assign a unique data identifier to the data file being added and the new record will be inserted in the Data File Catalog after which the data file is considered to be contained within the Non-Relational Data Base.

3.7.2 Removing a Data File from the Non-Relational Data Base

An existing data file can be removed from the non-relational data base only by the UNCATALOG command. The command must contain the data identifier of the data file to be removed. The UNCATALOG command is a privileged command and can be issued only by the Data Base Administrator and only while attached to the Global Data Base which contains the system tables.

The execution of the UNCATALOG command causes the record containing the specified data identifier to be deleted from the SYSCATL system table. Any records in the Data File Directory in the Global Data Base that contain the data identifier are deleted from the directory tables. If an on-line copy of the data file exists, it is deleted and the direct access storage is freed. Any off-line copies of the data file will continue to exist but will not be accessible via the Data File Directory or the Data File Catalog.

3.7.3 Loading a Data File

The action of loading a data file refers to the transference of an off-line data file on magnetic tape to a direct access device. A data file which has been loaded will always be in one of the system standard formats while the off-line data file may have been in its original data file format or in the same system standard format. Thus, the loading process will always transform a data file to a system standard format, if necessary. The term "on-line" will be used to refer to a loaded data file on a direct access device. However, the data file is still sequentially organized and should not be confused with tabular data.

To transfer an off-line data file on magnetic tape to an on-line direct access device, the user issues a LOAD command. The data identifier of the data file to be loaded must be specified in the command. The execution of the LOAD command may require that a format conversion routine be loaded from the system library. The basic purpose of the routine would be to convert the original data file format of the off-line data file to a system standard format. However, other operations could be performed by the load routine. Further parameters required to control the operation of the load routine would be dependent upon the particular routine being used and the content of the

data file being loaded. For example, if the load routine had the capability of extracting a single physical variable from a data file which contained several physical variables, it might be required that the user indicate if physical variable selection is desired and, if so, which physical variables were to be extracted.

If the loading of a data file causes it to be modified, as described in the example above, a new data identifier will be assigned to the loaded data file since its content is different from that of the original data file. A Data File Catalog entry will be created for the loaded data file and will be inserted, as a record, in the SYSCATL system table. If the content of the loaded data file is the same as that of the original data file, the record in the SYSCATL system table corresponding to the original data file is updated to reflect the existance, location and format of the on-line, loaded data file.

When a data file is placed on a direct access device via the LOAD command, the on-line data file is marked as a temporary file. Temporary data files will be periodically scratched from the on-line environment by a utility program. To prevent an on-line data file from being scratched, a user must issue a KEEP command. The data identifier of the data file to be marked as permanent must be included in the command. The execution of the KEEP command causes the on-line copy of the data file whose data identifier is specified in the command to be marked as permanent. This action does not affect any off-line copies of the same data file.

There are several reasons why a user might wish to cause a data file to be loaded on-line. Some data files contain massive amounts of data and, for these types of data files, the load operation would perform windowing functions. Thus, by loading

the data file a user can select only the subset of the data in which he is interested. Additionally, having a data file on-line will reduce the time required for an application program to access that data file by eliminating the tape mounting delay. Many of the procedures which can be invoked via the PERFORM command will require that input data files be in system standard format. Therefore, it may be required that a data file be loaded prior to performing some procedure on it. The PERFORM command is discussed in a subsequent subsection. Also, prior to copying a data file to a table, it may be necessary to load that data file to either reduce the amount of data placed in the table or to convert a data file, whose original data file format is not compatible with the COPY command, into a compatible system standard format. A discussion of the COPY command is contained in a subsquent subsection.

3.7.4 Unloading a Data File

The unloading of a data file refers to the transference of a data file on a direct access device to an off-line magnetic tape. The on-line data file will be in a system standard format and no conversion or modification, such as windowing, can occur when the off-line data file is created. Thus, the unloading of a data file produces an off-line copy of the data file in the same system standard format as the on-line data file. The data identifier associated with the data file to be unloaded must be specified in the command. If an on-line copy of the data file does not exist or an off-line copy in system standard format already exists, the command will not be executed. Otherwise, the data file will be copied to a magnetic tape in the same system standard format as the on-line data file and the record corresponding to the data file in the SYSCATL system table will be updated to reflect the existance and physical location of the off-line copy of the data file.

The on-line copy of the data file will not be affected by the execution of the UNLOAD command. It will not be scratched or modified in any way. However, a user can cause an on-line copy of a data file to be removed from the system at any time by issuing a SCRATCH command. The data identifier for the data file whose on-line copy is to be scratched must be included in the command. If no on-line copy of the data file exists, no action is taken. Otherwise, the on-line copy of the specified data file is scratched, whether or not it has been marked as temporary or permanent. The SCRATCH command has no effect on off-line copies of the data file. The execution of the SCRATCH command frees the direct access space allocated to the on-line copy of the data file and updates the record corresponding to the data file in the SYSCATL system table to indicate the removal of the on-line copy. If, when the SCRATCH command is issued, no off-line copy of the data file either in the original data file format or in a system standard format, exists, the command will not be executed since this would cause the ultimate loss of the data file. In this case, the user should issue an UNCATALOG command to remove the data file, if that is what is desired.

3.7.5 Invoking Data File Processing Procedures

The interactive display and manipulation of the contents of data files is an important feature of the Integrated Data Base Management System. This facility is invoked via the interactive command, PERFORM. The name of the procedure being invoked must be included in the command.

The execution of the PERFORM command requires that a subroutine be loaded from the system library. Any number of such routines may exist in the system library and new routines to perform additional procedures can be added at any time. Thus, the system supports an open-ended facility for the display and manipulation of data files. Routines could be included

that display or plot the contents of a data file, perform regridding operations on a data file, perform windowing, slicing or splitting operations on a data file or merge several data files onto a single grid.

Each of the procedures invoked by the PERFORM command will use, as input, one or more data files residing on a direct access device in a system standard format. Any new data files created by the procedure would be stored on a direct access device in one of the system standard formats. A new data identifier would be assigned to the resulting data file and a Data File Catalog entry would be created for each new data file and inserted, as a record, in the SYSCATL system table. Any new data files created by a performed procedure would be marked as a temporary file. The concept of temporary data files and their handling is discussed in the previous subsection entitled Loading a Data File.

3.7.6 Data File/Table Conversion

Although the Integrated Data Base Management System is based on the division of data into two types or forms, tabular data and sequentially organized data files, there are times when it is convenient for a user to have the capability of converting data from one form to the other. To provide this capability, the interactive command language includes a COPY command which copies data files to tables and tables to data files performing the necessary conversion of physical data structure.

To copy an existing data file to a table, the user issues a COPY command which includes the data identifier of the data file to be copied and the name of the table into which the data is to be copied. The table specified in the command must already exist in the data base to which the user is

attached when the command is issued. The data file identified in the command may be in a system standard format or its original data file format. If a system standard format copy of the data file exists, it will be used as the source of the data records. No conversion takes place during the copy operation. Each logical record in the data file is placed in the tabular data storage area as a record in the specified table. Thus, data fields in the table should match, in type and length, those in the data file being copied. Any superstructures defined on the table are updated as the records are inserted into the table. The table must have been defined prior to issuing the COPY command, however, the table need not be empty. The execution of the COPY command which copies an existing data file to a table does not affect the data file in any way.

To copy a table to a data file, the user also issues a COPY command which specifies only the name of the table to be copied. The table must exist in the data base to which the user is attached when the command is issued. The execution of the COPY command to copy a table to a data file causes the table to be read sequentially, whether or not any superstructures exist for it. Only data records in the table, not superstructure records, are copied to the new data file. The data records in the table are written to a new data file on a direct access device in a system standard format. The system assigns a data identifier to the newly created data file. A Data File Catalog entry is created for the data file and is inserted, as a record, in the SYSCATL system table. The execution of a COPY command which copies a table to a data file does not affect the contents of the table in any way.

3.7.7 Data File Processing by Application Programs

The Application Program Command Language contains commands that manipulate not only tabular data but data files, as well.

All application program commands are issued via a CALL statement which uses the same subroutine name. For example purposes only, we have used IDBMS as the subroutine name. The first argument will be the application program command to be executed (e.g., SELECT, OPEN, READ). The remaining arguments will be a function of the command being issued. Section 5 describes the Application Program Command Language and includes the argument list for each command.

Several of the interactive commands from the data file category can be issued from an application program. These include COPY, LOAD and UNLOAD. However, a number of additional commands are available to an application program for the processing of data files. These commands permit the opening and closing of data files, the reading and writing of data records in a data file, the reading and writing of header and processing history records, the searching of a data file for a particular string and the retrieval of format information pertaining to the data file from the Data File Catalog. These commands are discussed briefly below.

The OPEN command logically connects a data file to an application program. The argument list contains the mode in which the application program will access the data file. The available modes are INPUT, OUTPUT and OUTIN. The first two access modes are self-explanatory. The third, OUTIN, indicates that the data file will be created by the application program and then modified by the application program. If the file is being opened in the INPUT mode, the argument list must specify the data identifier of the data file to be opened. If the data file is being opened in the OUTPUT or OUTIN mode, the system will assign a data identifier to the data file to be created and will return it to the application program via the data identifier argument. The OPEN command will also perform all operating system dependent open functions for the data file.

The CLOSE command logically disconnects a data file from an application program. The argument list must specify the data identifier of the data file to be closed. If the access mode associated with the data file being closed is OUTPUT or OUTIN and the data file was written successfully, a Data File Catalog entry is created and inserted, as a record, in the SYSCATL system table to reflect the existance and physical location of the new data file. The CLOSE command also performs all operating system dependent close functions for the data file.

The READ command retrieves into a work area within an application program, all or part of a data record from a data file. The data identifier of the data file to be accessed must be specified in the argument list. If a portion of the logical record is to be retrieved, the starting byte location and the length of the portion to be retrieved must also be specified in the argument list. The READ command includes, in its argument list, a logical record number which allows the data file to be positioned to a specific logical record for retrieval.

The WRITE command writes a new logical data record into a data file. The data identifier for the data file must be included in the argument list. As in the READ command, a logical record number can be included in the argument list of the WRITE command to position a data file to an existing record such that it can be overwritten. The overwriting of records in a data file is permitted only if the file was opened in the OUTIN access mode and has not been closed in the interim. The WRITE command permits all or a portion of a logical record to be written. If a new record is being written and only a portion of that record is specified in the argument list, the remainder of the record will contain binary zeros. If an existing record is being overwritten, only the portion of the record specified in the argument list is overwritten while any other fields in the existing record are retained.

The SEARCH command permits the scanning of data files to locate a particular string of characters. As in the READ and WRITE commands, a logical record number can be specified to position the data file prior to beginning the search. The argument list must contain the start byte and length of the string to be checked in each logical record. Also, the argument list must contain one of the relational operators, EQ, NE, LT, LE, GT or GE, indicating the type of comparison to be made. Finally, the argument list must point to a work area containing a string which is to be compared with the string retrieved from each logical record in the data file. During the execution of the SEARCH command, a string of characters defined by the start byte and the length specified in the argument list is retrieved from each data record read, and is compared with the string in the work area using the relational operator. When the relation condition is true or an end-of-file is encountered, the execution of the SEARCH command is terminated. If a match occurs prior to the end-of-file condition, the logical record number of the matching record is returned to the application program. A READ command specifying that logical record number can be used to retrieve data from the record.

The GET command permits an application program to retrieve records from a table based on the logical ascending sequence imposed on a table by a B-tree index. Each time that an application program issues a GET command, data fields from the record containing the next highest key value in the specified B-tree index are returned to the application program. To facilitate the traversal of a B-tree index, each such index has a cursor associated with it. These cursors are maintained by the system and move independently through their associated B-tree index whenever a GET command is issued referencing the key field on which it is created. An additional feature of the GET command is the ability to specify the starting point in the key sequence at which retrieval should begin.

Two other commands are associated with the use of the GET command. They are the LOCK and UNLOCK commands. Since the GET command uses a B-tree index to determine which record in a table to retrieve, no modifications to that table or the B-tree index can be permitted. Thus, an application program must issue a LOCK command for a table prior to issuing any GET commands referencing that table. The LOCK command can also be used at any time that an application program requires control over a table. This could occur prior to modifications of the table as well as prior to issuing GET commands. When issued, the LOCK command prohibits any interactive user or application program from modifying the table if the READ mode is specified in the command or, if the MODIFY mode is specified, it prohibits all access to the table. The UNLOCK command simply releases control over a table which was established by a previous LOCK command.

The FORMAT command permits an application program to retrieve information from the Data File Catalog concerning the existance of off-line and on-line copies of a data file and their associated formats. The data identifier of the data file for which the information is to be obtained must be included in the argument list. The system will return to the application program an indication of whether or not an off-line copy of the data file exists in its original data file format, whether an on-line copy exists in a system standard format and whether an off-line copy also exists in system standard format. Also, it will indicate in which system standard format or data file format a copy exists.

Four other commands, GETHEAD, PUTHEAD, GETHIST and PUTHIST, are available to read and write header records and processing history records, respectively, for data files in system standard format. Each of these commands must include the data identifier of the data file which is being accessed by the command. Additionally, the argument list must reference a work area which contains either the header record or history

record to be written for output or a work area into which the header or history record can be placed for input.

SECTION 4 - THE INTERACTIVE COMMAND LANGUAGE

4.1 Introduction to the Interactive Command Language

The Integrated Data Base Management System commands available to an interactive user (i.e., commands other than the operator commands) may be divided into five categories: data definition commands, data manipulation commands, administrative commands, utility commands, and file operations. This section describes the syntax and briefly discusses the function of these commands on a category-by-category basis. While nothing in the design of the Interactive Data Base Management System would preclude having every command described in this chapter made available for use by application programs, consideration of projected user requirements suggests that certain commands available for interactive users would be superfluous for application programs. Therefore, certain commands will be restricted for interactive use only. Moreover, since certain commands will be restricted to specific classes of users, the discussion of the function (or "semantics") of these commands will include a statement of the restrictions, if any, on the use of these commands.

It should be emphasized that this is not intended to be a substitute for a detailed users' manual, and the discussions of semantics are correspondingly brief. In particular, the reader will find more detailed discussions of file operations in Section 7, Data File Processing.

The notation used for describing the syntax of these interactive commands owes much to the CODASYL Data Base Task Group Report[5] and to the Backus Normal Form notation used to describe Algol 60[28]. The following rules apply:

- Key words are indicated with capital letters.
- Generic terms are indicated by lower case letters and are included in angled brackets (<,>). Generic terms

are replaced with appropriate values when the format is used. The use of subscripts on generic terms is not meant to imply different generic terms, but rather that the values used when replacing the generic terms will normally be different (see example at bottom of page).

Example: If 'BTS100' is a user-id and 'ALPHA' is a password then ENTER BTS100, ALPHA is an instance of the format ENTER <user-id>,<password>.

- Square brackets ([,]) indicate optional alternatives. At most one, but possibly none, of the alternatives may be present.

Example: MENU, MENU ON, and MENU OFF are valid instances of the format MENU [ON / OFF] .

- Braces ({,}) indicate mandatory alternatives. Precisely one alternative must be present.
- Vertical placement and vertical lines are both used to indicate alternatives.

Example: MENU [ON|OFF] is equivalent to MENU [ON / OFF] .

- An ellipsis (...) indicates that repetition is permitted. The portion of the format to be repeated is determined by the open bracket or brace ([or {) which matches the closed bracket or brace (] or }) immediately to the left of the ellipsis.

Example: If XYZ1, XYZ2, and XYZ3 are data file identifiers then LOAD XYZ1,XYZ2,XYZ3 is an instance of LOAD <file-id$_1$>[,<file-id$_2$>]... .

- The special symbol "::=" means "is defined to be". It is used to break up what would otherwise be a very complicated definition into simpler and easier to grasp parts.
- Other punctuation marks such as commas and asterisks must be present, as shown.
- Key words and generic items must be separated by blanks or punctuation marks when the command is entered into the system.

Example: ENTERBTS100,ALPHA is an unacceptable instance of the format ENTER <user-id>,<password>. However, both ENTER BTS100,ALPHA and ENTER BTS100, ALPHA are valid.

4.2 Utility Commands

Utility commands will perform a variety of necessary services for interactive users. All of these commands will be available to any interactive user without restriction, and certain of these commands shall furthermore be available to application programs as well. The seven utility commands will be:

ENTER - Connect the user to the Integrated Data Base Management System.

EXIT - Disconnect the user from the system.

ATTACH - Designate a particular data base for information processing.

DESCRIBE - Display a textual description of data entities, commands, user authorizations, or group memberships.

USE - Establish a one-character or two-character alias for a table.

MENU - Display a menu of interactive commands or toggle from full menu display mode to menu suppressed mode or vice versa.

PASSWORD - Change passwords.

4.2.1 ENTER

A user will connect to the Integrated Data Base Management System with the ENTER command, whose syntax is described below:

ENTER <user-id>,<password>

where the password must be correct for the indicated user before the system will process the command. The password will be selected by the user, and -- due to the manner of enciphering it when the password is stored internally -- not even the DBA will be able to learn the password except by communicating with the user or expending an inordinate amount of time and effort. This is intended to provide a user with a certain measure of security.

After successful execution of an ENTER command the user will always be attached to the Global data base.

4.2.2 EXIT

A user will exit from the Integrated Data Base Management System by keying in the single key word:

EXIT

When the user issues an EXIT command, any alias names established for tables via USE commands (see Section 4.2.4) will be erased and the contents of the user's Workspace Table will be deleted.

4.2.3 ATTACH

A user may leave one data base and begin processing data in another data base via the ATTACH command. The syntax of the ATTACH command will be:

ATTACH [TO] <data base name>

Subsequent commands which refer to tables or fields will be presumed to reference entities of the specified data base. However, an ATTACH will be accepted only if the user -- or some user group to which the user belongs -- has access rights to the specified data base.

Successful execution of an ATTACH command will not change any alias names established while the user was previously connected to some other data base. However, the user will not be permitted to reference those tables until re-attached to the data base where the aliases apply. Nor will successful execution of an ATTACH change the contents of the workspace table. Therefore, the workspace table can provide a convenient mechanism for transportation of data between different data bases.

4.2.4 USE

The syntax of the USE command will be:

USE <alias$_1$>[,<alias$_2$>]... FOR <table name>

where aliases will be one or two alphanumeric characters with the first character restricted to be alphabetic (e.g., S, SC, S3, etc.). This command will specify short aliases for tables, and the user may then substitute the alias for the table name in any command in which the table name is required. This will have two benefits:

(1) the number of keystrokes required to enter a command will be reduced, and
(2) at least one alias will be required when a table is cross referenced against itself in the performance of a retrieval using the relational calculus syntax*.

A user will be permitted to have more than one alias name on a single table, and the same alias name may be applied to different tables provided the tables reside in different data bases. However, any given alias can be used on at most one table in any given data base at any time. Thus, if a given alias is bound to a table in some data base and a user issues a new USE command binding that alias to a different table in the same data base, then the old binding will be overwritten by the new binding.

There will be one important restriction on the use of alias names. By convention, the name "W" will always refer to the user's "Worskpace" Table*. The user will not be required to make this binding formally (with a USE command) and may not redefine W.

* See the description of the SELECT command in Section 4.5.1.

Detaching from a data base will not alter the alias name bindings established by the user for that data base, and the same set of alias bindings will be in effect when the user reattaches to that data base. Upon EXIT from the system, however, all alias name bindings will be destroyed.

4.2.5 PASSWORD

Users may wish to alter their passwords for reasons of security, or the DBA may be called upon to reset the password of some user. The mechanism for changing passwords will be the PASSWORD command:

PASSWORD [FOR <user-id>]=<password>

Barring a system crash necessitating data base recovery and restart, the new password for the user will be in effect the next time the user issues an ENTER command.

Only the DBA will be allowed to use the PASSWORD command with the FOR clause. The DBA may find it necessary to issue such a command if a user forgets his password, or if it becomes necessary to deny access to the system temporarily for some user and the DBA does not wish to take the drastic step of issuing a REMOVE on that user.

The PASSWORD command will not be available to application programs.

4.2.6 MENU

The syntax of the MENU command will be:

When a user enters the Integrated Data Base Management System through an interactive remote terminal, the system will inquire whether he or she wishes to work in "full menu display" mode or "menu suppressed" mode, that is, whether the user wishes to have a menu of interactive commands displayed between transactions, or whether menu displays are to be inhibited.

If the user is working in full menu display mode then the only form of the MENU command which the system will accept will be MENU OFF, which will toggle the user into menu suppressed mode. If the user is working in menu suppressed mode, then he or she can issue a MENU ON command to toggle into full menu display mode, or the user will be permitted to input MENU with no qualifier to get a menu listed without switching out of menu suppressed mode.

4.2.7 DESCRIBE

DESCRIBE will be a multipurpose command used to pass information from the Integrated Data Base Management System back to a user. The syntax of a DESCRIBE command will be:

```
                  ⎧ ⎧DATABASE⎫ ⎧<entity name1>[,<entity name2>]...⎫ ⎫
                  ⎪ ⎨TABLE   ⎬ ⎨              *                 ⎬ ⎪
                  ⎪ ⎩FIELD   ⎭ ⎩WHERE <qualification>             ⎭ ⎪
DESCRIBE          ⎨ COMMAND {<command name>|TYPE=<category>}         ⎬
                  ⎪ GROUP <group name>                               ⎪
                  ⎩ RIGHTS [FOR <user-id1>[,<user-id2>]...]          ⎭
```

where the entity name is a data base name (if the command is DESCRIBE DATABASE), a table name (for DESCRIBE TABLE), or the name of a field in the Data Dictionary (for DESCRIBE FIELD). As may be inferred from the above syntactic description, a user will be able to request information about data bases, tables within a data base, the contents of a data dictionary, system commands, and user groups. Also, a user may inquire about the extent of his or her data access rights. Each of these six variants are described in greater detail below.

The DESCRIBE command is not available to application programs.

4.2.7.1 DESCRIBE DATABASE

The DESCRIBE DATABASE command will cause the Integrated Data Base Management System to output the following information for each data base specified by the user:

(1) data base name
(2) data base owner
(3) type (working vs. applications)

(4) creation date
(5) short textual description.

This is a minimal set of items of information, and the DBA may elect to define even more fields in a data base control block and, if so, these fields would surely also be listed. For example, the DBA may choose to define meaningful classes of data bases, so that the data base class would be output as well as the five items specified above.

The user will be able to specify one or more particular data bases by name, or the user can fetch a listing of all the data bases managed by the system by using the asterisk ("*") option. A third approach will be to specify a subset of the data bases through use of a WHERE clause and some qualification. The syntax of a WHERE clause and qualification are spelled out in greater detail in the subsection of this report devoted to the SELECT command but, basically, a qualification is a Boolean combination of predicates, and in this case the predicates will attach values to the fore-mentioned items of information in the data base control block. For example, a user may wish to see a description of all working data bases owned by BTS100, and do so with the command:

```
DESCRIBE DATABASE WHERE OWNER=BTS100 AND TYPE=WORKING
```

Other fields which might occur in a qualification predicate are creation date and class (if defined by the DBA).

It is important to note that a user will not have to be attached to any particular data base to issue a DESCRIBE DATABASE command, and that issuing a DESCRIBE DATABASE command will not transfer the user from his or her current data base.

4.2.7.2 DESCRIBE TABLE

A user will be able to retreive detailed descriptions of tables and data bases to which he or she is attached † by issuing a DESCRIBE TABLE command. For each table specified by the user the system will respond by listing:

(1) table name
(2) table owner
(3) creation date
(4) access control for READ (public, private, or restricted)
(5) access control for INSERT (public, private, or restricted)
(6) access control for UPDATE (public, private, or restricted)
(7) access control for DELETE (public, private, or restricted)
(8) a list of domains by name
(9) for each domain, the field in the data dictionary to which it corresponds and assertions, if any
(10) a list of search keys and superstructures.

The user will be able to specify a list of tables by name, or may ask for all tables in the data base with the asterisk option, or may use a WHERE clause and qualification. Fields available for use in predicates would include -- but not be limited to -- owner, creation date, access control status, and the names of one or more domains.

4.2.7.3 DESCRIBE FIELD

A user can examine the contents of the Data Dictionary for the data base to which he or she is attached †† by means of the DESCRIBE FIELD command. For each field specified by the user the system would list the following data:

---

† To maintain system security, system tables such as SYSDOM, SYSUSER, SYSCATL, and others will be invisible to the user and shall not be described to the user even if the user is attached to the Global Data Base.

†† Again, certain fields of the Global Data Dictionary would be blocked from user knowledge.

(1) field name
(2) type
(3) size
(4) units (if present).

Again, the user will be able to specify one or more particular fields by name, or may cause the entire Data Dictionary to be listed (with the asterisk option), or may use a WHERE clause and qualification to specify a subset of the Data Dictionary implicitly.

4.2.7.4 DESCRIBE COMMAND

The DESCRIBE command with the COMMAND option is similar to the "HELP" command of other interactive, user-friendly systems. The DESCRIBE COMMAND command will not be available through the system's Batch Command Reader.

The user will be able to name a specific command (e.g., DESCRIBE COMMAND INDEX) or may request details on all of the commands in a particular category (e.g., DESCRIBE COMMAND TYPE= UTILITY). The system will respond by giving the syntax and a brief description of the function for the command(s) specified by the user. The Integrated Data Base Management System will be selective in what it outputs -- there is obviously no need to tell the user about ENTER, for example, nor will the user be given any information on commands available for the DBA's use only. Thus, use of the USER clause and GROUP clause will not be explained if the user asks about DEFINE or REMOVE, nor will the system inform users about CATALOG, UNCATALOG, INCLUDE, EXCLUDE, or the GROUP clause for the DESCRIBE command. If it should happen that a user does request information about one of these restricted commands, the system will respond by stating that the command in question is restricted to use by the DBA only.

4.2.7.5 DESCRIBE RIGHTS

The RIGHTS clause of the DESCRIBE command will allow a user to discover what his or her data acess rights are. The system will respond to the DESCRIBE RIGHTS command by listing all the data access rights authorized to that user directly, then listing all the rights indirectly authorized to the user by group membership on a group-by-group basis. (A more thorough discussion of precisely what those rights are may be found in the subsection devoted to the GRANT command.)

Only the DBA will be permitted to use the DESCRIBE RIGHTS command with the FOR clause. This variation of the DESCRIBE RIGHTS command is designed to give the DBA the ability to determine the data access rights for users of the system.

4.2.7.6 DESCRIBE GROUP

The DBA will also be able to examine the membership of and access rights authorized to particular groups by the DESCRIBE GROUP command. The output will be a listing of the users belonging to the group, sorted by user-id, and a listing of the data authorizations granted to the group on the whole, sorted on table within data base.

4.3 Data Definition Commands

The six data definition commands will allow interactive users of the Integrated Data Base Management System to create and remove data bases, tables, and fields; to alter table layouts; to specify data validation tests; and to define access method superstructures on fields of a table to facilitate rapid data retrieval. The Data Base Administrator may also use data definition commands to introduce new users to the system, to remove inactive users, and to define user groups. The commands will be:

DEFINE - Identify new data bases, new fields, new tables, new users, new user groups, and data validation tests to the system.

REMOVE - Remove inactive users, obsolete user groups, unused fields, tables, or even whole data bases from the system.

EXPAND - Append one or more columns to an existing table.

INDEX - Establish hierarchical index superstructures on given fields or combinations of fields in a table.

INVERT - Create inverted file indices for a given field or combination of fields.

DROPINDEX - Delete index superstructures (both hierarchical and inverted) from specified tables.

Most of these commands will be restricted with respect to the list of users who may apply them in any given situation. These restrictions will be specified in greater detail in the following exposition.

None of these commands will be available through the Applications Program Interface, although, again, this is a philosophical point rather than a requirement of the system design or proposed implementation.

4.3.1 DEFINE

DEFINE is to be a multipurpose command used to introduce a variety of entities to the Integrated Data Base Management System. The syntax of a DEFINE command will be:

```
                 ⎧ DATABASE            <data base clause>       ⎫
                 ⎪ [DIRECTORY] TABLE   <table clause>           ⎪
DEFINE           ⎨ FIELD               <field clause>,          ⎬
                 ⎪ ASSERTION           <data validation clause> ⎪
                 ⎪ USER                <user clause>            ⎪
                 ⎩ GROUP               <group clause>           ⎭
```

To avoid the confusion which can be caused by overwhelming detail, each of the six variants is discussed in a separate subsection, below.

4.3.1.1 DEFINE DATABASE

The syntax of a DEFINE command with the DATABASE clause will be:

DEFINE DATABASE <data base name>

where the specified data base name must not duplicate the name of some already-existing data base. Any user will be able to define a data base, thereby becoming its owner. A user who issues a DEFINE DATABASE command will implicitly become attached to the new data base for the purpose of further processing.

4.3.1.2 DEFINE TABLE

The syntax for a DEFINE command with the TABLE clause will

be:

DEFINE [DIRECTORY] TABLE <table name>(<field def'n$_1$>[,<field def'n$_2$>]...)

where

$$\text{<field def'n>} ::= \left\{ \begin{array}{l} \text{<field name>} \\ \text{<field name}_1\text{>=<field name}_2\text{>} \end{array} \right\}$$

If the field definition is in the first form, a single field name, then that name must be in the data bases's Data Dictionary. If the field definition is in the second form then the second field name (on the right hand side of the equals sign) must be in the Data Dictionary while the first field name (on the left side of the equals sign) should not be in the Data Dictionary. Field definitions in the second form will allow the user to attach his or her own names to pre-defined fields (e.g., X=LON, Y=LAT). This will be an absolutely necessary feature for the case when two columns of the table span the same domain of values and are defined in terms of the same field (e.g., START-DATE=DATE, END-DATE=DATE).

Although the order of fields within a table is not significant for information retrieval purposes, the order in which the field definitions are listed in the table clause will define the internal sequence in which they will be stored by the Integrated Data Base Managements System's Physical Interface.

A user may not DEFINE a new table in a data base unless attached to that data base. The right to DEFINE tables will be limited to the DBA, the owner of the data base, and such users as the data base owner permits (see the GRANT command, in a later subsection). The user who defines the new table will become its owner. To avoid conflict with alias names (see

Section 4.2.4) a table name must be three or more characters long.

Only the DBA may DEFINE a DIRECTORY TABLE, and a DIRECTORY TABLE can only be created in the Global Data Base. Such tables will differ from normal tables in that they will automatically become a part of the special virtual table, SYSDIR, which references data files cataloged in the non-relational portion of the system.

4.3.1.3 DEFINE FIELD

The mechanism for entering data field names into a data base's Data Dictionary will be the DEFINE command with the FIELD clause. Its syntax will be:

```
DEFINE FIELD <field name>[TYPE=]<type>,[SIZE=]<size>[,[UNITS=]<units>]
```

The order of type, size, and units (if present) will not be significant if the key words TYPE, SIZE, and UNITS, respectively, are used, but when the key words are not used then they must be in the sequence specified above. All key words should be present or none should be used.

The type parameter may take on any one of five values: REAL, ALPHANUMERIC, INTEGER, LOGICAL, or DECIMAL* (the system will accept any reasonable abbreviation beginning with R, A, I, L, or D, respectively). The size parameter will indicate the size of this field in bytes. An integer field size must not exceed the number of bytes per word on the machine where this system is implemented, and a floating point field must be precisely one or two times the word size. The Integrated Data Base

*Short for "packed decimal" and available only if supported by the machine on which the Integrated Data Base Management System is implemented.

Management System will retain a table of acceptable unit names and abbreviations; use of a unit name (or abbreviation) for the units parameter which is not in that table will cause the command to be rejected.

Only the DBA or the data base's owner will be allowed to DEFINE fields.

4.3.1.4 DEFINE USER

The Data Base Administrator will introduce new users to the Integrated Data Base Management System through the DEFINE command with the USER clause. The syntax is shown below:

DEFINE USER <user-id>,<password>[,GROUPS=<group$_1$>[,<group$_2$>]...]

This will create a new entry in the SYSUSER system table for a user with the given user-id and password, and will include the new user in the indicated groups.

4.3.1.5 DEFINE GROUP

One useful feature of the Integrated Data Base Management System will be the ability of the DBA to establish "user groups". These groups will exist for purposes of authorizing data access rights to sets of users engaged in the same or similar projects without specifically enumerating that list of users. The mechanism for establishing a group will be the DEFINE command with the GROUP clause, and its syntax will be:

DEFINE GROUP <group name>

Note that this command will merely introduce a group to the system and will not assign any users to that group.

Users will be included in a group by a DEFINE USER command or an INCLUDE command (described in a later subsection).

4.3.1.6 DEFINE ASSERTION

An important function of any data base management system is protection of the quality of the data it manages. The Integrated Data Base Management System will make provision for limited, automatic data validation tests to be specified by authorized users to maintain the semantic integrity of its data. The normal mechanism in a relational data base management system for defining and applying these data validation tests is the "integrity assertion" -- a statement about the data in a table which is expected to be true unless one or more records is incorrect. A more thorough discussion of the theory and use of integrity assertions may be found in Appendix B. These data validation tests will be established by the DEFINE command with the ASSERTION clause, and its syntax will be:

DEFINE [SOFT] ASSERTION <assertion name> ON <table name>:<predicate>

where the predicate will be a true/false test of the form:

$$\text{<field name}_1\text{><relationship>}\left\{\begin{array}{l}\text{OLD <field name}_1\text{>}\\ \text{<field name}_2\text{>}\end{array}\right\}\left[\left\{\begin{array}{l}*\\ /\end{array}\right\}C_2\right]\left[\left\{\begin{array}{l}+\\ -\end{array}\right\}C_3\right]\Bigg\}^{C_1}$$

The c_i's represent arbitrary floating point or integer constants and the relationship tests will be indicated by the keywords LT, LE, EQ, GE, GT, and NE, or by the signs <,=, and >. In other words, a predicate will test the relationship of the values of a field against a constant (e.g., TEMP>-273.16) or against a function of another field (e.g., START-DATE LE END-DATE),

or a function of its former value during an update (e.g., AGE GT OLD AGE). The complexity of a function will be limited to:

(1) the field name itself (preceded by the keyword "OLD" if the same field as on the left side of the predicate),
(2) a field times or divided by a constant,
(3) a field plus or minus a constant, or
(4) a field times or divided by one constant and plus or minus another constant.

A "soft" assertion will not block a transaction from being processed, but will merely output a warning message when an update or insertion causes it to be violated. The default will be to block the particular transaction which violated the assertion, although the system shall process other updates or insertions which are valid. If an update is blocked by a violated data validation assertion then the system will print out (1) the (unupdated) record, (2) the field which was to have been changed, (3) the new value that field would have had, and (4) the assertion which was violated. When an insertion is blocked by a violated assertion then the Integrated Data Base Management System will list (1) the rejected record and (2) the assertion(s) which were violated.

It is anticipated that assertions will normally be defined for a table at the time the table is defined and before any data has been inserted. However, the Integrated Data Base Management System will accept new assertions being established on nonempty tables. When that happens, the system will list the records already in the table which violate the assertion and give the user the choice between keeping the assertion (and thereby deleting the records) or keeping the records (and thereby implicitly removing the assertion). If there are no records already in the table which violate the assertion then the assertion will always be accepted.

The right to establish an assertion will be limited to the DBA, the data base owner, and the owner of the table on which the assertion is to be established.

4.3.2 REMOVE

Anything which can be identified to the Integrated Data Base Management System by a DEFINE command may be removed from the system by a REMOVE command. The syntax of a REMOVE command will be:

```
                 ⎧ DATABASE   <data base name>                                  ⎫
                 ⎪ TABLE      <table name>                                      ⎪
                 ⎪ FIELD      <field name>                                      ⎪
REMOVE           ⎨ ASSERTION  <assertion name> FROM  <table name>               ⎬
                 ⎪ USER       <user-id>[,<user-id>] ...                         ⎪
                 ⎪ GROUP      <group name>[,<group name>]...                     ⎪
                 ⎩                                                              ⎭
```

The ramifications of each of the six variations of the REMOVE command -- corresponding to the six variations of DEFINE -- are discussed in greater detail below.

4.3.2.1 REMOVE DATABASE

Only the DBA or the owner of a data base will be allowed to REMOVE it from the system, and that individual must be attached to the data base before issuing the command. After a REMOVE DATABASE has been issued, no new user will be permitted to access that data base, but users already attached to the data base will be permitted to complete their information processing before the data base is destroyed.

4.3.2.2 REMOVE TABLE

A user must be attached to the data base which contains the table to be removed before a REMOVE TABLE command will be accepted by the system and, moreover, the user who issues the REMOVE TABLE command must be either the DBA, the data base owner, or the owner of the table.

As with the REMOVE DATABASE command, once a REMOVE TABLE command has been issued and accepted no new users will be permitted to access the table, although transactions already accessing the table will be permitted to complete before the table is destroyed.

4.3.2.3 REMOVE FIELD

A REMOVE FIELD command will delete a field name from a data base's Data Dictionary. A REMOVE FIELD command will be accepted only if issued by the DBA or the data base owner while attached to the appropriate data base, and even then the system will refuse to execute the command unless no table in the data base includes that field.

4.3.2.4 REMOVE ASSERTION

Only the individuals who can DEFINE an assertion will be allowed to REMOVE it (i.e., the DBA, the data base owner, or the table owner) and they must be attached to the data base containing the table to do so. Assertion removal is postponed if there is an insertion or update in progress at the time a REMOVE ASSERTION command is issued, to prevent anomalies from arising when some portion of the transaction is rejected for violating the assertion in question while other portions of the transaction (executed after the REMOVE ASSERTION has been issued) are accepted despite violating that data validation check.

4.3.2.5 REMOVE USER

The Data Base Administrator (and only the DBA) will be able to REMOVE a user from the system with a REMOVE USER command. If the user in question is currently active at the time the REMOVE USER is issued, then the effect of that command will be delayed until he or she performs an EXIT from

the system. (In such a case, the DBA will be notified by a printed message.)

4.3.2.6 REMOVE GROUP

Only the DBA will be allowed to establish a user group, and only the DBA will be permitted to issue a REMOVE GROUP command. It should be noted that the REMOVE GROUP command will merely dissolve the group(s); it will not cause the removal of any member of that group from the system.

Performance of a REMOVE GROUP will take place immediately upon receipt of the command by the system. However, if any user should be accessing a table at the time the REMOVE GROUP is issued and that user received the authorization to access that table or its data base only through membership in the group being removed, then that transaction will be permitted to go to completion.

4.3.3 EXPAND

The EXPAND command will permit a qualified user to add one or more fields to an existing table. The syntax of an EXPAND command is shown below:

EXPAND <table name> [BY] (<field def'n_1>[,<field def'n_2>]...)

where the fields are defined as described in the DEFINE TABLE subsection. For storage purposes, these new fields will be added to the end of the table. Only the DBA, the data base owner, and the owner of the table will have the right to EXPAND a table, and expansion will be the only alteration to the layout of a table (other than total removal) supported by the system.

After an EXPAND is executed on a table, the records already stored in the table will be treated as having "null" values for the new fields. Unlike most data base management systems, however, when the Integrated Data Base Management System performs an EXPAND it will not cause any immediate alteration of existing records. Consequently, an EXPAND will be quite inexpensive to perform in this system.

4.3.4 Generating Data Access Superstructures: INDEX and INVERT

Access method superstructures to facilitate rapid and efficient data retrieval may be placed on tables by the INDEX and INVERT commands, which have the syntax:

$$\left\{\begin{array}{l}\text{INVERT} \\ \text{INDEX [UNIQUE]}\end{array}\right\} \text{<table name> [ON] <key}_1\text{>[,<key}_2\text{>]...}$$

where

$$\text{<key> ::=}\left\{\begin{array}{l}\text{<field name>} \\ \text{<key name> = (<field name}_1\text{>[,<field name}_2\text{>]...)}\end{array}\right\}$$

That is, a superstructure key may be a single given field of the table or may be formed by concatenating more than one field into a combined key.

The INDEX command will establish hierarchical index structures* on the specified fields or combinations of fields and INVERT will establish inverted file indices on the specified fields or combinations of fields. Hierarchical indices are useful when the values of the search keys are unique, or nearly unique, and inverted file indices facilitate rapid data access when a given value of a search key defines a set of records. Since the type of index superstructures created by INVERT and INDEX are useful in such different contexts, the system will not support both a hierarchical and an inverted index on the same field or on identical combinations of fields.

If a sequence of fields is combined to make a single search key, then the same index used for that key also facilitates rapid access of any leading subsequence of fields. For

*specifically, B-trees.

example, if a table T has the fields A, B, and C, then the index established by the command

```
INDEX T ON K=(A,B,C)
```

will also serve to effect rapid access for A alone or for A and B combined. Therefore, the Integrated Data Base Management System will ignore requests for identical types of indices on leading subsequences of combined field search keys, although it shall accept requests to establish different types of indices for leading subsequences and any type of index request for non-leading subsequences. Hence, the following commands are compatible with the INDEX command above:

```
INVERT T ON A,K1=(A,B)
INDEX T ON B,C,K2=(B,C)
```

The optional UNIQUE qualifier on the INDEX command will mean that duplicate values of the specified search key(s) are not permitted. If a transaction such as an insertion or update would cause this requirement to be violated, then that transaction will be blocked and an appropriate error message will be output to the user.

Although access method superstructures will normally be generated at the time the table is created, the Integrated Data Base Management System will support the establishment of both types of indices on pre-existing, non-empty tables. In the case where an INDEX UNIQUE command is issued against a table with duplicate values for that search key the system will respond by listing the erroneous records and the user may elect to delete the duplicates or to rescind the command.

As with EXPAND, use of the INDEX and INVERT commands will

be limited to the DBA, the data base owner, and the owner of the table, and the individual issuing the command must be attached to the data base which contains the table.

4.3.5 DROPINDEX

The rapid access superstructures created through the INDEX and INVERT commands may be dropped from a table through the use of the DROPINDEX command. The syntax of a DROPINDEX command will be:

DROPINDEX <key name$_1$>[,<key name$_2$>]...FROM <table name>

where the key names may be single fields or may be the key names attached to combinations of fields when the index was created. Since the same field -- or key -- may not have both a hierarchical superstructure and an inverted file superstructure simultaneously, the DROPINDEX command can be, and shall be, used to drop both kinds of indices.

Only someone with the authorization to generate a superstructure (i.e., the DBA, the data base owner, and the table owner) will be allowed to issue a DROPINDEX command, and only when attached to the appropriate data base.

4.4 Administrative Commands

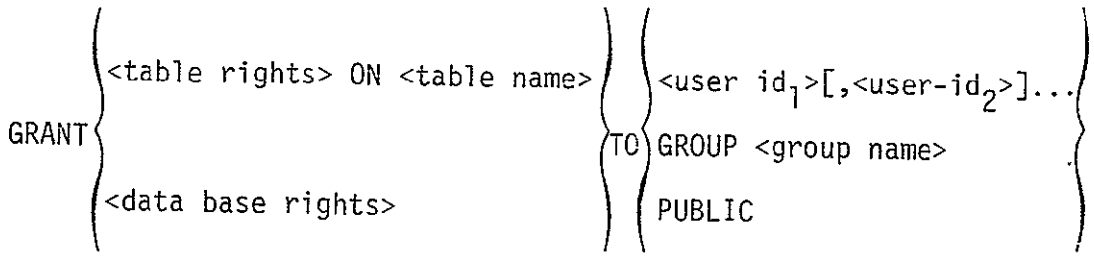
The four administrative commands will provide users in an administrative position (i.e., the DBA and owners of data bases and tables) with control over the state and/or accessibility of entities under their purview. The administrative commands will be:

GRANT	-	Authorize users or user groups the right to access and/or modify data.
REVOKE	-	Cancel previously granted rights.
INCLUDE	-	Add a user to a user group.
EXCLUDE	-	Remove a user from a user group.

None of these commands will be available to an application program.

4.4.1 GRANT

The syntax of the GRANT command is depicted below:



When a new table or data base is created by a DEFINE command the right to access its contents will be strictly limited to its owner (that is, its creator) and to the DBA and (in the case of the creation of a new table) the data base owner. The GRANT command is designed to make it possible for the DBA or the owner of the data base or table to make its contents available to a wider circle of users. The authorization may be extended to all users of the Integrated Data Base Management System through the key word "PUBLIC", or the authorization may be restricted to a particular collection of users, either named explicitly or identified implicitly through membership in some user group. Since different key words will be used to distinguish rights associated with tables from rights associated with data bases, these two topics are discussed separately below.

4.4.1.1 Granting Rights on Tables

The following key words will be associated with table rights: READ, INSERT, UPDATE, and DELETE which will have their obvious interpretations, plus ALL RIGHTS, which will refer to all four rights simultaneously. It will be possible for the owner of the table or the DBA to GRANT any combination of the four access rights (listed in any order), or ALL RIGHTS, to the whole community of data bases users,

or any particular user group, or any explicitly-listed users. Neither the table's owner nor the DBA will need to GRANT rights to himself since the Integrated Data Base Management System will always assume that the table owner has all four access rights on his own table, while the DBA will always have all rights to everything.

The system will reject a SELECT command from any user unless that user has READ rights on all tables referenced by the SELECT. Nor will the system accept an INSERT, UPDATE, or DELETE command from a user unless he or she has the appropriate right on the table being edited, plus READ rights on all other table referenced by the command. The user may hold these rights explicitly, by having been named in a GRANT command on that table, or the user may hold these rights implicitly, by belonging to the appropriate user group or if the rights are PUBLIC.

Although ownership of a table is a privilege, rather than a right, the GRANT command may also be used by the DBA to change the ownership of a table. The syntax for a GRANT of OWNERSHIP will be:

GRANT OWNERSHIP ON <table name> TO <user-id>

By issuing a GRANT OWNERSHIP command, the DBA will be taking ownership of the table away from the former owner and assigning it to a new owner. One side effect of this command will be that the former owner will no longer have full access rights to the data in the table unless (1) he belongs to one or more groups which have been granted all access rights on the table or (2) he is or has previously been explicitly granted ALL RIGHTS on the table. In other words, a former owner of a table will become just another user as far as that table is concerned once the privilege of ownership has been removed.

4.4.1.2 Granting Rights on Data Bases

There will be two rights associated with data bases: ACCESS and MODIFY. As with tables, it will be possible for the data base owner or the DBA to issue either right separately or to issue them jointly with the "ALL RIGHTS" key word, and these rights may be authorized to the entire user community, to a specific user group, or explicitly to individual users.

A user will not be permitted to ATTACH to a data base without having ACCESS authorization for that data base. Once attached, he or she will be forbidden to issue a DEFINE*, INSERT, UPDATE, or DELETE command unless authorized the right to MODIFY the contents of the data base. Note that a user may well have READ rights on some table in the data base and yet be blocked from retrieving the data in that table by not having been authorized to ACCESS the data base. Similarly, some particular user might have, for example, UPDATE rights on a table in a data base and yet would not be able to perform an update on the table if he or she lacks MODIFY rights on the data base itself.

Just as the DBA will be able to transfer the privilege of ownership of a table, so the DBA will have the ability to transfer the privilege of ownership of an entire data base to another user. The syntax for a GRANT of data base ownership will be:

GRANT OWNERSHIP TO <user-id>

The DBA will be required to be attached to the data base

---

*Except for DEFINE DATABASE. It should be noted that by logical extension this also forbids all other commands in the data definition and administrative categories.

in question before changing its ownership, and hence the data base need not be named in the GRANT OWNERSHIP command. Again, the status of the former owner of the data base will be no different from the status of any other user after the privilege of ownership has been granted to the new owner, and the former owner may not even retain ACCESS or MODIFY rights on the data base unless implicitly (via group membership) or explicitly granted those rights. However, the former owner of a data base will retain ownership of any tables in the data base for which he or she is the listed owner.

4.4.2 REVOKE

Access authorizations which have been granted may be revoked. The syntax for a REVOKE command is:

```
                  ⎧<table rights> ON <table>⎫                ⎧ PUBLIC                                          ⎫
REVOKE            ⎨                         ⎬ [FROM         ⎨ <user-id1>[,<user-id2>]...                      ⎬ ]
                  ⎩<data base rights>       ⎭                ⎩ GROUP <group name>                              ⎭
```

where the table rights are READ, INSERT, UPDATE, and DELETE and the data base rights are ACCESS and MODIFY (i.e., the same rights which may be authorized to a user by a GRANT command). Since ownerless tables and data bases are not permitted, there is no REVOKE OWNERSHIP command corresponding to the GRANT OWNERSHIP command -- the GRANT of OWNERSHIP implicitly revokes the former owner's ownership.

Rights may only be revoked from users or user groups which have explicitly been given those rights. For example, suppose users A, B, and C constitute group X. It is not permissible to GRANT rights to users A, B, and C (explicitly) and then to REVOKE them from Group X or to GRANT to group X and then REVOKE from A, B, and C (listed explicitly), even though the collection of users which constitutes group X and the list of users A, B, and C are one and the same. Nor is it permissible to GRANT some rights to group X and then REVOKE them from user A unless those rights were explicitly granted to user A, as well. Similarly, it is not permissible to GRANT some rights on a table or data base to PUBLIC and then REVOKE those rights from a specific user unless the rights were granted explicitly to that user.

If a REVOKE command is issued without specifying from whom the rights named in the command are to be taken (i.e., no FROM

clause), then those rights named will revert to "owner only" status and none but the table owner and the DBA will have those rights on the table.

4.4.3 INCLUDE

It will be the responsibility of the Data Base Administrator to assign users to user groups. This will be accomplished by the INCLUDE command, whose syntax will be:

INCLUDE <user-id$_1$>[,<user-id$_2$>]...IN [GROUP] <group name>

The group named in the INCLUDE command will have to have been previously created by a DEFINE command with the GROUP clause. Each of the users included in the group will be implicitly authorized all data access rights explicitly authorized to the entire group. A given user may be included in more than one user group, but the system will regard the user's rights granted through the group membership as belonging to the user and not just to the group. That is, if group X has READ rights on table S but not table T and group Y has READ rights on T but not S , a user belonging to both groups may simultaneously read both tables, even though neither group's set of rights are sufficient alone for the transaction.

4.4.4 EXCLUDE

Users will be removed from user groups by the EXCLUDE command. The syntax of an EXCLUDE command will be:

EXCLUDE <user-id$_1$>[,<user-id$_2$>]...FROM [GROUP] <group name>

If it happens that a user being removed from a group is concurrently accessing some data where his or her authorization to access that data comes solely from membership in the group, then the system will permit the transaction to complete itself before detaching the user from the group.

4.5 Data Manipulation Commands

The six commands in the data manipulation category will provide a user with the ability to retrieve information from tables, to edit a table, and to view the contents of tables and the results of data retrievals. This category of commands, taken together, will constitute the data sub-language (or, in relational data model jargon, the "query language") for the system. These data manipulation commands are:

SELECT - Retrieve data from one or more tables.

INSERT - Add new records to an existing table, updating indices as needed.

UPDATE - Edit one or more existing records in a given table, updating indices as needed.

DELETE - Remove one or more records from a given table, updating indices as needed.

DISPLAY - List the contents of a table or the results of a retrieval on a user's terminal.

PRINT - List the contents of a table or the results of a retrieval on a line printer.

None of these commands will be restricted, except insofar as the data base to which the user is attached must contain all tables referenced by his command and the user must have authorization to perform whatever operation on the tables he or she requests. All of these commands (except DISPLAY, for obvious reasons) will be available for use by application programs.

4.5.1 SELECT

SELECT is perhaps the most important command in the entire Integrated Data Base Management System interactive command language, as it is the mechanism by which data is retrieved and cross referenced from one or more tables in a given data base. There will be <u>two</u> syntaxes for the SELECT command, as depicted below:

$$\text{SELECT} \left\{ \begin{array}{l} \text{[FROM <table list>]<template}_1\text{>[;<template}_2\text{>]...} \\ \\ \text{(<target list>) WHERE <qualification>} \end{array} \right\} \#$$

The first syntax is based on the Query-by-Example retrieval language of Zloff[40] and the second is founded on the relational calculus-based Quel language devised by Stonebraker, et. al.[37] The first non-blank character after the keyword "SELECT" will tell the command interpretor whether to expect the Query-by-Example syntax (if it is alphabetic) or whether to expect the relational calculus syntax (if that character is a left (open) parenthesis). Since a user may be constrained to issue one or more carriage returns while entering a single SELECT command into the system, a special character ("#") will be used to terminate the command.

The results of a SELECT retrieval will be placed in the user's special "workspace table" (always indicated by its alias, W). Before a SELECT command will be accepted from a user, he or she must be attached to the appropriate data base and have READ authorization for every table referenced by the command. Upon completion of a SELECT command the system will output the number of records instantiated in W as a result of the SELECT, plus an explanation of what went wrong if no records are retrieved. This explanation will be

in the form of a status word if the SELECT was input from an application program.

The sample table layouts depicted in Figure 4-1 will be used to illustrate the concepts and syntax of Query-by-Example and relational calculus.

Since they merit considerable attention in their own right, the operation of the Workspace Table and the syntax of a template, a target list, and a qualification are dealt with in greater detail below.

4.5.1.1 Query-by-Example Syntax

Query-by-Example is designed for interactive users working in a conversational mode. The user requests a table by name, the system responds by displaying a skeleton of the table (i.e., table name, column headings, column outlines), and the user provides a query "template" by filling in the appropriate rows of the sample table with an example answer. It is not possible to describe "templates" for interactive Query-by-Example since much of what the user enters must be based on interaction with the system. In each example response input by the user, there will be two types of entries:

(1) "example elements", which - in this system - will be preceded by an asterisk†, and
(2) "constant elements", which are not starred.

The example elements represent hypothetical answers, but the system will treat them as variable names and it would not be wrong for the user to input a variable name of his or her choosing that bears no relationship to a typical data item in the corresponding field. Example elements for

†A reader who is familiar with Query-by-Example as published by Zloof will notice a number of minor differences between that syntax and this. The changes are in part due to data characteristics and in part designed to accomodate non-graphics terminals.

IMAGE-DATE (alias: D)

IMAGE-ID	DATE	TIME	TOP-LAT	BOTTOM-LAT	LEFT-LON	RIGHT-LO

EVENT (alias: E)

NAME	CLASS	START-DATE	END-DATE	START-LAT	START-LON	END-LAT	END-LO

Figure 4-1: Sample Tables

which the user is interested in seeing the results should have an "S" for ("SELECT") in front of their asterisk. Figure 4-2a shows how a Query-by-Example style query might be input for names and dates of all hurricanes in the EVENT table and figure 4-2b illustrates the similar query for hurricanes which occurred in 1977 (dates will be presumed to be day, month, and year in DDMMYY form). Note that in figure 4-2b it is necessary to use comparison operators to qualify the constant elements.

A user may key in multiple rows (hence the need for the special end-of-query mark, "#", instead of a carriage return to terminate a retrieval) to form Boolean sums or products of simpler queries. Use of the same example elements in the rows represent an AND and different example elements represent a Boolean OR. Figure 4-3 illustrates an alternative formulation for the query in figure 4-2b, using an AND operation††. Figure 4-4 shows a retrieval involving an OR.

Not every example element need be selected out of a table. Another use for example elements is to cross reference two tables over fields whose values are derived from a common domain. Figure 4-5 illustrates this use (note the use of a semi-colon to inform the system when the user has completed the input of sample queries for one table).

Finally, Query-by-Example will provide a number of standard "aggregate functions" whose values are computed over an entire column of a table. These functions will be MIN, MAX, AVG, SUM, COUNT, and SDEV (for standard deviation). They will be preceded by an asterisk and placed after the S (if present) but before the example element. The range of each aggregate function will be the entire column to which

---

†† Mathematically speaking the two queries are different, but practically speaking, given the extent of the hurricane season, they are equivalent.

QUERY 1: Retrieve name and date for all hurricanes.

SELECT FROM E

EVENT

NAME	CLASS	START-DATE	END-DATE	START-LAT	START-LON	END-LAT	END-L
S*AGNES	HURRICANE	S*010772	S*010772				
#							

Figure 4-2a: Sample Retrieval

QUERY 2: Retrieve names of all hurricanes in 1977.

SELECT FROM E

EVENT

NAME	CLASS	START-DATE	END-DATE	START-LAT	START-LON	END-LAT	END-L
S*AGNES	HURRICANE	>010177	<311277				
#							

Figure 4-2b: Sample Retrieval Using Comparison Operators

QUERY 2 (again): Retrieve names of all hurricanes in 1977.

```
SELECT FROM E
```

EVENT

NAME	CLASS	START-DATE	END-DATE	START-LAT	START-LON	END-LAT	END-LON
S*AGNES	HURRICANE	>010177					
S*AGNES	HURRICANE	<311277					
#							

Figure 4-3: Sample Retrieval Using AND

QUERY 3: Retrieve names and starting locations of all hurricanes or tropical storms since 1976

```
SELECT FROM E
```

EVENT

NAME	CLASS	START-DATE	END-DATE	START-LAT	START-LON	END-LAT	END-LON
S*AGNES	HURRICANE	>311276		S*X1	S*Y1		
S*BRUCE	TROP-STORM	>311276		S*X2	S*Y2		
#							

Figure 4-4: Sample Retrieval Using OR

QUERY 4: Retrieve data file id's for all images which are likely to show the formation of Hurricane Agnes

```
SELECT FROM E,D
```

EVENT

NAME	CLASS	START-DATE	END-DATE	START-LAT	START-LON	END-LAT	END-I
AGNES	HURRICANE	*010072		*X	*Y		

IMAGE-DATA

DID	DATE	TIME	TOP-LAT	BOTTOM-LAT	LEFT-LON	RIGHT-LON
S*XYZ #	*010772		>*X	<*X	<*Y	>*Y

Figure 4-5: Sample Retrieval With Cross Referencing

it is applied, counting duplicate values if necessary. The user will be able to restrict the function to apply to unique data items by inserting the qualifier "*UNIQUE" between the function name and the example element. Figure 4-6 illustrates the use of the COUNT function.

The templates in a Query-by-Example retrieval may be linearized so that the necessity for system output to the user may be eliminated. This will be necessary when the SELECT command is issued via the Batch Command Reader facility or from an application program, and may be useful when a very knowledgable user is constrained to work with a slow terminal or when operating system response time is slow. The system will expect the linearized Query-by-Example mode if the first non-blank character after the key word "SELECT" is alphabetic, but the key word "FROM" is not present. In linearized Query-by-Example each template will have the form:

<template>::=<table name>(<row$_1$>)[<row$_2$>)]...

where each row represents a row the user would have entered in the interactive Query-by-Example syntax. The syntax of rows is defined to be:

<row>::=<column entry$_1$>[,<column entry$_2$>]...

<column entry>::=<column name><operator><value>

The value part, of course, will correspond to whatever the user would have keyed in for that column in the interactive version of the query, except that column entries with null value fields should not be present. The relational operator will normally be an equals sign, unless the ">" or "<" comparison operators would have been used in the query. Column entries need not be in order since they are explicitly

QUERY 5: Count the number of hurricanes in the data base

```
SELECT FROM E

EVENT
```

NAME	CLASS	...
S*COUNT*AGNES	HURRICANES	

Figure 4-6: Sample Retrieval Illustrating the COUNT Function

identified by name.

Figure 4-7 illustrates sample queries 1 through 5 using the linearized Query-by-Example notation.

4.5.1.2 Relational Calculus Syntax

As stated earlier, the syntax of a query using the relational calculus will be:

SELECT (<target list>) WHERE <qualification> #

The syntax for an entry in the target list is illustrated below:

<target list entry>::={ <data field name> | <result field name>=<function> }

where the function may be a data field, an aggregate function of a data field (e.g., MIN, MAX, AVG), or an arithmetic combination of data fields and/or aggregate functions. The data fields named in a target list entry must appear in one or more tables in the data base. To avoid possible ambiguity, a data field name must be qualified by the name (or alias) of the table to which it belongs. Consequently the syntax of a data field name used in a relational calculus query will be:

<data field name>::=<table name>.<field name>

The names of result fields need not be in the data base's Data Dictionary, as they will only exist with respect to the user's Workspace Table, W. Moreover, the system will compute type, size, and unit parameters for the result fields named in the

QUERY 1: Retrieve name and date for all hurricanes.

```
SELECT E(NAME = S*AGNES, CLASS = HURRICANE,
        START-DATE = S*010772, END-DATE = S*010772)#
```

QUERY 2: Retrieve names of all hurricanes in 1977.

```
SELECT E(NAME = S*AGNES, CLASS = HURRICANE, START-DATE > 010177)
       (NAME = S*AGNES, CLASS = HURRICANE, START-DATE < 311277)#
```

QUERY 3: Retrieve names and starting locations of all hurricanes or topical storms since 1976.

```
SELECT E(NAME = S*AGNES, CLASS = HURRICANE, START-DATE > 311276,
        START-LAT = S*X1, START-LON = S*Y1)
       (NAME = S*BRUCE, CLASS = TROP-STORM, START-DATE > 311276,
        START-LAT = S*X2, START-LON = S*Y2) #
```

QUERY 4: Retrieve data file id's for all images which are likely to show the formation of Hurricane Agnes.

```
SELECT E(CLASS = HURRICANE, NAME = AGNES, START-DATE = *010772,
        START-LAT = *X, START-LON = *Y);
      D(DATE = *010772, TOP-LAT > *X, BOTTOM-LAT < *X,
        LEFT-LON < *Y, RIGHT-LON > *Y) #
```

QUERY 5: Count the number of hurricanes in the data base.

```
SELECT E(NAME = S*COUNT*AGNES, CLASS = HURRICANE) #
```

Figure 4-7: Sample Linearized Retrievals.

target list based on the data fields and mathematical transformations in the functions which define the result fields, so that there will be no requirement for the user to define these fields in advance.

The qualification will be a Boolean combination of true/false predicates, and the syntax of the Boolean expression will be:

$$\text{<Boolean exp>::=}\left\{\begin{array}{l}\text{(<Boolean exp>)}\\ \text{<predicate>}\\ \text{NOT<Boolean exp>}\\ \text{<Boolean exp>}\left\{\begin{array}{l}\text{AND}\\ \text{OR}\end{array}\right\}\text{<Boolean exp>}\end{array}\right\}$$

while the syntax of a predicate will be:

<predicate>::=<data field name><operator><value>

The valid operators will be the six key words LT, LE, EQ, GE, GT, and NE, plus the characters >,=, and <. A value may be a constant of the appropriate type, another data field, or a function of another data field.

Figure 4-8 depicts sample data retrievals using the relational calculus syntax.

As mentioned above, both the right-hand side of a target list entry and the right-hand side of a qualification predicate may include arithmetic functions of fields (e.g., AREA= D.DELTAX*D.DELTAY) and/or library functions. The Integrated Data Base Management System will provide both aggregate functions defined on whole columns of a table and non-aggregate functions defined with respect to single data items. The aggregate

QUERY 6: Retrieve the names of all tropical storms since 1976.

```
SELECT (E.NAME) WHERE E.START-DATE GE 010177
                AND E.CLASS = TROP-STORM #
```

QUERY 7: Retrieve name and formation date for all hurricanes or tropical storms formed in the northern hemisphere since 1976.

```
SELECT (E.NAME, FORMATION=E.START-DATE)
 WHERE (E.CLASS = TROP-STORM OR E.CLASS = HURRICANE)
   AND E.START-LAT > 0 #
```

QUERY 8: Retrieve data file id's for all images which may cover the formation of hurricane Alice.

```
SELECT (D.DID) WHERE D.TOP-LAT GE E.START-LAT
                 AND D.BOTTOM-LAT LE E.START-LAT
                 AND D.LEFT-LON GE E.START-LON
                 AND D.RIGHT-LON LE E.START-LON
                 AND D.DATE = E.START-DATE
                 AND E.CLASS = HURRICANE
                 AND E.NAME = ALICE#
```

Figure 4-8: Sample Retrievals Using Relational Calculus Syntax

functions which will be provided are MIN, MAX, SUM, AVG, COUNT, and SDEV. Aggregate functions may not be nested. The non-aggregates shall include the standard Fortran library functions such as SQRT, SIN, COS, ATAN, ALOG, and EXP (but not MIN, MAX, AMIN, or AMAX, to avoid confusion). Type-conversion Fortran functions (e.g., IFIX, FLOAT, DBLE, INT, etc) will not be supported -- any necessary data type conversions will be handled automatically and transparently by the system itself. In addition, the system will provide certain specialized non-aggregate functions for unusual cases which can be expected to recur with some frequency. Typical functions might be SDIST, for example, to compute the spherical distance between two points on the earth's surface, and DURATN, to calculate the difference between two calendar dates. Non-aggregate functions will be permitted to be nested inside each other and inside aggregate functions. Figure 4-9 illustrates the use of functions inside queries. Where applicable, the Boolean expression in the qualification can be replaced by an asterisk to indicate that the data manipulation operation is to be performed on all records of the referenced table (e.g., WHERE *).

4.5.1.3 The Workspace Table

There will be precisely one Workspace Table, W, associated with each active user on the system. By convention, W will be empty when a user signs onto the system, and will be destroyed when that user issues an EXIT command from the system. W will have no pre-defined fields and no special access superstructures, nor will W be considered to be a part of any particular data base. Thus when a user detaches from one data base and attaches to another the contents of W will be undisturbed, and therefore W will constitute a mechanism for the transportation of data from one data base to another.

The contents of W will be generated by a SELECT

QUERY 9: Determine the average duration of hurricanes occuring since 1972.

```
SELECT (AVDUR = AVG(DURATN(E.START-DATE, E.END-DATE)))
 WHERE E.CLASS EQ HURRICANE AND E.START-DATE GE 010173#
```

QUERY 10: Retrieve the data file id's and area of all images which lie along a line running from 30°N latitude and 75°W longitude with an azimuth of 45°, terminating at 72°W longitude, for October 4, 1976. (Assume latitudes, longitudes, and azimuths expressed in radians and that west longitudes are negative.)

```
SELECT (D.DID, AREA = SDIST(D.TOP-LAT,D.LEFT-LON,D.BOTTOM-LAT,D.LEFT-LON)
                * SDIST(D.TOP-LAT,D.LEFT-LON,D.TOP-LAT,D.RIGHT-LON))
 WHERE D.TOP-LAT GE .5236 + TAN(.7854)*(D.LEFT-LON + 1.309)
   AND D.BOTTOM-LAT LE .5236 + TAN(.7854)*(D.RIGHT-LON + 1.309)
   AND D.RIGHT-LON LE -1.2566
   AND D.DATE = 041076 #
```

[Note: Retrieval is based on the fact that a line with a positive slope intersects a rectangle if and only if it lies between two lines with the same slope which intersect the rectangle at its upper left and lower right corners, respectively.]

Figure 4-9: Relational Calculus Retrievals Using Functions in the Target List

command. As records are retrieved from a data base they will be entered into W , and the sequence of fields, their names, and their definitions will be implicitly created by the system to correspond to the fields in the records being retrieved. If W should happen to be non-empty at the time a new SELECT is issued, then its former contents will be overwritten and lost, and the previous field definition for W will be replaced by one which reflects the format of the new records. Hence it follows that the workspace table may have a variety of different field definitions during a single user session, depending on the number of SELECT commands issued by the user and the specific data requested each time.

A particular user's workspace table will be associated with that user personally, and will be inaccessible to all other active users -- including the DBA. A user will always have the right to insert records into or delete records from his or her workspace table, even if lacking MODIFY rights on the database to which he or she is attached. Aside from that, however, the user will be free to treat his or her workspace table as if it were a part of any data base to which he or she is attached. That is, retrievals may be made against W itself, and other tables may be cross referenced against data stored in W . Moreover W may be used as a source of data for insertions and deletions on other tables in the data base. The only particular restriction of which the user must be aware is that the system will not permit an INSERT command into W if W is empty -- an empty workspace can be filled only by a SELECT command.

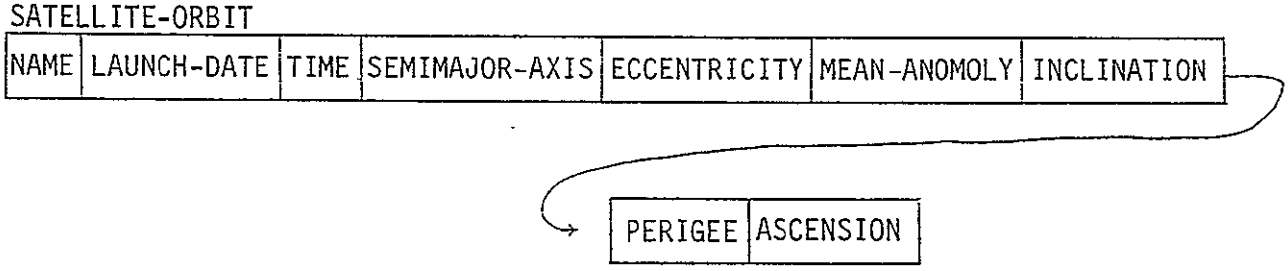
4.5.1.4 Comparing the Two Approaches

Both the Query-by-Example approach to information retrieval and the relational calculus have been demonstrated to be "relationally complete," that is, any relational operation which can be performed on a relational data base can also be performed using either of these two approaches. Why then, have two different methods to do the same thing? The answer lies in the words "user convenience" as each of these two approaches provides a user with certain niceties that are unavailable in the other.

The major advantage of the relational calculus-based approach is the ability to put functions of one or more fields in the target list and on the right-hand side of a qualification predicate. This makes the relational calculus exceedingly powerful, particularly with respect to a scientifically-oriented data base.

Query-by-Example, however, has a variety of advantages over the relational calculus. In the interactive mode, for one thing, Query-by-Example is very user friendly. A study of this point[38] has, in fact, demonstrated that Query-by-Example compares very well with other query languages for relational data bases, most especially with respect to ease of learning and user retention. Since one goal of this proposed data base management system is ease of use for casual, infrequent, and/or minimially experienced users, this point argues very strongly for supplying Query-by-Example as a retrieval language. A second point, also noted in the study, is that Query-by-Example is "behaviorally extendable" in the sense that a novice need only learn a small part of the language to write successful queries and can build on his or her knowledge as required. However, from the point of view of an information retrieval language for a scientific (as opposed to business) data base

perhaps the most important advantage of Query-by-Example concerns data units. Consider the example table describing satellite orbits depicted below



and let us suppose that semimajor axis is recorded in kilometers. But the user may wish to frame his or her request in terms of earth radii and not know (or be willing to take the time to calculate) the appropriate conversion factor. By inputting the query with units attached, as depicted in Figure 4-10a, the system could perform the conversion itself. Similarly, as in 4-10b, the units could be attached to the example element for automated conversion on output elements. It is possible to extend the relational calculus by attaching units to numerical quantities in the qualification, but it is not easy to see how to do the same for entries in the target list. Granted, nothing prevents suitable conversion functions from being established for the relational calculus approach, but the user would be forced to pay a penalty in burden on the memory and complexity of the resultant retrievals.

QUERY 11: Retrieve all satellites in high orbit (>4 earth radii).

SELECT FROM SATELLITE-ORBIT

SATELLITE-ORBIT

NAME	LAUNCH-DATE	TIME	SEMIMAJOR-AXIS	ECCENTRICITY	...
S*TIROS			>4 RADII		

Figure 4-10a: Retrieval with Automatic Units Conversion

QUERY 12: Retrieve name, launch date, and radius of all satellites in circular orbit.

SELECT FROM SATELLITE-ORBIT

SATELLITE-ORBIT

NAME	LAUNCH-DATE	TIME	SEMIMAJOR-AXIS	ECCENTRICITY	...
S*TIROS	S*010475		S*10 RADII	0	

Figure 4-10b: Retrieval with Automatic Output Units Conversion

4.5.2 INSERT

Like SELECT, the INSERT command will have two distinct syntaxes, as shown below:

$$\text{INSERT[INTO]<table name>} \begin{cases} (<\text{record}_1>)[,(<\text{record}_2>)]... \\ (<\text{target list}>)\ \text{WHERE}\ <\text{qualification}> \end{cases}$$

where:

<record>::=<assignment>[,<assignment>]...

<assignment>::=<field name>=<constant>

One special constant will be the key word NULL, which will represent a null value for the corresponding field of the table. Alphanumeric constants, other than the key word NULL, must be enclosed in apostrophes. A user -- provided he or she has INSERT rights on the particular table and MODIFY authorization for the data base which contains that table (see GRANT) -- will have the option to spell out the new records to be added to the table, or may generate the new records to be added to the table by retrieving data from and cross referencing other tables in the data base*. The latter approach will be equivalent to a SELECT using relational calculus syntax (see Section 4.5.1.2), except that the records retrieved will be placed in the specified table, rather than the Workspace Table.

Again, it should be noted that a special character ("#") must terminate this command, as the user may be forced to input one or more carriage returns before completion of the full command input sequence.

---

* Including the user's Workspace Table.

The table which receives the records may be the Workspace Table, W, except that the system will not accept an INSERT into W unless W is nonempty. Inserting into an empty Workspace Table can only take place via a SELECT command.

4.5.3 UPDATE

There will be only one syntax for the UPDATE command, as depicted below:

UPDATE <table name> (<change$_1$>[,<change$_2$>]...) WHERE <qualification> #

where the changes indicate the way fields in records are to be modified and have the form:

$$\text{<change>::=<field name> = } \left\{ \begin{array}{l} \text{<constant>} \\ \\ \text{<function>} \end{array} \right\}$$

A constant appearing on the right hand side of a change should agree with the definition of the field in type, and any function must, of course, be a function of the field being changed (e.g., ALTITUDE=ALTITUDE - 50.3). The syntax of a qualification is defined in Section 4.5.1.2.

In the case where the qualification consists of an asterisk rather than a boolean expression, the Integrated Data Base Management System will apply the changes to all records in the table. Note that, whether a boolean qualification is input or not, a special character ("#") must be used to terminate the command.

Again, users must have authorization to UPDATE the particular table and to MODIFY the data base which contains it before the system will accept an UPDATE command from them. Moreover, if the WHERE clause references another table then the user must have READ rights on that table.

Updates will not be accepted for the Workspace Table.

4.5.4 DELETE

The syntax of the DELETE command is:

```
DELETE [FROM] <table name> WHERE  <qualification> #
```

where the syntax of a qualification is described in Section 4.5.1.2. Like SELECT, INSERT, and UPDATE, the DELETE command has to be terminated by a special character ("#") as the entire command may span multiple lines of input. A null qualification will cause the entire contents of the table to be deleted, although the table itself will remain (albeit in an empty state).

A user will have to have been granted DELETE authorization on the specified table, MODIFY authorization on the data base which contains it, and READ authorization on any additional tables referenced in the qualification, except, of course, that he or she may always DELETE from W if W is nonempty (but the requirement for READ authorizations on the other tables must still be observed).

4.5.5 DISPLAY

The DISPLAY command will be used to list the contents of a table (including the workspace table) at an interactive user's terminal. The syntax of a DISPLAY command will be:

DISPLAY [<table name>][(<target list>)][FORMAT=(<format>)]

where the syntax of a target list is described in Section 4.5.1.2 and the syntax of a format will be identical to the format specifications inside a Fortran format statement. That is,

<format>::=<specification$_1$>[,<specification$_2$>]...

and

<specification>::= '<string>'
<integer>H<string>
[<integer>] { X | I<integer> | {F | E | D}<integer>.<integer> | A<integer> }
T<integer>

The default for output table name will be the workspace table, W, but the user may specify any table in the data base to which he or she is currently attached. The user will also be permitted to specify a target list, so that only certain columns of the table are shown, and the default will be to list the entire record for each record in the table. Finally, the Integrated Data Base Management System will allow the user to input any valid Fortran output format for listing the table, or else the user can let the system select its own format specifications,

based on the type and size of each field to be displayed. The choice of format specifications for fields will be by table lookup (a typical table, assuming four bytes per word is depicted below). Columns will be evenly spaced, with the spacing chosen to make the output readable. The system will output blanks for fields which are null.

TYPE	SIZE	FORMAT
alphanumeric	n	An
real	4	E12.7
real	8	D16.10
integer	1	I4
integer	2	I6
integer	3	I8
integer	4	I12
logical	1	A5*

TABLE 4-1: System-Generated Formats

Note that a special character will not be needed to terminate this command, and that a user may specify a table other than his or her workspace table only if the user has a READ authorization on that table.

There is one important *caveat* : the user should be aware that executing a DISPLAY on a table other than the user's workspace table may lock out other users' insertions, updates, and deletions for a considerable span of wall clock time and thus this feature should be avoided except on small tables or tables with low usage.

*Will print 'TRUE' or 'FALSE'.

4.5.6 PRINT

The Print command is similar to DISPLAY, except that the output will be directed to a line printer, and not to an interactive terminal.. The syntax of a PRINT command will be:

```
PRINT { TITLE <title>                                                          }
      { [<table name>][<target list>)][FORMAT=(<format>)]                      }
```

where target list is described in Section 4.5.1.2 and format in 4.5.5. Again, a user will have the option of specifying any table in the data base to which he or she is attached (and for which he or she has READ authorization) or — by default — having the system list the contents of the user's workspace table. Moreover, the user will further retain the option of singling out certain columns for listing versus having all columns listed, and of specifying an output format versus allowing the Integrated Data Base Management system to choose its own format. However, a user will also be able to use a PRINT command to output a title to be placed on the printer listing. The system itself will center the title.

4.6 Data File Commands

Not all of the data managed by the Integrated Data Base Management System will be stored in tables. Indeed, one of the most important functions of this system will be to provide its users with efficient and convenient access to data files maintained on tape or stored on-line. A more unified treatment of the way in which the Integrated Data Base Management System provides access to the data files may be found in Section 7, Data File Processing. This subsection will merely present the syntax and function of the eight file operations available to an interactive user, which will be:

COPY - Insert records from a data file into a table or vice versa.

CATALOG - Insert a new data file into the system catalog.

UNCATALOG - Purge a catalog entry.

LOAD - Create an on-line data file in (a) system standard format from a data file on tape.

UNLOAD - Create a backup copy on tape of an on-line data file.

KEEP - Mark a temporary on-line data file for permanent saving.

SCRATCH - Purge an on-line copy of a data file.

PERFORM - Manipulate the contents of a data file via loadable library routines.

The operation of these eight commands is governed by the following set of principles:

(1) Each data file is identified by a unique data file identifier (did).

(2) All on-line files will be stored in a self-describing standard format.

(3) All data files are read-only, and may be purged, but not edited in place or overwritten.

(4) All data files known to the system will have an entry in the system catalog.

The CATALOG and UNCATALOG commands will be restricted to the DBA, and KEEP and SCRATCH will be restricted to the owner of the on-line copy of the data file and the DBA. All other commands will be unrestricted, and available to any user. These file operations will be viewed by the system as being functions of the system "back end", as opposed to commands in the other groups which will be operations on the "front end". Hence, the user -- though forced by convention to be attached to some data base -- need not be attached to any particular data base to request a file operation, except the COPY operation.

Three of the above commands -- COPY, LOAD, and UNLOAD -- will be available for use by applications programs.

4.6.1 COPY

The syntax of a COPY command is depicted below:

```
        ⎧<table name>                  ⎫
COPY    ⎨                              ⎬
        ⎩<did> TO <table name>         ⎭
```

Although the function of this command (copying the contents of a table into a data file versus copying the contents of a data file into a table) will be symmetrical, the syntax will not. This lack of symmetry is explained by the fact that when a table is copied to a data file the system creates a new on-line file with a new data file identifier to receive the records from the table, while in the reverse case, when data coming in from a data file (on- or off-line) the table which receives those records must have been pre-defined.

When a COPY command is used to copy records from a table to a file, the system will display or print* the new data file id. The primary side effect of this command is that null fields are replaced with zero (if numeric) or filled with blanks (if alphabetic).

There are three important conditions which must be met before the system will accept a COPY command from a file into a table. The first of these, as expressed above, is that the table which receives the records must have been predefined. It need not be empty, however, and the new records will merely be appended to the end of the table. The second condition is that the sequence of fields in the table must agree in type, size, and number with the sequence of fields in a record of the file.

*Depending on whether the command is input from an interactive terminal or the Batch Command Reader, respectively.

Finally, the data file must be recognized as having tabular-like data, as opposed to image data or profile data, for example. This is due to the fact that the sequence of records in, say, an image file will bear a relationship to one another based on their order while, by definition, the order of records in a table is meaningless. Tabular operations take no cognizance, then, of the particular order of records in a table, but this order cannot be ignored for other types of data.

4.6.2 CATALOG

The DBA can enter tape files into the Integrated Data Base Management System's Data File Catalog (SYSCATL system table) with the CATALOG command. The syntax of a CATALOG command will be:

CATALOG (<file entry$_1$>)[,(<file entry$_2$>]...

where

<file entry>::=<reel number>,<file number>,<format code>

The system will verify that each entry is unique and, if so, the system will assign the file a unique data file identifier <did>, make the entry in the catalog, and output the <did> to the DBA. If, however, the file entry duplicates a previous entry, then the system will merely return the pre-existing <did>.

The uniqueness of a file entry depends on all three components of the entry, and not just the reel number and file number. Therefore, if one physical file contains multiple logical data files, then the separate logical files may be indicated with different format codes.

Only the DBA will have the authority to issue a CATALOG command. It should be noted that the DBA will also have to execute one or more INSERT commands into the system directory tables to reflect the new entries in the catalog.

4.6.3 UNCATALOG

The DBA may purge catalog entries by using the UNCATALOG command. The syntax of an UNCATALOG command will be:

UNCATALOG <did$_1$>[,<did$_2$>]...

Not only will the catalog entry be wiped out for each <did> named in an UNCATALOG command, but on-line copies of those files will also be purged and records referencing those <did>'s will be deleted from the system directory tables. However, the tape file itself will be untouched, and any records in tables belonging to local data bases which reference that <did> will also remain as they were before the UNCATALOG command was issued.

Only the DBA may issue an UNCATALOG command. With respect to an on-line file being purged as a side effect of the UNCATALOG command, the same rules apply as for the SCRATCH command. That is, if an application program has opened that file then it will not be erased until it has been closed.

4.6.4 LOAD

The function of the LOAD command will be to create an on-line data file by copying (and perhaps reformatting) a cataloged tape file (or portion of a tape file). The syntax of a LOAD command will be:

LOAD [<operation$_1$>(<parameter list$_1$>)[,<operation$_2$>(<parameter list$_2$>)]...]<did>

where the valid operations will be SLICE, WINDOW, and SUBSET, and the parameters to be specified will depend upon the operation.

The effect of a LOAD command will be to create an on-line file in a self-descriptive system standard format. Where no operation is specified, the entire tape file will be brought on-line and reformatted, if necessary, to conform with the appropriate system standard format. In such a case, the data file id for the on-line copy and the off-line copy will be the same.

The SLICE operation is designed to create subfiles from data files representing multi-dimensional grids. Such a file can have up to five dimensions, corresponding to the standard horizontal and vertical (x and y) axes on the ground, an altitude (z) axis, a time (t) axis, and a wavelength axis (λ). The SLICE operation will permit the user to take a 2-D "slice" along any pair of axes represented in the file. The parameter list, then, will be the pair of axes through which the SLICE is to be taken, plus a set of equations fixing the remaining axes. The BNF syntax description for a parameter list for SLICE is:

<parameter list>::=<axis$_1$>,<axis$_2$>,<eq'n$_1$>[,<eq'n$_2$>]...

where

<eq'n>::=<axis>=<constant>

<axis>::=X|Y|Z|T|L

An error message will be output and the command aborted if any of the specified axes are not present in the gridded data file or if an axis is left unspecified.

A WINDOW operation will only be performable on a two dimensional file (e.g., a digitized image or digitized cartographic terrain elevation model). The WINDOW operation will cause a rectangular subarea to be selected from the specified data file and copied into an on-line file. The user will have to specify four parameters: starting and ending line number and starting and ending column number, all specified relative to the first data point in the first record as (1, 1). The pair of line numbers and pair of column numbers may be in any order, but the user shall be required to specify both line numbers before specifying either column number. Zero starting values (line or column) will be the same as a one, and values greater than the number of lines or size of a line, respectively, will be rounded down to the appropriate value. However, starting values which are greater than the number of lines or size of a line will cause an error message to be returned and the command to be aborted.

It is anticipated that most data files will contain observed values for more than one physical variable at each observation point. By using the SUBSET operation a user will be able to create an on-line file which contains any non-empty subset of those variables. The parameter list for a SUBSET operation will be a list of names of physical variables, and the size of the list will be permitted to vary from command

to command. If the user should happen to specify a physical variable not represented in the specified file, or if the system fails to recognize one or more of the physical variable names, then the system will abort the command and return an error message.

As indicated by the syntax, these three operations may be combined in any order in a single LOAD command, if the user so desires. However, the use of one or more operations with a LOAD command will cause a new data file to be created, so that the system will have to generate a new data file id and assign it to the new data file. Whenever an on-line file is created, whether it is a new data file or a direct copy from tape, the user issuing the LOAD command will become the owner of the file. Anyone will be able to access the file, but only the owner (or the DBA) will be allowed to issue an explicit purge on it. However, unless marked "permanent" by the DBA or its owner, any on-line file will be classed by the system as "temporary" and automatically purged a specified number of days after its last access.

4.6.5 UNLOAD

The syntax of an UNLOAD command will be:

UNLOAD <did_1>[,<did_2>]...

The UNLOAD command will create back-up tape copies in system standard format for each on-line data file listed in the command (unless the Integrated Data Base Management System determines that a back up tape file already exists for the specified on-line file). Anyone may UNLOAD an on-line file.

4.6.6 KEEP

The role of the KEEP command will be to take a temporary on-line file and mark it permanent. The syntax of a KEEP command will be:

KEEP <did_1>[,<did_2>]...

Anyone may issue a KEEP command, but with privilege comes responsibility and hence the user who issues a KEEP command will become the owner of the kept file.

KEEP commands will apply only to temporary on-line files, and KEEP commands issued for permanent files will be rejected by the system.

4.6.7 SCRATCH

The SCRATCH command will cause on-line files to be purged from mass storage. The syntax of a SCRATCH command will be:

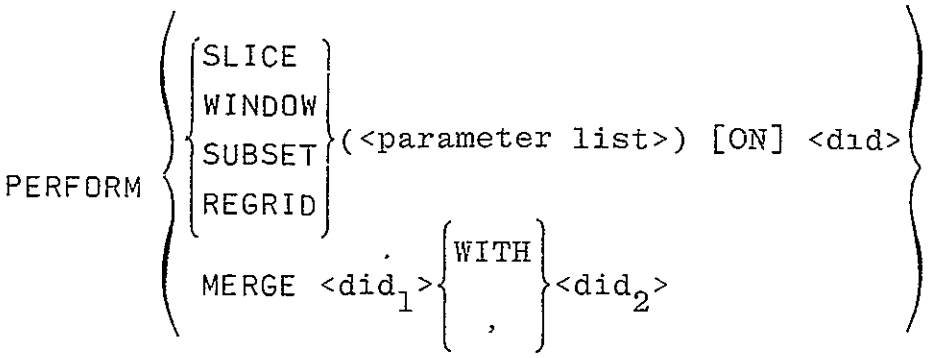
SCRATCH <did$_1$>[,<did$_2$>]...

Only the owner of an on-line file or the DBA may SCRATCH that file, and then only if the file is marked temporary or a tape version exists. An on-line file shall not be purged if it has been marked permanent and no back-up tape copy exists -- it will be necessary for the user to UNLOAD that file before issuing the SCRATCH or else an UNCATALOG will have to be used.

A SCRATCH command will affect only the on-line file unless the file is temporary and no tape copy exists, in which case the entire catalog entry will be deleted. In no case will tables in the front end of the system be affected by a SCRATCH command.

4.6.8 PERFORM

The syntax of a PERFORM command will be:



where parameter lists have been defined in Section 4.6.4 or are described below. Any data files specified in a PERFORM command must be on-line files in a system standard format, and the result of executing a PERFORM command will be a new, "temporary" on-line data file, also in system standard format. The system will display or print the data file id (did) for this new file, and the owner of the new file will be the user who issued the PERFORM command. As per system convention, the execution of a PERFORM command will not cause any change to occur to the input file(s) specified in the command sequence.

The SLICE, WINDOW, and SUBSET operations have been described in the subsection of this report which discusses the LOAD command. The REGRID operation will map observation points from one multi-dimensional grid coordinate system onto another grid coordinate system, interpolating new data values at each observation point as necessary. The new coordinate system must be derivable from the old coordinate system strictly by a translation of origin and a change of scale. For an n-dimensional coordinate system ($n \geq 2$) there will be $2n$ parameters in the parameter list. The first n parameters will be scaling factors (each strictly greater than zero) by which the corresponding coordinate axis is to be multiplied to

generate new grid spacings, and the second n parameters will be translation parameters which will dictate how the origin of the original coordinate system is to be shifted along each axis to generate the new origin. There are five possible coordinate axes: x and y (on the ground), z (altitude), t (time), and λ (wavelength). Any subset of two or more of these might be present in the original file, but the order of axes will always be presumed to obey the transitive ordering sequence t before x before y before z before λ .

The MERGE operation will cause two data files to be merged into a single data file. A number of preconditions must be met before the system will accept a PERFORM MERGE command:

(1) The two coordinate systems must have the same number and type of coordinates (except for wavelength, as discussed below).

(2) The grid spacing along all axes must be the same for both grids.

(3) For purposes of a MERGE operation, observations along each hyperplane defined by a fixed wavelength (λ) will be treated as observations of a single physical variable.

(4) The areal coverage along the x and y axes must overlap.

(5) If there is a t axis in the files' coordinate system, then the time periods must also overlap.

If the origin and termination of the axes for both grids are not the same, the merged data file will contain a grid consisting only of the overlapping portions of the original grid. The observations at each grid point in the new data file will be formed by concatenating the observations from the first data file with those in the second data file.

The five file operations presented in this subsection constitute a minimal and not necessarily complete set of operations. The Integrated Data Base Management System has been designed to permit additional operations to be added (e.g., histogramming) at a future date.

SECTION 5 - THE APPLICATION PROGRAM COMMAND LANGUAGE

5.1 Introduction to Application Program Command Processing

The Application Program Command Language is the method by which an application program communicates with the Integrated Data Base Management System. It permits an application program to access tabular data as well as data files. The Application Program Command Language is not a complete language by itself. It relies on a host language to provide a framework for it and to provide the procedural capabilities required to manipulate data. The command language consists of a set of CALL statements or its equivalent which are incorporated into a procedural host language program. The command language may be used with any host language (e.g., FORTRAN, COBOL, PL-1, ALC) that supports a CALL statement. A single entry point or subroutine name is used for all application program commands. The CALL statement will have a variable length argument list as a function of the command being issued. The first two arguments in every application program are the command itself (e.g., 'SELECT', 'READ') and an integer variable which, upon return from the Integrated Data Base Management System, will contain the status associated with the execution of the command.

To an application program, the Integrated Data Base Management system appears to be the single subroutine, IDBMS. To the Integrated Data Base Management System, an application program will appear to be a special type of user with its own User Control Block and its own Workspace Table. Between an application program and the system will be the Application Program Interface, which will create a Command Control Block for the application program command and place it on the proper queue for processing.

The Application Program Command Language contains several commands which are also included in the Interactive Command

Language, described in Section 4. However, after analysis of projected user requirements, several commands from the Interactive Command Language were omitted from the Application Program Command Language. These include commands to define and remove data bases, tables, fields, users and groups and commands to grant and revoke access rights as well as others. It should be noted that nothing in the design of the Integrated Data Base Management System would preclude those commands which were omitted from being included in the Application Program Command Language. The set of commands available to an application program will include only a subset of the commands available to interactive users, as detailed in Table 5-1. It should be noted that the data file processing operations such as REGRID, SLICE, etc. initiated interactively via the PERFORM command are available to application programs directly as part of the Application Program Command Language. Also, an application program will have available to it an additional set of commands which are not available to interactive users. This section provides an overview of how an application program interacts with the system, describes the calling sequence for issuing "interactive" commands through the Application Program Interface, and describes the calling sequence for and function of the remaining commands. Since most of these special commands access data files in the Non-Relational Data Base, the reader is expected to be familiar with Section 7, Data File Handling, as well as Section 4, The Interactive Command Language.

5.2 Issuing "Interactive" Commands from an Application Program

As stated previously, only a subset of the Interactive Command Language is included in the Application Program Command Language. Table 5-1 lists the interactive commands which can be issued by an application program and the following subsections describe the calling sequence for each of

UTILITY COMMANDS

- ENTER
- EXIT
- ATTACH
- USE

DATA MANIPULATION COMMANDS

- SELECT
- INSERT
- UPDATE
- DELETE

DATA FILE COMMANDS

- COPY
- LOAD
- UNLOAD

DATA FILE PROCESSING OPERATIONS (PERFORM)

- SLICE
- WINDOW
- SUBSET
- REGRID
- MERGE

Table 5-1: Commands Available to Both Interactive Users and Application Programs

these commands as well as additional commands which support them. The result of issuing any of these commands by an application program is the same as if they were entered interactively. Thus, the description of each of these commands in Section 4 is applicable and will not be repeated here.

5.2.1 Utility Commands

Interactive commands from the category of Utility Commands which can be issued by an application program are described below.

5.2.1.1 The ENTER Command

The ENTER command connects an application program to the Integrated Data Base Management System. This command must be issued by an application program prior to issuing any other command in the Application Program Command Language. It is coded as follows:

```
CALL IDBMS('ENTER',<status>,<program-id>,<user-id>,<password>)
```

where:

- <status> is a binary integer variable which, upon return from the Integrated Data Base Management System, will contain an integer value indicating whether or not the command was executed successfully. A status code of zero indicates successful execution of the command. A positive status code indicates unsuccessful execution of the command. The value of the positive integer defines the error condition which caused the unsuccessful execution.

- <program-id> is an alphanumeric literal or variable which uniquely identifies the application program which is attempting to connect to the Integrated Data Base Management System. The program-id is analagous to the user-id associated with each valid user of the system.

- <user-id> is an alphanumeric literal or variable which identifies the user running the application program. During this execution of the application program, it will assume the access rights associated with the user running the program.

- <password> is an alphanumeric literal or variable containing the password of the user identified by the <user-id> argument.

5.2.1.2 The EXIT Command

The EXIT command disconnects an application program from the Integrated Data Base Management System. This command should be issued by any application program which has connected to the system via the ENTER command. During EXIT command processing, the system will close any data files for which no CLOSE command was issued by the application program and reset all locks on tables which have not been explicitly reset by an UNLOCK command. The EXIT command is coded as follows:

```
CALL IDBMS('EXIT',<status>)
```

where <status> is as previously defined.

5.2.1.3 The ATTACH Command

The ATTACH command indicates the intent of the application program to access the specified data base. It is coded as follows:

```
CALL IDBMS('ATTACH',<status>,<data base name>)
```

where <status> is as previously defined and:

- <data base name> is an alphanumeric literal or variable which contains the name of the data base to which the application program is to be attached. After successful completion of this command, the specified data base will

be the application program's primary data base and any subsequent data manipulation or COPY commands issued by the application program prior to another ATTACH command will access that data base.

5.2.1.4 The USE Command

The USE command permits a one or two character alias name to be specified for a table. It is coded as follows:

```
CALL IDBMS('USE',<status>,<alias name>,<table name>)
```

where <status> is as previously defined and:

- <alias name> is an alphanumeric literal or variable which specifies a one or two character alias that can be used in place of the name of the table specified in the <table name> argument. The alias name can be used in subsequent commands wherever the associated table name can validly be used. The alias name remains until it is assigned to another table or until the application program terminates.

- <table name> is an alphanumeric literal or variable which contains the name of the table for which the alias is being established.

5.2.2 Data Manipulation Commands

Interactive commands from the category of Data Manipulation Commands are described in this subsection. These commands permit an application program to manipulate tabular data in much the same way as an interactive user can. All of the commands in this category contain one or more arguments in their calling sequence which is a variable length string. By including these strings, the calling sequence is simplified considerably and causes the argument list to resemble the interactive command syntax. To facilitate the use of variable length strings, each such string must be terminated with the special symbol #. Some examples of the use of these variable length strings is

illustrated in a subsequent subsection.

5.2.2.1 The SELECT Command

The SELECT Command retrieves data from one or more tables and places it in the Workspace Table associated with the application program. The tables referenced by the SELECT command must be contained in the data base named in the most recent ATTACH command issued by the application program. The SELECT command is coded as follows:

```
CALL IDBMS('SELECT',<status>,<record no>,<target list string>,
        <qualification string>)
```

where <status> is as previously defined and:

- <record no> is a binary integer variable which, upon return from the Integrated Data Base Management System, will contain the number of records which have been placed in the Workspace Table as a result of the execution of this SELECT command. This argument may contain zero upon return, if no records were placed in the Workspace Table.

- <target list string> is an alphanumeric literal or variable which defines the data fields for which values are to be retrieved from existing tables. This argument is exactly the same as the target list specified in the interactive SELECT command described in Section 4. It defines the data fields which constitute the Workspace Table constructed as a result of the execution of the SELECT command. The target list string must be terminated by the special character #.

- <qualification string> is an alphanumeric literal or variable which specifies the conditions that must be met by a record for it to be selected for retrieval. This argument is exactly the same as the qualification specified in the WHERE clause of the interactive SELECT command described in Section 4 except that application program variable names can be used as well as constants in the relation conditions (e.g., X EQ 2 could be replaced by X EQ V1 where V1 is an application program variable which has been set to 2 by another application program statement). The concept of using variable names in the qualification string will be discussed in more detail in

the description of the BIND command in a subsequent subsection. The qualification string must be terminated by the special character #.

5.2.2.2 The INSERT Command

The INSERT command adds one or more new records to a table. There are two forms of the INSERT command: one which permits a single record to be inserted directly into a table by supplying values for the data fields within the argument list and a second which will operate in a manner similar to the SELECT command in that data will be retrieved from one or more tables and the resulting records will be added to the table named in the argument list rather than being placed in the Workspace Table. All tables referenced by the INSERT command must be contained in the data base named in the most recent ATTACH command issued by the application program.

The implementation of two different types of INSERT commands requires that the Application Program Communication Module, IDBMS, be capable of recognizing two different argument lists. The "record" option, where data field values are specified explicitly in the argument list, requires only a <record string> argument, whereas the "selection" option requires both a <target list string> and a <qualification string> argument There are several techniques for handling this problem and the choice of one over the other may be operating system dependent. For example, if each compiler supported by the operating system marks the last argument in an argument list, the Application Program Communication Module can detect the shorter argument list of the "record" option. If the compilers do not mark the last argument, the <record no> argument could be set to a negative value by the application program prior to issuing the INSERT command to indicate the "record" option and a non-negative value to indicate the "selection" option or vice versa. Alternatively, open and close parentheses could be used in the

<record string> argument to distinguish it from the <target list string> argument which would not contain parentheses, thereby defining the type of INSERT command being issued (e.g., '(X=1,Y=2)#'). The description of the arguments below assumes that the compilers support variable length argument lists by marking the last argument in the list. The two forms of the INSERT command are coded as follows:

"record" option

CALL IDBMS('INSERT',<status>,<record no>,<table name>,<record string>)

where <status> is as previously defined and:

- <record no> is a binary integer variable which, upon return from the Integrated Data Base Management System, will contain the integer value one if the record defined in the <record string> argument was successfully added to the table named in the <table name>. Otherwise, the integer value zero will be returned.
- <table name> is an alphanumeric literal or variable which contains the name of the table to which the record is to be added.
- <record string> is an alphanumeric literal or variable which defines the values to be assigned to data fields in the record to be added. This argument is exactly the same as the record which can be specified in the interactive INSERT command described in Section 4 except that application program variable names can be used as well as constants. The data fields named in this argument must be data fields in the table specified in the <table name> argument. Any data fields which are not assigned a specific value will contain a null value in the added record. The record string consists of one or more assignment statements separated by commas. The record string must be terminated by the special character #. The form of the assignment statement is:

<assignment statement>::=<data field name>={<constant> | <program variable name>}

where <program variable name> must be a variable defined in the application program and used in a preceding BIND command.

"selection" option

```
CALL IDBMS('INSERT',<status>,<record no>,<table name>,<target list string>,
        <qualification string>)
```

where <status> is as previously defined and:

- <record no> is a binary integer variable which, upon return from the Integrated Data Base Management System, will contain the number of records which have been added to the table named in the <table name> argument as a result of the execution of this INSERT command. This argument may contain zero upon return if no records were added to the table.

- <table name> is an alphanumeric literal or variable which contains the name of the table to which the records are to be added.

- <target list string> is an alphanumeric literal or variable which defines the data fields in the table specified in the <table name> argument for which data values are to be retrieved from the records satisfying the criteria stated in the <qualification string> argument. This argument is exactly the same as the target list which can be specified in the interactive INSERT command described in Section 4. Any data fields in the table specified in the <table name> argument which are not included in this argument will contain a null value in all added records. The target list string must be terminated by the special character #.

- <qualification string> is an alphanumeric literal or variable which specifies the conditions that must be met by a record for it to be retrieved and used to construct a new record to be added to the table specified in the <table name> argument. The syntax of the <qualification string> argument is the same as that for the SELECT command described in a previous subsection.

5.2.2.3 The UPDATE Command

The UPDATE command modifies one or more data fields in one

or more records in a table. All tables referenced by the UPDATE command must be contained in the data base named in the most recent ATTACH command issued by the application program. The UPDATE command is coded as follows:

```
CALL IDBMS('UPDATE',<status>,<record no>,<table name>,<change list string>,
        <qualification string>)
```

where <status> is as previously defined and:

- <record no> is a binary integer variable which, upon return from the Integrated Data Base Management system, will contain the number of records which have been modified as a result of the execution of this UPDATE command. This argument may contain zero upon return if no records were modified.
- <table name> is an alphanumeric literal or variable which contains the name of the table in which the records are to be modified.
- <change list string> is an alphanumeric literal or variable which defines the data fields to be modified and the new values which are to be assigned to them. This argument is exactly the same as the change which can be specified in the interactive UPDATE command described in Section 4 except that application program variable names can be used as well as constants and functions. The data fields named in this argument must be data fields in the table specified in the <table name> argument. The change list string consists of one or more assignment statements separated by commas. The change list string must be terminated by the special character #. The form of the <change list string> argument is:

<change list string>::='<assignment statement$_1$>[,<assignment statement$_2$>]...#'

<assignment statement>::=<data field name>={<constant> | <function> | <program variable name>}

where <program variable name> must be a variable defined in the application program and used in a preceding BIND command.

- <qualification string> is an alphanumeric literal or variable which specifies the conditions that must be met by a record for it to be modified. The syntax of the <qualification string> argument is the same as that for the SELECT command described in a previous subsection.

5.2.2.4 The DELETE Command

The DELETE command deletes one or more records from a table. The table referenced by the DELETE command must be contained in the data base named in the most recent ATTACH command issued by the application program. The DELETE command is coded as follows:

```
CALL IDBMS('DELETE',<status>,<record no>,<table name>,<qualification string>)
```

where <status> is as previously defined and:

- <record no> is a binary integer variable which, upon return from the Integrated Data Base Management System, will contain the number of records which have been deleted as a result of the execution of this DELETE command. This argument may contain zero upon return if no records were deleted.
- <table name> is an alphanumeric literal or variable which contains the name of the table from which the records are to be deleted.
- <qualification string> is an alphanumeric literal, or variable which specifies the conditions that must be met by a record for it to be deleted. The syntax of the <qualification string> argument is the same as that for the SELECT command described in a previous subsection.

5.2.3 Operations which Support "Interactive" Commands

There are several commands which can be issued by an application program that provide additional tabular data handling or support capabilities but are not available to interactive users. These commands are required in an application program because of the procedural environment in which an application program operates. Each of these commands is described below.

5.2.3.1 The BIND Command

The BIND command permits the Integrated Data Base Management System to recognize a program variable name and to associate that program variable with its proper location in the main storage allocated to the application program. After being named in a BIND command, a program variable can be used in an alphanumeric string argument in an "interactive" type command wherever a constant could be validly used. Thus, program variables can be used in <qualification string>,<change list string> and <record string> arguments in the data manipulation commands described in the preceding subsections. The BIND command is coded as follows:

```
CALL IDBMS('BIND',<status>,<program variable name>,<program variable>)
```

where <status> is as previously defined and:

- <program variable name> is an alphanumeric literal or variable which contains the name of the program variable as it will appear in an alphanumeric string argument in a subsequent command. While the program variable name will represent the program variable in string type arguments in subsequent commands, they need not match (e.g., 'LATITUDE',LAT where LATITUDE would be used in string arguments to represent the program variable LAT).

- <program variable> is a variable which is defined and assigned values within the application program. Whenever the program variable name specified in the preceding argument is encountered in an alphanumeric string argument, the Integrated Data Base Management System will substitute the location of the program variable and will use the current value stored at that location during its processing.

5.2.3.2 The FETCH Command

The FETCH command retrieves data values from a record in the Workspace Table and makes them available to the application program for processing. The data fields named in the command must be data fields defined for the Workspace Table by the most recent SELECT command issued by the application program. The first FETCH command following a SELECT command will cause values of the specified data fields to be retrieved from the first record in the Workspace Table and to be placed into the specified work area within the application program. Subsequent FETCH commands will retrieve data from the next record in turn until all records in the Workspace Table have been accessed. Each SELECT command issued by an application program will cause subsequent FETCH commands to begin accessing records in the Workspace Table at the first record. The FETCH command is coded as follows:

```
CALL IDBMS('FETCH',<status>,<target list string>,<work area>)
```

where:

- <status> is a binary integer variable which, upon return from the Integrated Data Base Management System, will contain an integer value indicating whether or not the command was executed successfully. As in the <status> argument in other commands, a code of zero indicates successful execution while a positive code indicates unsuccessful execution and defines the error condition. However, for the FETCH command, a negative status code may be returned indicating an end-of-table condition.

- <target list string> is an alphanumeric literal or variable which defines the data fields in the Workspace Table for which values are to be retrieved and returned to the application program in the work area. The data field names in the <target list string> argument must be separated by commas. The order of the data field names will determine the order in which the corresponding data values will be stored in the work area. The target list string must be terminated by the special character #.

- <work area> is a variable which defines a contiguous area of main storage into which data values from the Workspace Table will be stored. The work area must be large enough to contain the data values corresponding to the data fields specified in the <target list string> argument.

5.2.3.3 The LOCK Command

The LOCK command permits an application program to gain processing control over a table. The table specified in the LOCK command must be contained in the data base named in the most recent ATTACH command issued by the application program. Two modes of processing control are available to an application program: read and modify. If an application program specifies a read mode lock for a table, other application programs and interactive users can read the contents of the table but can not modify them. If an application program specifies a modify mode lock for a table, no other application programs or interactive users can access the contents of the table in any way. Once a lock is set by an application program, it can be reset only by the UNLOCK command, by the EXIT command or if the application program terminates abnormally prior to issuing either of these commands. The LOCK command is coded as follows:

```
CALL IDBMS('LOCK',<status>,<table name>,<mode>)
```

where <status> is as previously defined and:

- <table name> is an alphanumeric literal or variable which contains the name of the table for which the lock is to be set.

- <mode> is an alphabetic literal or variable which defines the type of lock to be set. The <mode> argument has only two valid values: READ and MODIFY. There meanings are as follows:

  READ - the application program intends to read the contents of the table. No other user should be permitted to update the contents of the table while this lock is active. If another application program already has set a READ lock on the table, this READ lock will also be set on the table. If another application program has set a MODIFY lock on the table, this READ lock will be rejected.

  MODIFY - the application program intends to modify the contents of the table. No other user should be permitted to access the contents of the table in any way. If another application program already has set either a READ or MODIFY lock on the table, this MODIFY lock will be rejected. If no lock of any kind has been set on the table, this MODIFY lock will be set on the table.

5.2.3.4 The UNLOCK Command

The UNLOCK command releases processing control over a table which was established by the application program via a previous LOCK command. The table referenced by the UNLOCK command must be contained in the data base named in the most recent ATTACH command issued by the application program. This command resets both READ and MODIFY locks, whichever type of lock had been last set for the table by the application program. The UNLOCK command is coded as follows:

```
CALL IDBMS('UNLOCK',<status>,<table name>)
```

where <status> is as previously defined and:

- <table name> is an alphanumeric literal or variable which contains the name of the table for which the lock is to be reset.

5.2.3.5 The GET Command

The GET command retrieves data values from a record in a table in a data base and makes them available to the application program for processing. The table referenced by the GET command must be contained in the data base named in the most recent ATTACH command issued by the application program. The GET command follows the logical ascending sequence imposed on the table by a B-tree index to determine which record should be accessed. The particular B-tree index to be used, should more than one exist for a table, is defined by specifying its associated key name in the argument list of the command. If no B-tree indices exist for a table, the GET command can not be used to retrieve data from that table.

To support the GET command, the Integrated Data Base Management System maintains a cursor for each B-tree key field such that access to records in the table through a B-tree index can be based on the current position of the cursor. A cursor is a logical pointer which moves through a table following the logical sequence imposed by the B-tree index with which it is associated. A cursor may be set at the record in a table which is associated with the lowest key value in the B-tree index or at any other record by specifying the key value associated with that record in the argument list of the GET command. Each time a GET command is issued, the specified data values are retrieved from the record containing the next highest key value and the cursor associated with the B-tree index being used is logically positioned by the system to the record accessed. All cursors associated with B-tree indices for a table are independent from one another. Thus, a GET command which causes one cursor to move will not change the position of any other cursor.

Since the GET command retrieves records from a table based

on a cursors set on a B-tree index, no updates to the associated table which might modify the B-tree index can be permitted while an application program is issuing GET commands. Thus, before an application program issues a GET command to access a table, a LOCK command must have been successfully executed by the application for that table. The lock mode specified in the LOCK command can be either READ or MODIFY, but need only be READ to disallow updates to the table. After all GET commands have been issued for the table, the UNLOCK command should be issued. If a table named in a GET command has not been locked by the application program issuing it, the GET command will be rejected.

The GET command is coded as follows:

```
CALL IDBMS('GET',<status>,<table name>,<key name>,<key area>,
          <target list string>,<work area>)
```

where:

- <status> is a binary integer variable which, upon return from the Integrated Data Base Management System, will contain an integer value indicating whether or not the command was executed successfully. As in the <status> argument in other commands, a code of zero indicates successful execution while a positive code indicates unsuccessful execution and defines the error condition. However, for the GET command, a negative status code may be returned indicating an end-of-table condition. Additionally, the <status> argument is used to indicate that this GET command will set a starting point for retrieval of records via the B-tree index. When the status contains a negative value upon issuing the GET command, the contents of the key area will be used to set the cursor associated with the B-tree index identified by the <key name>.

- <table name> is an alphanumeric literal or variable which contains the name of the table from which data values are to be retrieved.

- <key name> is an alphanumeric literal or variable which defines the B-tree index whose cursor is to be used to

determine from which record the data fields specified in the <target list string> argument will be retrieved. The key name must have been previously specified interactively in an INDEX command which created a B-tree index on the table named in the <table name> argument.

- <key area> is a variable which defines a contiguous area of main storage into which the values associated with the key field specified in the <key name> argument will be stored. Should the <key name> argument represent a combination key field, values from each of the fields which constitute the combination key field will be stored in the key area. Additionally, the <key area> argument is used to establish a starting point for retrieval within the specified B-tree index. If the <status> argument contains a negative value, the contents of the key area will be used to set the starting point for retrieval, in that the record accessed will be the one with a matching key or, if no matching key exists, the one with the next highest key. The cursor associated with the specified B-tree index will be logically positioned at the accessed record and subsequent GET commands will retrieve records with the next higher key in the B-tree index.

- <target list string> is an alphanumeric literal or variable which defines the data fields in the table named in the <table name> argument for which values are to be retrieved and returned to the application program in the work area. The data field names in the <target list string> argument must be separated by commas. The order of the data field names will determine the order in which the corresponding data values will be stored in the work area. The target list string must be terminated by the special character #.

- <work area> is a variable which defines a contiguous area of main storage into which data values from the table specified in the <table name> argument will be stored. The work area must be large enough to contain the data values corresponding to the data fields specified in the <target list string> argument.

5.3 Data File Commands

Interactive commands in this category permit an application program to move entire data files from an off-line to an on-line storage device and to create a backup on magnetic tape of an on-line data file. Additionally, an application program

can transform a table to a data file and a data file to a table. Each of the commands which can be issued by an application program are described below.

5.3.1 The COPY Command

The COPY command permits an application program to transform a table to a data file and a data file to a table. The direction of transformation is controlled by the contents of the <did> argument. The COPY command is coded as follows:

```
CALL IDBMS('COPY',<status>,<did>,<table name>)
```

where <status> is as previously defined and:

- <did> is a variable containing either the identifier of the data file which is to be transformed into a table or spaces (blanks), which indicate that the table is to be transformed into a new data file. If a new data file is to be created from the contents of the table, upon return from the Integrated Data Base Management System, the <did> argument will contain the identifier assigned to the newly created data file.
- <table name> is an alphanumeric literal or variable which contains the name of the table which will receive the contents of the data file or from which the new data file is to be created.

5.3.2 The LOAD Command

The LOAD command permits an application program to create an on-line data file in system standard format from a data file on magnetic tape. For a discussion of system standard formats, see Section 7, Data File Handling. If the contents of the on-line data file are exactly the same as that of the off-line data file, they will have the same data identifier. If the contents of the two data files are different after loading, a new data identifier is assigned to the on-line data file. The LOAD command is coded as follows:

```
CALL IDBMS('LOAD',<status>,<old did>,<new did>,<parameter array>)
```

where <status> is as previously defined and:

- <old did> is a variable containing the identifier of the off-line data file on magnetic tape which is to be copied on-line. The data file will be converted to one of the system standard formats during the loading process, if necessary.

- <new did> is a variable which, upon return from the Integrated Data Base Management System, may contain the data identifier of a new on-line data file in system standard format if it contains, in some form, a subfile of the original off-line data file. If the original data file is copied on-line in its entirety, no new data identifier is assigned to it and the <new did> argument will contain spaces (blanks) upon return to the application program. If some operation is performed during loading, as defined by the <parameter array> argument, that causes the contents of the on-line data file to be different from those of the off-line data file, a new data identifier will be assigned to the on-line data file by the system and will be returned in the <new did> argument.

- <parameter array> is a one-dimensional array variable whose values will be a function of the data file being copied on-line. The parameter values will control the loading in that they will permit window and subset operations to be performed as the data file is being loaded. See Section 4, The Interactive Command Language, for a more thorough discussion of the parameters required to control a LOAD operation.

5.3.3 The UNLOAD Command

The UNLOAD command permits an application program to create a backup copy on magnetic tape of an on-line data file. The UNLOAD command is coded as follows:

```
CALL IDBMS('UNLOAD',<status>,<did>)
```

where <status> and <did> are as previously defined.

5.4 Data File Processing Operations

There will be (at least) five operations which will manipulate whole data files. These operations will be available to interactive users via the PERFORM command, but they will be more directly available to application programs. These operations will take one or two data files, plus certain parameters, as input and create a new data file as output. The input data file(s) must have been loaded on-line beforehand either interactively or by the application program. The new data file will also be an on-line file in system standard format. It will be marked as a temporary data file and will be opened as a side effect of its creation and will be closed before control is returned to the application program. The data identifier assigned to the newly created data file will be returned to the application program. The new data file can be opened and read by the application program but cannot be overwritten.

The five operations will be:

SLICE - Create a new data file by taking a two-dimensional slice of a multi-dimensional gridded file.

WINDOW - Create a new data file by extracting a rectangular subarea from an image, cartographic, or two-dimensional gridded file.

SUBSET - Create a new data file by extracting only a specified subset of physical variables from the original file.

REGRID - Create a new data file by interpolating the data from a gridded file in one coordinate system into a new coordinate system.

MERGE - Form a single file by merging the data from two other files.

A precise description of the above operations may be found in Section 4, The Interactive Command Language.

5.4.1 Performing a SLICE Operation

The slice operation creates a new data file from an existing data file by taking two dimensions from the n-dimensional grid (n > 2) of the original data file. The two dimensions to be extracted are defined within the argument list along with the remaining n - 2 dimensions and an array of constants which, together, define a set of n - 2 equations of the form DIMENSION = CONSTANT . Each equation defines a hyperplane through one of the remaining axes, the intersection of which defines the two-dimensional plane (slice) with the desired set of axes. The SLICE operation is coded as follows:

```
CALL IDBMS('SLICE',<status>,<old did>,<new did>,<axis array>,<constant array>)
```

where <status> is as previously defined and:

- <old did> is a variable containing the data identifier associated with the existing multi-dimensional gridded data file from which a two dimensional slice is to be extracted.
- <new did> is a variable which, upon return from the Integrated Data Base Management System, will contain the data identifier of the two-dimensional gridded data file created as a result of the SLICE operation.
- <axis array> is a one-dimensional array variable whose values define the axes of the grid from which the slice is to be taken. The acceptable values for the elements are X, Y, Z, T and L which represent the two earth coordinates, altitude, time and wavelength, respectively. The first two elements of this array must be the two axes of the extracted slice while the remaining n - 2 elements are the remaining axes of the grid. They must match, one-for-one, the n - 2 constants specified in the <constant array> argument.
- <constant array> is a one-dimensional floating point array variable whose values must match the axes in the 3rd through n^{th} elements of the <axis array> argument to form the equations which define the hyperplanes.

5.4.2 Performing a WINDOW Operation

The WINDOW operation extracts a rectangular subarea from an existing data file to create a new data file. The input data file can be an image file, a cartographic terrain elevation model or a two-dimensional gridded file. The new data file created by the WINDOW operation will be of the same type as the original. For the purpose of performing a WINDOW operation, a data file appears to contain records numbered from 1 to N where N is the number of records in the data file. Each record in the data file appears to contain fixed length fields (e.g., pixels) numbered from 1 to M where M is the number of fields in each record. The length of a field is defined in the header record for all system standard formatted data files. Thus, the WINDOW operation is performed by specifying beginning and ending record numbers and field numbers to define the subarea. In addition to providing the capability of extracting a contiguous subarea from an existing data file, the WINDOW operation also permits sampling of the entire data file or a subarea to create a new data file. By specifying a record step size, j, which is greater than one, every j^{th} record can be selected within any defined subarea. Similarly, by specifying a field step size, k, which is greater than one, every k^{th} field can be extracted from each selected record to create the new data file. The WINDOW operation is coded as follows:

```
CALL IDBMS('WINDOW',<status>,<old did>,<new did>,<1st record>,<last record>,
        <record step size>,<1st field>,<last field>,<field step size>)
```

where <status>,<old did> and <new did> are as previously defined and:

- <1st record> is a binary integer variable whose value indicates the first record from which data fields will be extracted to form the subarea. If it contains zero,

the subarea will begin with the first data record in the data file. If the <1st record> argument contains a positive integer, i , the subarea will begin with the i^{th} data record in the data file. If the first record exceeds the number of records in the data file, the WINDOW operation will be rejected.

- <last record> is a binary integer variable whose value indicates the last record from which data fields will be extracted to form the subarea. If it contains zero, the last data record in the data file will be the last record in the subarea. If the <last record> argument contains a positive integer, i , the subarea will end with the i^{th} data record in the data file. If the last record exceeds the number of records in the data file, the last data record in the data file will be the last record in the subarea. If the first record exceeds the last record, the WINDOW operation will be rejected.

- <record step size> is a binary integer variable whose value indicates the sampling interval to be used for data records to create the subarea. If it is zero, all records beginning with the record specified in the <1st record> argument and ending with the record specified in the <last record> argument will be used to create the subarea. If the <record step size> contains a positive integer, j, only every j^{th} record beginning with the record specified in the <1st record> argument will be used. The last record used will be the j^{th} record not exceeding the record specified in the <last record> argument.

- <1st field> is a binary integer variable whose value indicates the first field in each data record to be extracted to form the subarea. If it contains zero, the subarea will begin with the first field in each selected data record. If the <1st field> argument contains a positive integer, i , the subarea will begin with the i^{th} field in each selected data record. If the first field exceeds the number of fields in the data records, the WINDOW operation will be rejected.

- <last field> is a binary integer variable whose value indicates the last field in each data record to be extracted to form the subarea. If it contains zero, the last field in each selected data record will be the last field in each record in the subarea. If the <last field> argument contains a positive integer, i , the i^{th} field in each selected data record will be the last field in each record in the subarea. If the last field exceeds the number of fields in the data records, the last field in each data record will be the last field in each record in the subarea. If the first field exceeds the last field, the WINDOW operation will be rejected.

- <field step size> is a binary integer variable whose value indicates the sampling interval to be used for fields to create the subarea. If it is zero, all fields beginning with the field specified in the <1st field> argument and ending with the field specified in the <last field> argument will be extracted from the selected records to create the subarea. If the <field step size> argument contains a positive integer, k, only every k^{th} field will be extracted from every selected record beginning with the field specified in the <1st field> argument. The last field used in each selected record will be the k^{th} field not exceeding the field specified in the <last field> argument.

5.4.3 Performing a SUBSET Operation

The SUBSET operation extracts the value of one or more physical variables at each point in a data file containing gridded data to create a new data file. The input data file must contain data on an n-dimensional (n = 2, 3, or 4) grid. The new data file will contain data on the same grid as the original data file. The SUBSET operation permits specific variables to be extracted from the original data file based on their relative position in the vector of data values at each point in the grid. As an example, consider a three-dimensional gridded data file (the dimensionality of a gridded data file is defined in its header record) where the three dimensions are longitude, latitude and altitude. At each grid point, a vector exists which contains values of wind velocity, wind direction, temperature and pressure in that order. To create a new data file containing only temperature and pressure data at each grid point, the integer array defined by the <variable array> argument must contain 3 and 4, indicating that the third and fourth data values in the vector at each grid point are to be extracted. The resulting data file will contain the same longitude, latitude and altitude at each grid point contained in the original file as well as the temperature and pressure values at each grid point. The SUBSET operation is coded as follows:

```
CALL IDBMS('SUBSET',<status>,<old did>,<new did>,<variable array>)
```

where <status>, <old did> and <new did> are as previously defined and:

- <variable array> is a one-dimensional binary integer array variable whose values indicate the relative position in the vector at each grid point from which variables are to be extracted. The array identified by the <variable array> argument must be dimensioned at least one greater than the number of variables to be extracted at each grid point. Each element of the array must contain a positive integer indicating the relative position in the grid point vector of the variable to be extracted. The element immediately following the last element in the array which defines the relative position of variables to be extracted, must contain zero as an array terminator.

5.4.4 Performing a REGRID Operation

The REGRID operation creates a new gridded data file from an existing gridded data file. A scale factor and/or a translation can be applied to the existing grid points to obtain the new grid points while one of several interpolation schemes can be used to obtain the values of the variables at each of the new grid points. Separate scaling factors and translations can be applied to each axis in the coordinate system of the original gridded data file. An array is used in the argument list to contain both the scaling factors and the translations. The number of entries in each array will depend upon the number of coordinate axes (not considering wavelength as an axis) in the grid (the number and type of axes for a gridded data file are defined in its header record). The i^{th} entry in each array will correspond to the i^{th} axis of the multi-dimensional grid, where the order of the axes is defined by the following total ordering:

$$T < X < Y < Z$$

As an example, consider a three-dimensional gridded data file where the three dimensions, or axes, are time (T), longitude (X) and latitude (Y). Then the first element in both the scale factor and translation arrays would contain the scaling factor and translation, respectively, to be applied to the T axis (time), the second elements would contain the scaling factor and translation for the X axis (longitude) while the third elements would contain the scaling factor and translation for the Y axis (latitude). Note, that no dummy elements need be supplied for the non-existent Z axis. The REGRID operation is coded as follows:

```
CALL IDBMS('REGRID',<status>,<old did>,<new did>,<scale factor array>,
        <translation array>,<interpolation indicator>)
```

where <status>,<old did> and <new did> are as previously defined and:

- <scale factor array> is a one-dimensional binary floating point array variable whose values define the scaling factor for each of the dimensions or axes of the grid. The scale factor array must contain one scale factor for each axis of the grid contained in the data file. Each scale factor must be greater than zero. A scale factor of one indicates no scaling of the grid points along the corresponding axis.

- <translation array> is a one-dimensional binary floating point array variable whose values define the offset of the origin of the new grid from that of the original grid. The translation array must contain one translation value for each axis of the grid contained in the data file. A translation value of zero indicates no translation for the corresponding axis.

- <interpolation indicator> is a binary integer variable whose value indicates the interpolation scheme (e.g., linear, cubic spline, etc.) to be used to obtain the values of the variables at the new grid points.

5.4.5 Performing a MERGE Operation

The MERGE operation combines the contents of two gridded

data files into a single, new gridded data file. Both of the input data files must be in system standard format and must be defined over the same grid. That is, both grids should be defined with respect to the same set of coordinate axes. Additionally, the grid spacing for all axes must be the same for both grids. If the origin and termination of the axes for both grids do not match, the merged data file will contain a grid consisting only of the overlapping portions of the original grids. If no portion of the two input grids overlap, the MERGE operation will be aborted. The vector of data values at each grid point in the new data file will be formed by concatenating the values from the corresponding grid point in the first data file with those from the corresponding grid point in the second data file. The MERGE operation is coded as follows:

CALL IDBMS('MERGE',<status>,<old did_1>,<old did_2>,<new did>)

where <status> and <new did> are as previously defined and:

- <old did_1> is a variable containing the identifier of one of the input data files which must be a gridded data file.
- <old did_2> is a variable containing the identifier of the second input data file which must be a gridded data file. The data identifier contained in the <old did_2> argument must be different from that in the <old did_1> argument.

5.5 Examples of the Use of "Interactive" Commands

As an example of the use of "interactive" commands and supporting operations in an application program, consider a table named EVENT with data fields NAME, CLASS, STRTDATE, ENDDATE, STARTLAT, STARTLON, ENDLAT, ENDLON and STRENGTH which describes a series of storms under study. The EVENT table is contained within a data base named STORMS. Also,

```
      REAL*8 PGMID,USERID,DBNAME,ALIAS,TABLE,DID,DATE(2),BLANKS
      INTEGER RECNO,STATUS
      DATA DBNAME/'STORMS'/,TABLE/'EVENT'/,ALIAS/'E'/,KEY/'NAME'/,
     *BLANKS/' '/,PGMID/'TEST'/
C     READ USER-ID AND PASSWORD OF USER RUNNING PROGRAM AND
C     START AND DATES FOR RETRIEVAL
      READ(5,10) USERID,PASSWD,DATE
   10 FORMAT(A6,1X,A4 /, 2A6)
C     CONNECT TO THE INTEGRATED DATA BASE MANAGEMENT SYSTEM
      CALL IDBMS('ENTER',STATUS,PGMID,USERID,PASSWD)
      IF(STATUS.GT.0) GO TO 9000
C     BIND START AND END DATE NAMES TO DATE ARRAY FOR USE IN STRINGS
      CALL IDBMS('BIND',STATUS,'SDATE   ',DATE(1))
      IF(STATUS.GT.0) GO TO 9000
      CALL IDBMS('BIND',STATUS,'EDATE   ',DATE(2))
      IF(STATUS.GT.0) GO TO 9000
C     ATTACH TO STORMS DATA BASE FOR PROCESSING
      CALL IDBMS('ATTACH',STATUS,DBNAME)
      IF(STATUS.GT.0) GO TO 9000
C     ASSIGN ALIAS NAME,E, TO EVENT TABLE IN STORMS DATA BASE
      CALL IDBMS('USE',STATUS,ALIAS,TABLE)
      IF(STATUS.GT.0) GO TO 9000
C     RETRIEVE DATA FIELDS FROM EVENT TABLE INTO WORKSPACE TABLE
      CALL IDBMS('SELECT',STATUS,RECNO,'E.NAME,E.STRENGTH#',
     * 'E.STRTDATE GT 770531 AND E.ENDDATE LT 770901#')
      IF(STATUS.GT.0) GO TO 9000
C     CALL SUBROUTINE TO READ AND PRINT ANY DATA IN WORKSPACE TABLE
      IF(RECNO.GT.0) CALL WORKRD
C     READ EVENT TABLE SEQUENTIALLY USING INDEX ON NAME FIELD
      CALL SEQRD(TABLE,KEY)
C     ADD NEW RECORD TO EVENT TABLE
      CALL IDBMS('INSERT',STATUS,RECNO,TABLE,'NAME=DORA,CLASS=HURR,
     * STRTDATE=SDATE,ENDDATE=EDATE,STARTLAT=20.5,ENDLAT=41.7,
     * STARTLON=300,ENDLON=309.7#')
      IF(STATUS.GT.0) GO TO 9000
C     COPY CURRENT CONTENTS OF EVENT TABLE TO A DATA FILE
C     SET DATA IDENTIFIER (DID) TO BLANKS TO INDICATE COPY
C     FROM TABLE TO NEW DATA FILE
      DID=BLANKS
      CALL IDBMS('COPY',STATUS,DID,TABLE)
      IF(STATUS.GT.0) GO TO 9000
C     DISCONNECT FROM THE INTEGRATED DATA BASE MANAGEMENT SYSTEM
 9999 CALL IDBMS('EXIT',STATUS)
      IF(STATUS.GT.0) WRITE(6,9010) STATUS
      STOP
C     ERROR HANDLER
 9000 WRITE(6,9010) STATUS
 9010 FORMAT('0',10X,'IDBMS ERROR - STATUS CODE = ',I4)
      GO TO 9999
      END
```

Figure 5-1: Using the Application Program Command Language

```
      SUBROUTINE WORKRD
      DIMENSION NAME(3)
      INTEGER STATUS
      LOGICAL*1 WKAREA(16)
      EQUIVALENCE (NAME(1),WKAREA(1)),(STREN,WKAREA(13))
C   RETRIEVE DATA FROM WORKSPACE TABLE
   10 CALL IDBMS('FETCH',STATUS,'NAME,STRENGTH#',WKAREA)
      IF(STATUS) 9999,20,9000
   20 WRITE(6,30) NAME,STREN
   30 FORMAT('0',6X,3A4,5X,A4)
      GO TO 10
C   ERROR HANDLER
 9000 WRITE(6,9010) STATUS
 9010 FORMAT('0',10X,'FETCH ERROR - STATUS CODE = ',I4)
 9999 RETURN
      END
```

```
      SUBROUTINE SEQRD(TABLE,KEY)
      DIMENSION NAME(3)
      REAL*8 TABLE,KEY
      INTEGER STATUS
      LOGICAL*1 WKAREA(20)
      EQUIVALENCE (CLASS,WKAREA(1)),(SLAT,WKAREA(5)),(ELAT,WKAREA(9)),
     * (SLON,WKAREA(13)),(ELON,WKAREA(17))
C   LOCK EVENT TABLE PRIOR TO SEQUENTIAL RETRIEVAL
      CALL IDBMS('LOCK',STATUS,TABLE,'READ')
      IF(STATUS.GT.0) GO TO 9000
C   SET KEY RETURN ARGUMENT TO BINARY ZEROS AND STATUS NEGATIVE TO
C   START READING AT RECORD IN EVENT TABLE CONTAINING LOWEST NAME
      DO 10 I=1,3
        NAME(I)=0
   10 CONTINUE
      STATUS=-1
C   RETRIEVE FIELDS FROM RECORD IN EVENT TABLE WITH NEXT HIGHEST NAME
   20 CALL IDBMS('GET',STATUS,TABLE,KEY,NAME,'CLASS,STARTLAT,ENDLAT,
     * STARTLON,ENDLON#',WKAREA)
      IF(STATUS) 9999,30,9000
   30 WRITE(6,40) NAME,CLASS,SLAT,SLON,ELAT,ELON
   40 FORMAT('0',10X,3A4,3X,A4,2(5X,2(F6.2,2X)))
      GO TO 20
C   ERROR HANDLER
 9000 WRITE(6,9010) STATUS
 9010 FORMAT('0',10X,'IDBMS ERROR - STATUS CODE = ',I4)
C   UNLOCK EVENT TABLE
 9999 CALL IDBMS('UNLOCK',STATUS,TABLE)
      IF(STATUS.GT.0) WRITE(6,9010) STATUS
      RETURN
      END
```

Figure 5-1 (Continued): Using the Application Program Command Language

the EVENT table has a B-tree index associated with it which has been defined on the data field NAME. Excerpts from an application program which accesses the EVENT table are shown in Figure 5-1.

5.6 Commands Which Access Data Files

The Application Program Command Language contains several commands which are available only to application programs (and not to interactive users). These commands provide an application program with the ability to access and create data files. The nine commands in this category are:

OPEN	-	open an existing data file for input or a new data file for output.
CLOSE	-	close a data file.
READ	-	copy a data record (or portion thereof) into an indicated work area.
WRITE	-	output a data record (or portion thereof) from an indicated work area.
SEARCH	-	scan a data file record-by-record to locate a specific character string.
GETHEAD	-	fetch the header record from a data file in system standard format.
PUTHEAD	-	output a header record to a new data file.
GETHIST	-	fetch a history record from a data file in system standard format.
PUTHIST	-	output a history record to a new data file.

The last four commands will not be accepted by the Integrated Data Base Management System unless the data file referenced by the operation is known to be in a system standard format. The following subsections describe each of the commands for handling data files and gives the calling sequence for each.

5.6.1 The OPEN Command

The OPEN command logically connects a data file to an application program for processing. If the <access mode> argument indicates that a new data file is to be created, the resultant file will always be a temporary on-line file, regardless of the format. The owner of this file will be taken from the user-id in the ENTER command by which the application program became connected to the system. If an OPEN is issued against a data file which has not been loaded on-line and for which no system standard format version exists then the OPEN operation shall be aborted. The OPEN command is coded as follows:

```
CALL IDBMS('OPEN',<status>,<did>,<format>,<access mode>)
```

where <status> is as previously defined and:

- <did> is a variable which contains the identifier of the data file (if an existing file) or else will receive the identifier assigned to the file (if a new file is being created).
- <format> is an alphanumeric variable whose contents define the system standard format in which a new data file is to be written or which receives the format type of an existing data file which is to be read.
- <access mode> is an alphabetic literal or variable which defines the way in which the application program intends to access the data file being opened. The <access mode> argument has three valid values: INPUT, OUTPUT and OUTIN. Their meanings are as follows:

  INPUT - The application program intends to read an existing data file identified by the data identifier specified in the <did> argument. The format of the existing data file will be returned in the <format> argument.

  OUTPUT - The application program intends to create a new data file. The format in which the new data file is to be written is defined by the <format> argument. The data identifier

assigned to the new data file by the system will be returned in the <did> argument.

OUTIN - The application program intends to create a new data file and also modify the new data file. The <format> and <did> arguments are processed as described for the OUTPUT mode.

5.6.2 The CLOSE Command

The CLOSE command logically disconnects a data file from an application program. If the data file being closed is a new data file, an entry is created for the data file and is inserted in the Data File Catalog. The CLOSE command is coded as follows:

```
CALL IDBMS('CLOSE',<status>,<did>)
```

where <status> and <did> are as previously defined.

5.6.3 The GETHEAD Command

The GETHEAD command causes a header record from a data file in one of the system standard formats to be retrieved and returned to the application program. The GETHEAD command can be issued at any time following an OPEN command for the data file and preceding a CLOSE command. However, if the GETHEAD command follows one or more GETHIST or READ commands, the data file is repositioned by the system at the physical beginning of the data file prior to attempting to read the header record. The GETHEAD command will be rejected if the data file is not in one of the system standard formats. The GETHEAD command is coded as follows:

```
CALL IDBMS('GETHEAD',<status>,<did>,<work area>)
```

where <status> and <did> are as previously defined and:

<work area> is a variable which defines a contiguous area of main storage within the application program into which the header record will be stored. The work area must be large enough to contain the entire header record.

5.6.4 The GETHIST Command

The GETHIST command causes a processing history record from a data file in one of the system standard formats to be retrieved and returned to the application program. The initial GETHIST command issued for a data file will return the first processing history record, if any exist, to the application program. Subsequent GETHIST commands will return succeeding processing history records to the application program in the order in which they appear in the data file. If a GETHIST command follows a READ command, the system will position the data file to the second record before attempting to retrieve the processing history record. It is not necessary that any GETHIST commands be issued by an application program. It is possible for an application program to open a data file that is in system standard format, process the data file and close it without ever retrieving a processing history record. The GETHIST command will be rejected if the data file is not in one of the system standard formats. The GETHIST command is coded as follows:

```
CALL IDBMS('GETHIST',<status>,<did>,<work area>)
```

where <did> and <work area> are as previously defined and:

- <status> is a binary integer variable which, upon return from the Integrated Data Base Management System, will contain an integer value indicating whether or not the command was executed successfully. As in the <status> argument for other commands, a code of zero indicates successful execution while a positive code indicates unsuccessful execution and defines the error condition. However, for the GETHIST command, a negative status

code may be returned indicating the end of the processing history records has been encountered. A negative status code value will be returned for the first GETHIST command issued if the data file contains no processing history records.

5.6.5 The PUTHEAD Command

The PUTHEAD command causes a header record to be written on a new data file. A PUTHEAD command must be issued prior to issuing a PUTHIST or WRITE command for the data file. Subsequent PUTHEAD commands can be issued by the application program. They will overwrite the existing header record if this capability is supported by the operating system for the peripheral storage device on which the data file is being written. The PUTHEAD command is coded as follows:

```
CALL IDBMS('PUTHEAD',<status>,<did>,<work area>)
```

where <status> and <did> are as previously defined and:

- <work area> is a variable which defines a contiguous area of main storage within the application program. Prior to issuing the PUTHEAD command, the header record must be constructed in the work area.

5.6.6 The PUTHIST Command

The PUTHIST command causes a processing history record to be written on a new data file. Unlike the PUTHEAD command, all processing history records must be written before a single data record is written, and any PUTHIST commands issued after a WRITE command will be rejected by the system. Each PUTHIST command will cause precisely one processing history record to be written on the data file, immediately after the previous history record. The PUTHIST command is coded as follows:

```
CALL IDBMS('PUTHIST',<status>,<did>,<work area>)
```

where <status> and <did> are as previously defined and:

- <work area> is a variable which defines a contiguous area of main storage within the application program. Prior to issuing the PUTHIST command, the processing history record must be constructed by the application program in the work area. Alternatively, the work area might be one which was specified in a previous GETHIST command thus causing the transfer of an exsiting processing history record from an existing data file to a new data file.

5.6.7 The READ Command

The READ command causes a data record, or portion thereof, from a data file to be retrieved and returned to the application program. The READ command need not be preceded by either a GETHEAD or GETHIST command. It can be issued for data files in one of the system standard formats or in their original data file format. To an application program, the data records in a data file appear to be numbered sequentially from 1 to N where N is the number of data records in the data file. The data record retrieved by a READ command is a function of the integer value placed in the <record no> argument described below. All, or any part, of the retrieved record can be returned to the application program. This is controlled by the <start> and <length> arguments described below. The READ command is coded as follows:

```
CALL IDBMS('READ',<status>,<did>,<record no>,<start>,<length>,<work area>)
```

where <did> is as previously defined and:

- <status> is a binary integer variable which, upon return from the Integrated Data Base Management System, will contain an integer value indicating whether or not the command was executed successfully. As in the <status> argument in other commands, a code of zero indicates successful execution while a positive code indicates unsuccessful execution and defines the error condition.

However, for the READ command, a negative status code may be returned if, in an attempt to position the data file to the data record specified by the <record no> argument, the end-of-file is encountered.

- <record no> is a binary integer variable whose value indicates the relative position of the data record to be retrieved from the data file. If it contains zero, the record retrieved will be the data record immediately following the last data record retrieved, except for the initial READ command, in which case, the first data record in the data file will be retrieved. If the <record no> argument contains a positive integer, the data file will be positioned, either forward or backward, to that data record prior to retrieving the data record. A negative value in the <record no> argument will cause the READ command to be rejected.

- <start> is a binary integer variable which indicates the first byte in the retrieved data record that is to be returned to the application program. If the <start> argument contains zero, the first byte returned in the work area will be the first byte in the retrieved record. If it contains a positive integer, i , the first byte returned in the work area will be the ith byte in the data record. If the start byte exceeds the number of bytes in the data record, the READ command will be rejected.

- <length> is a binary integer variable which indicates the number of bytes to be returned to the application program. If the <length> argument contains zero, the remainder of the retrieved record, beginning with the start byte, is returned. If the length plus the start byte exceeds the length of the data record, the portion of the data record beginning with the start byte and going to the end of the record will be returned.

- <work area> is a variable which defines a contiguous area of main storage within the application program into which the retrieved data record, or portion thereof, is stored. The portion of the data record returned in the work area is defined by the <start> and <length> arguments. The size of the work area must be greater than or equal to the number of bytes specified by the <length> argument unless it is zero. In which case, its size must be greater than or equal to (record length - <start> + 1).

5.6.8 The WRITE Command

The WRITE command causes a data record, or portion thereof, to be written to a new data file. The initial WRITE command issued by an application program must be preceded by a PUTHEAD command but need not be preceded by any PUTHIST commands. However, if any processing history records are to be placed in the new data file, they must all be written, using the PUTHIST command, before the first WRITE command is issued. As with the READ command, the data records in a new data file appear to be numbered sequentially from 1 to N where N is the number of data records written at any given time during the execution of the application program. The placement of the data record to be written is a function of the integer value placed in the <record no> argument described below. Thus, existing data records can be overwritten by a WRITE command. However, this can only occur for a data file which is in the process of being created by the application program and only prior to the first CLOSE command issued by the application program. All, or any part, of a data record can be written to a data file. Any portion of a new data record which is not provided by the application program will contain binary zeros when the data record is placed in the data file. Any portion of an existing data record which is being overwritten and is not provided by the application program will contain the original contents of the existing data record. The portion of a data record transferred from an application program by a WRITE command is controlled by the <start> and <length> arguments described below. The WRITE command is coded as follows:

```
CALL IDBMS('WRITE',<status>,<did>,<record no>,<start>,<length>,<work area>)
```

where <status> and <did> are as previously defined and:

- <record no> is a binary integer variable whose value indicates the relative position in the data file where the data record, or portion thereof, is to be written. If it contains zero, the record written will immediately follow the last command written, except for the initial WRITE command, in which case, the record written will immediately follow the header record or the last processing history record. A zero in the <record no> argument may cause an existing data record to be overwritten if a preceding WRITE command positioned the data file such that sequential writing of data records would cause existing data records to be overwritten. If the <record no> argument contains a positive integer which is greater than N+1, the WRITE command will be rejected. If it contains a positive integer less than or equal to N, the existing data record at that relative position in the data file will be overwritten.

- <start> is a binary integer variable which indicates the first byte in the data record into which data from the application program is to be stored. If the <start> argument contains zero, the first byte in the work area will be stored in the first byte of the data record. If it contains a positive integer, i , the first byte in the work area will be stored in the i^{th} byte of the data record. If the start byte exceeds the number of bytes in the data record, the WRITE command will be rejected.

- <length> is a binary integer variable which indicates the number of bytes to be stored in the data record. If the <length> argument contains zero, data is stored in the remainder of the record, beginning with the start byte. If the length plus the start byte exceeds the length of the data record, data are stored in that portion of the record beginning with the start byte and going to the end of the record.

- <work area> is a variable which defines a contiguous area of main storage within the application program from which data are transferred to a data record. The portion of the data record to which the contents of the work area are transferred is defined by the <start> and <length> arguments. The data to be transferred must be placed into the work area by the application program prior to issuing the WRITE command. The size of the work area must be greater than or equal to the number of bytes specified by the <length> argument, unless it is zero. In which case, its size must be greater than or equal to (record length - <start> + 1).

5.6.9 The SEARCH Command

The SEARCH command initiates a record-by-record scan of a data file, or portion thereof, to locate particular data values. The SEARCH command need not be preceded by any other command except an OPEN command for the data file to be searched. The SEARCH command scans only data records and can be issued for data files in one of the system standard formats or in their original data file format. As with the READ and WRITE commands, the data records in a data file appear to be numbered sequentially from 1 to N where N is the number of data records in the data file. The SEARCH operation will begin with the record immediately following that designated in the <record no> argument. Each data record will be retrieved in turn and the portion of the retrieved record defined by the <start> and <length> arguments will be compared, as defined by the <comparison operator> argument, with the contents of the work area. When the result of the comparison is true, the SEARCH operation will terminate and the record number of the record satisfying the comparison will be returned in the <record no> argument. It should be noted that the same contiguous string of bytes from each data record retrieved will be used in the comparison, as determined by the <start> and <length> arguments. For example, if the <start> argument contains 6 and the <length> argument contains 4, then bytes 6, 7, 8 and 9 (and only those bytes) from each record will be compared with the contents of the work area. The SEARCH command is coded as follows:

```
CALL IDBMS('SEARCH',<status>,<did>,<record no>,<start>,<length>,
                <comparison operator>,<work area>)
```

where <did> is as previously defined and:

- <status> is a binary integer variable which, upon return from the Integrated Data Base Management System, will contain an integer value indicating whether or not the command was executed successfully. As in the <status> argument in other commands, a code of zero indicates successful execution while a positive code indicates unsuccessful execution, and defines the error condition. However, for the SEARCH command, a negative status code will be returned if an end-of-file was encountered before a data record was found for which the comparison was true.

- <record no> is a binary integer variable whose value indicates the relative position of the data record after which the SEARCH operation will begin. If it contains zero, the SEARCH operation will begin with the first data record in the data file. If it contains one, the SEARCH operation will begin with the second record and so on. If a data record is found for which the comparison is true, the record number of that record will be returned in the <record no> argument. Since the SEARCH operation begins with the data record immediately following that specified by the input value of the <record no> argument, a search can be continued following a successful comparison by using the value returned in the <record no> argument as the input value for the next SEARCH command. If an end-of-file is encountered during the SERACH operation, the contents of the <record no> argument will not be modified.

- <start> is a binary integer variable which indicates the first byte in each data record that is to be compared with the contents of the work area in the application program. If the <start> argument contains zero, the comparison will begin with the first byte in each data record. If it contains a positive integer, i , the first byte that is compared is the i^{th} byte in each data record. If the start byte exceeds the number of bytes in each data record, the SEARCH command will be rejected.

- <length> is a binary integer variable which indicates the number of bytes in each data record to be compared. If the <length> argument contains zero the remainder of each record, beginning with the start byte, is compared with the contents of the work area. If the length plus the start byte exceeds the length of each data record, the portion of each data record beginning with the start byte and going to the end of the record will be compared.

- <comparison operator> is a two byte alphabetic literal or variable which defines the comparison operation to be performed between that poriton of each retrieved

data record defined by the <start> and <length> arguments and the contents of the work area in the application program. The valid comparison operators are EQ, NE, LT, LE, GT and GE.

- <work area> is a variable which defines a contiguous area of main storage within the application program whose contents are compared with all or a portion of the contents of each retrieved data record. The data to be compared must be placed into the work area by the application program prior to issuing the SEARCH command. The size of the work area must be greater than or equal to the number of bytes specified by the <length> argument, unless it is zero. In which case, its size must be greater than or equal to (record length - <start> + 1).

5.7 Miscellaneous Commands

Commands in this category do not fit easily into any of the previous categories of commands. Currently, only one command is included in this category, the FORMAT command. However, other commands may be added to this category as required.

5.7.1 The FORMAT Command

The FORMAT command permits an application program to determine which copies of a data file currently exist and in what format. By accessing the Data File Catalog, this command will indicate whether or not an original off-line version of the data file exists and, if so, in what format; whether or not an on-line version exists and, if so, in what system standard format; and whether or not an off-line, backup version exists in system standard format. The FORMAT command is coded as follows:

```
CALL IDBMS('FORMAT',<status>,<did>,<original copy>,<on-line copy>,
           <backup copy>)
```

where <status> and <did> are as previously defined and:

- <original copy> is a variable which, upon return from the Integrated Data Base Management System, will contain an indication of whether or not an original copy of the data file exists on magnetic tape and, if so, in what format. If no such copy exists, spaces (blanks) will be returned. Otherwise, a character string indicating the original data file format in which the tape file exists will be returned.

- <on-line copy> is a variable which, upon return from the Integrated Data Base Management System, will contain an indication of whether or not a copy of the data file in system standard format exists on a direct access device and, if so, in which system standard format. If no such copy exists, spaces (blanks) will be returned. Otherwise, a character string indicating the system standard format (e.g., gridded, image, etc.) in which the on-line copy exists will be returned.

- <backup copy> is a variable which, upon return from the Integrated Data Base Management System, will contain an indication of whether or not a copy of the data file in system standard format exists on magnetic tape and, if so, in what format. If no such copy exists, spaces (blanks) will be returned. Otherwise, a character string indicating the system standard format in which the backup copy exists will be returned. If both an on-line copy and a backup copy exist, the returned contents of this argument will match that of the <on-line copy> argument, since both copies must be in the same system standard format.

SECTION 6 - THE PHYSICAL STORAGE OF TABULAR DATA

6.1 The Tabular Data Storage Area

All data managed by the "front end" of the Integrated Data Base Management System will be logically organized into tables. These tables may have one or more indices associated with them. All tables and indices maintained by the relational front end of the system must be stored on one or more direct access devices (drums, disks, data cells, etc.). All direct access space which has been allocated and initialized at system generation time for the storage of tables and their associated indices is referred to as the tabular data storage area. The tabular data storage area is subdivided into physical pages and a page map is constructed at system generation time which relates physical pages to a specific direct access device.

6.2 Physical Pages

The basic unit of storage for data managed by the "front end" of the Integrated Data Base Management System will be the physical page -- a fixed size block of bytes capable of being rolled into or out of main memory with a single I/O command. In certain computers (e.g., DEC's PDP-11) the physical page size is predetermined by the machine architecture. Where there are no constraints imposed by the mainframe architecture itself, the main considerations in choosing a physical page size are (1) that there should be an integral number of pages per track on the direct access device and (2) if the direct access device is such that a track is divided into sectors, there should be an integral number of sectors per page. Within these two constraints, the normal desire will be to make the physical page size as large as possible to cut down I/O requests during record-at-a-time processing. However, it must be recognized by the system implementors that when the page size is too large the number of physical pages which can reside in main memory simultaneously will be limited and that this

can lead to thrashing* problems for multiple users. Notice that the size of the physical page is determined solely by hardware considerations and not by data-related considerations, such as typical record size.

Physical pages will be associated with a unique physical page id identifying its location on disk. There are two approaches which can be used:

(1) Assign consecutive numbers to physical page locations following some order and use an auxiliary table or system function to map physical page id's into direct access device addresses (and, incidentally, to do some checking for validity).
(2) Create a physical page id directly from the address, for example, by concatenating disk pack number, cylinder number, track number, and sector number, if used.

A choice between these two alternatives will have to be made at implementation time. The primary tradeoff will be between number of bytes required to store a physical page id vs. extra time and core required for page id decoding, and these cannot be analyzed until machine and on-line mass storage specifications are known.

One physical page id which will not be used is zero. If, in the numbering scheme, it makes sense to have a "zero-th" page, that page will be reserved for system use only and will not be used to hold data. Hence, a physical page id of zero can be used as a "null pointer" to indicate the end of a linked list.

6.3 Managing Mass Storage

Depending upon the sophistication of the operating system on which this system is implemented, it may or may not be

*"Thrashing" describes a condition where throughput has degenerated due to a higher demand for pages in core than can be accommodated.

advantageous to make use of the machine's own file handing system to manage mass storage. By far and away the easiest approach would be to have each relation implemented as a single on-line file, where the files are maintained by the machine's operating system. Such a file would be constrained by the DBMS software to grow or shrink one physical page at a time, and an addressing scheme based on physical page id within file would have to be designed. This in turn would have an impact on the numbering scheme for selecting physical page id's.

In the absence of precise knowledge as to the operating system under which the DBMS will be implemented, the conservative decision is to assume that the DBMS will have to handle its own disk management. The DBMS can view the physical pages as being of two types: free or in use. Those pages which are in use are linked to a Relation Control Block and those which are free are also chained together in a last-in-first-out singly-linked list, as depicted in Figure 6-1. To reserve a free page, the system simply removes the first page from the chain, while liberating a given page merely requires saving a link to the first free page and changing the head pointer to point to this new page. The system will have to be initialized during system generation by linking all pages in the tabular data storage area together.

6.4 Buffers and the Buffer Control Table

The existence of fixed-size physical pages requires fixed-size buffers in main memory to hold these pages. These buffers will be managed by a buffer control table. There will be one entry in the table for each buffer in main memory and each entry will contain the following fields:

(1) page number - the physical page id of the page in this buffer.

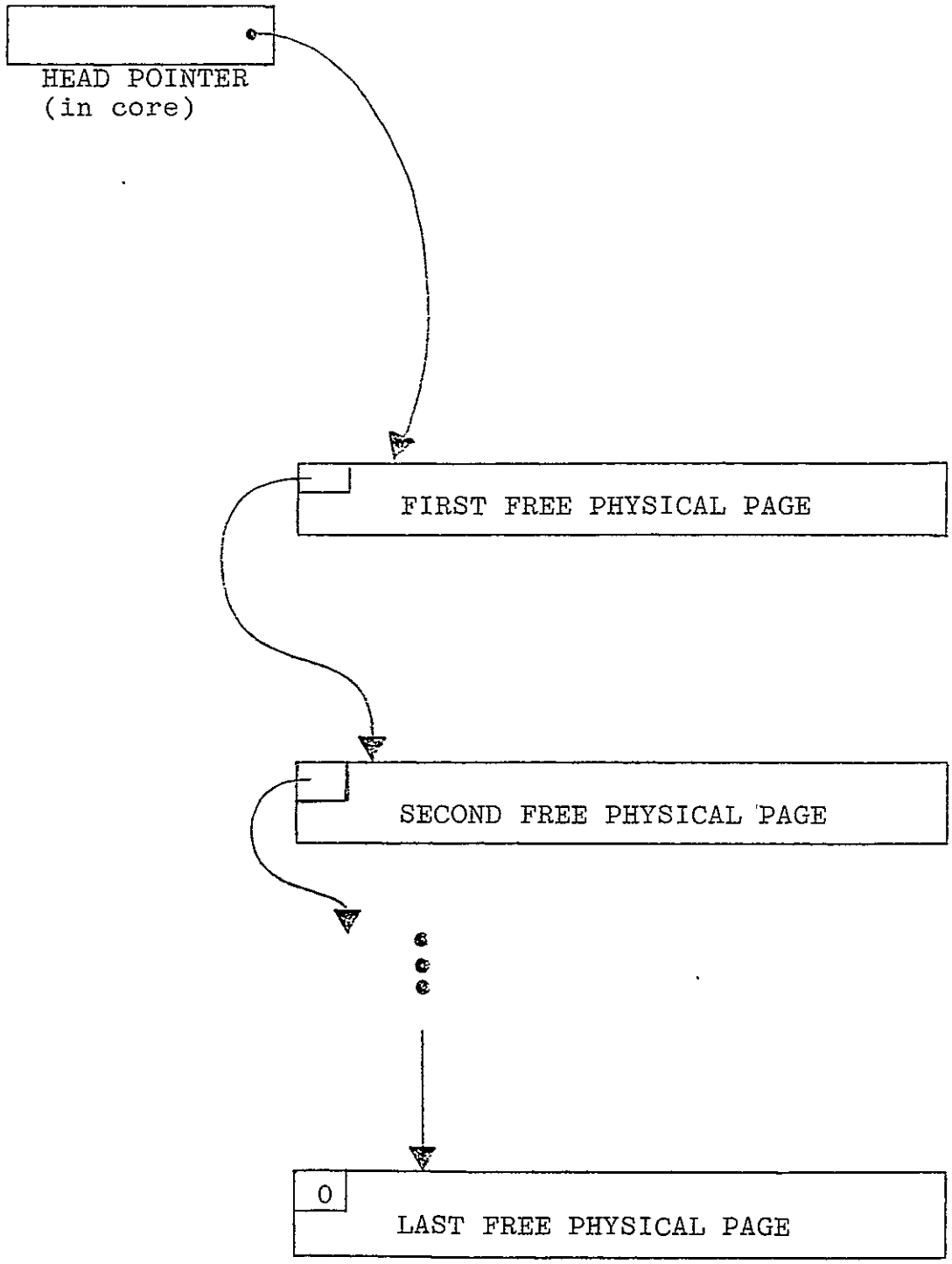


Figure 6-1: Linked List of Free Pages

(2) update flag - indicates whether the contents of the page have been altered.
(3) hold flag - indicates whether the user expects to have further need of this page or not.
(4) LRU counter - used to identify the least-recently used physical page in main memory.
(5) user id - of the user currently accessing this page.

When an active user wishes to access a physical page, the system begins by searching the buffer control table to determine whether that page is already in core. If so, then the user turns on the "hold" flag and begins processing immediately. If not, then the system must select a buffer and roll the desired page into that buffer. The first choice for a buffer is one which is empty (an unlikely event after the first few seconds the system is up). If there are no empty buffers, then the system must select one of the buffers and prepare it for use. The selection criteria (counting empty buffers as a "zero-th" level) is:

(1) hold flag off, update flag off	(00)
(2) hold flag off, update flag on	(01)
(3) update flag off, hold flag on	(10)
(4) both flags on	(11)

The buffer which fits into the lowest numerical category is selected. Where two or more buffers tie as candidates for swap-out, then the least-recently used among the candidates is chosen. For example, if there is more than one buffer in category two but no empty buffers and no buffers in category one, then the least-recently used buffer in category two is chosen. There is, however, one caveat -- no buffer should be swapped out if its user id matches that of the user requesting a page and its hold flag is on.

The LRU counter works as follows. The system stores a master counter. Every time a buffer is accessed, this master counter is incremented and copied into the LRU counter for that buffer. Then, the least-recently used page is that page with the smallest counter. If the master counter overflows then remedial action may be taken, e.g., divide all counters, including the master counter, by two. This would be more rapid than subtraction (since it can be done by right shifting each counter one bit) although counter adjustment will normally occur about twice as often.

6.5 The Structure of Tables

6.5.1 Storing Records on a Physical Page

There is one basic rule which guides the entire design of the physical file structure: no record shall be split across physical page boundaries. One immediate consequence of this rule is that a certain amount of space at the end of a physical page might be wasted -- space left over which is too small to contain another record and which, therefore, is unusable. This wasted space is referred to as "internal fragmentation" and it can be a significant overhead factor when records are large relative to the size of a physical page. However, an approach which allowed records to be split across page boundaries would cause a considerable increase in processing time (since some records would require two page accesses to be read) and in the complexity of the software. Since the wasted fragment must always be smaller than the size of a record, when records are reasonably small relative to the page size the gains in decreased complexity and computation time are ample compensation for the wasted space.

Another corollary of the above rule is that the physical page size will impose a systemic upper bound on the size of a

record. Records which are too large to fit on a single page must be redefined by the user to make them fit (this may result in the user splitting his table into two or more tables). Notice that this test would be performed by the system at the time the table is defined and before physical pages are allocated for record storage or an entry is inserted in the SYSREL table.

The first several bytes in a physical page will be reserved for pointers, two of which are used to chain all of the data pages of an on-line file into a doubly-linked list. The RCB will have the physical page id of the first data page in the list and the final page in the list, and each data page will have the physical page id of its predecessor and successor in the list. The forward pointers facilitate look-ahead buffering when processing the file on a record-by-record basis. When a data page is brought into core, the system can retrieve the page id of the next page in the chain from the forward page pointer, locate a page buffer, and overlap bringing in the next page with processing the current one.

The records are stored in the remainder of a page following the pointers. Each record will be preceded by a bit map showing which fields in the record are null. The record itself will be stored beginning at the next byte boundary after the bit map. Within a bit map a zero will indicate a null field and a one will indicate a non-null field. Since a string of n ones represents the integer (2^n-1). the test for no null fields is straightforward.

Figures 6-2a and 6-2b depict record storage within a physical page and the linked list structure of a table, respectively.

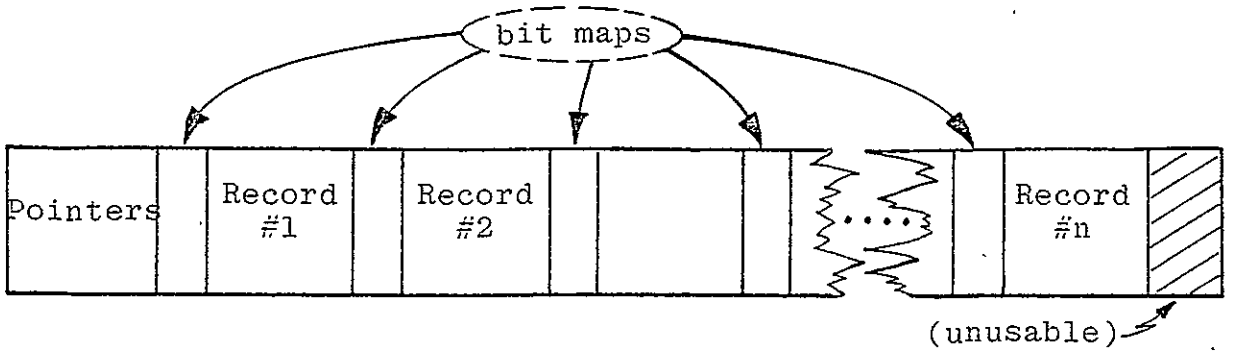


Figure 6-2a: Storage of Records Within a Physical Page

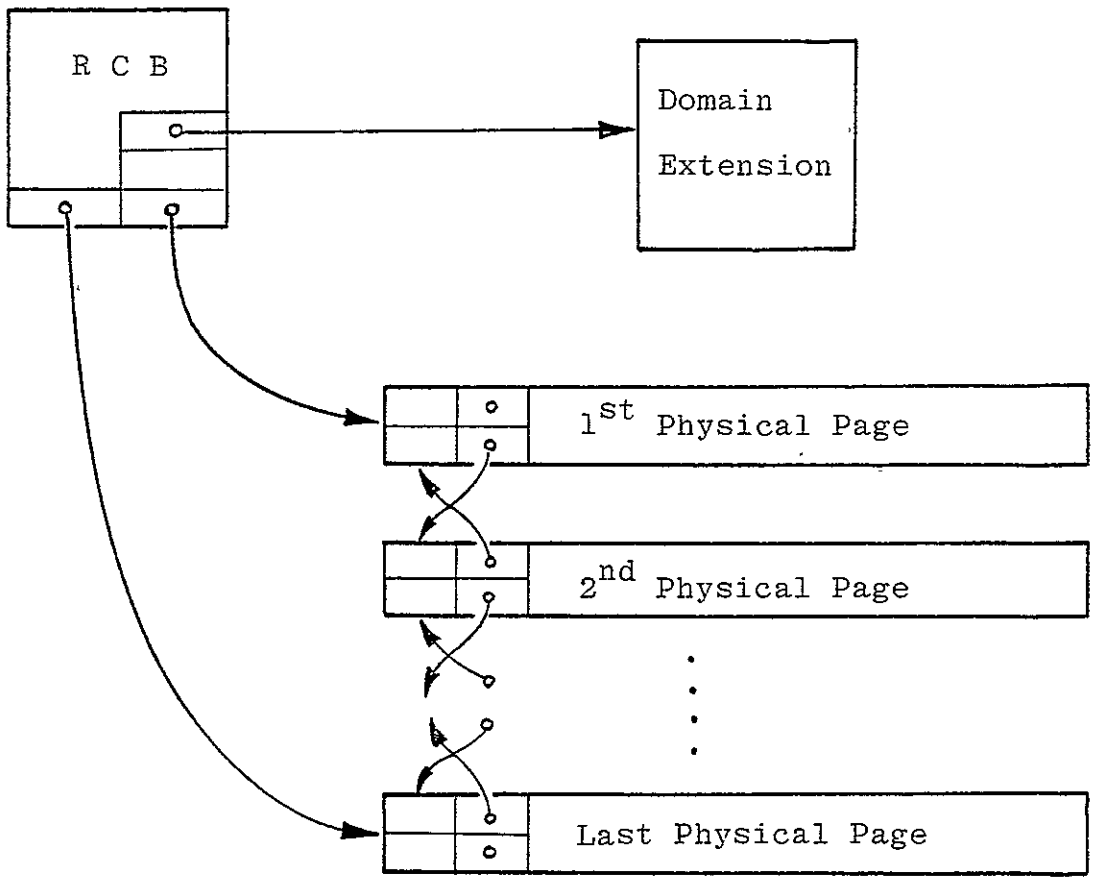


Figure 6-2b: Physical File Structure for Tables

6.5.2 Holes in a Page

The presence of a bit map associated with each record makes it an easy matter to delete a record merely by setting the bit map to all zeros. This creates a "hole" inside a data page, which can be filled during a later insert operation. To save time locating these holes they will be chained together in a linked list within a page, and pages with holes in them will be placed on a last-in-first-out, doubly-linked list. Note that the order of pages in the list of pages with holes in them will not necessarily correspond to the sequence of pages in the data page list.

The within-page list of holes will be ordered on ascending location within the page. This will permit the system to collapse adjacent holes into a single large hole using standard dynamic core allocation algorithms.* Since space within a hole is not otherwise being used, the first couple of bytes can be used to hold the size of the hole and a pointer to the next hole in the chain.

The use of a doubly-linked list makes it relatively easy to delete a page from the list. This can happen in two ways: (1) the last hole in the page has been filled by an insertion or (2) the only record in a page has been deleted.

If we assume that insertions take place only in the first page in the list of pages with holes in them then the double-linked list is slightly inferior to using a singly-linked list. However, deleting an empty page from the list can come anywhere in the list and would be quite expensive without the existence of a back pointer to the page's predecessor (this is why the regular list of data pages is also double-linked, since the empty page must be deleted from that list as well).

---

*Specifically, Algorithms A and B in section 2.5 of Knuth[20].

Figure 6-3 depicts the pointer structure within a table. Note that the RCB needs pointers to both ends of the primary list of data pages since new pages must be added to the end of the list, while the list of pages with holes needs only a head pointer.

6.5.3 Variable-Sized Records

The physical page structure described in this section can be adapted for use with variable-sized records (provided that the records are shorter than a physical page). There will be four major differences:

(1) The size of the wasted fragment at the end of a page will vary, and will normally wind up being treated as a hole.
(2) The size of holes will be more variable.
(3) Insertions will often require multiple probes into the list of pages with holes.
(4) Record size, as well as the bit map, must be stored in the record header.

However, the main outline of the data structure, algorithms for maintaining the two lists of pages, and even the algorithms for maintaining the within-page list of holes can be used unaltered.

6.6 Access Method Superstructures

6.6.1 B-Trees

6.6.1.1 Description

The use of tree-structured indices with two-way decision nodes (i.e., binary trees) appears to have been invented in the

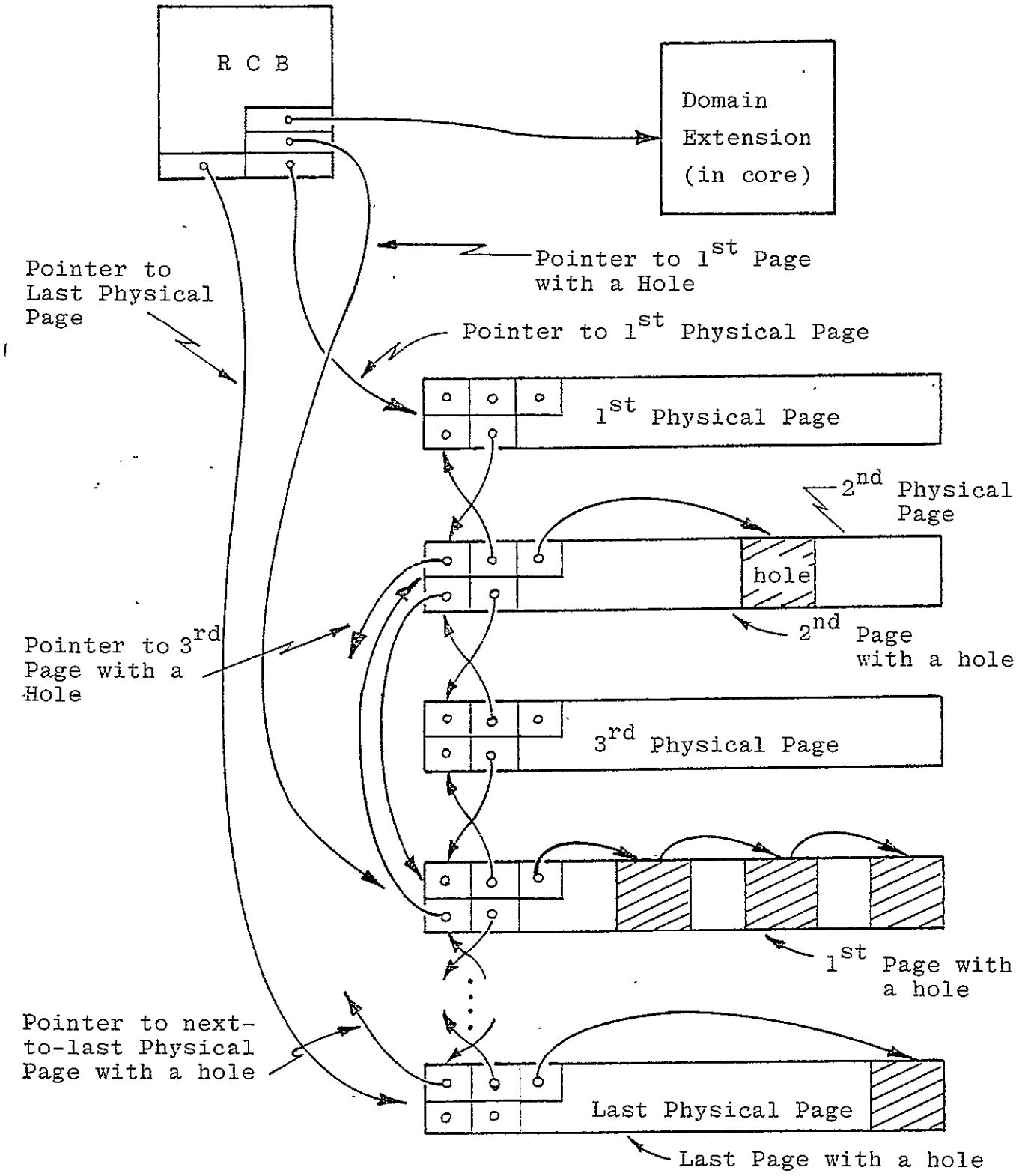
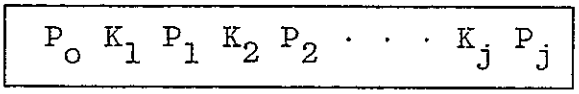


Figure 6-3: Pointer Structure of a Table

1950's*. More recently this notion has been generalized to three-way decision nodes and from these to m-way decisions[1]. An m-way decision tree is called a "B-tree" and it is formally defined as follows:

(1) Every node has m or fewer sons.
(2) Every node - except the root and the leaves - has at least $\frac{m}{2}$ sons.
(3) All leaves are on the same level and have no sons.
(4) The root has at least 2 sons (unless it is a leaf).
(5) A non-leaf node with k sons has $k-1$ keys.

A node with $j+1$ pointers $P_0, P_1, \ldots, P_j$ and j keys $K_1 < K_2 < \ldots < K_j$ can be depicted as:



To search for a key K in the above node, simply test K against K_i for $i = 1, 2, \ldots, j$. If $K = K_i$ then we are done, otherwise if $K < K_i$ search for K in the node whose address is P_{i-1}. Finally, if $K > K_j$ go search the node whose address is P_j. If the above node is a leaf, so that the pointers are null, then either K will equal K_i for some i or else K is not in the file.

B-tree indices lend themselves to large paged files where both the index and the file must be stored on a direct access device. B-trees are quite efficient for search purposes, since the number of disk accesses required to locate a key will be less than or equal to the number of levels in the tree. A worst case analysis of the maximum number of levels L in an m-ary tree

*A history can be found in section 6.2.2 of Knuth[21].

as a function of the number of records (N) and m is only moderately difficult to compute:[21]

$$L < 1 + (\log_2 N - 1)/(\log_2 m - 1)$$

In other words, with m as small as 32 ($=2^5$) it is possible to locate a single record out of two million with at most five disk accesses.

Not only are B-trees efficient for searching, they are also easy to update. Inserting a new key and pointer into a less than full node is a simple matter of shifting keys and pointers already in the node to perserve the ordering of the keys. If the node is full (i.e., the node contains m-1 keys) then the node must be split to make room for the new key and pointer. Let K' be the middle key of the m keys (counting the new one). Then an unused node P' is fetched and all keys and pointers to the right of K' are moved into P' and K' and P' are inserted into the father of P . This procedure is illustrated in figure 6-4 for m equal to seven. If P has no father (i.e., P is the root), then in order to accomodate the split node a new root containing P , K' , and P' must be created (hence the exception to rule 1 described in rule 4). This adds a level to the tree. It can be shown[21] that the likelihood of any split is less than 2/(m-2).

Deletions are only slightly more difficult than insertions. When a node falls below the minimum size due to a deletion the first step is to examine the node's right brother. If that node is above the minimum, then keys and pointers can be taken from that node to balance the two. If there is no right brother, or if the right brother is also of minimum size, then try to take some keys and pointers from the left brother. If

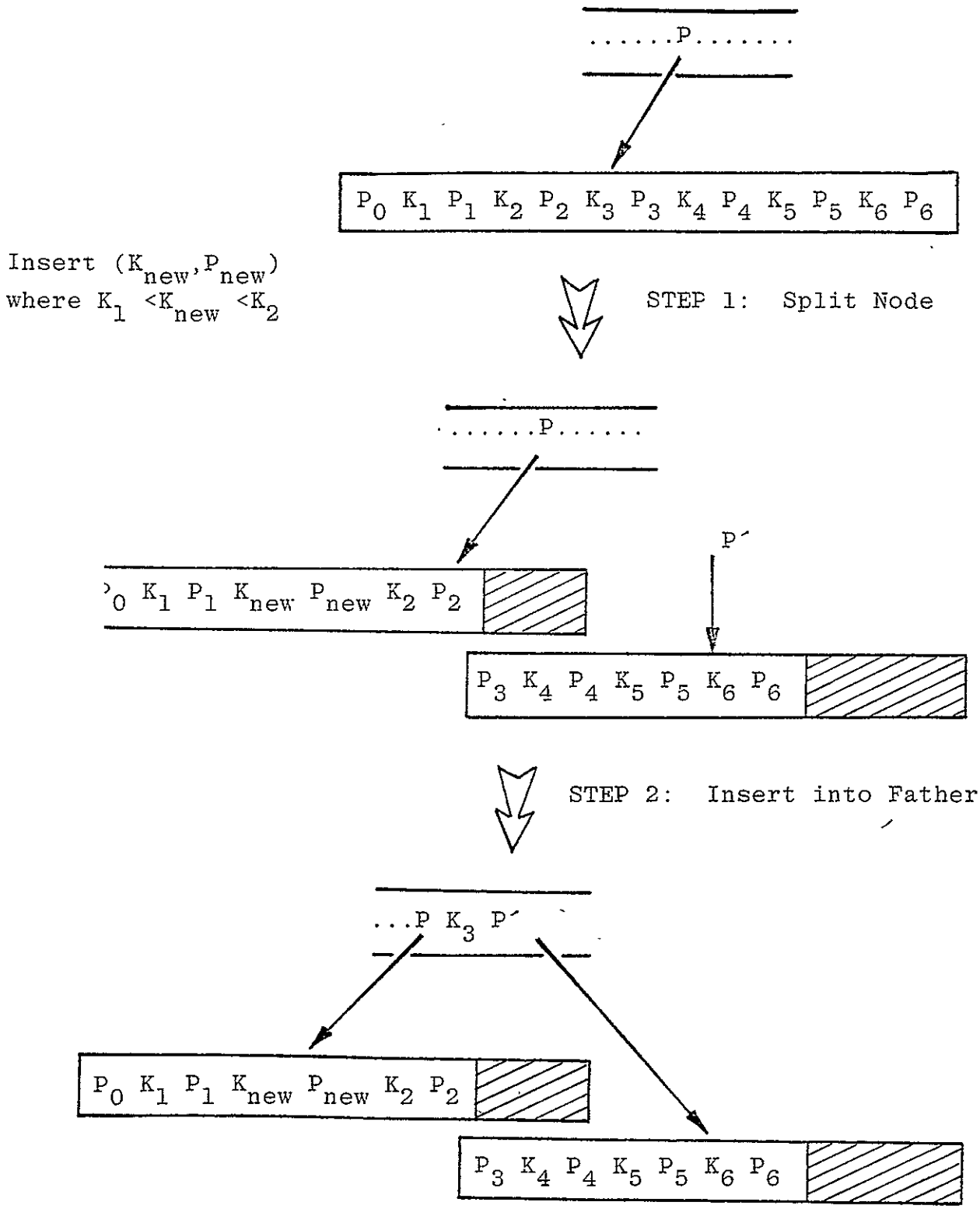


Figure 6-4: Splitting a B-Tree Node During Insertion

that also is impossible, then this node and one brother should be collapsed into a single node and the appropriate key and pointer deleted from the father.

6.6.1.2 Implementation Within the System

The concept of B-trees as outlined within the preceding section will be extended for use within the Integrated Data Base Management System. Specifically, the following conditions will be added to the five which define B-tree structures:

(6) Leaf nodes will contain entries of the form (K,T), where T is the record identifier for the record whose key is K . The maximum number of entries in a leaf may be different from m , though the constraint that a node must always be at least half full will be observed. No pointers (except record id's) will be stored in leaves.

(7) No record id's will be stored in non-leaf nodes, and therefore a search cannot terminate until a leaf is reached.

(8) The keys stored in non-leaf nodes will be the value of the largest key on any leaf which is a descendant of that node.

(9) Leaves will be linked together so as to preserve an ascending key sequence.

Figure 6 -5 depicts a B-tree of order four (i.e., at most three keys per node), where the leaves (not depicted) hold five keys apiece and the keys are the integers 1-90. Notice that the maximum key on each leaf (i.e., every integer between 1 and 90 divisible by 5) is repeated on precisely one non-leaf (or branch) node with the exception of the maximum key value (90 in this case). Fetching the record id for key K is done by walking down the tree following the same search procedure as described

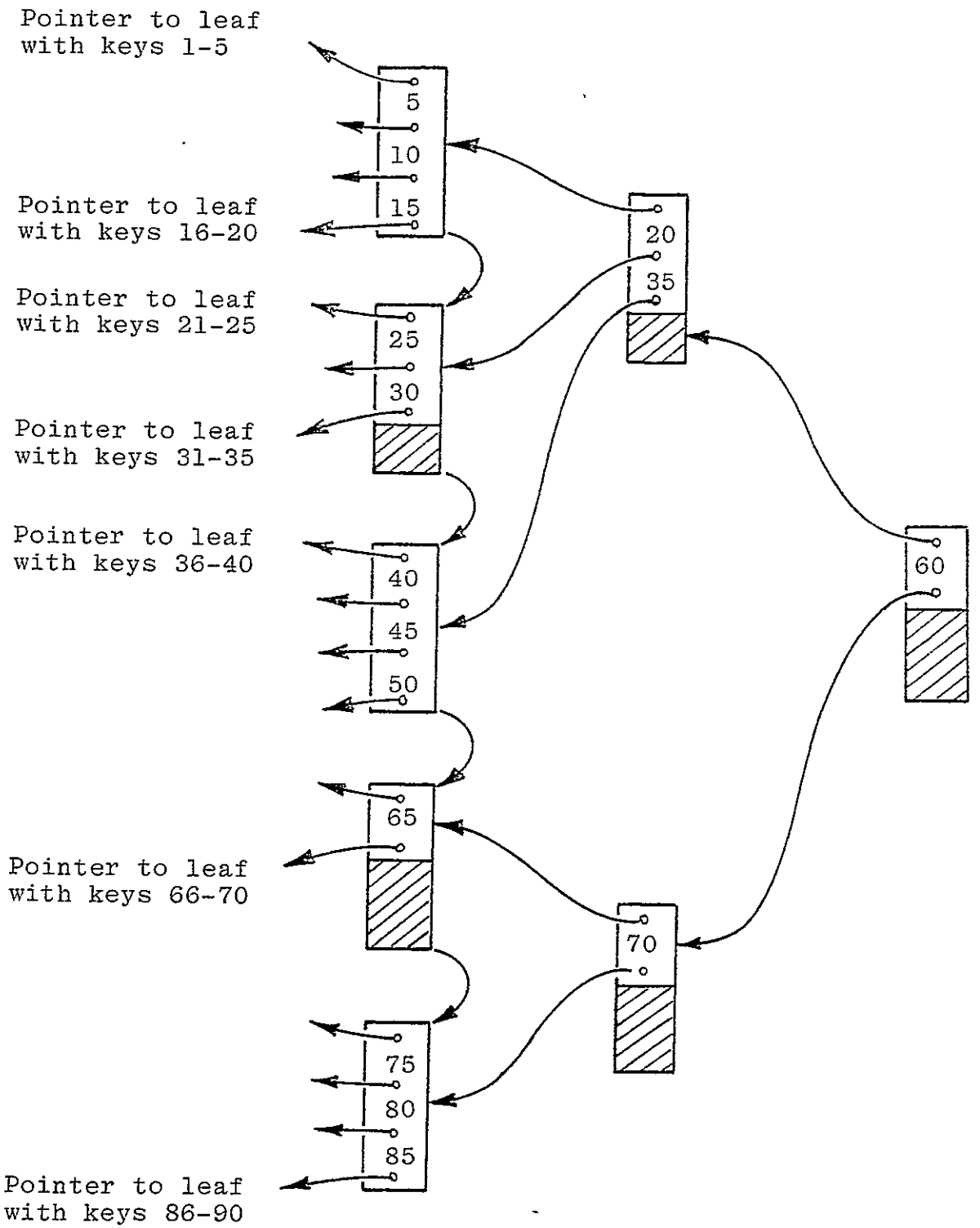


Figure 6-5: A Sample B-Tree of Order 4

previously until the leaf where K is stored has been located, then searching the leaf with a binary or sequential search. Notice that the test for going to node P_{i-1} must be changed from strict inequality to " < ", but that if K is found in a branch node then its position in the leaf will be known in advance. This revised B-tree structure is similar to IBM's VSAM access method,[19] but the two methods are not identical.

The insertion algorithm outlined in the preceeding subsection must also be modified. The first stage is a search to locate the leaf node where the key and record id are to be inserted. If this leaf is full then split it (keep track of which half the key and record id belong in), then insert this new entry. If this key is the largest one on its page, make the appropriate change in the father. If a split occurred then insert the largest key from the left half and the address of the right half into the father using the normal insertion algorithm described previously. Deletion of a key and record id entry from this modified B-tree structure is almost precisely identical to the normal deletion procedure, except that an additional test must be added to handle the case where a leaf is still sufficiently full after the deletion to not warrant shifting of entries, but where the entry deleted is the last one on the leaf. In such a case the copy of that key in the father (or more remote ancestor) must be altered. This is not necessarily as complicated as it may seem, since one need only test for equality of a key match on the way down from the root, and save the node where the match occurred.

6.6.1.3 Enhancements

There are a number of minor improvements which can be made to the basic B-tree structure to enhance performance. For example, some gains could be made by going to a binary search on key values (since keys in a node are in sorted order). An alternative approach would be to use data compaction schemes. Consider the set of keys ROBERT, ROBERTS, ROBERTSON, ROBEY, ROBIN, ROBINETTE, ROBINSON. With four bytes per pointer and

nine bytes per key it will require a total of eighty-eight bytes to store these keys and their eight pointers. But a number of different compaction schemes can cut this dramatically. For example, one could use one byte to hold the length of the preceeding key to be duplicated, another byte to hold the number of bytes for the remainder of this key, then the remainder of the key itself. The above seven keys could be compressed into:

```
0 6 ROBERT    ——→    ROBERT
6 1 S         ——→    (ROBERT)S
7 2 ON        ——→    (ROBERTS)ON
4 1 Y         ——→    (ROBE)Y
3 2 IN        ——→    (ROB)IN
5 4 ETTE      ——→    (ROBIN)ETTE
5 3 SON       ——→    (ROBIN)SON
```

The total storage required would be 33 bytes for the keys plus 32 bytes for the pointers, or 65 bytes all together. Another possible compaction scheme would place the keys into a tree form (figure 6-6) and then linearize the tree with parenthesized notation:

```
(ROB(E(RT(*)(S(*)(ON)))(Y))(IN(*)(ETTE)(SON)))
```

This scheme would require only 22 bytes for the keys, plus space for the begin-end subtree marks. Further analysis of data characteristics would have to be made before specific recommendations on whether to implement key compaction and, if so, what scheme(s) to use could be made. Key compaction would have three major impacts upon "B-tree structure":

(1) Binary search could no longer be used.
(2) The maximum number of keys per page would no longer be fixed (i.e., it could vary from page to page).
(3) Extra execution time would be required to unpack the keys and the software to handle searches and insertions would be more complex.

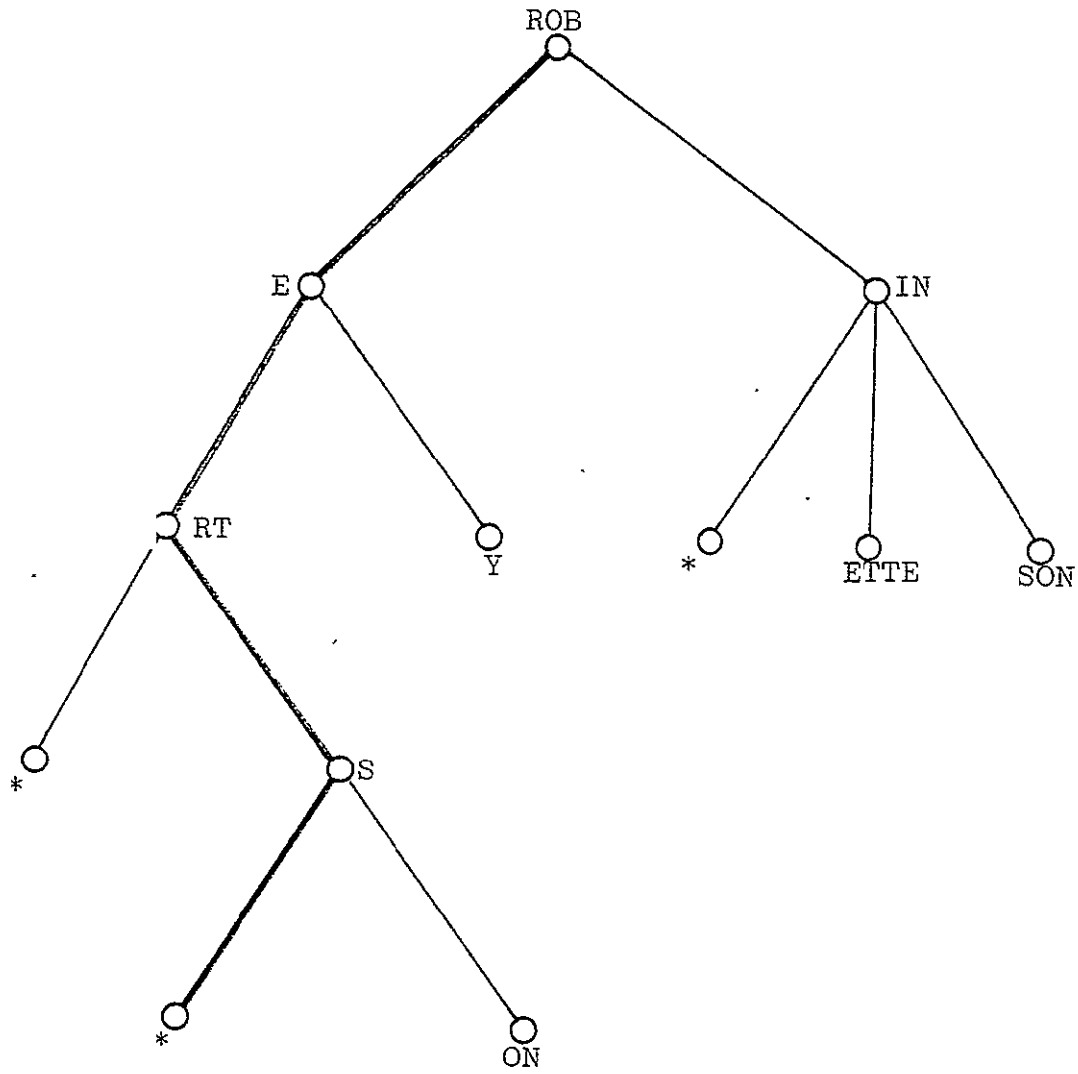


Figure 6-6: A Prefix Tree Compression for Seven Keys
(Heavy Line Indicates "ROBERTS")

It has been noted that there is no *a priori* need for a fixed value of m -- the same insertion, deletion, and search algorithms can be used with the more nebulous rule that each node (other than the root) in the tree should be at least half full. The advantages of key compression would be that fewer pages would be needed to store the same set of keys and this could quite possibly result in fewer levels in the tree (i.e., fewer disk accesses to locate a specific entry). A simulation study[24] addresses the impact of key compaction on an access method superstructure (VSAM) similar to the one proposed here. McCreight[25] also discusses algorithms for handling variable-sized and/or compacted keys.

One particular drawback of B-trees is the possibility that the root will be very small -- it can, after all, have as few as two pointers and a single search key. When this happens an extra disk access can be required just to make a binary decision. This can be avoided by resisting node splitting for the root. One method for doing this is a variant of the B-tree called the B*-tree, which resists node splitting at all levels by preferring to balance nodes between brothers (i.e., passing nodes off to brothers of the overfilled node) and splitting only when the brothers are full. In B*-trees the number of sons range between m and 2/3 m, however this does not necessarily deal with the problems of the root -- a node which by definition has no brothers. Shneiderman[35] suggests allowing the root node to have an overflow page (using P_m to point to the overflow node). In such a scheme the root would not split until it had 2m sons. If such a tree has L levels and the root has m+n sons, then the probability that a search would require L disk accesses is $\frac{m}{m+n}$ while the probability that a search would take L+1 disk accesses is $\frac{n}{m+n}$. By contrast, if the root had been split after the $m+1^{st}$ insertion into the root, then all of the searches would take L+1 disk accesses.

6.6.1.4 Arguments Against B-Trees

A recent article by Stonebraker and Held[17] compared B-trees rather unfavorably to ISAM-like, static tree-structured indices. It is felt that their analysis is incomplete and that many of the arguments advanced by Stonebraker and Held simply do not apply in the anticipated operating environment for the proposed Integrated Data Base Management System. Specifically, Stonebraker and Held are supposing that the files (tables) will have precisely one key and be indexed with precisely one tree structure. The system being designed makes no such suppositions, and needs a file structure capable of handling zero, one, two, ... an arbitrary number of tree-structured keys. Moreover, Stonebraker and Held further suppose that the records can be input initially to the system in sorted sequential order so that the leaves will have their entries in sorted sequential order. By contrast the entries for records in the system being designed will most certainly not come in initially in sorted sequential order. This is one of the true beauties of B-trees. If the leaves are accessed one by one from left-most leaf to right-most, it will be seen that the entries are in sorted sequential order, yet when the entries are inserted it never takes more than m-2 compares and m-2 physical shifts of entries. (Moreover, the sum of shifts and compares is m-1).

Another assumption of the Stonebraker and Held article which will not necessarily hold in the operating environment of the proposed Integrated Data Base Management System is that insertions will come into the system at a steady pace after the initial file creation. Instead, a situation such as depicted in figure 6-7 can be expected. Stonebraker and Held argue that even though later insertions must go into overflow areas, reducing search efficiency, it will be possible to justify periodic file reorganization. It is not clear that the expected

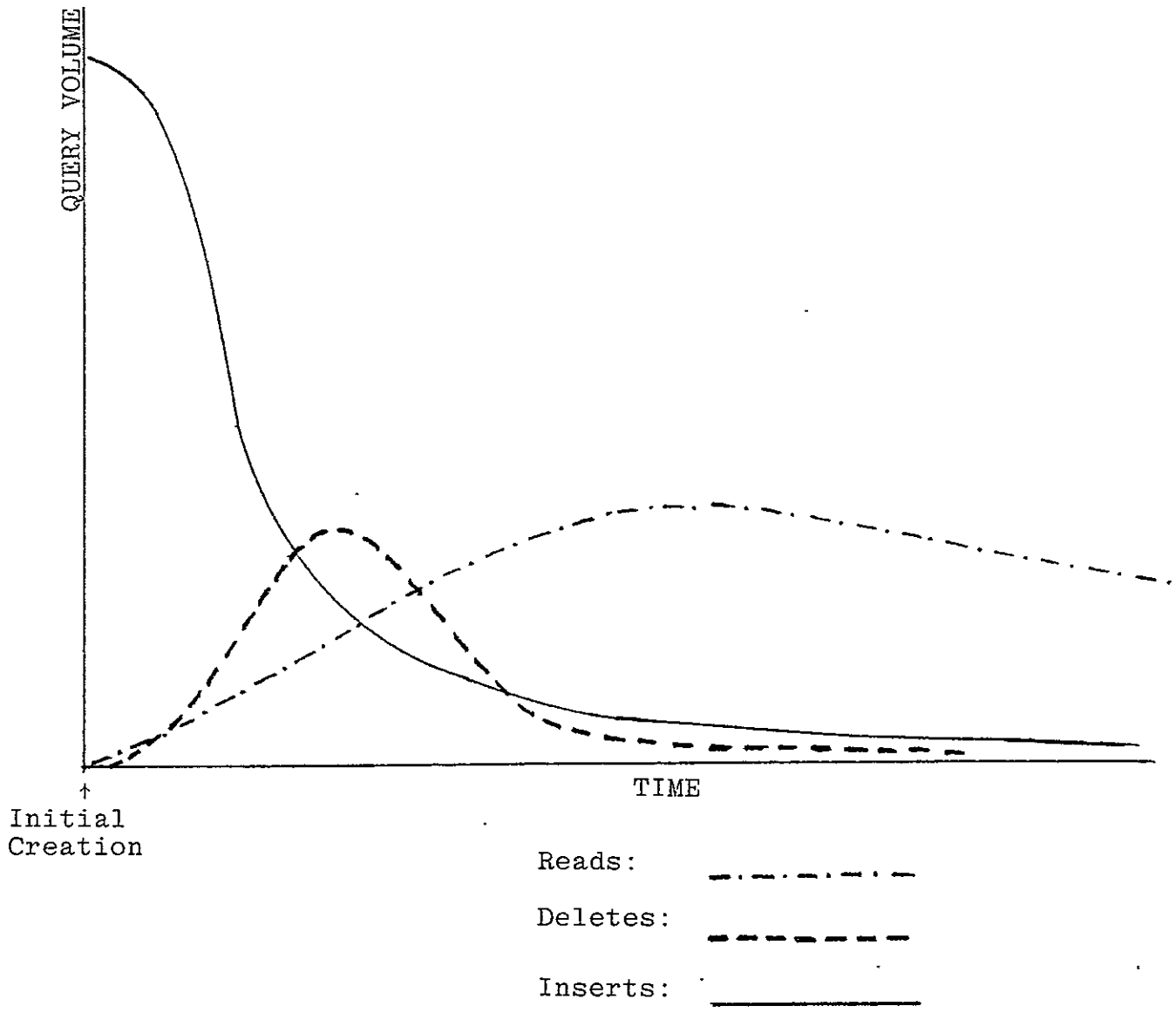


Figure 6-7: Projected Pattern of Usage for Typical Tables in the System (Other than Directories)

pattern of inserts/deletes in the proposed environment will justify the cost of reorganization.

The presence of overflow pages in the static tree structure advocated by Stonebraker and Held are its Achilles heel. Their article compares minimum number of levels for static vs. dynamic tree-structured indices, but what should be compared are expected number of disk accesses for the one against the other.

6.6.2 Inverted Indices

6.6.2.1 Description

Hierarchical data structures (B-trees, binary trees) are quite useful for efficient retrieval of data where the relationship between distinct key values and individual records is 1:1, or nearly so. However, when the ratio of distinct key values to separate records is 1:n for n somewhat greater than one, then a set-oriented data structure is more useful. One of the most efficient data structures for set-oriented indexing operations is the inverted file.

An inverted index for a search key of a table consists of two parts: a domain directory, with one entry for each distinct value the search key adopts in this particular table and a set of index tables, one for each entry in the domain directory. An inverted index is depicted in Figure 6-8. Each entry in the domain directory consists of a search key value and a pointer to an index table which contains a list of record id's of records in the table which have that value for the specified key. For example, to locate all records with the value "C" in a specified data field, one locates the entry corresponding to C in the domain directory and thereby discovers the address of a list of all record id's of records which have the value of C for that data field (i.e., record numbers 2, 7, 8, and 9).

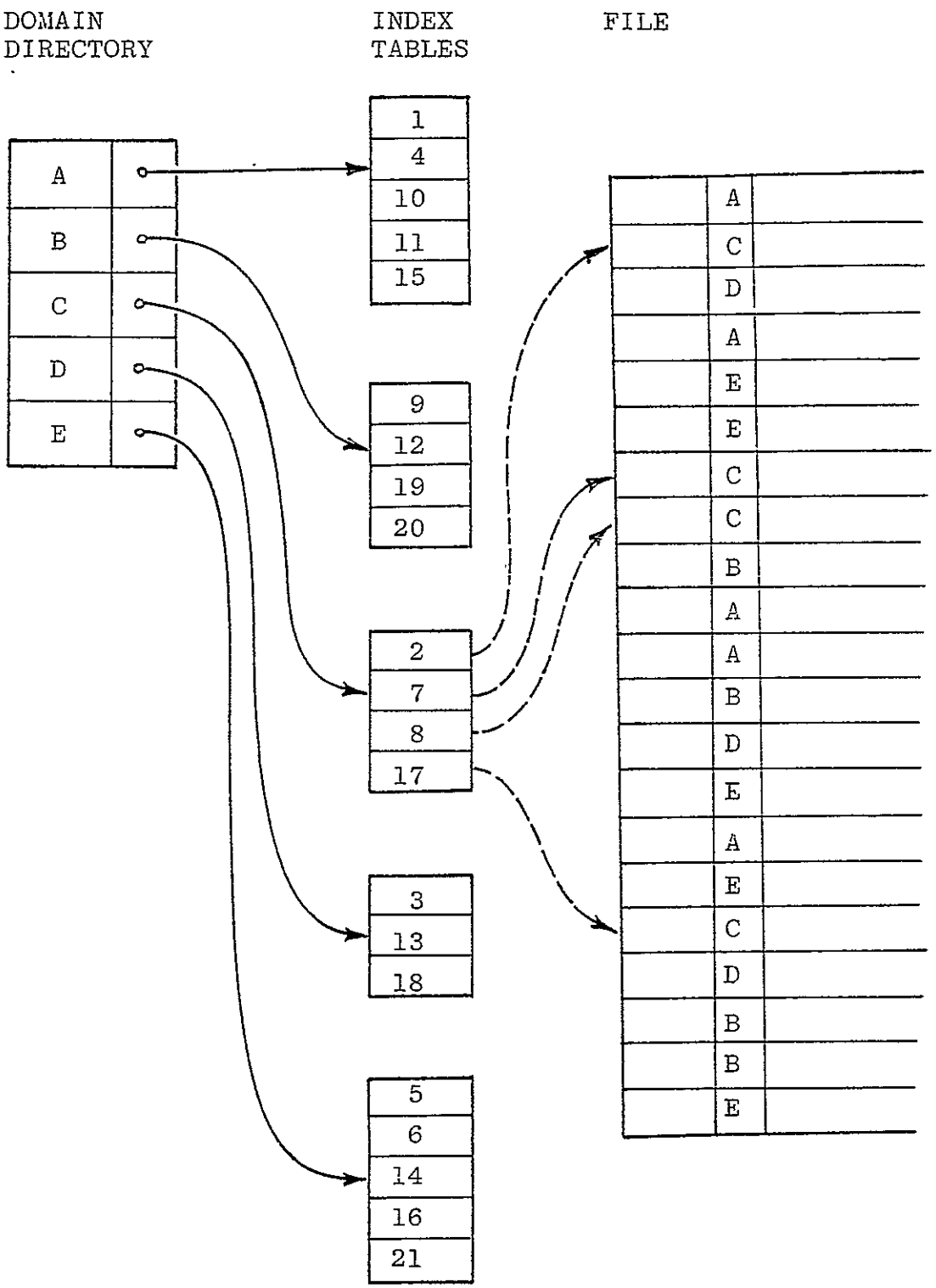


Figure 6-8: An Inverted Index

Inverted indices are particularly useful for retrieving records satisfying multiple constraints. Suppose, for example, records satisfying a combined query for LAT = 50° AND LON = 90° and SENSOR = MSS were requested, where LAT, LON, and SENSOR have inverted indices. This can be satisfied by looking up the value 50 in the domain directory for LAT and retrieving those record id's, looking up the value 90 in the domain directory for LON and intersecting that set of record id's with the first set (i.e., the set of record id's for LAT = 50), and finally retrieving the record id's where SENSOR = MSS and intersecting the sets one more time.

6.6.2.2 Logical Pages

It is not difficult to reconcile the concept of a physical page with the requirements for data record storage and for tree-structured indices. It is much more difficult to link the concept of a fixed-size physical page with the highly variable-sized domain directories and index tables. A domain directory might have only three or four entries (e.g., spacecraft name and launch date in a table containing information on active spacecraft) or it might have hundreds of entries. Similarly, an index table might have only a few record id's (the minimum is one record id since unused key values are deleted from the domain directory) or it might have thousands of record id's. The mechanism to decouple the variable-sized tables from the fixed physical pages is the logical page. As its name implies the logical page is a logical, rather than physical, entity. Logical pages are variable-sized and do not have a fixed physical address.

Logical pages are accessed through a logical-to-physical map. A logical-to-physical map is a table whose entries have the following fields:

(1) logical page number
(2) physical page number
(3) base address
(4) size
(5) old size
(6) continuation logical page

Given a logical page to locate, one begins by searching the logical-to-physical map for that page's entry. The location of the logical page is specified by a base address within a physical page, where the base address points just ahead of the page's true location. Thus to reference the i^{th} byte in logical page, one adds i to the base address in the indicated physical page.

It may happen that a logical page is larger than a physical page. In such a case the logical page is split, and the overflow is assigned to a new logical page whose number is then stored in the "continuation" slot in the map.

6.6.2.3 Searching an Inverted Index

All of the data structures associated with an inverted index -- the domain directory and the index tables -- are stored on separate logical pages. The pointer field for an entry in a domain extension which has a B-tree index will contain a physical page number, representing the root node. The pointer field for an entry in the domain extension which has an inverted index will be a logical page number, representing the logical page which contains the domain directory. Each domain extension must have its own logical-to-physical map, or the overhead for searching the map will be prohibitive.

Index tables will hold only record id's, in sorted order. Strictly speaking, there is no need for sorting the index

tables, but intersecting two sets of record id's will be made much more efficient if they are known to be sorted. The entries in a domain directory will be in two parts: a search key value and a logical page number. Since the size of a field will vary from field to field, it follows that these entries will vary in size from domain directory to domain directory. Regardless of entry size, however, these entries will be sorted on search key value. At this point it is possible to describe algorithms for searching an inverted index.

There are two types of searches to consider -- searches which locate all records for which a given search key takes on a single, specific value, and searches which locate records where the key falls within a specified range of values. The Integrated Data Base Management System treats the former type of search as a special case of the latter, where the upper bound of the range coincides with the lower bound. The search begins by locating the first entry in the domain directory such that the key value in the index is greater than or equal to the lower bound of the range and less than or equal to the upper bound of the range. The corresponding index (logical) page is then retrieved and its list of record id's is extracted. Since the domain directory is presumed to be sorted, the search continues by examining the next entry in the domain directory and either (1) terminating the search if the value of that entry exceeds the upper bound of the range, or (2) retrieving the corresponding index page, adding those record id's to the set of record id's already extracted, and then continuing to the next entry in the domain directory to repeat this cycle.

6.6.2.4 Maintaining Logical Pages

Deleting a record id, T, with search key value, V, from an inverted index begins with a search for V in the domain directory to retrieve the index page corresponding to V. Either T is in that page or it is not, and if it is present then it is

removed, the index page is compacted, and the size of the index page in the logical-to-physical map is decremented. If T is the only entry in the index table then the entire entry for that page in the logical-to-physical map must be deleted and the entry for V deleted from the domain directory.

Inserting record id T with key value V is slightly more complex since (1) V may or may not already be in the domain directory, and (2) inserting T into an index table or V into a domain directory may cause overflow past the end of a physical page or onto another logical page. If V is a new value then the first step is to create an entry for another logical page (the index table to correspond to V) in the logical-to-physical map. If there is enough free space in the physical page that contains the domain directory to hold both T and V, then the new logical page will be placed on the same physical page as the domain directory (to minimize physical page accesses in later searches). If an overflow occurs then there are three cases to consider:

(1) There is sufficient free space elsewhere in the page to accomodate the overflow entry, in which case the logical pages on that physical page are reshuffled using Garwick's algorithm (Knuth[20] section 2.2).

(2) The physical page is full, but there are multiple logical pages on this physical page, in which case the overflowing logical page is shifted to a new physical page.

(3) The physical page is full and this is the only logical page thereon, in which case the overflow is passed to a continuation page, if one exists, or else a continuation logical page is begun on a new physical page.

Special care must be taken if the inserted entry comes at the end of a logical page that has a continuation page, since it is important to maintain the relationship that the last entry in any given page is lower in the collating sequence than the first entry in the continuation page.

SECTION 7 - DATA FILE HANDLING

7.1 An Overview of Data File Processing

The Integrated Data Base Management System will maintain two different classes of data -- tabular data, stored in tables set up under user control and managed by what is, effectively, a relational data base management system, and "non-tabular" data files managed by a portion of the system which is, in effect, a file management system. The relational portion of the Integrated Data Base Management System is normally referred to as the "front end" of the system, while the on-line and off-line data files and the file management software are collectively referred to as the "back end."

It is presumed that the off-line data files will contain remotely-sensed and directly-sensed data about the earth and its environment. The remotely-sensed weather and climate[15] data shall certainly include level three data files and may well include level two data files. Nothing in the system's design precludes the inclusion of level one data files, and a decision on whether to include level one and two data files will have to be made by the Data Base Administrator in accordance with the needs of the user community.

Tape files will be introduced to the system by the CATALOG command, which is an interactive command restricted to use by the Data Base Administrator only. Each tape file will be identified by its location (e.g., reel number, physical file number) and by a format code. The Integrated Data Base Management System will respond by examining its Data File Catalog to determine whether this file duplicates another cataloged file and, if not, then the system will assign a unique data file identifier (did) to that file, output the did to the DBA, and enter the file into the Data

File Catalog. However, this process will merely make the file known to the system. Before the system can make the file known to the user community it will be necessary for the DBA to make one or more entries for that file in the Data File Directory in the Global Data Base.

Once the data file has been cataloged and inserted into the appropriate directory tables by the DBA, a user will have the ability to retrieve sets of did's representing files of interest to him or her by querying a particular directory table or by querying all the directory tables at once. The latter can be accomplished by querying the special table name "SYSDIR" which can be imagined to be a single, comprehensive table implicitly defined to be the union of all directory tables projected over common columns.* Note that SYSDIR will be a virtual table and will not physically exist.

Interactive users will not be allowed to access data files directly from tape. A necessary intermediate step will be for the files to be copied on-line with the LOAD command. The on-line files created by a LOAD command will always be in one of the system standard formats, which is a special file format with a fixed-length header, zero or more fixed-length processing history records, and then the data records themselves. The header will contain a code telling the system (and user application programs) how to interpret the remainder of the header, and the remainder of the header will inform the system (and user application programs) how to interpret the remainder of the data. A user need not LOAD an entire data file if interested in only a portion of the file. It is proposed that the system support three types of subfile-creating operations in conjunction with a LOAD: SLICE, SUBSET and WINDOW. In certain types

*The union and projection operations are defined in Appendix A.

of files a data observation point can be viewed as a node in a multi-dimensional grid, where the dimensions include not only the x and y coordinates on the ground, but also altitude (z), time (t), and/or wavelength (λ). The SLICE operation will take a 2-D slice through such a file. Since each observation point in a data file may contain observations for more than one physical variable, the SUBSET operation will exist to permit taking only a subset of the physical variables recorded in the file. Finally, the WINDOW operation will cause only a rectangular subarea of a two dimensional file, such as a sliced grid, an image, a cartographic terrain elevation model, etc., to be loaded. If a data file is loaded on-line without manipulation then it will retain its original identifier, while a new did must be issued if one or more operations cause a subfile to be loaded (since the contents of the on-line and off-line files would be different).

Once a data file is on-line, an interactive user may, if the contents of the file represent tabular data, COPY the on-line file into a pre-defined table in the front end of the system. Alternatively, the user may choose to manipulate the files further with a PERFORM command. Present plans call for five operations to be performable: the SLICE, SUBSET, and WINDOW operations described above, plus a REGRID operation to cause the grid system of a multi-dimensional gridded file to be redefined and the data observations interpolated to fit the new grid, plus a MERGE operation to merge two data files (provided they are defined with respect to the same axes and represent overlapping areas). The result of a successful PERFORM will be a new on-line data file (in a system standard format) with its own did. This is in accordance with the principle that all data files maintained by the system shall be read only.

As indicated above, certain operations are applicable only to specific types of data files. For example, the MERGE operation can be performed on gridded data files but not image or cartographic data files.

Unless converted to permanent status by a KEEP command, all on-line files will be classed as temporary and will be automatically purged from the on-line mass storage by the system some fixed span of days after the last access. A temporary on-line data file may be purged sooner than that with a SCRATCH command, but a SCRATCH will not be permitted on a permanent data file unless it has been backed up to tape with an UNLOAD command beforehand. Also, the DBA may purge all on-line and off-line copies of any file with an UNCATALOG command.

Except for data files (e.g., level two GARP reports from NOAA[15]) which are reasonable to COPY into tables, an interactive user will not be able to access data in data files directly. Matters will be rather different with an application program, which will be able to OPEN and CLOSE data files, READ and WRITE data records, and GET and PUT header and processing history records, as well as issuing LOAD, UNLOAD, and COPY commands and performing file manipulations. An application program may OPEN a file in input mode or output mode (in the case of the latter the system will generate a new did), and also in "direct" mode or "system standard mode." In system standard mode the files being opened must be in system standard format, and will be presumed to have a header record and could have one or more history records as well, while in direct mode the files are expected to <u>not</u> be in system standard format. A file opened in input/system standard mode may be an on-line data file or a backed-up tape copy of an on-line file (if the on-line file has been unloaded and scratched). A file opened in

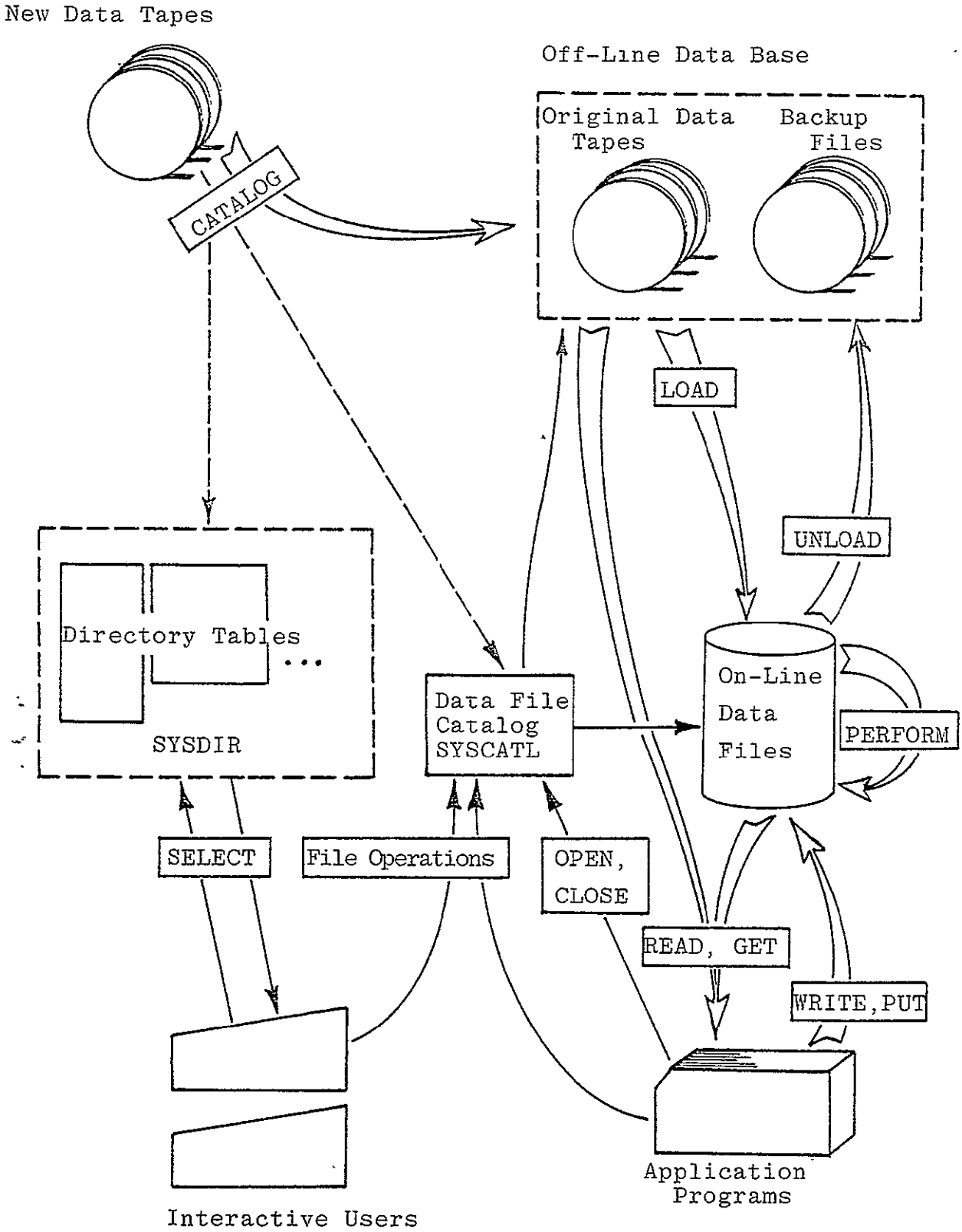


Figure 7-1: Flow of Data Through the System

input/direct mode will always be read from tape in the original data file format. Files opened in output/system standard mode will be on-line files in system standard format while files opened in output/direct mode will be on-line files in a special format* In either case the output file will be assigned a new data file identifier by the system when it is opened and the new file will become read-only when closed.

Figure 7-1 illustrates data paths within the system. The interactive data file processing commands are described in greater detail in section 4.6, and file operations available through the system to an application program are described in section 5. The remainder of this section will cover the topics of the Data File Catalog, the Data File Directory, and system standard formats in greater detail.

7.2 The Data File Catalog

The Data File Catalog will be a system table named SYSCATL. Like the other system tables (e.g., SYSREL, SYSUSER, SYSDB) the SYSCATL table will reside in the Global Data Base and will be invisible to normal users. Records may be inserted into this table by the DBA using the CATALOG command or by the system when a user creates a new on-line file. Records in the catalog will change only in response to commands such as LOAD, UNLOAD, KEEP, SCRATCH, etc., and cannot be edited by the DBA using INSERT, UPDATE, or DELETE commands. This is because changes may very well have non-obvious side effects and may require a certain amount of collateral processing.

The most important field in the SYSCATL table will be the one which contains the data file identifier. Since

* Described in Section 7.5.4.

virtually all references against the catalog will be based on the did, a hierarchical (B-tree) index superstructure will be established on that field, and, moreover, it will be a "unique" index. That is, the software shall be prepared to test for duplicate entries, and to reject an insertion which would create a duplicate value for that field. Null did's will never be accepted.

The remaining fields can be partitioned into three groups representing data about the original off-line tape file, data about the on-line version of the file, if any, and data about the off-line back-up copy of the file, if any, respectively. If any field in a particular group is null then all in that group must be null. Any group, or even any pair of groups may be null at any given time, though it will not be possible for all groups to be empty, since that would mean that the file does not exist at all.

The fields of the group describing the original tape file will include:

(1) reel number, or some means of identifying the tape on which it resides
(2) file number, or some means of identifying which (physical) file on that tape contains this data file
(3) format code

There may or may not be additional fields in this group, depending upon the specific characteristics of the tape file I/O system of the computer on which this system is implemented. Since the system will check for duplicate data files when it inserts a new entry into the catalog it will be useful to maintain a hierarchical index on a combined key formed by concatenating the reel number and file number. This index

need not be unique, however, since a given physical data file may well contain more than one logical subfile. An example of this situation would be NIMBUS-G SMMR MAP-LO tapes, where a single six-day file contains five frames and each frame contains two Mercator map matrices. Thus, there are fifteen logical subfiles of potential interest which could be derived from a single physical MAP-LO file*. The individual logical files could be distinguished from one another by having different format codes.

Note that this group may well be null -- if the data file in question happened to be created by the LOAD command with a subfile operation or a PERFORM command or if the file was created by an application program.

The fields of the group describing the on-line version of the data file will include:

(1) name or disk address of the on-line copy of the data file
(2) owner of the on-line copy
(3) temporary/permanent flag
(4) date last accessed
(5) format code

The existence of the name/address field depends upon implementation details and may, under certain circumstances, be superfluous. For example, if it is decided to use an alphanumeric character string for the did's, and if the operating system under which the Integrated Data Base Management System is implemented has a good file management subsystem, then one implementation approach for managing on-line data files would be to create a file name from the did, open a disk file under that name using the operating

* Each Mercator map could be a logical subfile and each frame could be a logical subfile.

system, and then copy the tape file into the disk file using normal operating system utilities.

The owner of an on-line file will be the user who loaded it onto disk, unless a KEEP command is later issued, in which case the user who wants the file kept would assume ownership of the file. Only the owner of the file or the DBA may SCRATCH it, although anyone may access it. If it appears likely that disk space will become a problem then it may be useful from the DBA's point of view to invert the catalog table on the owner field, so that the DBA could efficiently determine which users were making the heaviest demands on disk storage.

Finally, the fields describing the back-up tape copy of the file will duplicate the first group, to some extent·

(1) reel number, or some means of identifying the tape on which it resides
(2) file number, or some means of identifying which file on that tape contains the data
(3) format code

The only difference between the two groups is that the format code for this version of the file will necessarily represent a system standard format. Notice that this field is <u>not</u> superfluous since it is possible to imagine a sequence of operatons which leaves this the only non-null group of the three (e.g., a PERFORM creating the file, a later UNLOAD, then a SCRATCH) and it will be more difficult and time-consuming to access the header of a tape file in system standard format than to access the header of a disk file.

7.3 The Data File Directory

The purpose of the Data File Catalog is to provide the system with the information it needs to respond to interactive and application program data file processing commands. It will be the function of the directory tables, which constitute the Data File Directory, to provide information to the user community about the logical contents of data files managed by the system. Whereas the Data File Catalog will be invisible to users (other than the DBA), the Data File Directory, which is also contained in the Global Data Base, will be known and visible to all users.

There will be a number of directory tables, perhaps as many as one directory table for each class of data file entered into the back end of the system (e.g., one directory table for SMMR PARM files, one directory table for SMMR MAP files, one directory table for LANDSAT images, etc.). The number, content, and layout of these tables will be under the control of the DBA, who will be the only user authorized to issue a DEFINE DIRECTORY TABLE command, the only user with INSERT, UPDATE, or DELETE rights against these tables, and, for that matter, the only user with MODIFY rights against the Global Data Base. It will be the responsibility of the DBA to tailor the definitions of the directory tables to suit the needs of the user community.

With one exception, the Integrated Data Base Management System will treat directory tables just like any other table maintained and managed by the system. The DBA will be able to issue EXPAND commands, INDEX commands, INVERT commands, etc., on directory tables as well as being able to issue INSERT commands and DELETE commands as necessary to reflect the changing contents of the Non-Relational Data Base. The exception to this rule is that all directory

tables shall implicitly become a part of the virtual overall directory table, SYSDIR. SYSDIR will be a table which users will be able to query, but which will not physically exist. (The term for this type of table in relational data model jargon is "view.") While updates and deletes can be made against SYSDIR, causing modifications to be made to the underlying directory tables, the insertion of new records into the Data File Directory can only be made by inserting the records into the underlying, physically existing, directory tables.

Nothing in this section should be construed to imply that the only legitimate directory tables in the system will be the ones set up by the DBA in the Global Data Base. In point of fact, users may -- indeed, users are encouraged to -- set up their own directory tables in applications or working data bases. These directory tables may well include files which are not included in SYSDIR, files which have been created via PERFORM commands, for example, in response to specific application requirements. However, such directory tables will not be part of SYSDIR.

If a file is purged from the system via an UNCATALOG command, then any record in any directory table which references that file will automatically be deleted as well. This feature will be in addition to the ability of the DBA to issue DELETE commands against the directory tables without altering the Data File Catalog. However, automatic directory deletion will not occur as a function of a SCRATCH command (unless the SCRATCH command results in the file being purged), nor will this feature be extended to tables which are not part of SYSDIR. Presumably, the DBA will not be purging files which are actively in use so that the overhead of testing all tables in all data bases for references to files being purged would be wasted effort.

7.4 The Data File Identifier

The link between the front end, or relational, portion of the Integrated Data Base Management System and the back end will be the data file identifier, or "did." Each did will uniquely distinguish a data file managed by the back end of the system, and each reference to a data file in the front end of the system will be via the did.

There are a number of methods which might be employed to generate unique did's, and this document will not attempt to choose between them at this point since "best" almost certainly will depend to some extent on the characteristics of the machine(s) and operating system(s) on which the Integrated Data Base Management System is implemented*. However, three general approaches can be described:

(1) use a random number generator to generate a random number between 0 and 1, then convert this random number to a string of digits or alphanumerics

(2) concatenate the year and (Julian) date to create the first five characters of an identifier, then append two or three more digits as a counter (so that 7819432 is the thirty-second did generated on July 13, 1978)

(3) keep a universal counter and increment it each time another did was requested by the system for a new data file

All three of the above approaches have good points and bad points. The random number generator approach would work well if the back end was, in fact, implemented on a separate

*The reader should bear in mind the fact that the "dual system" design of the Integrated Data Base Management System would permit its being implemented on more than one computer.

computer. In that case the Data File Catalog table, SYSCATL, probably would be moved from the Global Data Base to the data file processing software, thereby allowing the introduction of scatter storage techniques. That is, it would be possible to take advantage of the uniqueness of the did field and the fact that virtually all references to catalog entries would be via the did to "hash" the catalog on the did field, thereby reducing the average number of disk accesses to retrieve an entry in SYSCATL. However, when the system is implemented on a single computer then the increased software complexity to support scatter storage access methods for one specific table would likely outweigh the search efficiency advantages. A drawback of the random number approach is that there is no guarantee that the did's so generated are, in fact, unique. The non-uniqueness of a given identifier would be detected when the new entry was inserted into the catalog, and this would necessitate the generation of another random identifier.

The two counter-based approaches can guarantee uniqueness, and while their ability to function efficiently with scatter storage techniques would depend upon the effectiveness of the hash function, these deterministic approaches would be quite efficient when used with the table storage management and look-up techniques used in the front end of the system (see Section 6), particularly when a series of data files was entered all at once. The main drawback of these counter-based approaches is that the updated counter must be saved on a non-volatile storage medium every time a new entry is made in the catalog, or else this approach would be highly vulnerable to a system crash.

Just as there is more than one reasonable approach to generating data file identifiers, so there is more

than one reasonable format for the did's. Should they be numeric id's, all digits? Or alphanumeric? How many characters? Again, resolution of this issue must await actual implementation of the system on some machine, since most of the tradeoffs cannot be properly evaluated without knowledge of which machine the system will be implemented on.

Another consideration in defining the format of an identifier is the need for detecting mistakes made by users when entering did's to the system. The most common such mistake is a transposition of characters, and the standard defense against this is the check digit. If a did is composed of n characters (alphanumeric or digits) then n-1 of them would perform the function of identifying the file while the n^{th} character would be uniquely determined as a function of the previous n-1 characters and their relative order. If the check function is well chosen, then the system can detect erroneous input characters (5 for S, 2 for Z, 1 for I, zero for 0) or the correct characters out of order by computing the proper check character for the n-1 identifying characters of the given input did and comparing it with the given check character. If they agree then the did will be accepted, and if they disagree then the input did must be wrong. Of course a double error or triple error may make it past this test, but some double and triple errors will still be detected and the overhead for detecting all double and triple errors would be more expensive than the likely gains.

7.5 System Standard Formats

As described earlier, all data files which are loaded on-line from tape will be reformatted to conform with the

relevant system standard format for the type of data contained in the file. There are a number of advantages to this convention:

(1) It facilitates the development of software interfaces between the Integrated Data Base Management System and other applications systems at Goddard Space Flight Center.
(2) It facilitates the implementation of data manipulation modules internal to the system (e.g., the modules which carry out the REGRID or SLICE operations of the PERFORM command).
(3) It simplifies the task of writing application programs which will make use of the data files, particularly if data files of the same type but from different or unknown sources are to be used.

There will be a number of system standard formats, one for each major broad category of data file. That is, there could be one system standard format for image data, one system standard format for cartographic data, one system standard format for uniformly gridded data (i.e., where the data observation can be viewed as occurring at a lattice point of a multi-dimensional network), one system standard format for chain-coded contour plots, etc. System standard formats will be alike in that each file in system standard format will include a fixed-size header record, zero or more fixed-size history records, and some number of data records, where the length and number of data records will depend upon the data itself. Header records will include a code describing which type of data is contained in the file (i.e., which system standard format the file is in), and the remainder of the header will describe how the data records are to be

interpreted. How the remainder of the header is to be interpreted will depend upon the format code.

This document shall not attempt to define the number and layout of all system standard formats which will be included in the final version of the Integrated Data Base Management System. Instead, the remainder of this section shall concentrate on describing what certain, selected system standard formats might look like. In the final implementation the fields and/or their type, size (in bytes), and units (if any) may well change from what is written here, but any changes will presumably be minimal. Since the number of bytes needed to store the information in a header will vary from format type to format type while the size of a header record shall be fixed, some headers will have to be padded with blanks. The following format descriptions will ignore this padding.

7.5.1 A System Standard Format for Image Data

A system standard format for image data must, at a minimum, be compatible with the data record layout and header record formats for image files used by systems at the Goddard Space Flight Center which handle image data such as AOIPS[2] and SMIPS/VICAR[27]. Compatibility, as it is used here, means that the data record format for image data files in system standard format should agree with the data record layouts normally handled by AOIPS and SMIPS/VICAR, and that the fields in an AOIPS header or SMIPS/VICAR label record should be available in the system standard format header or else derivable by a software interface routine.

7.5.1.1 The Header Record in an Image Data File

Table 7-1 illustrates a possible layout for a header record, based on the fields included in the Image Description portion of an AOIPS image label and a SMIPS/VICAR label. The order, type, and/or size of the data fields listed in that table are particularly dependent upon implementation details when the system is developed. For example, the size of a data file identifier (did) is not yet finalized. All fields are integer unless otherwise noted.

The first six fields shown in table 7-1 (above the dashed line) will probably be included in all system standard formats. The remaining fields of the header are designed to preserve header information if the image file is entered into the Non-Relational Data Base from AOIPS and later, perhaps after some manipulation, is passed back to AOIPS. Not every AOIPS image description field is duplicated in an image header, however, since certain fields will be superfluous given the data storage conventions. For example, secondary records would not be stored within the image file but would be saved in some number of history records. Likewise, by convention, there will be no top edge, bottom edge, left edge or right edge fill, so that words 35 through 38 of the AOIPS image label will be superfluous.

7.5.1.2 Data Records in an Image Data File

One record of an image file in system standard format will contain precisely one row (line) of the image matrix. Each pixel will occupy one or more bytes but will always occupy an integral number of bytes. There will be no "empty" records or lines with non-grey scale data in an image file in system standard format, although there may

field	size*	meaning
format code	1	system standard format code
history count	1	number of history records
record size	2	size of a data record in bytes
record count	2	number of data records
blocking	2	blocking factor
did	4	data file id for this image
name	8	user-assigned name of scene (alpha)
parent	8	did or name of parent image (alpha)**
master	16	reel and file id of master image
year	2	date and time (alphanumeric)
month	2	
day	2	
hours	2	
minutes	2	
seconds	2	
fractions	4	fractions of a second (real)
sensor	4	sensor name (alphanumeric)
generation	1	generation of image (master = 0)
no. images	1	number of images, if multi-image
pixel size	1	number of bytes per pixel
sig. bits	1	number of significant bits per pixel
storage	3	storage code (BSQ,BIL,BIP) (alpha)
orbit no.	4	number of orbit on which image was recorded
center lat.	4	latitude of frame center
center lon.	4	longitude of frame center
sun el.	4	sun elevation
resolution	4	spatial resolution of each pixel
zoom info.	22	AOIPS image related master/parent zoom relationship information

Table 7-1: A Possible Layout of the Header Record for Image Data

*In bytes. All sizes are tentative.
**Field equals zero if no parent (i.e., if this image is a master).

be some pixels which are only used for padding out a record if this system is implemented on a word-oriented, rather than byte-oriented computer.

It would be desirable to standardize pixel storage for a multi-image data file. The AOIPS and SMIPS/VICAR systems are oriented towards the band sequential format (BSQ) for multi-images, and it may, therefore, be correct for the Integrated Data Base Management System to standardize on band sequential format as well. Other possibilities are band interleaved by line (BIL), where the record containing the i^{th} line of the j^{th} band would follow the record containing the i^{th} line of the $j\text{-}1^{st}$ band (for $j > 1$), and band interleaved by pixel (BIP), where the k^{th} pixel in line i of band j follows the k^{th} pixel of line i in band j-1. Certain SMIPS/VICAR programs (e.g., BAYES, KARLOV) can accept line interleaved (BIL) multi-images and both systems have provision for indicating BIL and BIP formats in their respective image headers. Consequently it may be useful -- if not in initial versions of the system then perhaps in later enhancements -- to have the system support all three formats for multi-image data files.

7.5.2 A System Standard Format for Gridded Data

Certain level three data files can be viewed as representing multi-dimensional grids, where the dimensions are a subset of the x,y, and z spatial dimensions, plus the temporal (t) dimension and a wavelength (λ) dimension. Such files are presumed to possess at least two dimensions, and possibly all five. For purposes of the Integrated Data Base Management System's internal processing, the observations taken at a fixed wavelength shall be treated as observations of a single physical variable. The exception to this rule will be the SLICE operation, which will treat wavelength as a fifth dimension.

For expository purposes, the remaining four dimensions shall be regarded as having the following ordering: t before x before y before z . Therefore, when the conceptual layout of an n-dimensional grid is described (n = 1,2, 3 or 4) the first, second, ..., n^{th} dimensions will be uniquely determined by the above ordering. Thus, if n equals two and the grid has z and t axes, then the "first dimension" will be t and the second will be z . Similarly, a three dimensional grid with x, y, and z axes will have x as its first dimension, y as its second dimension, and z as the third (rather than some permutation such as z, x, y or y, z, x).

7.5.2.1 The Header Record in a Gridded Data File

Table 7-2 shows a possible layout for a header record for gridded data. The latitude, longitude, altitude, and date provide the initial y, x, z, and t coordinates, respectively, for the first lattice point in the first record. The maximum time coordinate will be the initial date and time plus $(n_t-1)\cdot\Delta t$, and, similarly, the maximum altitude will be the initial altitude plus $(n_z-1)\cdot\Delta z$. Computation of maximum latitude and longitude will be rather more complicated since the azimuth, if any, must be factored in.

Four byte and eight byte fields below the dashed line will be type real and one byte and two byte fields will be integer, unless otherwise indicated. Units for Δt, Δx , Δy , and Δz will have to be established at implementation time, as will the reference for altitude. These units will probably be fixed and will not vary from file to file. Otherwise, extra fields would be needed in the header to show which units were used for each delta (Δ) field.

field	size*	meaning
format code	1	system standard format code
history count	1	number of history records
record size	2	size of a data record in bytes
record count	2	number of data records
blocking	2	blocking factor
did	4	data file id for this file
year	2	date and time (alphanumeric)
month	2	
day	2	
hours	2	
minutes	2	
seconds	2	
latitude	8	latitude of first observation point in radians
longitude	8	longitude " " " " " "
azimuth	8	azimuth in radians clockwise from North
altitude	4	altitude of first observation in meters
Δt	4	spacing along t axis
Δx	4	" " x "
Δy	4	" " y "
Δz	4	" " z "
n_t	1	number of observations along t axis
n_x	1	" " " " x "
n_y	1	" " " " y "
n_z	1	" " " " z "
n_{obs}	1	" " " per lattice point
point size	1	size of a lattice point in bytes
λ_1	4	first wavelength
λ_2	4	second wavelength
⋮	⋮	⋮
$\lambda_{n_{obs}}$	4	last wavelength

Table 7-2: A Possible Layout of the Header Record for Gridded Data

*In bytes. All sizes tentative.

7.5.2.2 Data Records in a Gridded Data File

When n equals two, the grid can be visualized as a rectangular area as depicted in figure 7-2a, and each lattice point will have four neighbors (except for those on the boundary of the area). Each lattice point will be assumed to have observations for one or more physical variables, and the same set of physical variables are measured at each lattice point. When such a grid is stored in a system standard format file, each record will contain the observations for a fixed value of the first dimension, and for each given value of the first dimension (for which there is data) there will be precisely one record in the file. In other words, each record in the file will correspond to one column of the grid. Within the file the records will be ordered on increasing value for the first dimension, and within a record all observation for a given lattice point will be stored together and the lattice points will be ordered on increasing value of the second dimension.

Three-dimensional files may be viewed as a parallelepiped with rectangular sides, as depicted in figure 7-2b. To visualize the way in which lattice points from a three-dimensional grid are mapped into records, imagine the grid being sliced into a stack of two-dimensional grids along the <u>first</u> dimension. The resulting two-dimensional grids are then handled as described above. Thus, any pair of lattice points in a record from a three dimensional gridded file have the same values for the coordinates of their first two dimensions, and they vary only according to their third dimension.

Four dimensions are hard to visualize, but since the four dimensions in this case <u>must</u> be t, x, y, and z then one way to view a four dimensional grid is as a time-ordered

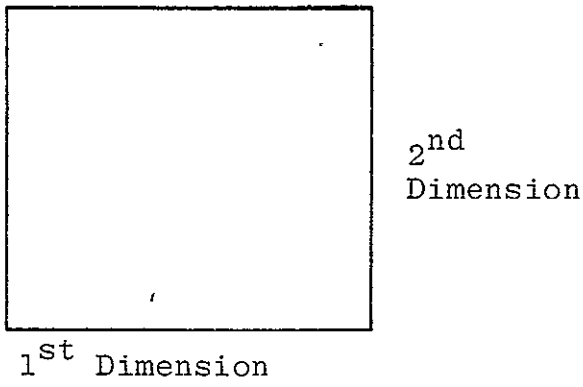


Figure 7-2a: Conceptual Layout of a Two Dimensional File

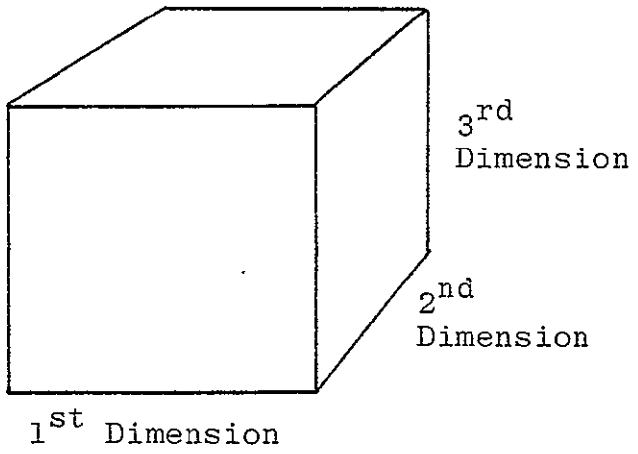


Figure 7-2b: Conceptual Layout of a Three Dimensional File

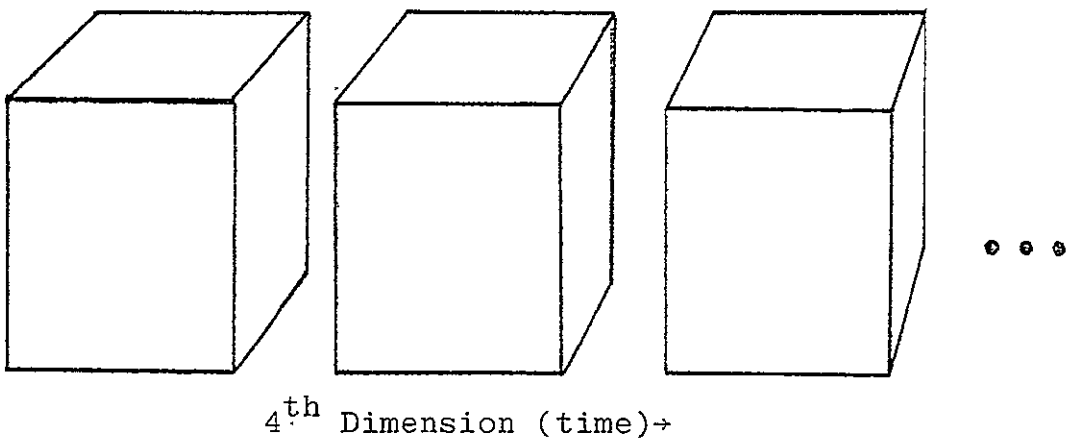


Figure 7-2c: Conceptual Layout of a Four Dimensional File

sequence of three dimensional grids, as depicted in figure 7-2c. This suggests the way in which the data can be mapped into records, namely, all lattice points in a given record fixed with respect to t, x, and y coordinates, and records ordered on ascending values of y within x within t . Figure 7-3 illustrates the process for converting the storage problems of an n-dimensional grid into n-1 dimensions for n=4, 3, 2, and figure 7-3 depicts the layout of records in a file for n=4.

Note that this arrangement is consistent with the view of wavelength as a fifth dimension, provided the observations stored at a lattice point are ordered in increasing value of λ .

Finally, there is the special case when n=1 . In that case there would be one observation point per record, unless the size of an observation point was very small, in which case there would be one record in the file.

7.5.3 A System Standard Format for Cartographic Data

Digitized terrain elevation models may be viewed as two-dimensional grids, with a ground elevation value at each lattice point. This suggests that the data for such a file could be stored as if it were a conventional two-dimensional gridded file, that is, each record will have the elevation data for a fixed x coordinate and ascending values of the y coordinate. However, it would not be sufficient to treat a cartographic data file as a conventional gridded data file since the cartographic data may be any one of a variety of map projections and this information must be present in the header. On the other hand, barring a natural or manmade catastrophe, the elevation value for a

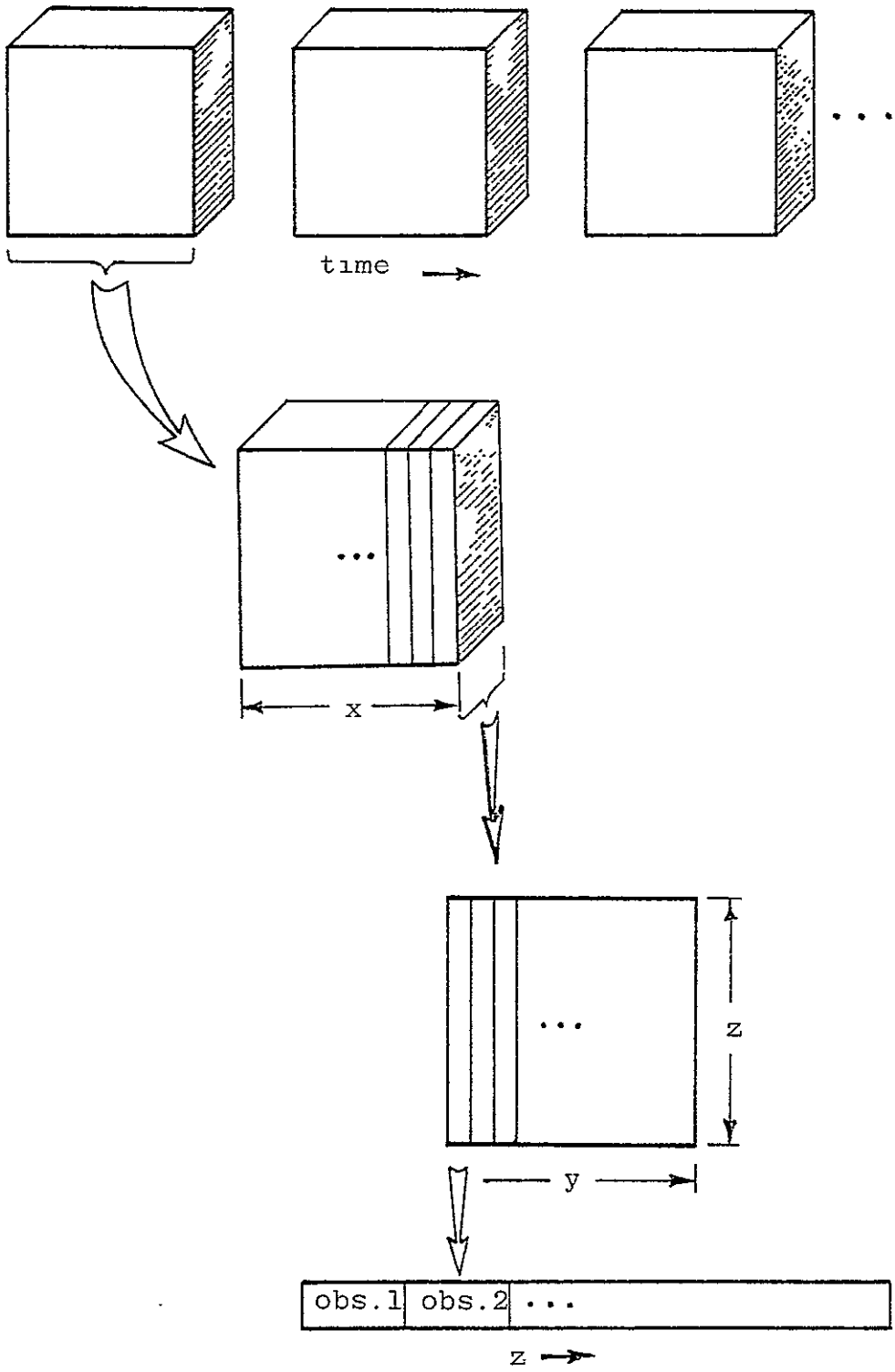


Figure 7-3: Stages for Mapping a Four Dimensional Grid into Records in a File

given location on the earth's surface is likely to be constant with respect to time (at least for spans of time less than a century) and thus the date fields would be superfluous.

Table 7-3 shows one way in which a header could be established for a cartographic data file. The header establishes the coordinates of the first point in the first record (lower left corner), but how these coordinates are established (and hence how the fields of the header are to be interpreted) will depend upon the projection code. If the data is in a tangent plane projection then the coordinates will be specified as an x offset and y offset in meters on the ground relative to a Cartesian coordinate system whose origin is at the point of tangency and whose y axis is aligned with some specified azimuth*. For a tangent plane projection, the "zone" field would be ignored, the latitude and longitude fields would be the coordinates of the point of tangency, the azimuth would specify the azimuth of the Cartesian coordinate system for the file, and the lower left x and lower left y are the x-offset and y-offset of the lower left corner of the file from the point of tangency. If the data is in the normal Mercator projection then rows will be lines of constant latitude and columns (records) will be lines of constant longitude. Here the zone, azimuth, lower left x and lower left y fields would all be zero, and latitude and longitude would represent the latitude and longitude of the lower left corner. In a Universal Transverse Mercator (UTM) projection, the coordinates of the lower left corner are specified by the UTM zone and UTM easting and northing. For a UTM projection, the latitude, longitude, and azimuth would all be zero and the easting and northing would be stored in lower left x and lower left y. Finally, for a Lambert projection only lower left x and lower left y would be used. The units for

*Except, of course, if the latitude is $\pm$ 90°, in which case (a) longitude and azimuth are superfluous and (b) the projection must be a polar stereographic projection.

field	size*	meaning
format code	1	system standard format code
history count	1	number of history records
record size	2	size of a data record in bytes
record count	2	number of data records
blocking	2	blocking factor
did	4	data file id for this file
projection code	1	map projection for this file
zone	1	UTM zone (if UTM projection)
latitude	8	latitude of file in radians N latitude
longitude	8	longitude of file in radians E longitude
azimuth	8	azimuth of file's coordinate system
lower left x	4	coordinates of file's lower left corner
lower left y	4	
Δx	4	spacing between columns
Δy	4	spacing between rows
z-offset	4	offset applied to each data point**

Table 7-3: A Possible Layout of the Header Record for Cartographic Data

*In bytes. All sizes are tentative.

** Permits elevation data to be positive. The z-offset will be subtracted from each elevation value in the file.

Δx and Δy will also depend upon the projection.

All fields after "zone" in Table 7-3 will be type real.

7.5.4 "Format X"

There will be one special "system standard format" which will not be descriptive of the logical contents of the file. This format, provisionally designated "Format X", will be used in the following two circumstances:

(1) The system does not know the format of the tape file or knows the format of the tape file but does not know how to translate the file into a system standard format (i.e., a LOAD routine has not been written to handle files in that format).

(2) An application program is creating a file in "direct" mode (i.e., not in a system standard format).

Basically, Format X consists of a header record, followed by a record-by-record copy of the tape file.* The header itself will be quite minimal, and will contain only the first six fields (twelve bytes) common to all system standard header records. If the record size field in a Format X header is zero, then the records in the data file will be variable length.

7.6 The LOAD and UNLOAD Commands

The data tapes to be managed by the "back end" of the Integrated Data Base Management System will exist in a variety of different tape formats, but, fortunately, not an infinite variety of tape formats. That is, the format

---

*Nothing would prevent a user from writing history records to a data file in Format X, but analysis of user requirements suggests that such will be rare.

specifications for the data tapes entered into the Non-Relational Data Base will be known in advance and presumably have been formally documented. Each such tape format will be mapped into a unique system standard format during a LOAD, although the mapping will not be one-to-one and there will be somewhat fewer system standard formats than tape formats.

The foregoing analysis permits the function of the LOAD command to be specified as follows:

(1) Given the data file identifier of the tape file to be loaded, look up that file in the Data File Catalog and determine whether it is already on-line or whether it is backed up in a system standard format*.
(2) If not on-line and not backed up then retrieve reel number, file number, and format code.
(3) Open the file and simultaneously determine the system standard format into which the file is to be translated.
(4) Create the on-line file record by record. Construct the header record while so doing.
(5) After all records in the data file have been written to direct access storage, write the header record over a dummy header record written prior to writing the first data record.

Rather than attempt to write one large LOAD module to handle all possible tape formats, there will be a number of LOAD routines, each routine handling a small number of different tape formats (possibly one tape format per routine). Each of these separate conversion routines will probably be implemented as a co-routine and would therefore operate independently of the rest of the system

* If backed up in Format X then it will be necessary to determine whether a conversion routine has been added to the system since the most recent LOAD.

until completion of the LOAD. Benefits of a co-routine approach would include the following:

(1) Memory requirements for I/O buffers would not impinge upon memory requirements for the remainder of the system. Hence, the number of LOAD commands which could be processed concurrently would not be limited by the main storage allocated to the data base management system.
(2) The co-routines could take over I/O message handling with interactive users, relieving the system of message traffic overhead.

Of course, a co-routine approach to implementation of the LOAD command depends upon whether the operating system of the host computer supports co-routines.

In contrast to the LOAD command, the UNLOAD command can be implemented with a single, reasonably simple and straightforward, subroutine. The special point to note is that an UNLOAD command will be rejected if the Data File Catalog indicates that a backed up copy of the file already exists. The exception to this rule is that an UNLOAD will be accepted if the backed up copy is in Format X and the current on-line copy is not in Format X. This can happen if a conversion routine is added to the system between the first LOAD and UNLOAD of the file and the most recent LOAD.

SECTION 8 - SYSTEM INTERNALS

8.1 Control Structure Concepts

This section describes the control structures around which the internal architecture of the Integrated Data Base Management System is designed. The control structures consist of control blocks, control block extensions, dictionaries, lists and queues. All control structures are transient in nature. That is, main storage is allocated for a control structure when it is to be used and freed when the control structure is no longer required to support processing. The control structures have been categorized as a function of their usage within the system and are described below.

8.2 Communications Control Structures

The category of communications control structures includes the Remote Terminal Communications list and the Application Program Communications list. These two lists provide logical entries or ports into the Integrated Data Base Management System for remote terminal users and application programs, respectively.

8.2.1 The Remote Terminal Communications List

The Remote Terminal Communications List performs the function of associating a remote terminal, an interactive user and a command being processed. It consists of one entry for each remote terminal connected to the Integrated Data Base Management System. The Remote Terminal Communications List can be implemented in several different ways, one of which is a two-way linked list of remote terminal entries ordered in ascending logical sequence by terminal-id. Each entry in the Remote Terminal Communications List will contain at least the following information: the terminal-id of the remote terminal for which the entry was created; a pointer to the User Control Block for the user who connected to the system via the remote

terminal; a pointer to the Command Control Block for any currently active command that was received from the remote terminal; a continuation flag that indicates whether or not a continuation message is expected for the last command received from the remote terminal and a message routing flag that indicates whether or not the next message received from the remote terminal is to be passed directly to an active procedure where it will be processed.

Initially, the Remote Terminal Communications List will be empty. When a message is received from a remote terminal, the entries, if any, in the Remote Terminal Communications List are searched to determine whether an entry exists for the remote terminal. The terminal-id of the remote terminal from which the message was received is compared with the terminal-id in each entry in the Remote Terminal Communications List. If a match occurs, the message is processed. If no match occurs, an entry containing the terminal-id of the remote terminal from which the message was received is created and the message is processed. If the message contains a valid command connecting a user to the Integrated Data Base Management System, the entry is completed with the necessary pointers and is inserted in the Remote Terminal Communications List so as to preserve the ascending logical sequence by terminal-id. Otherwise, the new entry is discarded since the only valid command for initiating an interactive session is the one which connects a user to the system.

When subsequent messages are received from a remote terminal, the Remote Terminal Communications List is searched to locate the entry corresponding to the remote terminal. Since a pointer to a User Control Block is contained within the entry, the user issuing the command can be identified. When a command is received disconnecting a user from the Integrated Data Base Management System, the entry associated with the remote

terminal from which the message was received is deleted from the Remote Terminal Communications List.

8.2.2 The Application Program Communications List

The Application Program Communications List is analgous to the Remote Terminal Communications List and performs the function of associating a region in main storage, an application program and a command being processed. It consists of one entry for each application program connected to the Integrated Data Base Management System. As for the Remote Terminal Communications List, the Application Program Communications List can be implemented as a two-way linked list of application program entries ordered in ascending logical sequence by program-id. The choice of a program-id probably will be operating system dependent. It must uniquely identify a particular application program executing in a particular region of main storage since the same application program may be executing in different regions of main storage at the same time. Each entry in the Application Program Communications List will contain at least the following information: the program-id of the application program for which the entry was created, a pointer to the User Control Block for the application program and a pointer to the Command Control Block for any currently active command that was received from the application program.

Initially, the Application Program Communications List will be empty. When a CALL statement is executed in an application program transferring control to the Integrated Data Base Management System, the entries, if any, in the Application Program Communications List, are searched to determine whether an entry exists for the application program. The program-id of the application program executing the CALL statement is compared with the program-id in each entry in the Application Program Communications List. If a match occurs, the request

is processed. If no match occurs, an entry containing the program-id of the application program executing the CALL statement is created and the request is processed. If the request contains a valid command connecting the application program to the Integrated Data Base Management System, the entry is completed with the necessary pointers and is inserted in the Application Program Communications List so as to preserve the ascending logical sequence by program-id. Otherwise, the new entry is discarded and an error code is returned to the application program since the only valid command for initiating applica tion program activity is the one which connects an application program to the system.

When subsequent requests are received from an application program, the Application Program Communications List is searched to locate the entry corresponding to the application program. Since a pointer to the User Control Block is contained within the entry, the application program making the request can be identified. When a command is received disconnecting an application program from the Integrated Data Base Management System or, if the application program abnormally terminates execution, the entry associated with the application program is deleted from the Application Program Communications List.

8.3 The Command Control Block

The Command Control Block is the primary repository of information for the processing of a command. It is created for each interactive command and application program command that enters the Integrated Data Base Management System. The main storage required for a Command Control Block is allocated dynamically when the command enters the system. Although the contents of a Command Control Block created for an interactive command and one created for an application program command will differ somewhat in content, the basic format of a Command

Control Block will be the same so that the software processes that use the Command Control Block can operate on them in the same manner when necessary.

The Command Control Block is the primary control structure for command processing. It is passed between software processes by means of queues which are discussed below. Each Command Control Block will contain an indication of which command it represents, an indication of whether the command was received from a remote terminal or an application program, a pointer to the User Control Block of the user or application program responsible for the command, a pointer to the communications list entry associated with the command and several other data fields, flags, pointers and storage areas required for command processing. The Command Control Block exists until the processing of the command that it represents is terminated by the Integrated Data Base Management System or, in the case of an interactive command, is aborted by the remote terminal user. When the processing of a command has been completed, the main storage used for its Command Control Block is freed.

8.4 System Control Structures

The category of system control structures includes the control blocks, control block extensions and dictionaries that are stored in system tables. As stored in system tables, these control structures represent the current information state of the Integrated Data Base Management System. As resident in main storage, these control structures represent the current processing state of the system. Permanent changes to the information state of the system, such as the creation of a new data base, are reflected by updating the system tables. Temporary changes to the processing state of the system, such as the attaching of a user to a data base for processing, are reflected within the control structures resident in main storage

but do not affect the system tables. System control structures are loaded from system tables as required to support the processing state of the system.

8.4.1 User Control Blocks

A User Control Block exists for each individual who has been defined to the Integrated Data Base Management System as a valid user by the Data Base Administrator. Likewise, a User Control Block exists for each application program that has been authorized access to the system by the Data Base Administrator. User Control Blocks are stored, as records, in the SYSUSER system table. Each User Control Block will contain the user-id of the individual user or application program which it represents as well as a password, in the case of an individual user, and other data fields, flags and pointers.

A User Control Block is created when the Data Base Administrator issues a command to define a new user or application program to the system. Main storage is allocated for the new User Control Block, after which it is initialized and inserted in the SYSUSER table. Data fields in a User Control Block can be updated at any time by the Data Base Administrator. However, only the password data field in the User Control Block for an individual user can be changed by that user.

When an interactive user connects to the Integrated Data Base Management System, the User Control Block for the user is retrieved from the SYSUSER table and placed on a two-way chain of User Control Blocks for users and application programs currently connected to the system. This chain is maintained in main storage in ascending logical sequence by user-id. If a User Control Block containing the user-id already exists on the chain, the user is not permitted to connect to the system. Thus, in the current system design only one interactive user

can be connected to the system under the same user-id at any one time. Each User Control Block may have both an Authorization Extension and a Group Extension associated with it in main storage.

When an application program connects to the Integrated Data Base Management System, it must supply not only its own user-id, but the user-id and password of the individual user who initiated execution of the application program. The User Control Block associated with the application program is retrieved from the SYSUSER table and placed on the User Control Block chain in main storage. If a User Control Block containing the user-id of the application program already exists on the chain, a character string is appended to the application program user-id so that it is unique. Thus, multiple copies of the same application program can gain access to the system simultaneously. The User Control Block associated with the user running the program is retrieved from the SYSUSER table and the password supplied by the application program is verified. The Authorization Extension and the Group Extension which are associated with the User Control Block for the application program will be those of the user running the application program. Thus, the access rights associated with this execution of the application program are those that have been granted to the user who is running the program.

When a command is received from either a remote terminal or an application program, it is always associated with a User Control Block via one of the communication lists described previously. Thus, the system can identify, in effect, the user issuing the command and, thereby, control access to information and regulate the definition, modification, and removal of control structures (i.e., users, access rights, data bases, data fields and tables). Also, since the User Control Block contains pointers to both the Data Base Control Block

for the data base to which the user or application program is currently attached and the Data Base Control Block for the previously attached data base, processing can be directed to the proper data base via the User Control Block. When a user or an application program disconnects from the Integrated Data Base Management System, the corresponding User Control Block is removed from the User Control Block chain and the main storage allocated for the User Control Block is freed.

Existing users or application programs can be denied access to the system by the Data Base Administrator. If the Data Base Administrator removes the User Control Block for a user or an application program from the SYSUSER table, that user or application program can no longer gain access to the Integrated Data Base Management System.

8.4.2 Group Extensions

A Group Extension is always associated in main storage with a User Control Block. It defines the groups to which the user belongs for the purpose of sharing common access rights to tables. If the user does not belong to any groups, no Group Extension to the User Control Block will exist. A Group Extension contains one entry for each group to which the user belongs. Each entry in a Group Extension is stored in the SYSGROUP system table. Each Group Extension entry contains the name of a group to which the user belongs and a pointer to the Authorization Extension which defines the access rights of that group. If the group has not been granted any access rights, no Authorization Extension will exist for the group so the pointer will be null. During the processing of commands, the access rights of the group are treated logically as if they had been granted directly to the user.

A Group Extension entry is created when the Data Base Administrator includes a user in an existing group so that the

user can share the access rights assigned to that group. The new Group Extension entry is inserted into the SYSGROUP table. It will be included in the Group Extension constructed when the user next connects to the system.

When a user or an application program connects to the Integrated Data Base Management System, the amount of main storage required for the Group Extension is computed. The number of entries in the user's Group Extension is stored in the User Control Block. After allocating the main storage necessary to contain the Group Extension, the Group Extension records are retrieved from the SYSGROUP table and are stored in the Group Extension. Whenever it becomes necessary, during the processing of a command, to determine a user's right to access a table, the Authorization Extension attached to the User Control Block is searched. If the required authorization is not contained therein, each entry in the Group Extension attached to the User Control Block is used to locate the Authorization Extension associated with the group specified within the entry. Each Authorization Extension for a group to which the user belongs is searched until the required authorization is located or until all Authorization Extensions have been searched. When a user or an application program disconnects from the Integrated Data Base Management System, the main storage allocated for the corresponding Group Extension is freed. The main storage allocated for each of the Authorization Extensions for groups to which the user belongs, will be freed only if no other members of the various groups are connected to the system.

At any time, the Data Base Administrator can remove a user from a group or remove a group from the system. In either case, one or more records will be deleted from the SYSGROUP table and the change will be reflected by the absence of the corresponding entry in the Group Extension for the affected user or users when they next connect to the system.

8.4.3 Authorization Extensions

An Authorization Extension may be associated in main storage with a User Control Block if it contains rights granted directly to the user or an Authorization Extension may be associated with entries in one or more Group Extensions if it contains rights granted to a group. If a user has not been explicitly granted any access rights, no Authorization Extension to the User Control Block will exist. However, if a user is a member of one or more groups, he will assume any access rights contained in the Authorization Extensions for those groups. Additionally, the user can access tables on which PUBLIC rights have been granted and tables of which he is the owner. An Authorization Extension contains one entry for each table on which the user or group has been explicitly granted one or more operational rights (READ, INSERT, UPDATE, DELETE) by the table's owner. Each entry in an Authorization Extension is stored as a record in the SYSAUTH system table. Each Authorization Extension entry contains the name of the table on which the rights were granted, the name of the data base containing the table, flags that indicate which access rights were explicitly granted, the user-id of the owner of the table and flags indicating which rights were granted by the owner of the table and which were granted by the Data Base Administrator.

An Authorization Extension entry is created when the owner of a table or the Data Base Administrator issues a command to grant one or more operational rights on the table to an individual user or a group. The new Authorization Extension entry is inserted into the SYSAUTH table. If the entry represents an access right granted to an individual user, it will be included in the Authorization Extension constructed for that user when he next connects to the system. If the entry represents an access right granted to a group, it will

be included in the Authorization Extension constructed for that group when any member of the group next connects to the system.

When a user or an application program connects to the Integrated Data Base Management System, the amount of main storage required for the Authorization Extension is computed. The number of entries in a user's Authorization Extension is stored in the User Control Block. After allocating the main storage necessary to contain the Authorization Extension, the authorization records are retrieved from the SYSAUTH table and are stored in the Authorization Extension. If the user is a member of one or more groups, the system determines whether the Authorization Extension associated with each of the groups is resident in main storage. If the Authorization Extension for a group is already resident in main storage, a pointer to it is stored in the corresponding entry in the user's Group Extension. If not, the record containing the group-name and a blank user-id is retrieved from the SYSGROUP table. This record contains the number of entries in the group's Authorization Extension. After allocating the main storage necessary to contain the Authorization Extension, the authorization records for the group are retrieved from the SYSAUTH table and are stored in its Authorization Extension. Whenever a user attempts to access data in a table which is owned by another user or the Data Base Administrator, the user's right to access the table must be determined. If access rights have not been granted to the PUBLIC but have been granted to individual users, the Authorization Extension associated with the user is searched to determine whether or not the user has been granted the right to perform the attempted data manipulation operation on the table. When a user or an application program disconnects from the Integrated Data Base Management System, the main storage allocated for the corresponding Authorization Extension is freed. Main storage allocated

for Authorization Extensions for any groups to which the user belongs may be freed if the user is the only member of the group who is connected to the system.

Access rights granted on a table can be revoked at any time. If one or more access rights granted to a user or group are revoked by the owner of the table or the Data Base Administrator, the corresponding authorization record in the SYSAUTH table is updated to reflect the new rights of the user or group. If the revocation of rights is such that the user or group retains no access rights to the table, the corresponding authorization record is deleted from the SYSAUTH table.

8.4.4 Data Base Control Blocks

A Data Base Control Block exists for each data base maintained by the Integrated Data Base Management System. Data Base Control Blocks are stored, as records, in the SYSDB system table. Each Data Base Control Block will contain the data base name, the user-id of the owner of the data base, a description of the data base, the data base classification, the date on which the data base was created and other data fields, flags and pointers.

A Data Base Control Block is created when a user issues a command to define a new data base to the system. Main storage is allocated for the new Data Base Control Block after which it is initialized and inserted in the SYSDB table. Data fields in a Data Base Control Block can be updated at any time by the Data Base Administrator.

When a user or an application program connected to the Integrated Data Base Management System attaches to a data base for processing, the Data Base Control Block for the data base is retrieved from the SYSDB table and placed on a two-way chain

of Data Base Control Blocks for data bases to which one or more users are attached. This chain is maintained in main storage in ascending logical sequence by data-base-name. If a Data Base Control Block for the data base to which a user is attaching already exists on the chain, there is no need to access the SYSDB table to retrieve the Data Base Control Block. A pointer to the Data Base Control Block for the data base to which a user is attached is stored in the User Control Block.

When a command is received that references the data base to which the user or application program issuing the command is attached, the corresponding Data Base Control Block is located using the attached data base pointer in the User Control Block. Since the Data Base Control Block contains a pointer to the Data Dictionary associated with the data base and a pointer to the chain of Relation Control Blocks for tables in the data base, it can be used to locate other control structures required to execute a command. Also, the user-id of the owner of the data base, which is contained within the Data Base Control Block, is used to assign either a temporary or permanent status to new data fields and tables defined for the data base. When there are no longer any users attached to a data base, its Data Base Control Block is removed from the Data Base Control Block chain and the main storage allocated for the Data Base Control Block is freed.

An existing data base can be removed from the system by its owner or by the Data Base Administrator. When a data base is removed from the Integrated Data Base Management System, the record containing the corresponding Data Base Control Block is deleted from the SYSDB table. This may cause records to be deleted from other system tables, as well. All data contained in tables in the data base are deleted from the system, also.

8.4.5 Data Dictionaries

A Data Dictionary is always associated in main storage with a Data Base Control Block. It contains a description of each data field contained in the corresponding data base. The Data Dictionary contains one entry for each data field defined for the data base. Each entry in a Data Dictionary is stored, as a record, in the SYSDD system table. Each Data Dictionary entry contains the name of the data field, the length of the data field, the storage format of the data field, the user-id of its owner and an indication of the units, if any, that are associated with the data field. The Data Dictionary will contain only one entry for each data field, no matter how many tables that data field is used in.

A Data Dictionary entry is created when a user issues a command to define a new data field for the data base to which the user is currently attached. The new Data Dictionary entry is placed in the existing Data Dictionary in main storage and is inserted into the SYSDD table. It will be marked as a permanent entry if the user defining it is the owner of the data base. Otherwise, it will be marked as a temporary entry and will be deleted from the SYSDD table when the user is no longer attached to the data base.

When the control structures associated with a data base are being loaded into main storage, the amount of main storage required for the Data Dictionary is computed. The number of entries in the Data Dictionary is stored in the Data Base Control Block. After allocating the main storage necessary to contain the Data Dictionary, the Data Dictionary entry records are retrieved from the SYSDD table and stored in the Data Dictionary. Whenever a user defines a new table in the data base or expands an existing table, the Data Dictionary associated

with the data base is searched to insure that all data fields in the table have been previously defined. When there are no longer any users attached to a data base, the main storage allocated for the corresponding Data Dictionary is freed.

An existing data field can be removed from a data base by its owner, the owner of the data base, or the Data Base Administrator, only if the data field is not currently being used within a table in the data base. When a data field is removed from a data base, the corresponding entry in the Data Dictionary is removed from main storage and the corresponding Data Dictionary entry record is deleted from the SYSDD table.

8.4.6 Relation Control Blocks

A Relation Control Block exists for each table maintained by the Integrated Data Base Management System. Relation Control Blocks are stored, as records, in the SYSREL system table. Each Relation Control Block will contain the table name, the user-id of the owner of the table, a description of the table, the temporary/permanent status of the table, the date on which the table was created and other data fields, flags and pointers.

A Relation Control Block is created when a user issues a command to define a new table to the system. Main storage is allocated for the new Relation Control Block after which it is initialized and inserted in the SYSDD table. It is also inserted in the chain of Relation Control Blocks pointed to by the Data Base Control Block for the data base containing the new table. Data fields in a Relation Control Block can be updated at any time by the Data Base Administrator.

When the control structures associated with the data base are being loaded into main storage, the Relation Control Blocks

for tables within the data base are retrieved from the SYSREL table and placed on a two-way chain originating at the Data Base Control Block. This chain is maintained in main storage in ascending logical sequence by table name. When a command is received that references a particular table in the data base to which the user or application program issuing the command is attached, the corresponding Relation Control Block is located by searching the chain of Relation Control Blocks emanating from the Data Base Control Block. Access to all data contained in a table is controlled through the Relation Control Block. When there are no longer any users attached to a data base, the main storage allocated for the Relation Control Blocks is freed.

An existing table can be removed from a data base by its owner, the owner of the data base or the Data Base Administrator. When a table is removed from a data base, the Relation Control Block is removed from the Relation Control Block chain, the main storage allocated for the Relation Control Block is freed and the record containing the corresponding Relation Control Block is deleted from the SYSREL table. This may cause records to be deleted from other system tables, as well. All data contained in the table and all superstructures created for the table are deleted from the system, also.

8.4.7 Domain Extensions

A Domain Extension is always associated in main storage with a Relation Control Block. It contains information about the data fields in the corresponding table. The Domain Extension consists of two sections; a primary section which contains one entry for each data field in the table and an auxiliary section which contains one entry for each data field used in a combination B-tree or inverted key field associated with the table.

Each entry in a Domain Extension is stored, as a record, in the SYSDOM system table. Each Domain Extension entry in the primary section contains the data field name, the column number of the data field in the table, the dimensionality of the data field, the starting location of the data field in each record, the length of the data field, a flag indicating whether or not an index exists on the data field and, if so, what type of index and, if an index does exist, a pointer to an index page. Domain Extension entries in the auxiliary section contain, essentially, the same information as those in the primary section except that they also include the key name.

Domain Extension entries are created when a user issues a command to define a new table, when new data fields are added to an existing table and when a new B-tree or inverted index is created on a combination of data fields in the table. When a new table is created, main storage is obtained for the Domain Extension. An entry is created for each data field in the table and stored in the Domain Extension. It is inserted in the SYSDOM table, also. When an existing table is expanded by adding one or more data fields, an entry is created for each data field. It is placed in the primary section of the existing Domain Extension and inserted in the SYSDOM table. When a B-tree or inverted index is defined on a combination of data fields in a table, an entry is created for each data field specified as part of the key field. They are placed in the auxiliary section of the Domain Extension and inserted in the SYSDOM table.

When the control structures associated with a data base are being loaded into main storage, the Domain Extension entries associated with each table in the data base are retrieved and placed in the appropriate Domain Extension. Each Domain Extension is linked to the corresponding Relation Control Block in main storage by a pointer in the Relation

Control Block. When a data manipulation command accesses data in a table, the Domain Extension is used to validate that the data fields or key fields specified in the command exist in the table, to select the access path which minimizes the number of records which must be accessed to satisfy the command and to locate the referenced data fields within the retrieved records. When the Relation Control Block which points to the Domain Extension is removed from main storage, the main storage allocated for the Domain Extension is freed.

When a table is removed from a data base, the main storage allocated for the Domain Extension to the Relation Control Block is freed and all records containing the corresponding Domain Extension entries are deleted from the SYSDOM table.

8.5 Queues

A queue is a two-way chain of Command Control Blocks. A queue is empty when it contains no Command Control Blocks. Some queues are used to transfer Command Control Blocks from one asynchronous process to another. Other queues are used to hold Command Control Blocks for commands that are awaiting the completion of an event.

8.5.1 The Command Queue

The Command Queue contains Command Control Blocks associated with interactive and application program commands which have been syntax checked. Command Control Blocks for interactive commands are placed on the Command Queue by the Interactive Command Processor. Command Control Blocks for application program commands are placed on the Command Queue by the Application Program Interface. When a Command Control Block is added to the Command Queue, it is placed at the end of the chain of Command Control Blocks already on the queue. Also, a

flag is set which activates the Monitor, if it is not already active.

When activated, the Monitor determines whether or not the tables required by the command whose Command Control Block has just been added to the Command Queue can be allocated to the command. If so, the Command Control Block for the command is removed from the Command Queue and the command is initiated. If not, it remains on the Command Queue until the necessary tables can be allocated to the command. Whenever a table becomes available, the Monitor is activated. The Monitor scans the Command Queue to determine if any command can be initiated.

8.5.2 The Initiator Queue

The Initiator Queue contains Command Control Blocks associated with commands for which execution can be started. That is, all tables required for execution of the commands can be allocated to them in the required mode. Command Control Blocks are placed on the Initiator Queue by the Monitor. When a Command Control Block is added to the Initiator Queue, it is placed at the end of the chain of Command Control Blocks already on the queue. Also, a flag is set which activates the Logical Interface, if it is not already active.

When activated, the Logical Interface attempts to select a command for execution from the Wait Queue, described below. Failing that, the Logical Interface selects the command whose Command Control Block is the first one in the Initiator Queue to be started. When a command is selected for execution from the Initiator Queue, its Command Control Block is removed from the Initiator Queue and the command becomes the executing command.

8.5.3 The Wait Queue

The Wait Queue contains Command Control Blocks associated with commands for which execution has been started but are now awaiting the completion of a I/O event. Command Control Blocks are placed on the Wait Queue by the Physical Interface when an I/O operation is started for the command. When a Command Control Block is added to the Wait Queue, it is placed at the end of the chain of Command Control Blocks already on the queue. Then, control is returned to the Logical Interface.

The Logical Interface scans the Wait Queue to determine if any of the commands therein can continue execution. A command whose Command Control Block is on the Wait Queue can continue execution when the I/O operation on which it is waiting completes. Since the Wait Queue is a first in - first out queue, the scan always begins with the first Command Control Block in the queue. When a command is encountered that can continue execution, its Command Control Block is removed from the Wait Queue and that command becomes the executing command. If no command on the Wait Queue can continue execution, the Logical Interface starts the execution of the command whose Command Control Block is first in the Initiator Queue.

8.5.4 The Output Message Queue

The Output Message Queue contains Message Request Blocks associated with output messages that are to be transmitted from the Integrated Data Base Management System to remote terminals. Message Request Blocks can be placed on the Output Message Queue by any software process that handles interactive commands. Each Message Request Block contains an indication of the message to be transmitted and the remote terminal to which it is to be sent. When a Message Request Block is added to the Output Message Queue, it is placed at the end of the

chain of Message Request Blocks already on the queue. Also, a flag is set which activates the Output Message Processor if it is not already active.

When activated, the Ouput Message Processor removes the first Message Request Block from the Output Message Queue and transmits the corresponding message to the specified terminal.

8.5.5 The Interactive Terminator Queue

The Interactive Terminator Queue contains Command Control Blocks associated with interactive commands for which execution has completed or has been aborted. Command Control Blocks for interactive commands which have completed normally are placed on the Interactive Terminator Queue by the Logical Interface. Command Control Blocks for interactive commands which have been aborted may be placed on the Interactive Terminator Queue by any software process that handles interactive commands. When a Command Control Block is added to the Interactive Terminator Queue, it is placed at the end of the chain of Command Control Blocks already on the queue. Also, a flag is set which activates the Interactive Command Terminator, if it is not already active. When activated, the Interactive Command Terminator removes the first Command Control Block from the Interactive Terminator Queue and performs the required processing to terminate the associated command.

8.5.6 The Application Terminator Queue

The Application Terminator Queue contains Command Control Blocks associated with application program commands for which execution has been completed. Command Control Blocks for application program commands which completed normally are placed on the Application Terminator Queue by the Logical Interface. Application Program Commands which contain syntax

errors are terminated abnormally. Their Command Control Blocks are placed on the Application Terminator Queue by the Application Program Interface. When a Command Control Block is added to the Application Terminator Queue, it is placed at the end of the chain of Command Control Blocks already on the queue. Also, a flag is set which activates the Application Program Terminator, if it is not already active. When activated, the Application Program Terminator removes the first Command Control Block from the Application Terminator Queue and performs the required processing to terminate the associated application program command.

SECTION 9 - SYSTEM SOFTWARE

9.1 System Architecture

This section describes the architecture of the Integrated Data Base Management System. The software has been divided into several asynchronous processes based on the functions which must be performed as a command proceeds through the system. Each of the processes are event driven. That is, a software process is executing in a "run" state only when an event occurs which indicates the software process must perform some function. Otherwise, the process is in a dormant or "wait" state. This software architecture, coupled with the use of the queues which are described in Section 8, permits several commands to be in different stages of processing with a minimum of delay.

The following subsections describe briefly each of the software processes that constitute the Integrated Data Base Management System. Certain programs, such as the System Generation Program and utility programs, are separate from the software which supports user processing. Also, there will exist a set of routines which will be stored in a library managed by the Integrated Data Base Management System. These routines will be loaded into main storage only when needed to support a user processing requirement. Figure 9-1 shows the software processes which constitute the system and the command flow through the system via the queues.

9.2 The System Generation Program

The System Generation Program is a stand-alone program which, except under extraordinary circumstances, is run only once to initialize the Integrated Data Base Management System. When invoked, the System Generation Program reads a set of input parameters specified by the Data Base Administrator which

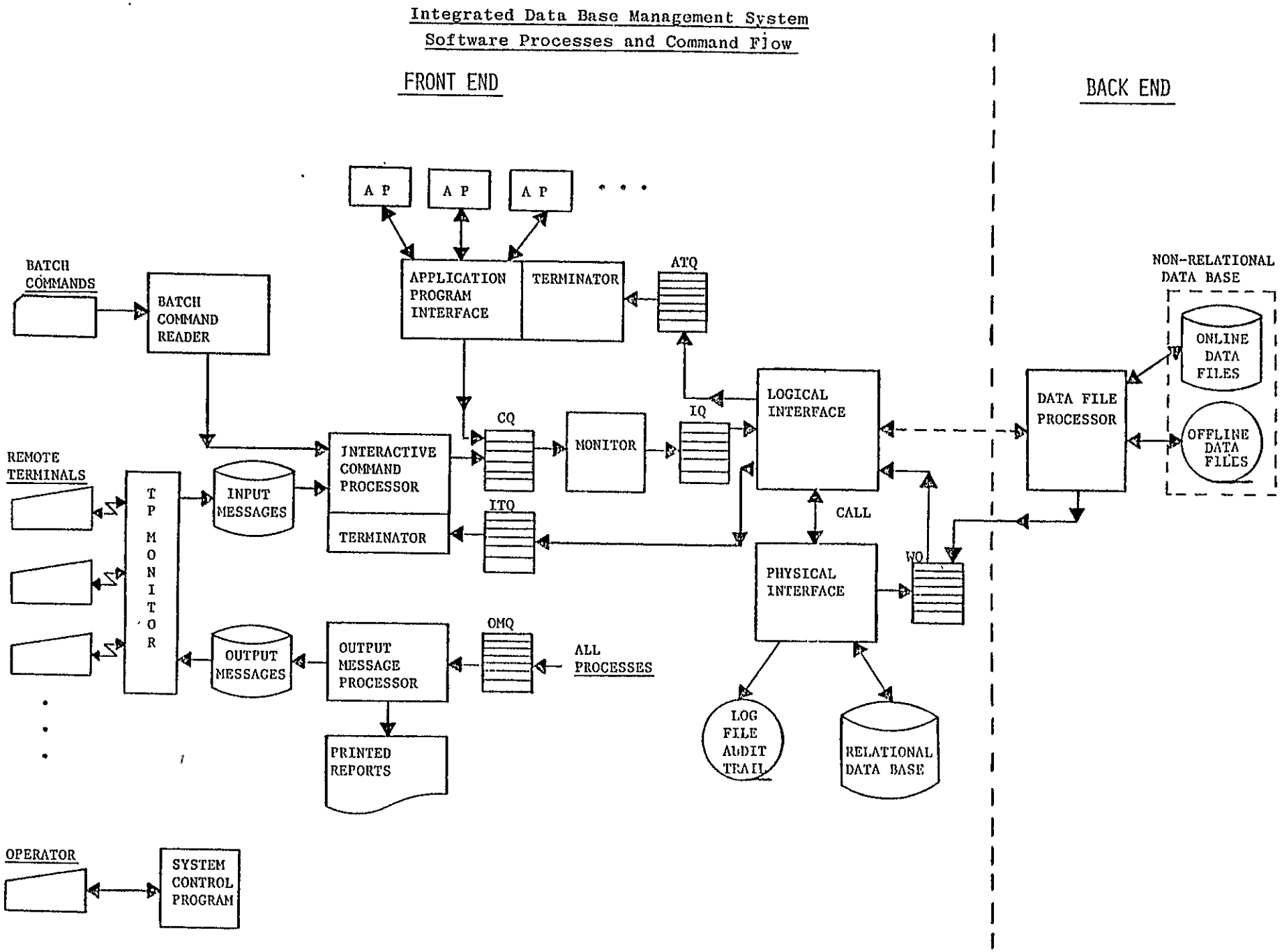

Integrated Data Base Management System
Software Processes and Command Flow
FRONT END
BACK END
A P
A P
A P
BATCH COMMANDS
BATCH COMMAND READER
APPLICATION PROGRAM INTERFACE
TERMINATOR
ATQ
NON-RELATIONAL DATA BASE
ONLINE DATA FILES
OFFLINE DATA FILES
LOGICAL INTERFACE
DATA FILE PROCESSOR
CQ
MONITOR
IQ
REMOTE TERMINALS
T P M O N I T O R
INPUT MESSAGES
INTERACTIVE COMMAND PROCESSOR
TERMINATOR
ITQ
CALL
PHYSICAL INTERFACE
WQ
OUTPUT MESSAGES
OUTPUT MESSAGE PROCESSOR
OMQ
ALL PROCESSES
PRINTED REPORTS
LOG FILE AUDIT TRAIL
RELATIONAL DATA BASE
OPERATOR
SYSTEM CONTROL PROGRAM

Figure 9-1

9-2

control the system initialization process. The input parameters include the initial password for the Data Base Administrator, the page size for the tabular data storage area and a list of online storage devices which can be mapped into the tabular data storage area.

After accepting and checking the input parameters, the modules which constitute the nucleus of the Integrated Data Base Management System are loaded. This code is used to perform most of the output during the initialization of the system. The online storage area which will contain tabular data is initialized by writing empty pages throughout the area. The System Generation Program will contain within it, the control structures associated with the Global Data Base, as well as the User Control Block for the Data Base Administrator. The control structures include the Data Base Control Block for the Global Data Base, its Data Dictionary, the chain of Relation Control Blocks for all system tables in the Global Data Base and their Domain Extensions. The control structures will reflect an empty Global Data Base. The System Generation Program will insert the control structure records representing the Data Base Administrator and the Global Data Base into the appropriate system tables. The order of processing of the various control structures is significant since the processing of one control structure may affect the contents of another.

After completing the output of the control structure records, the System Generation Program will output a block of control information to be used each time the program load operation is performed for the Integrated Data Base Management System. The control information will include a map to be used in converting page pointers in the tabular data storage area to online device addresses, a page pointer to the first free page in the tabular data storage area, page pointers to the pages containing the control structures which constitute the Global Data

Base and any other information which must be retained from one program load operation to the next. The control information may be stored in a specific location on a system resident pack so that the block can be loaded by the System Control Program when the Integrated Data Base Management System is started.

9.3 The System Control Program

The System Control Program provides the operator with control over the execution of the Integrated Data Base Management System. All operator communication with the system is via the System Control Program using a set of operator commands. Operator commands are accepted, checked and executed by the System Control Program. The operator commands allow the operator to start the system, obtain information concerning the operation of the system, stop the system and restart the system after a major malfunction.

During the program load operation which starts the system, the System Control Program loads the control information block, attaches all of the asynchronous processes and initializes all queues to an empty state. When an operator command is received which requests information, the System Control Program collects the required information and transmits it to the operator. When the operator stops execution of the system, the System Control Program performs the required procedures to terminate the processing of any commands in progress and stop the system. When the operator requests a system restart, the System Control Program performs the required restore operations such that the system can be restarted.

9.4 The Interactive Command Processor

The Interactive Command Processor actually consists of two asynchronous software processes; the Interactive Command Input

Processor and the Interactive Command Termination Processor. Each of these software processes is described briefly below.

9.4.1 The Interactive Command Input Processor

The Interactive Command Input Processor accepts interactive commands from either the telecommunications message handler or the Batch Command Reader. Messages received from the telecommuncations message handler were entered via a remote terminal while messages received from the batch command reader were entered via a card reader. Each message contains an identifier indicating its point of origin; either a particular remote terminal or the card reader.

After receiving a message, the Remote Terminal Communications List is searched to determine if an entry already exists containing the identifier. If not, a new entry is created containing the identifier and the message is associated with that entry. If an entry already exists, the message is associated with the existing entry in the list. If the message is not a continuation of a previously received message, the Interactive Command Processor constructs a Command Control Block for the message. The message is parsed and the syntax is checked. If, after parsing the message, the system expects a continuation message, a flag is set in the corresponding entry in the Remote Terminal Communications List and the next message is accepted. If no continuation is expected and a syntax error occurs, a Message Request Block is placed on the Output Message Queue which will cause a diagnostic message to be transmitted to the terminal from which the message originated or, if the message entered via the batch command reader, the diagnostic message is printed. If a syntax error is encountered while processing a command, the command is terminated by placing its Command Control Block on the Interactive Terminator Queue. If no errors are found in the command, the command

is introduced to the monitor by placing its Command Control Block on the Command Queue.

A command entered through a remote terminal can be aborted by sending a special "break" character in a message. When a message is received containing the special character, the command being processed from that terminal, if any, is immediately terminated no matter what stage of processing it may be in. The termination may require some amount of restoration of information to remove the effects of the command on the system. The Command Control Block is placed on the Interactive Terminator Queue so that it may be purged from the system.

9.4.2 The Interactive Command Terminator

The Interactive Command Terminator is an asynchronous software process which performs all actions necessary to complete the processing of an interactive command. The Command Control Blocks for the interactive commands to be terminated are obtained from the Interactive Terminator Queue. When a Command Control Block is placed on the Interactive Terminator Queue by another software process, a flag is set placing the Interactive Command Terminator into the run state, if it is not already executing. The Interactive Command Terminator removes the first Command Control Block from the Interactive Terminator Queue. The entry in the Remote Terminal Communications List with which the command is associated is modified to remove all reference to the command being terminated. The main storage allocated for the Command Control Block is freed and all other processing required to purge the command from the system is performed.

9.5 The Application Program Interface

The Application Program Interface consists of two modules

which provide the facilities by which an application program in one region of main storage can communicate with the Integrated Data Base Management System in another region and two asynchronous software processes: the Application Program Command Processor and the Application Program Command Terminator. Each of the communication modules and the software processes are described briefly below.

9.5.1 The Communication Modules

Two modules are used to provide communication between an application program and the Integrated Data Base Management System. They are the Application Program Communication Module and the Cross-Boundary System Routine.

A copy of the Application Program Communication Module must be included in the load module for each application program which issues commands to the Integrated Data Base Management System. The Application Program Communication Module is entered when a CALL to the Integrated Data Base Management System is executed in the application program. It creates an Application Program Request Block containing the address of each argument in the command and invokes the Cross-Boundary System Routine.

The Cross-Boundary System Routine places the Application Program Request Block on the Application Program Request Queue and sets a flag to place the Application Program Command Processor in the run state, if it is not already executing. Control is returned to the Application Program Communication Module where the application program is placed in a non-executing wait state. When the Integrated Data Base Management System completes the processing of a command, control is returned to the Cross-Boundary System Routine where argument values are transferred to the application program and the application program is placed in the run state again.

9.5.2 The Application Program Command Processor

The Application Program Command Processor is an asynchronous software process which accepts commands from application programs. When an Application Program Request Block is placed on the Application Program Request Queue by the Cross-Boundary System Routine, a flag is set placing the Application Program Command Processor into the run state, if it is not already executing. The Application Program Command Processor removes the first Application Program Request Block from the Application Program Request Queue and searches the Application Program Communication List to determine if an entry already exists for the application program that issued the command. If not, a new entry is created for the application program and the Application Program Request Block is associated with that entry. If an entry already exists, the Application Program Request Block is associated with the existing entry in the list.

A command Control Block is constructed for the command and the contents of the argument list associated with the command are checked. If an error is detected in the argument list, the command is terminated by placing its Command Control Block on the Application Terminator Queue. If no errors are found in the argument list, the command is introduced to the Monitor by placing its Command Control Block on the Command Queue.

9.5.3 The Application Program Command Terminator

The Application Program Command Terminator is an asynchronous software process which performs all actions necessary to complete the processing of an application program command. The Command Control Blocks for the commands to be terminated are obtained from the Application Terminator Queue. When a Command Control Block is placed on the Application Terminator Queue by another Software Process, a flag is set placing the

Application Program Command Terminator into the run state, if it is not already executing. The Application Program Command Terminator removes the first Command Control Block from the Application Terminator Queue. The entry in the Application Program Communications List with which the command is associated is modified to remove all reference to the command being terminated. The main storage allocated for the Command Control Block is freed and all other processing required to purge the command from the system is performed. Finally, the Cross-Boundary System Routine is invoked to transfer argument values to the application using addresses in the Application Program Request Block and to place the application program back into the run state.

9.6 The Monitor

The Monitor is an asynchronous software process which handles resource allocation for commands and dispatches commands to the Logical Interface for execution. The Command Control Blocks for the commands to be dispatched by the Monitor are obtained from the Command Queue. When a Command Control Block is placed on the Command Queue by either the Interactive Command Input Processor or the Application Program Command Processor, a flag is set placing the Monitor into the run state, if it is not already executing. Whenever a Command Control Block is placed on the Command Queue, the Monitor determines whether or not the command can be dispatched immediately. A command can be dispatched if all tables which it requires for processing can be allocated to the command in the proper mode. If the command can be dispatched by the Monitor, its Command Control Block is removed from the Command Queue and placed on the Initiator Queue for processing by the Logical Interface. If the command cannot be dispatched immediately by the Monitor, its Command Control Block remains at the end of the Command Queue.

Whenever the status of a table in the system changes, a flag is set. This flag causes the Monitor to enter the run state, if it is not already executing. When the status of one or more tables changes, the Monitor scans the Command Queue to determine whether or not any command on the queue can be dispatched. If so, the Command Control Block for the command to be dispatched is removed from the Command Queue and placed on the Initiator Queue for processing by the Logical Interface. If no command on the Command Queue can be dispatched, the Monitor is placed in the wait state until either a new command is placed on the Command Queue or the status of a table within the system changes.

9.7 The Logical Interface

The Logical Interface, along with the Physical Interface and the Data File Processor, forms a single asynchronous process which performs the command dependent processing. The Command Control Blocks for commands to be processed by the Logical Interface are obtained from the Initiator Queue. When a Command Control Block is placed on the Initiator Queue by the Monitor, a flag is set placing the Logical Interface into the run state, if it is not already executing. The Logical Interface removes the first Command Control Block from the Initiator Queue and begins the command dependent processing for that command. When the command that is currently being processed by the Logical Interface performs an Input/Output operation or requires the loading of a library routine, its Command Control Block is placed on the Wait Queue and the execution of that command is suspended.

After placing a Command Control Block on the Wait Queue, the Logical Interface scans the Wait Queue to determine whether or not any of the Input/Output or library load operations on which the commands are waiting have completed. If

so, the corresponding Command Control Block is removed from the Wait Queue and execution of the command continues. If no command on the Wait Queue can continue execution, the Logical Interface removes the first Command Control Block from the Initiator Queue and starts the execution of that command. If the Initiator Queue is empty, the Logical Interface is placed in the wait state until a command is placed on the Initiator Queue or a command on the Wait Queue can continue execution.

When a command completes execution or is aborted, the Logical Interface places its Command Control Block on either the Interactive Terminator Queue or the Application Terminator Queue, depending upon the source of the command. A flag is set to place the corresponding terminator into the run state, if it is not already executing.

9.8 The Physical Interface

The Physical Interface consists of subroutines which support Input/Output operations for tabular data. The subroutines are entered from the Logical Interface via a CALL. The Physical Interface provides the buffer control facilities for transferring pages between main storage and the tabular data storage area on direct access devices. It also maintains the superstructures associated with tabular data. The Physical Interface controls the logging of page images and provides the capability of dynamically restoring data bases should a command terminate abnormally. Also, the Physical Interface is responsible for the logging of transactions to provide an audit trail and record images to allow recovery should a malfunction cause system failure.

Of course, the primary function of the Physical Interface is the transfer of tabular data into and out of main storage. When an Input/Output operation is to be performed, the Input/Output subroutine is entered. This routine starts the data transfer and then places the Command Control Block for the command which initiated the data transfer on the Wait Queue. After placing the Command Control Block on the Wait Queue, the Input/Output subroutine returns control to the Logical Interface which selects the next command to be executed.

9.9 The Data File Processor

The Data File Processor consists of a set of subroutines which supports Input/Output operations for sequential data files. It is entered from the Logical Interface via a CALL. The Data File Processor is responsible for locating an existing data file for an input operation and, if necessary, issuing device mounting instructions to the operator. For an output operation, the Data File Processor assigns a unique data identifier to a new data file, locates direct access space for storage of the data file, if necessary, and updates the SYSCATL system table to reflect the existence of a new data file or another copy of an existing data file.

The Data File Processor supports the loading of library routines to perform special processing on data files. This includes functions such as regridding, windowing and plotting. These routines will be resident in the system library until a command is issued which specifies one of the functions performed by a library routine. The Data File Processor will cause the proper module to be loaded from the library and pass control to the routine after it has been loaded. When the routine has completed its processing, control is returned to the Data File Processor.

The Data File Processor uses several system standard formats for the internal handling of data files. Data files in their original format are converted to one of the system standard formats automatically by the Data File Processor using library routines. Thus, when an off-line copy of a data file in its original format is loaded onto a direct access device, the Data File Processor uses information contained in the corresponding record in the SYSCATL table to locate and load the proper format conversion routine from the system library. The format conversion routines read a data file in its original format and write either a copy of the data file or a new data file, which is a subset of the original, in one of the system standard formats.

The technique of using loadable routines to perform operations on data files and to perform format conversion provides an open-ended facility for data file processing. New routines can be added to the system library to perform new functions on data files. Also, new format conversion routines can be added to convert new original formats into system standard formats. Naturally, there will be certain programming conventions that must be adhered to when creating the new routines and it is expected that the Data Base Administrator will control the addition of new routines to the system library.

The Data File Processor supports the processing of data files by application programs. It provides the capability of positioning a data file to a particular logical record based upon data values in each record or based upon a relative record number in the data file. It also provides for the deblocking of physical records into logical records during input and the blocking of logical records into physical records during output.

9.10 The Output Message Processor

The Output Message Processor is an asynchronous software process which constructs and transmits messages to either remote terminals or a line printer. The Message Request Blocks for the messages to be transmitted by the Output Message Processor are obtained from the Output Message Queue. When a Message Request Block is placed on the Output Message Queue by another software process, a flag is set placing the Output Message Processor into the run state, if it is not already executing. The Output Message Processor removes the first Message Request Block from the Output Message Queue. The message identified in the Message Request Block is constructed and transmitted to the remote terminal specified in the Message Request Block or to the line printer. After transmitting the message, the main storage allocated for the message request block is freed and the next Message Request Block is obtained from the Output Message Queue. If the Output Message Queue is empty, the Output Message Processor is placed into the wait state until a Message Request Block is placed on the Output Message Queue.

APPENDIX A - THE RELATIONAL MODEL OF DATA

A.1 Description and Definition

"Data model" is the technical term used to describe a user's conceptual view of the contents and logical structure of a data base. The relational data model is, at the same time, one of the conceptually simplest models, yet one of the most sophisticated. To begin with, within the relational data model all information is contained in one or more flat tables. Certainly, this is a common enough approach to data organization; human beings have been tabulating data and looking up information in tables for as long as there have been writing materials and some sort of script to write in.* Given the ubiquity of tables of information in our daily lives, it may come as something of a surprise that the relational model of data is founded on a rigorous mathematical base. Moreover, it can be mathematically demonstrated that any data relationship which can be represented in a competing data model (hierarchical, network) is representable in the relational model of data, while the converse is not always true.

The term "relation" has a rigorous mathematical definition. Given sets D_1, D_2, ..., D_n (not necessarily distinct), R is a relation on those sets if it is a set of n-tuples $<d_1, d_2, \ldots, d_n>$ such that d_1 is from D_1, d_2 is from D_2, and, in general, d_i is from D_i for $i = 1, 2, \ldots, n$. To be mathematically consise, R is a relation on the sets D_1, D_2, ..., D_n if it is a subset of the Cartesian cross product $D_1 \times D_2 \times \ldots \times D_n$.

*See, for example, Knuth, D. E., "Ancient Babylonian Algorithms", Communications of the ACM, vol. 15, No. 7 (July 1972) or Boyer, C.B., A History of Mathematics, Wiley & Sons, (1968).

This mathematical definition gives rise to much of the nomenclature used in the relational data model. The sets D_1, D_2, etc., are called domains and n is called the degree of the relation. When n = 1 the relation is called "unary", when n = 2 it is called "binary" (and the n-tuples are called "ordered pairs"), when n = 3, the relation is called "ternary", and for larger values of n (or when n is unknown) the term "n-ary" is generally used. In the standard nomenclature the term n-tuple is usually shortened to tuple, and it is a property of the above formal definition of relations that the tuples are assumed to be in random order.

One term which requires careful definition is "attribute". An attribute is a name assigned to a domain set reflecting its usage within the relation. Whereas the domains are not distinct, the attributes of a relation must be distinct. To see the difference between domains and attributes consider a relation describing a group of tropical storms. One attribute would be the name of the storm and two others might be the date it formed and the date it broke up. Both of these latter two attributes are from the same domain - the set of all calendar days covered by the study - but the meaning of the elements of that domain when used in the "start date" attribute is different from the meaning of dates used in the "end date" attribute.

At this point, it is worthwhile to stop to examine the correspondences between tabular nomenclature and relational nomenclature. An n-ary relation is equivalent to a flat table with n columns. The attributes are equivalent to the columns and the tuples represent the rows of the table. It is also possible to relate entities in the relational data model to terms and concepts used in standard data processing, but this requires a caveat or two. For example, it is possible to think of a tuple as a record and an attribute as the name of a field, but a file is a physical entity, as well as a logical entity, while

tables and relations are abstract concepts. Depending upon the implementation details, a single physical file may hold more than one relation, or a single relation may span multiple physical files. The table below, summarizes the correspondences between tabular, relational, and data processing nomenclature.

Relational	Tabular	Data Processing
relation	table	(logical) file
attribute	column	field name
tuple	row	record
degree	No. of columns	No. of fields

Table A-1: Terminology Correspondences

Two terms have been borrowed from data processing nomenclature which do not have a common name in tabular terminology. One of these is data item*, which refers to the contents of a single field of a record, and which, by extension, is used to refer to the value of a particular attribute in a given tuple. The other term is key, which refers to an attribute or collection of attributes whose values uniquely determine the tuples they belong to. If there are multiple keys (i.e., "candidate keys"), then one of them is usually designated as a primary key. In business-oriented implementations, it is not unusual to see the tuples stored in sorted order on the primary key, but this is an implementation detail of specific systems and is not a property of the relational model per se.

For the balance of this appendix, the terms "column" and "attribute" will be used interchangeably and, likewise, for "table" and "relation". The terms "row", "tuple", and "record" will also be treated as synonyms.

*The term "component" is sometimes used for "data item" in relational terminology.

A.2 Normalization

A.2.1 First Normal Form

Except for very trivial examples, there will not be a single, unique way to represent a collection of data (i.e., a data base) as a group of relations. Some table layouts are easier to work with than others and, particularly when the data base is dynamic, careless structuring of the data base can lead to problems. Fortunately, it is possible to define "normal forms" for table layouts based on data dependencies which will circumvent most of these problems.

One type of problem occurs when an attribute can be decomposed into sub-data items which may be of interest to a user. For example, it is possible to store a date as a six character alphanumeric string representing day, month, and year or year, month, day or some such combination. In this form "date" can be a single attribute. However, if this is done it becomes impossible, within the relational framework, to handle a request such as "fetch all table entries where the year is 1977". Since year is not an attribute of the relation, it is necessary to rephrase the request in the more awkward form "fetch all table entries where date is between 1 January 1977 and 31 December 1977". Similarly, a location on the earth's surface can be a single attribute (named, perhaps, "location") or expressed as a pair of attributes - latitude and longitude. The former approach, using the single attribute "location", would make it impossible to retrieve tuples based on latitude value even though the information is implicitly present, because "latitude" would not be an attribute of the relation in that formulation.

A related problem can be illustrated by the following example. Suppose we wish to set up a data base for presidential elections. One table might have election year, (primary

key), winner, winning party, winner's electoral votes, loser, losing party, and loser's electoral votes as its set of attributes. (Note, by the way, that winner and loser are defined over the same domain, the set of all presidential candidates and, likewise, the two "party" attributes are defined over a common domain. Also, both "electoral vote" attributes are defined over the set of all nonnegative integers.) However, while we are accustomed to thinking of the United States as having a two party system, in many years there have been more than two major party candidates (in the election of 1860, Lincoln ran in a four candidate field) and, of course, George Washington ran unopposed. Thus, while most tuples would have single values for the attributes loser, losing party, and loser's electoral votes, some tuples would have two values for each of this set of attributes, some would have three values, a couple of tuples would have none. As awkward as this is from an implementation standpoint, it is even more awkward for a user to work with. The solution is to split this table into two tables, one (keyed on election year) with the attributes "election year", "winner", "winning party", and "electoral votes", while the other (keyed on election year and loser, together) would have "election year", "loser", "losing party", and "electoral votes" as its attributes, with unique values of each attribute of each tuple.

These considerations lead to the concept of a first normal form. First normal form has a rigorous mathematical definition, but it can be easily summarized as follows:

> A relation is in first normal form if each attribute is single-valued and nondecomposable.

In order for a relational data base management system to work properly, all relations must be in first normal form.

A.2.2 Anomalies and Higher Normal Forms

A relational data base can exhibit three kinds of misbehavior, called "anomalies" in the literature, even when all of its relations are in first normal form. The first of these is called the update anomaly. Consider again the "election" relation with the attributes election year, winner, winning party, and electoral votes. Suppose we have the two tuples:

<1968, Nixon, Democrat, 301>
<1972, Nixon, Democrat, 520>

If we discover this mistake while processing the 1972 tuple, for example, we must be careful to change Nixon's party in both tuples. Moreover, we must check the "election losers" relation to look for tuples containing Nixon to correct his party affiliation there as well, or else the data in this hypothetical data base would have inconsistent facts about Nixon. Since one of the most important goals of a data base management system is to maintain data consistency, this is perhaps the most serious of the three anomalies.

The second type of anomaly is called the insertion anomaly. If the relations are not well chosen, it may be impossible to represent certain facts in the data base. For example, it is not possible to represent the fact that Ronald Reagan is a Republican or that Scoop Jackson is a Democrat in this data base since these men were not their party's candidate in 1976. More seriously, between August, 1974, and November, 1976, it was not possible to represent the fact that Gerald Ford is a Republican in the hypothetical data base, since he was not a candidate for either the presidency or vice presidency until the 1976 election. This could be a serious problem if the point of the data base was to supply data about presidents.

The final anomaly, the deletion anomaly, is harder to

illustrate. Suppose that a data base exists to support a manufacturing or wholesaling activity, with a relation having the name "item reorder" and attributes "item number", "supplier", "supplier address", and "minimum reorder qty". Suppose further that supplier XYZ is a supplier for item W, and that W is the only item XYZ supplies. If the firm decides not to order any more of item W and deletes this tuple then, since the tuple has the only occurrence of supplier XYZ in the entire relation, we also lose XYZ's address. If the firm ever intends to deal with XYZ in the future, this loss of information will be an unwanted side effect.

Second and third normal forms were developed as tools to help data base designers select good sets of relations, relations which avoid the three anomalies described above. Second normal form is primarily of historical interest, as a step towards development of third normal form. Let A and B be two attributes in a relation R. If knowledge of the value of A uniquely determines the value of B, (e.g., "supplier" determines "supplier address" and "winner" determines "winning party") then we can call A a determinant. R will be in third normal form if it is in first normal form and every determinant is a candidate key.

Recently a fourth normal form has been defined to handle yet another type of problem. If an attribute A determines a <u>set</u> of values for attribute B, then the relation is in fourth normal form if A is a determinant of all remaining attributes in the relation. There is no specific anomaly associated with relations in third but not fourth normal form, but such relations are not as easy to maintain properly as fourth normal form relations.

These higher normal forms are simply formal ways to structure sets of relations so that each relation expresses a

single concept. It should be emphasized that these higher normal forms are not required for a relational data base to function. For that, only first normal form is required. Rather, these normal forms should be considered by a data base designer to be guidelines for selecting sets of tables and table layouts that are easy to work with and easy to maintain. In particular, the design of the Integrated Data Base Management System will allow it to function quite well without necessarily having the data in any normal form higher than first normal form. The features which permit this are:

(1) Use of higher level data sublanguages,
(2) Allowance for null attributes in tuples, and
(3) Non-necessity for keys and random ordering for stored tuples in data files.

Higher level languages are particularly useful for finessing the update anomaly. Instead of looking up the 1968 and 1972 tuples in the hypothetical "elections" relation to change Nixon's party affiliation to Republican, a command in our system's query language would say:

```
UPDATE ELECTIONS
(WINNING PARTY = REPUBLICAN)
WHERE ELECTIONS.WINNER EQ NIXON #
```

(This would still require looking for Nixon in the "losers" relation.)

Since keys are not an integral part of the storage and retrieval operations in the system's physical interface, and since null values are permitted for any attribute of a tuple (the latter feature is not possible without the former), it is possible to store facts in a relation maintained by this system even though a tuple cannot properly be defined for the fact.

For example, tuples such as < -, Reagan, Republican, - > and < -, Jackson, Democrat, - > could be added to the "losers" relation described earlier. While this does not provide a complete solution to the insertion and deletion anomalies, it does partially mitigate their effect.

A.3 Relational Operations and Query Languages

There were three views of relations offered previously and each of these views suggests a series of basic operations which ought to be applicable to relations in a relational data base.

One of the views was mathematical, the perception of the relations as sets of n-tuples. Hence, it follows that the common set operations such as union, intersection, and difference should be performable on relations provided, of course, that the two sets to be operated on are compatable (i.e., that the two relations are defined over the same set of domains taken in the same sequence).

The view of a relation as a table implies that table look-up operations are applicable to relations. A basic set of tabular operations can be defined: select a column or set of columns ("project"), select a row or set of rows based on some logical criterion ("restrict"), and create a new, larger table by cross-referencing two tables of lower degree over a common domain ("join"). Just as the set theoretic operators take two sets as input and yield a single set as output, these tabular operations take one or two (in the case of "join") tables as input and produce a new table as output. Finally, the view of a relation as a file of records suggests that data manipulation commands such as insert, delete, and update should be supported (data retrieval, of course, is equivalent to a restriction, or a restriction followed by a projection

over the desired attributes). These nine operations constitute a relational algebra for manipulating relations, and this relational algebra provides the foundation for query languages to support information handling in a relational data base. We say that a data sublanguage is relationally complete if it is possible to perform all relational operations in the relational algebra using that language.

A rather different point of view is adopted by query languages based on relational calculus. It is possible to view a relation as a proposition in the first order predicate calculus, and to view the individual n-tuples as "axioms." In languages based on relational calculus a user formulates his or her query as a statement in the first order predicate calculus defining a new relation, where that statement may well include universal and existential quantifiers ($\forall$, "for all," and $\exists$, "there exists", respectively). Codd[8] was the first to propose the relational calculus, and he went on to demonstrate that the relational calculus is relationally complete. The advantage of relational calculus over relational algebra is nonprocedurality, that is, the user formulates a query by defining the results of the retrieval and not as a series of processing steps. It is left for the system to interpret the query statement and to formulate its own retrieval procedures. Nonetheless, the unfamiliar and highly mathematical notation used by the relational calculus appears to have been an impediment to its widespread acceptance. Recently, however, a relational calculus-based language named QUEL has been developed by Stonebraker, et. al.[37], which dispenses with the need for the quantifiers and which makes heavy use of English key words. Such a language would presumably have a higher degree of user acceptance.

Another approach to a nonprocedural query language made palatable to casual users by use of English key words is embodied in IBM's experimental SEQUEL language[4], which is based on the concept of "mapping." In a mapping, known quantities -- specified in a Boolean predicate -- are mapped into an unknown quantity -- the data items to be retrieved by means of the relations in the data base, much as mathematical functions may be viewed as mapping sets into other sets. In SEQUEL, mappings may be nested inside other mappings. This feature gives SEQUEL its power and yet its most significant drawback; although the language is purportedly "directed at the nonprogramming professional,"[3] examples in Chamberlin[4] and Date[10] suggest that formulating queries in SEQUEL would be difficult for users not trained in recursive programming languages such as Algol.

A fill-in-the blanks language call Query-by-Example has been developed by Zloof[40]. The user enters the names of the relations against which a query is to be made on a graphics CRT terminal and the system responds by drawing in a skeleton table with columns and headings. At that point the user fills in one or more rows with examples of the desired result. The known values are keyed in directly while unknowns are represented by arbitrarily-chosen sample values which are flagged in some way. Psychological studies of user interactions with Query-by-Example[38] demonstrate that Query-by-Example is easy to use, particularly for the casual or novice user; has a high degree of retention for infrequent users; and is "behaviorly extendable," that is, a user can start by learning just enough of the language to get by and add to his or her knowledge as necessary. However, although a "linearized" version of Query-by-Example exists for use with batch input or non-graphics terminals, that form of the language is more bulky to use and not nearly as convenient.

Finally, there has been much interest in the use of natural language (specifically English) as a very high level query language. In particular, there has been much research on the part of artificial intelligence theorists on development of a natural language interface for relational data base management systems (the consensus among researchers is that the relational data model facilitates natural language interface development to such a degree that if a natural language front end cannot be developed for a relational system, then it cannot be developed for a data base management system at all). There has also been much work done on automatic translation of questions into first order predicate calculus, so that a relational calculus-based system may yet be commercially feasible. But a true natural language interface is still a long way off.

A.4 History

The late 1960's saw the development of several artificial intelligence-oriented systems based on the storage of data as a set of binary relations (such as MOTHER-OF<JACK,MARY>). However, textbooks on data base management systems (e.g., Date[10], Martin[23]) and survey articles (Chamberlin[3]) are unanimous in pointing to the 1970 article by E. F. Codd[6] as the seminal paper providing the impetus for the theoretical and practical development of the relational model of data. A subsequent series of articles by Codd continued to develop the theoretical foundations of the relational data model, including definition of a "relational algebra" and specification of prototype data sublanguages based on first order predicate calculus and on the relational algebra[8], development of the theory of normalization[7], and conceptual design of an English language interface between a casual user and a relational data base management system[9]. It is rare in any science for a single individual to provide nearly all of the theoretical basis for a major new

branch of technology, and rarer still in computer science. Nonetheless, it is clear that Codd is the founding father of the relational model of data.

Curiously enough, Codd does not appear to have made a direct contribution to the implementation of any working prototype relational data base management system. Most prototype systems have been constructed in a university environment or in IBM research laboratories (for a list and description of these systems and the following two see Chamberlin[3] or Date[10]). General Motors implemented a system called RDMS in 1972 (not to be confused with MIT's system of the same name) and RISS was built for Forest Hospital in Des Plaines, Illinois. Two of these systems - RISS and INGRES - are commercially available for use on DEC PDP-11's. The vast bulk of research on relational data base management systems has been conducted under IBM auspices (all of the people mentioned in this section, Chamberlin, Codd, Date, and Martin, are IBMers, as is Zloof the person who developed Query-by-Example). However, IBM has been careful to label all of its work on the relational data model as "experimental", and it is not likely that an IBM product in this area will soon be forthcoming.

A.5 The Advantages of the Relational Model of Data

There are two competitive data models, the "hierarchical" model typified by IBM's IMS system and the "network" model devised by the Data Base Task Group of the CODASYL committee[5]. It is sufficient, however, to compare the relational data model to the CODASYL model since the hierarchical data model is not general - there are data relationships which can be represented in the relational data model and in the CODASYL model which cannot be represented in the hierarchical model.

The primary advantage of the relational data model viz-a-viz the CODASYL model is data independence. There are two forms of data independence - physical independence and logical independence. Physical independence means that the user should be shielded from details of the physical storage of the data, including character representation methods, byte size, record blocking, physical access method, etc. The relational model functions quite well in this respect, since the user sees only the tables and attributes and sees nothing of the underlying physical implementation. In the CODASYL model, the files may be envisioned as labeled vertices of a directed graph with labeled arcs, where the labels on the arcs define relationships between the entities of one file and the entities of another. Typically, a CODASYL model is implemented with pointers from one record in a file pointing to a record or linked list of records in another file, and so the user must often be aware of decisions made by the Data Base Administrator concerning physical record placement and access paths. In the CODASYL model, both structural and nonstructural features, e.g., details of storage structure and access strategy, are interwoven with the logical structure. Physical structure changes which would be transparent to a user of a relational data base would not be transparent to the user of a CODASYL data base.

Logical independence is generally defined to mean that, within reason, application programs which operated properly before a change to the logical structure of the data base should continue to work after the change. Here, again, the relational model has a major advantage over the CODASYL model. Changes such as adding a column cause no problem to the user, although small changes in information can cause the data to be unnormalized and require major restructuring of the data base. However, it is equally possible that small changes, for example changing a one-to-one relationship to a one-to-

many or, especially, a one-to-many relationship to a many-to-many, can also cause nontrivial restructuring of a CODASYL data base. Consequently, it is generally conceded that the CODASYL data model has considerably less logical independence than the relational data model.

Another major advantage of the relational data model over the CODASYL model is its flexibility, although it is not so much that the relational model is flexible as that the CODASYL model is inflexible. Recall that relationships and access paths must be formally specified in the CODASYL data definition schema, and if the data base administrator should happen to overlook some relationship between records in a pair of files, then users of the system will be unable to respond to a query which requires that (missing) access path.

Two interwined issues are complexity and clarity and, here again, the relational data model has the advantage. The CODASYL model has no fewer than six data constructs, any of which can bear information which could not otherwise be derived. In the relational model there is precisely one such data construct (the n-ary relation). Moreover the set construct, which supports one-to-many relationships, performs three roles:

(1) It carries information,

(2) It defines access paths, and

(3) It provides a mechanism for integrity constraints.

The multiplicity of information-bearing constructs and the multiplicity of roles make it hard to present the contents and interrelationships in a CODASYL data base consisely. This is reflected in Martin's statement[23] that "a badly drawn schema can confuse rather than clarify, and one often sees badly drawn schemas" (pg. 83).

The simplicity and clarity of the relational data model contribute to ease of use, particularly for the untrained, nontechnical, and/or casual user. This is a point conceded by even the most outspoken advocates of the CODASYL model.

The reader might well ask why, if there are all of these fore-mentioned advantages of the relational model over the CODASYL model, there should be any advocates of the CODASYL data model at all. One reason may be that the CODASYL data model provides the user with highly visible navigation routes or access paths through the data base and, consequently, the data sublanguages for manipulating data in a CODASYL data base tend to be procedural in nature. That is, a query is input to a CODASYL data base management system as a series of steps to be taken which will derive the intended answer. While this will hold back the untrained or trained but casual user, direct control of data access provides a feeling of intimacy with the system which can be very important to some classes of user.

Associated with this is the generally-accepted (though not backed up by hard proof) point of view that CODASYL data base management systems are more efficient than relational systems. The CODASYL data model trades off data independence for efficiency's sake and, in some implementations, the quest for machine efficiency has been taken to the point where a user may be accessing a record while it is being updated to save the overhead involved in testing, setting, and releasing concurrency control locks. However, there are perils in evaluating system efficiency without consideration of whether the users are capable of making the most efficient use of their own time. Everyone understands that there is a tradeoff between main memory and execution time in designing a system, but fewer people seem prepared to grasp the fact that there is a tradeoff between system efficiency and user efficiency. The latter is, granted, hard to quantify since it includes not only the time a user must spend devising and inputting his query, but also

hidden costs in demands on the Data Base Administrator's time and ingenuity, training time to teach new users how to use the system, etc. However, problems with user efficiency can cause a system to be under-utilized and, oftentimes, abandoned (see Lucas[22]). It is believed that this system will be competitive with existing commercial CODASYL data base management systems in terms of system efficiency, and that the many user-oriented advantages offered by the relational data model will help the user and system together achieve their fullest potential.

APPENDIX B - ADDITIONAL TOPICS

B.1 Dynamic Memory Allocation

B.1.1 Approaches to Dynamic Memory Allocation

In order to maintain the User Control Blocks, Relation Control Blocks, the command queues, buffers for processing large tape files and on-line data files, etc., with any degree of efficiency it might be necessary for the Integrated Data Base Management System to hold a large, contiguous block of memory and to allocate and deallocate portions of this "free space" for control blocks, buffers, and so forth as needed. There are three basic approaches to managing this free space: fixed-size pages, variable-sized allocation with a free space list, and "buddy" methods. Each of these methods is discussed below. Whether or not one of these techniques will be included as part of the Integrated Data Base Management System will depend upon the operating system on which it is implemented.

Paging is the easiest approach to implement and its memory overhead -- a single bit map with one bit per page -- is quite low. However, even when the system supports more than one page size there will normally be a certain amount of wasted space within a page where the space required is less than the size of the page. This wasted space is called "internal fragmentation", and it can be a serious problem leading to system degradation.

A very different approach is to allocate precisely as much memory as is required to service any given request. Such an approach results in the free space being checker-boarded into blocks or areas which are in use and blocks which are available for allocation, where it is rare for any pair of blocks to be the same size. Typically, the

available blocks are chained together on some sort of linked list. Knuth[20], Section 2.5 describes algorithms for maintaining free space lists and allocation strategies for selecting which available block to allocate to a given request for memory. This class of dynamic memory management schemes has only negligible internal fragmentation, but it suffers from the more subtle problem of external fragmentation. External fragmentation describes a situation where the free space becomes choked with tiny available blocks, each too small to satisfy a typical request for memory. External fragmentation not only raises the cost of searching the free list of available blocks, but in the limit it can result in a system blockage where no pending request for space can be satisfied, even though sufficient free space exists to satisfy them, because the available memory is scattered in pieces too small to be of use.

The "buddy" methods represent a compromise between the above two approaches. In the binary buddy method all blocks -- allocated or available -- are of size 2^k for some integral value of k. If a request for a block of size x comes in, then the system would determine the smallest j such that $2^j \geq x$. If there is a block of that size available, then it would be allocated immediately. If not, then the system locates the smallest available block larger than x and splits it in half (repeatedly, if necessary) until a block of size 2^j results. Since each allocated block in the system must have been created by splitting a larger block in half, when a block is released the system checks to see whether the other member of the pair -- the buddy -- is also free. If so then the two are combined to reconstitute the original, larger, block and then the system looks for that block's buddy to rebuild even larger free blocks.

Intuitively, the buddy method approach should be worse than the other two, since it is susceptible to both internal and external fragmentation. Moreover, it is quite possible to have a large, contiguous block of unallocated memory which cannot be used in its entirety because it is composed of two smaller blocks which, though available, are not buddies and cannot be coalesced. Nonetheless, theoretical calculations and simulation studies[20,26] suggest that buddy methods do, in fact, outperform the variable-sized allocation method both from the point of view of total fragmentation and from the point of view of efficiency of the allocation and deallocation operations. A recent study by Nielsen[29] suggests that the fragmentation problems of the binary buddy method are too severe when used in simulation systems, but that study confirmed the execution efficiency of the buddy methods noted by Knuth[20].

B.1.2 The Fibonacci Buddy Method

A variation on the binary buddy method proposed by Hirschberg[18] uses the Fibonacci numbers instead of powers of two. Table B-1 shows the Fibonacci sequence, where each number in the sequence (after the second) is the sum of the previous two. The binary buddy method is based on the equality $2^k = 2^{k-1} + 2^{k-1}$ while the Fibonacci buddy method is based on the equality $F_k = F_{k-1} + F_{k-2}$, where F_k is the k^{th} Fibonacci number. Table B-2 shows the most important advantage of the Fibonacci approach over the binary approach, namely, that the Fibonacci method presents the user with a greater variety of block sizes for a given limit -- particularly at the low end of the spectrum.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Table B-1: The Fibonacci Numbers

Range	Fibonacci	Binary
1 - 100	10	7
100 - 1100	5	4
1100 - 12000	5	3

Table B-2: Count of Numbers in Various Ranges

For a while, the Fibonacci method was an academic curiosity. Hirschberg's calculations and his simulation study (using the data of Minker, et. al.[26]) demonstrated that the Fibonacci buddy method outperformed the binary buddy method, but he was not able to produce an efficient algorithm for implementing it. Then Thomas and Cranston[39] came up with an efficient method for implementing the Fibonacci scheme that they were able to prove was within one bit of the absolute minimum storage overhead needed to implement any buddy method. Each block -- freed or in use -- wastes one word (or one byte, for an eight-bit byte-oriented machine). This overhead word (byte) has four fields: a one-bit "A" ("allocated") field, a one-bit "B" field, a one-bit "M" field, and a "k-value" (indicating that the block is of size F_k, the k^{th} Fibonacci number). The B bit tells whether this is a left buddy or a right buddy, and the M bit saves half of the original block's B and M byte. If B = 0, then the M bit is the original block's B bit and if B = 1, the M bit is the original block's M bit.

The Fibonacci sequence is stored in an array (starting with 2 and 3 to generate the sequence) and associated with each number is a pointer to a linked list where free blocks of that size are chained together. If a block of size x is requested, then the array is searched to find the smallest block size (strictly) larger than x. If that list is empty, then the array is searched to find a larger block which can be split (and resplit, if necessary) to create a block of the appropriate size. When a block is freed, the B bit, its size, and its address are used to compute the address of the buddy. If the buddy's A bit shows that it, too, is free, then the two are combined to reconstitute the original block, and then the address of the original block's buddy is computed and the process repeats itself.

Not only does the Fibonacci method have the considerable advantage of permitting a variety of block sizes, but it lends itself to generalization and extension to permit a system designer to fine tune for a specific purpose. One approach to generalization is to define $F_k = F_{k-1} + F_{i-i}$* , where the first i numbers in the sequence are specified. Such a sequence is even denser than the usual Fibonacci sequence, as shown in Table B-3. Clearly, the more different block sizes that are available, the smaller the internal fragmentation will be. Another extension of this technique is to use different generating sequences. There is no particular reason why the sequence should begin 1, 2, 3, 5, ... and not 2, 4, 6, 10, ... or 3, 7, 10, 17, ... Therefore, if a system designer knows in advance that requests of certain particular sizes will be very frequent in the system he can try different values of i and different starting sequences to optimize for those sizes.

*Note that $i = 1$ and $F_1 = 1$ defines the binary buddy approach.

There is further evidence, beyond Hirschberg's own calculations and simulation to support the suggestion that a Fibonacci buddy method is likely to be the best choice for a dynamic memor allocation algorithm. This expectation is reinforced by Nielsen's study[29] although Nielsen did not test the Fibonacci buddy method directly, since the Fibonacci buddy method combines the best features of the top-rated "multiple free list" algorithm and the high-rated binary buddy method. Moreover, the efficacy of the Fibonacci buddy method in the face of an unknown distribution of request sizes and durations is supported by the conclusions of Peterson and Norma[30].

Range	i = 1	i = 2	i = 3	i = 4
2 - 100	7	10	12	14
100 - 1100	4	5	6	7
1100 - 12000	3	5	6	8

Table B-3: Count of Numbers in Various Ranges for $F_k = F_{k-1} + F_{k-i}$

B.2 Data Integrity, Consistency, and Quality

B.2.1 Sources of Erroneous Data

Sibley and Fry[13] have identified five sources of poor quality data in a data base. The data might be incorrect because it was:

(1) never any good (garbage in equals garbage out)
(2) altered by human error
(3) altered by a program bug

*Generated by first i numbers in the sequence 1, 2, 3, 4.

(4) altered by a machine error
(5) destroyed by a major system catastrophe.

In addition to these five problems, one can add problems relating to data consistency. Consider the effect of a system crash during an update. It is seldom possible to restart the update procedure at precisely the instant where the system malfunctioned, and simply restarting the update without some mechanism to recreate the data as it was before the crash will not often give the required results. For example, suppose the machine crashes while processing an update to give all systems analysts a 10% raise. If the update is simply restarted after the system comes back up without restoring the data base to its initial state, some systems analysts will get a 21% raise. A second source of consistency problems occurs when two users update the same table simultaneously. These problems will be addressed at greater length in another section dealing with user concurrency.

Ensuring data quality and consistency requires both the ability to detect erroneous data and the ability to restore affected portions of the data base to a previous state. To aid in the detection of erroneous data, there will be provision for user-input data validation rules through an integrity subsystem and procedures for automatic backout and recovery.

B.2.2 Backup and Restoration

B.2.2.1 Audit Trails

An audit trail (also called a "journal" or "log file") is a tape file which records:

(1) Beginning and end of all commands
(2) User-id for each command
(3) "Before" and "after" images of all changed records

(plus) images of inserted and deleted records

(4) Time and command identifier for each change.

The audit trail plays three roles. First, an audit trail is convenient for a quality audit of the data base, to detect data which is erroneous, but semantically plausible. Second, an audit trail helps detect the source of errors (whether discovered during a formal quality audit or detected informally by a human user). Finally, and probably most important, is the role of the audit trail in recovery from a system crash. The entries in the tape file can determine which commands have been initiated but not completed prior to the crash, and backout procedures can be initiated to recover the contents of those tables to their state prior to the initiation of the incomplete command. Moreover, if the data base has been checkpointed then the audit trail can help roll forward from that checkpoint to recover from major system malfunctions.

B.2.2.2 Internal Backout Provisions

It is very important that the system be able to undo any changes made to a table for two reasons: (1) the command may be blocked from completion by an I/O error or a semantic error, or (2) the system may crash during the course of executing a transaction. The latter can be handled by resorting to the audit trail, but there may be a more efficient method for restoring a table.

There are a variety of techniques in current use for providing backup and restore capabilities. The simplest but least efficient approach is to create and maintain a second copy of the original state of the file. This makes restoration an easy matter, but it is expensive and time-consuming. Differential files have been espoused as a means of getting around the dumping of entire files to provide backup copies. "After" images

of all changed records are kept in a separate file -- the differential file -- and record accesses begin by searching for the required record in the differential file, using the original file only if searching the differential file comes up empty. Then only the somewhat smaller differential file must be copied before a change transaction begins. Differential files have their drawbacks, most notably with respect to restoring deletions and due to the two-pass record access requirements. Severance and Lohman[33] describe a method for alleviating some of the latter difficulties, but deletions would still be a problem.

An approach with some similarities to differential files is taken in this system. This approach makes use of a linked list in the tabular data storage area of "before" images of changed physical pages and another linked list in main storage that is attached to a Command Control Block and which contains the page id of each page for which a "before" image has been recorded during processing of the associated command. While a physical page is being rolled in prior to being changed, the system scans the list of page ids to see whether this page is recorded on it. If not, then the first order of business once this page is in core is to select an empty page slot, create an entry in the list which records the physical id for the page to be changed and the page id for the empty slot, and then, before doing any further processing, copy the page in core into the slot. After the page is updated, it can go back into its proper location on the disk. Backing out the effects of a command and restoring a table to its state before initiation of the command is simple enough. The system merely scans down the list connected to the Command Control Block, entry by entry, and for each entry in the list it rolls in the before image of the physical page and writes it back to its proper location. Notice that this scheme preserves the pointer mechanism for the linked list of physical pages. If the command goes through to completion with no problems (hopefully the normal case!) then the entire list of "before" images

in the tabular data storage area can be freed up with no noticeable overhead providing they themselves are maintained in a linked list.

It will be necessary to record the new free page head pointer and backup page pointer in the audit trail when a free page is selected for copying.

B.2.3 The Integrity Subsystem

B.2.3.1 Integrity Assertions

When data base management systems designers and computer scientists refer to the "integrity" of data in a data base, they usually are referring to semantic correctness. Examples of semantically incorrect data would be a temperature less than -273.16° Celsius, a latitude greater than 90°, an employee record where the first name is "William" and the sex is "F", etc. In the days when data management systems consisted of a room full of filing cabinets and several clerks, the human beings responsible for data entry could catch most such errors with no perceptible overhead. In modern, computerized, data base management systems certain errors become very difficult to catch (e.g., "female" employees with obviously masculine names) and any data validation software included in the system must exact overhead penalties in the form of extra time to perform the tests and extra space to store the knowledge base and the code itself. Designing a space and time-efficient structure for the knowledge base, and resolution of the trade-offs between the expense of performing the tests and the utility of catching the errors, are major problems which confront the designer of a data base management system.

The normal approach to data validation in a relational data base management system is the "integrity assertion" (also

called an "integrity constraint"). An integrity assertion is a true/false predicate whose value will be "true" if and only if the records of the updated table are semantically correct. Suppose SPACECRAFT-DATA and EMPLOYEE are tables containing satellite data and personnel records, respectively. Then typical assertions might be defined as:

```
DEFINE ASSERTION A1 ON SPACECRAFT-DATA:  TEMP   > -273.16
DEFINE ASSERTION A2 ON SPACECRAFT-DATA:  LAT    < 1.5708.
DEFINE ASSERTION A3 ON EMPLOYEE:  AVG(SALARY)   < 30000
```

(where temperatures are stored in degrees Celsius and latitudes in radians north latitude). Eswaran and Chamberlin[12] identify five ways to classify integrity assertions:

(1) record vs. set
(2) state vs. transition
(3) immediate vs. delayed
(4) invoked on all changes vs. invoked only for specific types of changes (e.g., deletions only)
(5) hard vs. "soft"

Not all of the thirty-two ($= 2^5$) combinations of these labels are likely to be useful, and some of these are expensive to support. The Integrated Data Base Management System will support precisely the following combinations, with "hard" and "soft" variations on each:

(1) immediate-record-state-update only
(2) immediate-record-transition-update only
(3) immediate-record-state-insert only
(4) immediate-record-state-both insert and update

What is to be performed by the above assertion classes can be determined by the description below.

Immediate-type assertions are tested each time a data item is changed. Delayed assertions are tested at the end of a command or sequence of commands. The value of delayed assertions is that a proper and correct sequence of updates can cause a temporarily invalid state of the data base to occur. For example, consider a personnel data base with an assertion that the "number of employees" data field in a record of the "department" table must equal the sum of the number of records in the "employee" table with that department number. Adding or deleting a record in the "employee" table would cause this assertion to be violated until the appropriate data field in the "department" table was also updated. Delayed assertions are not necessarily expensive to implement, but analysis of projected user requirements suggests that their utility in a scientific environment, no matter how useful they are in a business environment, would not be worth the cost of implementing and supporting such a feature.

Set-type integrity assertions are distinguished from record assertions in having predicates which are functions of the entire table (e.g., assertion A3, above). These are expensive to perform since every record must be accessed when any tuple is changed. For that reason the Integrated Data Base Management System shall not support set assertions. It is not clear whether there would be any benefit to allowing the owner of a table to specify that certain assertions be tested only on insertion or only on update of a record, although this is simple enough to support and will be, in fact, supported by the system. One type of assertion which applies only to updates and which may be useful is the transition assertion, which relates allowable values of a data field in an updated record to the former value (e.g., DEFINE ASSERTION A4 ON EMPLOYEE: AGE > OLD AGE). Again this would be simple enough to implement and inexpensive to perform, and it will be implemented within the system.

One enhancement whose value is clear is the "soft" assertion. Unlike "hard" assertions, which abort commands when semantic errors are detected, a soft assertion would merely issue a warning.

One final consideration is the allowable complexity of predicates for immediate assertions. Clearly, the simpler the predicates are, the easier it will be to store, decode, and apply them. If we assume that predicates have the form:

<predicate> :: =<field name> <comp> <value>

where <field name> is the name of a data field in the table and <comp> is a comparison operator (i.e., =, <, >, LT, LE, EQ, GE, GT, or NE), then the allowable forms for <value> are

(1) a constant
(2) another data field
(3) a data field plus or minus a constant
(4) a data times or divided by a constant
(5) a data field times or divided by a constant plus or minus a constant

The "field name" will be either another data field of the same table or (if transition assertions are supported) the same field name as on the left side of the predicate, preceeded by the word "OLD". Some examples are:

```
DEFINE ASSERTION A5 ON EMPLOYEE:  AGE = OLD AGE + 1
DEFINE ASSERTION A6 ON SPACECRAFT-DATA:  START-DATE > END-DATE
```

By restricting the complexity of the predicates and the scope of the assertions it is possible to support immediate integrity assertions by a fairly simple table, attached to the Relation Control Block in main storage much as the Domain

Extension. The fields of this table would include:

(1) assertion name
(2) data field name (left side of comparison)
(3) applicability code (insertion only, update only, both)
(4) comparison operator
(5) constant
(6) data field name (right side of comparison)
(7) multiply/divide flag
(8) constant
(9) add/subtract flag

By a suitable choice of null entries, this set of fields is sufficient to describe all valid predicates.

B.3 A Locking Mechanism to Support Concurrency

B.3.1 Problems Introduced by Concurrent Updates

One major objective of a data base management system is to provide for the quality and integrity of the data. Therefore, it makes sense that the system should not itself introduce inconsistencies into the data. One source of system-induced inconsistencies are problems which can arise from permitting concurrent processing of the same files (tables) of data by two or more users.

There are two broad categories of problems which can arise due to concurrent processing. The first of these is called the "lost update", and it is a consequence of the fact that data in a data base management system is stored on external media (disk, drum) but must be copied into main memory before being read and/or edited. Suppose that user 1 initiates process 1 to update Record R, and suppose that user 2 simultaneously initiates process 2 to edit the same record. A possible sequence of events

is:

(1) Process 1 copies R into main memory and begins to update the record.
(2) Process 2 copies R into main memory and begins its own update.
(3) Process 1 finishes and copies R back to disk.
(4) Process 2 finishes and copies R back to disk.

Thus, the results of the first user's efforts are overwritten, hence the name "lost update". Fortunately, the design of the proposed Integrated Data Base Management System (which uses a common buffer pool and begins any data retrieval by searching for the required physical page in the buffer pool before initiating any disk I/O request) will alleviate the lost update problem to some extent. However, the case of a "pipelined CPU" or multi-CPU environment would still pose difficulties.

A more insidious problem is the so-called "phantom record". Suppose that user 1 is increasing the salary field of some set of records in a personnel file (e.g., giving all systems analysts a 10% raise) while user 2 is listing all records in the personnel file where the salary is above a certain threshold. If these two processes run concurrently, then depending upon the relative order in which the records were accessed some systems analysts whose salaries were increased above the threshold might be listed and some might not -- these are the phantom records. Note that this type of problem is not a loss of <u>data</u> integrity, but a loss of <u>process</u> integrity. One feels, intuitively, that the second process' results should list all the systems analysts where salary changed from below the threshold to above the threshold, or else none should be listed -- anything else is inconsistent.

The standard approach to retaining data consistency with

concurrent users is through the use of some sort of locking mechanism on the data. The basic rule is that data consistency can be maintained if and only if the results of two concurrent processes are indistinguishable from the results of the same two processes run sequentially in some order. Considering again the example used to describe the phantom record problem, it is acceptable to have all of the systems analysts whose salaries were lifted over the threshold included in the listing, since this is equivalent to executing process 1, followed by process 2. It is equally acceptable to have none of the systems analysts included, as that is equivalent to executing process 2, followed by process 1. As that exhausts the possibilities, nothing else is acceptable. Notice that the rule is always satisfied if the two processes in question only read the data and do not change it. However, if a process wishes to change some piece of data, it will be necessary to wait until all other processes operating on that piece of data finish, then that process must have sole access to the data. Normally, this is accomplished by means of two different types of locks -- a "share" lock for processes which do not intend to change the data, and an "exclusive" lock for processes which do. Processes which request a piece of data are allowed to proceed only if (a) no other process has a lock on that data, or (b) this process wishes to lock the data in shared mode and the data has already been locked in a shared mode (depending upon system strategy, the request may have to check for pending exclusive use requests). If neither criterion is satisfied then the request must be placed in some sort of "pending" queue.

B.3.2 High Level vs. Low Level Locking

There are two issues -- not entirely separable -- which must be clarified in the description of any locking mechanism. The first of these is whether the locks shall be set by some sort of high-level logical mechanism (e.g., predicate locks) or

whether the locks should be physically attached to single, indivisible, units of data. The high-level approach to locking in a relational system is seductive. In the command language of most relational systems, the set of records to be accessed is implicitly defined by a logical predicate (the "WHERE clause" of SEQUEL and QUEL). The predicate of the incoming request and the predicates for active requests are combined into a Boolean expression (usually in disjunctive normal form) and tested for satisfiability. An expression is satisfiable if there exists some consistent assignment of "true" or "false" to each term in the expression which makes the whole expression true. Consider the following pair of requests against a personnel file:

(1) NAME = SMITH AND SALARY > 20000
(2) TITLE = MGR OR SALARY < 19000

The resultant expression is:

(NAME = SMITH AND SALARY > 20000 AND TITLE = MGR)
OR (NAME = SMITH AND SALARY > 20000 AND SALARY < 19000)

The second parenthesized subexpression is always false, since a salary cannot simultaneously be greater than $20,000 and less than $19,000. However, the entire expression is satisfiable since the first subexpression is satisfiable if there is a manager named Smith making more than twenty thousand dollars. Therefore, the two requests conflict and one must be blocked. Note that this says nothing about whether there <u>is</u> such a record in the personnel file -- satisfiability does not guarantee that a conflict exists, only that there is a potential for problems. Rather, it is the unsatisfiability which is desirable, since it guarantees that conflict will not exist.

Certainly the high level logical locking approach is mathematically elegant. However, at a higher mathematical level it

is known that satisfiability is an NP-complete problem. Barring a major mathematical break-through of unprecedented proportions, there exists no efficient means for testing for satisfiability. Testing for satisfiability with r transactions and an average of n terms per predicate must take time proportional to $r \cdot 2^n$. (In fact, $r \cdot 2^n$ is a lower bound.) Stonebraker, one of the designers of the INGRES system, has proposed a similar approach[36]. It is easy enough to show that if the incoming predicate passes Stonebraker's test, than the resultant Boolean expression must be unsatisfiable, and in fact the test may be more restrictive than is necessary to detect potentially non-conflicting requests. Moreover, it is not clear whether there exist efficient procedures to perform the tests required for Stonebraker's algorithm.

B.3.3 Granularity

Granularity refers to the size of a lockable unit, and once the decision is made to use physical locks, the size of the data "granules" to be locked must be specified. Should the locks be set at the record level? Lower still, at the data field level? Or higher -- at the physical page or even data base level?

Gray, et al.,[16] have proposed a scheme which allows the user to place locks at a variety of different granule sizes, depending upon the needs of the particular transaction. This approach views the structure of a data base, conceptually, as a directed acyclic graph structure (see Figure B-1). The system may place explicit locks at any vertex in the graph, and an explicit lock at a vertex implicitly locks all descendants in the graph. These implicit locks do not have to be formally specified anywhere -- to prevent possible lock conflicts between an explicitly-set lock at a lower vertex in the graph from conflicting with an implicitly-set lock created by an explicit lock on

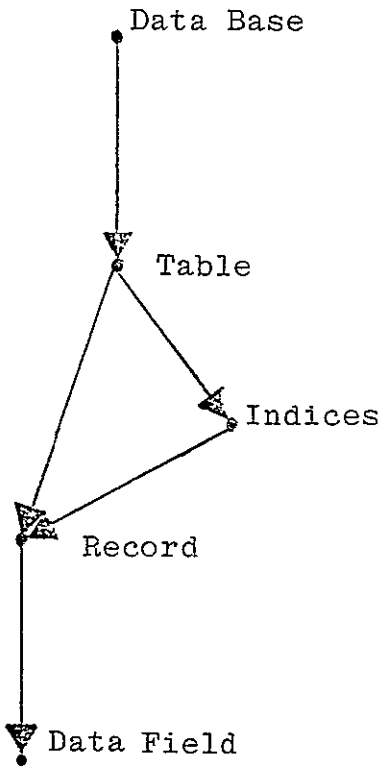


Figure B-1: Hierarchy of Lockable Units in a Data Base

an ancestor vertex, all lock requests are required to begin by walking down the graph from the top vertex placing "intention" locks at every ancestor of the vertex to be locked. This permits conflicts to be detected at the very highest level.

The advantage of being able to select a fine granularity is that it permits the maximum possible concurrency. If the system places locks on units of data which are larger than necessary, then there is the danger of blocking a second request which needs the unused portion of the locked data, but which otherwise does not conflict with the original request. The disadvantage of using a fine granularity is clear, however -- the more granules which can be locked, the more overhead to test, set, and maintain these locks will be required. In an effort to examine the tradeoffs, Stonebraker and Ries performed a simulation study to explore the desirable size of a "granule".[31] Their study can be critiqued on the grounds that it assumes that the transactions are uncorrelated, and thus it ignores the "80-20" law.* The study demonstrated that splitting the data base in ten equal-sized granules performed surprisingly well, particularly when transactions requested large portions of the data or when the number of I/O channels was restricted. Using fifty granules appeared to do best for multiple I/O paths (and it performed as well as ten granules for a single I/O path) and for requests for small portions of the data base.

B.3.4 A Physical Locking Mechanism

The results of Ries and Stonebraker's study suggest that the table should be the basic lock granule. Several factors have influenced this choice, most notably the fact that it is easy to determine which tables are needed for a command in advance of executing that command -- something not known for

*The 80-20 law for commercial data processing applications states that 80% of the transactions against a file deal with at most 20% of the records in the file, and the same applies to this 20%. Therefore, when the 80-20 law holds, a miniscule 4% of the records account for approximately 64% of the transactions.

records or physical pages -- and the Relation Control Block is always in core, making it easy to keep the list of pending requests queued at the RCB (again, not possible for records or physical pages).

Figure B-2 depicts the basic system elements included in the locking scheme. The monitor selects Command Control Blocks from the Command Queue and passes them to the Logical Interface for execution. The Monitor is an asynchronous process which is awakened by one of three sources, the Application Program Interface, the Interactive Command Processor, and the Logical Interface, depending upon circumstances. Although the Relation Control Blocks (RCB) are the "property" of the Logical Interface, they are also used by both the Monitor and Physical Interface.

Both the Application Program Interface and the Interactive Command Processor can activate the Monitor by passing it a Command Control Block (CCB). The Monitor examines the CCB to determine what action to take. The Monitor determines which tables the command will access and the type of lock needed to support that access (shared for reads, exclusive for insertions, updated, and deletes) (note that any CCB will need at most one exclusive lock). For each table needed by the CCB, the Monitor will locate the RCB and save a pointer to it. The Monitor can pass this CCB to the Logical Interface only if:

(1) Each RCB needed has an empty "pending" queue, and
(2) Either the RCB is unlocked or else it is locked in shared mode and the requested lock is also shared mode.

If any table fails either test then the CCB will not be placed on the Initiator Queue for execution and each lock request will also be enqueued at the RCB. The CCB will be given a copy of the list of pointers to RCB's.

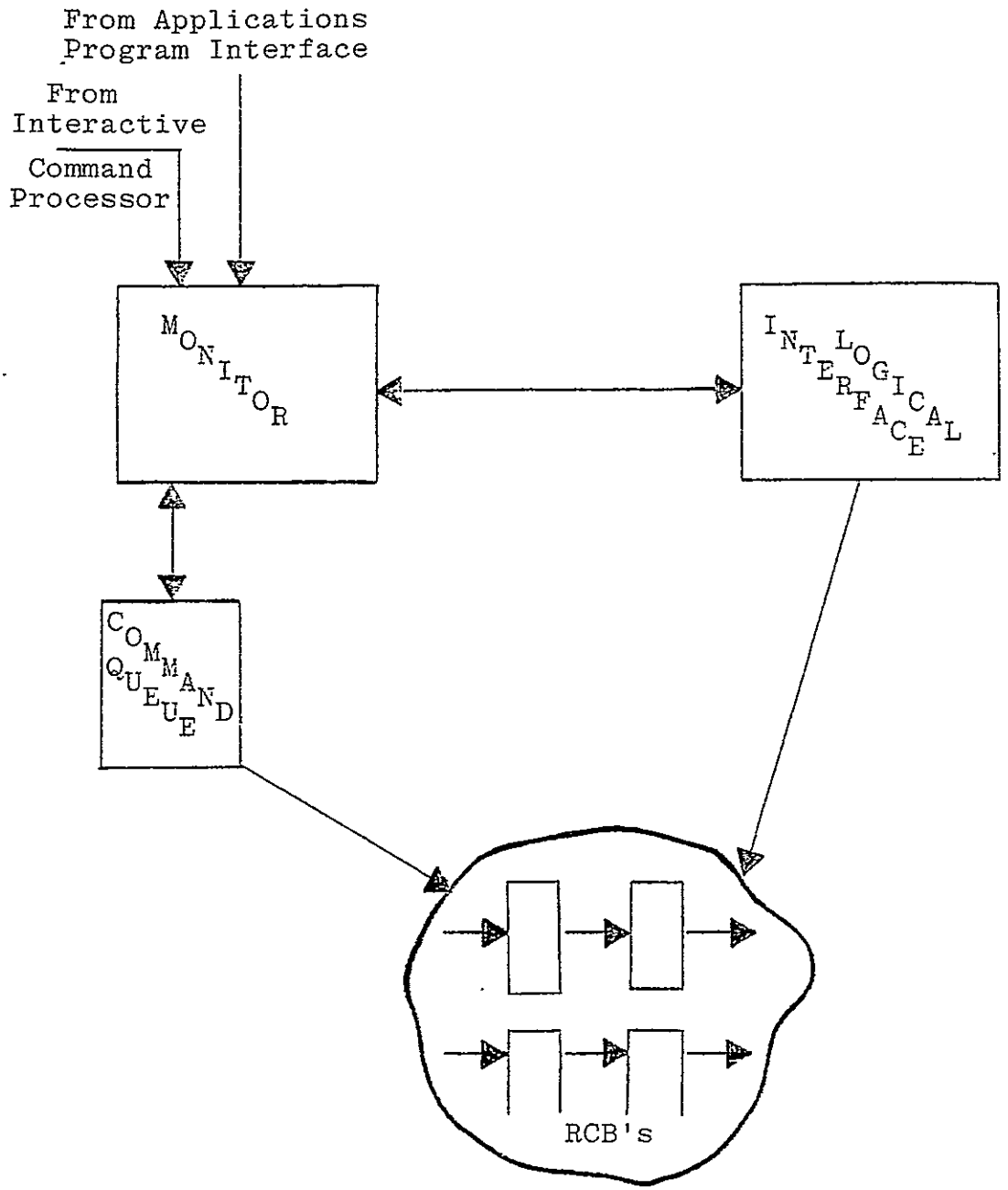


Figure B-2: Concurrency Support Substructure

To support this locking mechanism the RCB must have three fields: a bit indicator for lock mode (0 = shared, 1 = exclusive), a counter for the number of CCB's using this table (equals zero when the table is unlocked), and a pointer to a queue of pending requests. The queue will be a circularly-linked list, so that front and rear are equally accessible (since FRONT = LINK (REAR)). Entries in the queue will consist of the mode of the pending lock request plus a pointer to the CCB which is waiting for that table.
Accepted lock requests are handled by setting the lock mode bit to the appropriate value and incrementing the counter. Rejected requests cause the system to create a queue entry and to add it to the queue.

When the Logical Interface completes the execution of a command, it decrements the count fields of each RCB referenced by the command (this may happen incrementally, while the command is executing, as the system finishes using each RCB). If the count field on any RCB goes to zero during this step and the queue of pending lock requests is not empty then the Logical Interface will wake the Monitor and pass it the CCB address for the first entry in the queue.

When the Monitor is awakened by the Logical Interface, it will begin by examining the RCB's needed by that CCB. The requirements for activating the CCB are slightly different. First, each RCB needed must be unlocked or else locked in shared mode and the pending CCB's lock request must also be shared mode. Second, the CCB's queue entry must be at the head of the queue, or else the lock request must be for a shared lock and all lock requests ahead of it in the queue must also be for shared locks. If the above requirements are met then the queue entry is deleted for each RCB, following which the CCB is deleted from the Command Queue and placed on the Initiator Queue for processing by the Logical Interface.

Whether or not the CCB is activated at this point, all remaining CCB's which follow it in the Command Queue are tested to see whether they, too, can now be activated. Note that nothing ahead of the CCB whose address was passed back to the Monitor to initiate this procedure need be considered.

It is important to note that the proposed approach requires the Monitor to do no more work than the minimal amount necessary since the Monitor is only activated at points when there is a possibility of changing the contents of the Command Queue.

B.3.5 Scheduling Strategies

There are two approaches to scheduling which can be used. One approach is to initiate any new command which can be initiated, even if this siezes a table needed by a pending command in the Command Queue. This increases total system throughput, but it has the potential to leave one transaction stranded in the Command Queue while commands entered later get processed sooner. The second strategy, which is the strategy embedded in the previous section's procedure, maintains a strict first in, first out discipline. As a result, no single command will spend an inordinate amount of time in the Command Queue, but total system throughput may suffer. If the test for an empty "request pending" queue when the Monitor is activated by the Interactive Command Processor or the Applications Program Interface and the similar test for queue entry at the head of the queue when the Monitor is activated by the Logical Interface are deleted this will result in the initial strategy described in this section being employed, rather than the second strategy. As these are easy changes to make, it is possible that the Data Base Administrator could experiment with both strategies over a period of time and use whichever seems best. The DBA could also adapt the strategy to changing system requirements.

It should be pointed out that some sort of hybrid approach, following the first strategy of letting commands begin when relations were available regardless of pending requests, but preempting an active command if any CCB waits too long in the queue, would be expensive to implement. The preempted command would have to be backed out (cheap enough for a read, but otherwise expensive) and the work done to that point would be nullified. Moreover, the Monitor would have to start keeping track of the length of time each CCB spent in the Command Queue.

B.3.6 Deadlock

Deadlock is a condition where two or more processes permanently block each other. The simplest example is when command C_1 has an exclusive lock on table R_1 but cannot do any further processing without accessing table R_2. Meanwhile, command C_2 has an exclusive lock on R_2 but cannot go on without accessing R_1. Obviously, both are blocked and neither command can proceed unless one or the other is preempted.

It can be demonstrated that deadlock can occur only if all of the following conditions are met:

(1) Concurrency - two or more processes can run at the same time.
(2) Locking - a process can have exclusive access to some data.
(3) No Preemption - no data can be taken from a process which has locked that data.
(4) Expansion - a process may request additional locks without relinquishing locks already held.

If any of the above conditions are disallowed then deadlock

can occur, except that allowing preemption is insufficient by itself and must be used in conjunction with some algorithm for detecting a circular chain of blocked processes and a strategy for choosing the process to be preempted. The procedure outlined in the previous section prevents deadlock by disallowing expansion. No command is permitted to begin until it has available all tables which will be needed by that command.

B.4 Data Compatibility

B.4.1 The Scope of the Problem

One problem which is perhaps unique to scientifically-oriented data bases is the question of data units. If a data item representing a distance is stored in some table, that distance may be expressed in angstroms, microns, millimeters, centimeters, inches, feet, yards, meters, rods, kilometers, miles, earth radii, astronomical units, or light years. The problems with weight are even worse, as there are two different kinds of ounces and three different tons -- not to mention the difference between pounds as weight, pounds as mass and pounds as force. If two items in different tables, both representing the same measured quantity (e.g., distance, time, mass, area, volume) are to be compared or mathematically combined, it is imperative that they have the same units attached or be converted to equivalent units. This should be handled automatically by the data base management system.

A related problem occurs when data items representing different measured quantities are to be combined to produce a third quantity, as for example, if a mass is to be divided by a volume to produce a density. In such cases, it is important that the units all be part of the same system of measurements (cgs, kms, English), and the data base management system should see to it that they are.

B.4.2 An Approach to Data Compatibility

The problems outlined in the preceding section can be handled within the system with the aid of the following pair of tables:

(1) a system table, indicating the measurement system (cgs, kms, English) to which the units belong, and

(2) a conversion table, listing pairs of commensurate units and the conversion factor.

These two tables might be laid out as depicted in Figure B-3. If the Integrated Data Base Management System is called upon to compare the values of two data items or to add or subtract them, it will begin by examining the definition of these items in the appropriate Data Dictionary. If the data items are alphanumeric, then comparison will be allowed, but not addition or subtraction. If the data items are numeric, then comparison operations, additions, and subtractions will be allowed if and only if the data items have the same units or can be converted to the same units. If the units do not agree, then the system will try to retrieve a conversion factor from the conversion table and the operation will be aborted if no conversion factor can be retrieved. It should be noted that an internal data type conversion may also be necessary (integer to real, real to double precision), as well as a numeric conversion with the conversion factor. It should also be noted that retrieving the conversion factor would use standard search and access software. It is equivalent to the following retrieval command:

```
USE C FOR CONVERSION
SELECT (C. FACTOR)
WHERE C.GIVEN-UNITS = units1
  AND C.TARGET-UNITS = units2
```

CONVERSION

GIVEN-UNITS	TARGET-UNITS	FACTOR
INCHES	CM	2.54
CM	INCHES	0.3937
YARDS	INCHES	36.0
MILES	FEET	5280.0
SEC	DAYS	.0000198

UNITS-SYSTEM

UNITS	SYSTEM	QTY
FEET	ENG	DISTANCE
FPS	ENG	SPEED
KG	KMS	MASS

Figure B-3: Tables To Support Data Compatibility

The system table would be used to support multiplication and division. Multiplication and division would be permitted only if the two data items agreed as to measurement system. If the system does not agree (e.g., mass measured in grams but volume in cubic meters when computing a density) then the system will have to convert one or both data items until they agree as to measurement system. Again, standard system software could be used to handle the search and access, which would be equivalent to the following query:

```
USE C FOR CONVERSION
USE S, X, Y FOR UNITS-SYSTEM
SELECT (X.UNITS)
WHERE S.UNITS = units1
AND X.QTY = S.QTY
AND Y.UNITS = units2
AND X.SYSTEM = Y.SYSTEM
SELECT (C.FACTOR)
WHERE C.GIVEN-UNITS = units1
AND C.TARGET-UNITS = W.UNITS
```

Again, care must be taken to make certain that the data types agree, as well as data units.

The above approach will not handle all possible data conversions -- one type of conversion which cannot be handled by a multiplicative scale factor is temperature. It is possible to convert Centigrade to Kelvin (or back), Centigrade to Farenheit, or Farenheit to degrees Rankine. If desired, temperatures could be handled as a special case.

A third table -- relating abbreviations to unit names -- might also be useful for purposes of parsing queries (including DEFINE commands).

B.5 System Security

One goal of a data base management system is prevention of the dissemination of data to unauthorized recipients. Within a data base management system this requires three steps: identification of the user accessing the data base, authentication of that user, and validation of each operation requested by the user subsequent to logging on to the system. In the Integrated Data Base Management System, the identification and authentication steps will be handled by the ENTER command and the validation step will be embedded within the affected commands (e.g., ATTACH, DEFINE, SELECT, INSERT, etc.). Provided the user remains within the system, the weakest link in the system's security is the identification/authentication step since the system will only be capable of determining whether the password input as part of the ENTER command agrees with the specified user-id, and not whether the user logging on with the ENTER is, in fact, the user identified by the user-id. The onus will be on the user community to protect their passwords from becoming known by other users and to change them with some frequency.

If an unauthorized user bypasses the system, then the situation will be much more difficult. Since no known operating system can be guaranteed to prevent a knowledgeable and determined user from reading files which he or she is not authorized to access, it behooves the system to provide protection against this possibility. One important piece of information which must be protected is the list of user-ids and passwords in the SYSUSER table. The key to providing such protection is the one-way "trapdoor" encoding functions of the public key encryption systems discovered by Hellman and Diffie[11]. Public key encryption systems are such a fundamental and important advance that the topic has begun to receive attention outside scholarly circles in popular scientific journals such as Scientific American[14] and from there into

news media such as TIME[31] and The Washington Post[34]. Such systems rely on "one-way", or "trapdoor", encoding functions, where the mathematical manipulations which encipher the data are so very different from the mathematical manipulations that decipher the data that knowledge of the enciphered data and the key used to encipher it is insufficient to decode the data. Thus, the system could store the user-id for each user (unencoded), the user's password (encoded), and the encoding key (assigned by the system) in SYSUSER, and yet if an unauthorized user should break the operating system's security and read the SYSUSER table he would not be able to determine the passwords of any users on the system. Since all the system has to do to authenticate the user is to encode the input password using the stored key and match the result against the encoded password stored in the SYSUSER table, the scheme is not only fail-safe but efficient.

The problem of encrypting the data stored in other tables, including user-created tables as well as other system tables, poses more difficulties since the system will have to decode the data before using it (or presenting it to the user). Thus, the decoding keys will have to be stored somewhere and they, themselves, must be secured. One public key encryption scheme which might be used is the one proposed by Rivest[32], where the enciphering and deciphering keys are based on two secretly chosen prime numbers p and q. The decoding key is a pair of integers (d,n), where $n = p \cdot q$ and d satisfies certain conditions based on p and q. Finally, the encoding key is a pair of integers (e,n), where $e \cdot d$ satisfies certain criteria involving p and q. Since e and n are always known, this encryption method can only be compromised if d is discovered or, with some extra effort, if p and q can be deduced. The system could avoid the latter by selecting m triples (e,d,n) in advance and saving the (e,n) pairs in one table T_1 and the d's in another table T_2, where the order of the d's in T_2 is different from the order of their corresponding

encoding keys in T_1 and where the size of T_2 is much greater than the size of T_1 (i.e., many "red herrings" are scattered through T_2). The system would employ a hash function h, such that h(e,n) would be the address in T_2 where the decoding d for that particular encoding key can be found. Since p and q are never stored in the system, they are safe unless an intruder, upon discovering n, can find the prime factors of n -- a process which can always be done but which is computationally infeasible for large values of n. The alternative is to discover d, which is stored in the system. If the encoding key for a secure table is stored with that table's entry in SYSREL, then an unauthorized intruder would have to copy not only the data in the table, but also SYSREL and T_2 to compromise the table's security. Even that would not be enough without either knowing the hashing function or else trying all entries in T_2 until one is found which provides an inverse function for the encoding key. For that matter, SYSREL and T_2 could themselves be encrypted using a secret key embedded in the software of the system.

Unless the physical security of the tape files could be guaranteed, there would be no point in encrypting the on-line data files also maintained by the system.

Encryption of tabular data is not a necessity for an initial implementation of the Integrated Data Base Management System, given the proposed uses to which it is expected to be put. But it is not improbable that some time in the future there will be a need to protect the security of certain data maintained by the system, and this subsection provides an indication of how that might be done. One issue which must be resolved before this scheme could be implemented is the status of Rivest's patent application on his method.

B.6 The Macro Command Facility

The Macro Command Facility would permit an interactive user to enter a sequence of commands, specify a name for the sequence and, subsequently, execute the sequence by simply specifying its name. While this feature is not a necessity, it would provide users with a method of executing often used command sequences with a minimum of effort. To implement this feature, two additional commands, END and EXECUTE, would have to be added to the Interactive Command Language and the DEFINE and REMOVE commands would have to be extended with a SEQUENCE option. Each of these commands is described briefly below.

The DEFINE command with the SEQUENCE option would introduce the command sequence and place the Integrated Data Base Management System into macro mode. The name to be assigned to the command sequence would be included in the DEFINE SEQUENCE command. The syntax for the DEFINE SEQUENCE command would be as follows.

DEFINE SEQUENCE <sequence name>

The sequence name specified in the CREATE command must be unique among command sequence names already known to the system. If it duplicates an existing command sequence name, the command will be rejected. The DEFINE SEQUENCE command could be issued by an interactive user at any time. Additionally, it could be used in the input stream for the Batch Command Reader, thus permitting command sequences to be created via the Batch Command Reader facility.

The END command would terminate the sequence of commands initiated by the last DEFINE SEQUENCE command issued by the user. If no previous DEFINE SEQUENCE command issued by the user is active, the END command would be rejected. The syntax for

the END command is as follows.

END

All commands issued following a DEFINE SEQUENCE command and preceeding its matching END command would be included in the command sequence named in the DEFINE SEQUENCE command. If no interactive commands were issued between the DEFINE SEQUENCE and END commands, no command sequence would be created and an error message would be displayed.

As stated previously, the DEFINE SEQUENCE command would place the system in macro mode. While in this mode, all commands received from the user issuing the DEFINE SEQUENCE command would be parsed and syntax checked but would not be executed. Each error free command entered while in the macro mode would be stored in the Macro Library. Any comman containing an error would be rejected, but the user could immediately reenter the command. The Macro Library might be implemented as a new system table with an inverted index created on the sequence name field in each record of the tabl

The REMOVE command with the SEQUENCE option would remove an existing command sequence from the system. The command sequence name would have to be included in the REMOVE SEQUENCE command. The syntax for the REMOVE SEQUENCE command would be as follows.

REMOVE SEQUENCE <sequence name>

This command would remove all records associated with the command sequence named in it from the Marco Library. If no such sequence exists, the command would be rejected.

The EXECUTE command would cause an existing command sequence to be retrieved from the Marco Library and to be

executed as if entered interactively by the user. The sequence name of the command sequence must be included in the EXECUTE command. If no such command sequence exists in the Macro Library, the EXECUTE command will be rejected. The syntax for the EXECUTE command is as follows.

EXECUTE <sequence name>

The EXECUTE command would cause each record containing the sequence name specified in the command to be retrieved from the Command Library. Each record might contain a command or partial command in the command sequence as it was originally entered when the sequence was created. It would be displayed on the remote terminal of the interactive user and executed by the system. An alternative approach might be to save the Command Control Block for each command in the sequence in the Macro Library instead of an image of each command as it was originally entered. This would be possible since a Command Control Block is created when a command is parsed. This approach would alleviate the need for parsing the command sequence each time that it was executed. In either case, execution of the command would be carried out in exactly the same manner as if the command had been entered interactively from the remote terminal. The EXECUTE command could also be included in the input stream processed by the Batch Command Reader. Thus, command sequences could be initiated via the Batch Command Reader facility.

It might also be possible to permit EXECUTE commands to be included in newly defined command sequences. Thus, existing command sequences could be easily incorporated into other command sequences.

BIBLIOGRAPHY

(1) Bayer, R., and McCreight, E., "Organization and Maintenance of Large Ordered Indices," Proceedings of the 1970 ACM-SIGFIDET Workshop on Data Description and Access, Houston, Texas, pg. 107-141 (November 1970).

(2) Bracken, P.A., Dalton, J.T., Billingsley, J.B., and Quann, J.J., Atmospheric and Oceanographic Information Processing System (AOIPS) System Description, Document X-933-77-148, Goddard Space Flight Center, Greenbelt, Maryland (March 1977).

(3) Chamberlin, D.D., "Relational Data-Base Management Systems," ACM Computing Surveys, vol. 8, no. 1, pg. 43-66 (March 1976).

(4) Chamberlin, D.D., et. al., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control," IBM Journal of Research and Development, vol. 20, no. 6, pg. 560-575 (November 1976).

(5) CODASYL Data Base Task Group, April 1971 Report. (available from Association for Computing Machinery, Inc., 1133 Avenue of the Americas, New York, New York 10036).

(6) Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM, vol. 13, no. 6, pg. 377-387 (June 1970).

(7) Codd, E.F., "Further Normalization of the Data Base Relational Model" in Data Base Systems, Courant Computer Science Symposia Series, vol. 6, pg. 33-64, Prentice-Hall, Inc., Englewood Cliffs, Jew Jersey (1972)

(8) Codd, E.F., "Relational Completeness of Data Base Sublanguages," in Data Base System, Courant Computer Science Symposia Series, vol. 6, pg. 65-98, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1972).

(9) Codd, E.F., "Seven Steps to RENDEZVOUS with the Casual User," Proceedings of the IFIP TC-2 Working Conference on Data Base Management Systems, North-Holland Publishing Company, Amsterdam, the Netherlands (April 1974).

(10) Date, C.J., An Introduction to Database Systems (2nd ed.), Addison-Wesley Publishing Company, Reading, Massachusetts (1977).

(11) Diffie, W., and Hellman, M.E., "New Directions in Cryptography," IEEE Transactions on Information Theory, vol. IT-22, no. 6, pg. 633-654 (November 1976).

(12) Eswaran, K.P., and Chamberlin, D.D., "Functional Specifications of a Subsystem for Data Base Integrity," Proceedings of the 1975 Conference on Very Large Data Bases, Framingham, Massachusetts, pg. 38-68 (September 1975).

(13) Fry, J.P., and Sibley, E.H., "Evolution of Data-Base Management Systems, ACM Computing Surveys, vol. 8, no. 1, pg. 7-42 (March 1976).

(14) Gardner, M., "A New Kind of Cipher that would take Millions of Years to Break," Scientific American, vol. 237, no. 2, pg. 120-124 (August 1977).

(15) Gary, J.P., AOIPS Data Base Management System Support for GARP Data Sets, NASA Technical Memorandum 78042, Goddard Space Flight Center, Greenbelt, Maryland (October 1977).

(16) Gray, J.N., Lorie, R.A., and Putzolu, G.R., "Granularity of Locks in a Shared Data Base," Proceedings of the 1975 Conference on Very Large Data Bases, Framingham, Massachusetts, pg. 428-451 (September 1975).

(17) Held, G., and Stonebracker, M., "B-Trees Re-examined," Communications of the ACM, vol. 21, no. 2, pg. 139-143 (February 1978).

(18) Hirschberg, D.S., "A Class of Dynamic Memory Allocation Algorithms," Communications of the ACM, vol. 16, no. 10, pg. 615-618 (October 1973).

(19) Keehn, D.G., and Lacy, J.O., "VSAM Design Set Parameters," IBM Systems Journal, vol. 13, no. 3, pg. 186-212 (March 1974).

(20) Knuth, D.E., The Art of Computer Programming, vol. 1, "Fundamental Algorithms" (2nd ed.), Addison-Wesley Publishing Company, Reading, Massachusetts (1973).

(21) Knuth, D.E., The Art of Computer Programming, vol. 3, "Sorting and Searching," Addison-Wesley Publishing Compnay, Reading, Massachusetts (1972).

(22) Lucas, H.C. Jr., Why Information Systems Fail, Columbia University Press, New York (1975).

(23) Martin, J., Principles of Data-Base Management, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1976).

(24) Maruyama, K., and Smith, S.E., "Analysis of Design Alternatives for Virtual Memory Indices," Communications of the ACM, vol. 20, no. 4, pg. 245-254 (April 1977).

(25) McCreight, E.M., "Pagination of B*-Trees with Variable-Length Records," Communications of the ACM, vol. 20, no. 9, pg. 670-674 (September 1977).

(26) Minker, J., Crooke, S., and Yeh, J., "Analysis of Data Processing Systems," Computer Science Center Technical Report TR 69-99, University of Maryland, College Park, Maryland (December 1969).

(27) Moik, J.G., Users Guide for Batch Operation of the SMIPS/VICAR Image Processing System, Document X-933-76-114, Goddard Space Flight Center, Greenbelt, Maryland (May 1976)

(28) Nauer, P., et. al., "Revised Report on the Algorithmic Language Algol 60," Communications of the ACM, vol. 6, no. 1, pg. 1-17 (January 1963).

(29) Nielsen, N.R., "Dynamic Memory Allocation in Computer Simulation," Communications of the ACM, vol. 20. no. 11, pg. 864-873 (November 1977).

(30) Peterson, J.L., and Norma, T.A., "Buddy Systems," Communications of the ACM, vol. 20, no. 6, pg. 421-433 (June 1977).

(31) Ries, D.R., and Stonebraker, M., "A Study of the Effects of Locking Granularity in a Data Base Management System," Proceedings of the 1977 ACM-SIGMOD Conference on Management of Data, Toronto, Canada, pg. 10-25 (August 1977).

(32) Rivset, R.L., Shamir, A., and Adleman, L., "A Method for Obtaining Digital Signatures and Public Key Cryptosystems," Communications of the ACM, vol. 21, no. 2, pg. 120-126 (February 1978).

(33) Severance, D.G., and Lohman, G.M., "Differential Files: Their Application to the Maintenance of Large Data Bases," ACM Transactions on Database Systems, vol. 1, no. 3; pg. 256-267 (September 1976).

(34) Shapley, D., "The New Unbreakable Codes: Will They Put NSA Out of Business?", The Washington Post, pg. B-1, July 9, 1978.

(35) Shneiderman, B., Department of Information Systems Management, University of Maryland, College Park, Maryland (personal communication).

(36) Stonebraker, M., "High Level Integrity Assurance in Relational Data Base Management Systems," Electronics Research Laboratory Memorandum ERL-M473, College of Engineering, University of California, Berkeley, California (August 1974).

(37) Stonebraker, M., Wong, E., Kreps, P., and Held, G., "The Design and Implementation of INGRES," ACM Transactions on Database Systems, vol. 1, no. 3, pg. 189-222 (September 1976).

(38) Thomas, J.C., and Gould, J.D., "A Psychological Study of Query by Example," Proceedings of the National Computer Conference, vol. 44, pg. 439-445 (May 1975).

(39) Thomas, R., and Cranston, B., "A Simplified Recombination Scheme for the Fibonacci Buddy System," Communications of the ACM, vol. 18, no. 6, pg. 331-332 (June 1975).

(40) Zloff, M.M., "Query by Example," IBM Research Report RC 4917, IBM Thomas J. Watson Research Center, Yorktown Heights, New York (July 1974).

(41) "An Unbreakable Code?", TIME, vol. 112, no. 1, pg. 55-56 (July 3, 1978).

BUSINESS AND TECHNOLOGICAL SYSTEMS, INC.
10210 GREENBELT ROAD • SEABROOK • MARYLAND 20801
301/794-8800