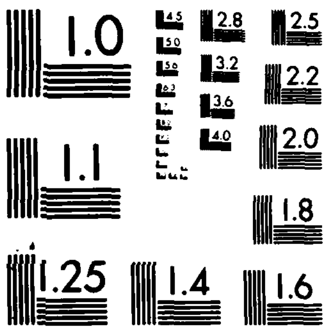
AD-A140 628 DESIGN OF AN INTEGRATED SOFTWARE SYSTEM BASED ON THE RELATIONAL DATA BASE MODEL(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P J HARRISON ET AL. DEC 83 1/2

UNCLASSIFIED F/G 5/1 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(2)

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

DESIGN OF AN INTEGRATED SOFTWARE SYSTEM BASED ON THE RELATIONAL DATA BASE MODEL

by

Patrick John Harrison
and
Gracie Lee Thompson

December 1983

Thesis Advisor: Dushan Badal

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A140 638	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Design of an Integrated Software System Based on the Relational Data Base Model		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December, 1983
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Patrick John Harrison and Gracie Lee Thompson		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93943		12. REPORT DATE December, 1983
		13. NUMBER OF PAGES 175
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18. SUPPLEMENTARY NOTES

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

ISS, Integrated Software System, Table, Relational Data Base Model, Data Base, Operators, Applications Database

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

Integration of application programs into a single system has become increasingly important as the workstation environment moves toward uniformity for easier learning and use. This thesis proposes an Integrated Software System (ISS) based on the Relational Database model as a suitable basis for integrating five common applications found in a business office. Relations, or tables, are defined as the common data objects (Continued)

ABSTRACT (Continued)

and it is shown how they are used to support each logical application. Operations based on relational algebra are defined which extend the functions of ISS beyond the aggregate of the five chosen applications. A simple graphical user interface is designed for the kernel of the system and a design for a kernel prototype using a Unix environment is presented. The results of this thesis are intended to lay the foundation for development of an ISS using the relational database model.

Design of an Integrated Software System Based on the Relational Data Base Model

by

Patrick John Harrison
Lieutenant, Royal Australian Navy
B.Sc., University of New South Wales, 1974

Gracie Lee Thompson
Lieutenant, United States Navy
B.A., University of South Florida, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1983

Authors: [illegible]

Gracie L. Thompson

Approved by: [illegible]
Thesis Advisor

[illegible]
Second Reader

David K. Hsiao
Chairman, Department of Computer Science

K.T. Marshall
Dean of Information and Policy Sciences

ABSTRACT

Integration of application programs into a single system has become increasingly important as the workstation environment moves toward uniformity for easier learning and use. This thesis proposes an Integrated Software System (ISS) based on the Relational Database model as a suitable basis for integrating five common applications found in a business office. Relations, or tables, are defined as the common data objects and it is shown how they are used to support each logical application. Operations based on relational algebra are defined which extend the functions of ISS beyond the aggregate of the five chosen applications. A simple graphical user interface is designed for the kernel of the system and a design for a kernel prototype using a Unix environment is presented.

TABLE OF CONTENTS

I. INTRODUCTION

The introduction of computers into the office and home environments has led to the development of numerous software packages to achieve many different tasks. Common applications such as word processing, form generation, database management, electronic mail and spreadsheet modeling have become almost essential in the workstation environment. The unintegrated software packages generally available have different operating instructions, different command vocabularies and often different underlying conceptual models of the system and its data. Data structures and files belonging to one application are often inaccessible to another. Users must learn the fundamentals and conceptual model of each application separately, even though many of the basic operations in the different applications are essentially identical.

An integrated software system is a software package which includes a number of logical applications and which attempts to minimize the disparity by providing a single conceptual model of the underlying system for all applications. The single unified view of the system overall allows a user to learn the system more easily and perceive it as an actual single integrated system.

A. THE ISS

An integrated software system combines the functions of its applications into a single package by using a single conceptual data object and command vocabulary. The vocabulary includes a basic set of commands, which apply to every logical application, and application specific commands for each application. The basic set of commands perform the operations which are common to all applications. Since only one conceptual data object is defined in the system, these commands operate uniformly on their operands regardless of the application, and represent the intersection of functions between the logical application areas. By providing only one set of common operators on one data structure, a user can learn to use the system easily and quickly. A user may become familiar with new applications by learning only a small additional number of application specific commands and functions.

This thesis describes the design of an Integrated Software System, the ISS, which uses the Relational Database moael as the underlying conceptual model, with the single data object being the relation. The relation is more easily discussed and understood as being simply a table where the tuples are just rows and the attributes just column names. Both of these terminologies will be used in this thesis. The ISS design integrates five applications or application areas: word processing, form generation, database management, electronic mail and spreadsheet modeling. A table format is designed to support each application area and the intersection of functions is described by defining the primitive operators which operate on all tables and on which applications and utilities may build to effect their particular transactions.

B. THESIS OBJECTIVES

The first objective of this thesis is the design of the system of tables to support the ISS. Data tables and system tables were designed also to support the concurrent research into multilevel security [Ref. 2].

Secondly, a set of primitive table operators and general system commands is defined which form the kernel of the ISS. These operators may be used by a single user and by applications to manipulate data tables in the desired fashion. A set of extended binary operators is defined which is traditionally used in database applications, but which is useful for other applications as well. Thirdly, a simple graphics oriented user interface for the ISS is defined.

Finally, a prototype implementation strategy for the data table object and the primitive operators is described for the Unix environment. This crude design when implemented may then be used to prove the utility of the ISS to support user needs in an operation environment.

The work in this thesis is a refinement and extension of the work carried out by Nishimura in his thesis, "Analysis Of The Relational Database Model In Support Of An Integrated Applications Software System." [Ref. 1] The tasks to re-evaluate the logical data bases and basic operators, to design the basic user interface and, to design a simple prototype were chosen from Nishimura's suggestions for follow-on research.

C. THESIS OVERVIEW

The system described in this thesis is the conceptual model of the ISS, its data object and operations. The design of tables and basic operators are complete in that they can support all the application areas. As described in section B above, the objectives of this thesis are to address this kernel to ensure its completeness and consistency with the overall conceptual model. This model is developed as a "single user at a time" relational database model.

The intention is to develop the conceptual model for the user without regard to physical realities which must be considered in implementation of the system. Despite this attempt at maintaining simplicity, a certain amount of knowledge regarding fundamental relational database theory is assumed, particularly regarding the material in chapter II and section C of chapter IV.

Provision is made for multilevel security enforcement by the inclusion of special tables and fields for that use and are included for the furtherance of concurrent research [Ref. 2] into multilevel security. Essentially multilevel security requires document access control by security classification, special group membership or job status (by compartment and caveats), or by explicitly named access lists. This access control must be enforceable at a document and paragraph level and when combined with generally accepted database controls on column values, amounts to the requirement of controlling access to table entries by column and row. In addition a table of legitimate users is required to be kept which includes the user's clearances and other covers. The maintenance of multilevel security in any system is a complex problem and is beyond the scope of this thesis. The framework for such controls is included, but the issue is not developed further. Interested readers may consult the references for further details of multilevel security requirements.

Chapter II surveys related work in the relational database (RDB) field, in particular the analysis which was done to show the feasibility of using the RDB model in an integrated environment. Other research efforts are discussed which bear little direct impact on the conceptual design of the ISS but which may be helpful in subsequent work on implementation and application development. The use of the RDB model for applications other than those chosen for the ISS is briefly described.

Chapter III describes the set of tables which make up the ISS: the system tables required to maintain the ISS, the data tables of each application, and how they both may be used to support their applications.

Chapter IV defines the abstract interfaces of the primitive operators available to manipulate data tables, and how they may be used to realize the basic operations common to all of the applications. These operators form the kernel on which applications may be constructed. A further set of combining operators is defined which is commonly found in database applications but which are shown to be of use in the other application areas and in some cases in combining tables from different applications.

Chapter V presents a simple, graphics oriented user interface for the kernel of the ISS. The general system operation and the form of the basic operators are defined in this design. The design resembles a form filling process similar but simpler than IBM's Query By Example database query language. Commensurate with the goal of uniformity for the ISS, this design is intended to provide a basis for the design of the commands in each application.

Chapter VI describes a simple prototype implementation strategy for the kernel primitives and general system, using the Unix programming environment. It describes the methodology which can be used to write very simplified I/O modules to map the designed user interface into the abstract interfaces of the lower level modules.

Finally, chapter VII presents our remarks, conclusions and suggestions for follow-on research.

II. RELATED WORK

In this chapter we discuss previous work in application systems which use a Relational Database (RDB) as the underlying model. Section A describes in detail the initial design of the Integrated Application Software System which motivated this thesis. Section B discusses previous work on editing and relation browsing, general material describing desirable features of editors, and how relation browsers and editing may be implemented in the Relational Model. The final section describes briefly other applications for which the Relational Model can be used.

A. INITIAL DESIGN

The premise of this thesis proposal is an extension of the research efforts presented by Nishimura, [Ref. 1]. In the initial design phase, Nishimura evaluated the utility of the Relational Database model to conceptually integrate the five application areas of interest: text processing, relational database management, form generation, electronic mail and spreadsh[illegible] [illegible]eling.

Motivation fo[illegible] [illegible]esis research stemmed from the realization that [illegible] of these five applications perform similar functions and are conventionally utilized as a non-integrated collection of application software. In this non-ISS, each application implements operations on dissimilar file types and the data among the files are not directly sharable. In addition, in order for the user to utilize them, he or she is required to learn a large number of commands for each application, many of which are synonymous. If the commonality of the application areas could be collapsed into one application like the kernel of an ISS, this would significantly reduce the redundancy and minimize the command vocabulary needed to utilize the applications.

After choosing the relational database model as the kernel for the ISS, it was decided that the most natural data object to use in the system was the table. In the ISS, each of the five applications is a logical database consisting of a set of tables. The set of tables are of three classes: Application Directory, Data Table Schema, and Data Table. Each column in the table represents one attribute of the file and each row represents a unique occurrence. The tables include columns which represent key values to uniquely identify each row. Any datum in a table can be accessed by specifying the name of the table, the value of the key, and the name of the attribute containing the datum.

The Application Directory Table contains descriptive and definitional data about the data tables in an application or logical database. Each row in the Application Directory

Table describes one data table and has a standard schema. For each application, the schema of the Application Directory Table can be augmented to include additional data table attributes.

A data table represents the logical file of an application. The data tables are typed in accordance with their primary use, i.e., text, form text, database, spread sheet, and mail. The data tables are categorized by types in order to logically organize those which are used primarily by the same application. Since one of the key objectives of an ISS is to be able to share data, strong data table typing is not a feature in the design.

The Data Table Schema Table contains a row for each column in a Data Table. In some sense this schema represents variable declarations in a conventional programming language.

Each row in the Application Directory Table is linked to a set of rows in the Data Table Schema Table and a Data Table. The same relationship among the tables exist for each application.

Since the Application Directory Table of each application is identical, no further discussion of that table type is necessary, however the data tables are different. All data tables in one application, except the database application, have the same structure. All tables have an ID column. The Text Data Tables have just one other column, TEXT LINE which is the text. Form data tables similarly have two columns except the second column, FORM LINE is meant to be used differently by the form processing application. A Mail Data Table has eight columns which in each row describe the origin and destination of each message, and hold the message itself or pointer to a large message body. Spreadsheet Data Tables have six columns so that every row describes an X,Y coordinate pair for the conventional spreadsheet view. The Database Tables are the only ones without a predetermined structure, they have the ID column and any number of other columns required by a user.

In order to realize the main objectives in an ISS, one must present the user with a single conceptual view of the system regardless of the context of its use. The practicality of this objective was demonstrated through the use of tables as described above. In addition, the user must be provided with a common set of table operators as well as a set of application dependent operators. The intersection of operations proposed to manipulate data in the files is comprised of the locate, insert, modify, delete, copy, and move operations. Nishimura gave conceptual descriptions of how the six basic ISS primitives can perform the operations in the functional intersection. The primitive operators suggested (insert, modify, delete, project, select, and union) are based on relational algebra.

In the initial design phase it was shown how these operators could be utilized on each application to yield the same result as all conceivable queries in a non-Integrated Software System environment.

It is the intent of this thesis to refine this conceptual design and view. The refinement will ultimately prove the feasibility of these concepts by implementing the fundamental ideas presented herein.

B. TEXT EDITING AND RELATION BROWSING

The need to edit and browse relations is important to each of the applications in an Integrated Software System. Editing of text particularly is the subject of much research, both psychological and technical, to determine the qualities of a good editor.

An ideal editor is a subjective term determined by the needs of the user, however the following general features may be identified as desirable: [Ref. 3]

(1) A consistent conceptual model of the the system with a clear and concise user interface including on-line help and documentation.

(2) Powerful facilities that take advantage of computing power, with an "infinite undo" capability.

(3) An ability to edit a format which closely resembles the intended target format. (Whether it be intended for hard copy or not.)

(4) Fast and visible responses to all commands however trivial or complex.

(5) Access to shared information and other contexts on the same display surface without leaving the editor, and the ability to access other applications or to be part of a larger integrated environment.

(6) The ability to edit mixed targets such as text, forms, programs and data.

The above points are worthy goals for editor and browsing interfaces and have been considered in our system design.

A Browser, as its name implies, is a software facility which allows the user to "browse" through information related to an application. TIMBER (Text, Icon, and Map Browser for Extended Relations), as discussed in [Ref. 4], is the design of a sophisticated user friendly, graphics oriented browser for a relational database.

TIMBER was designed to meet the objectives of four application areas. It provides user interface to a relational database system that can be:

(1) a relation browser for fixed format relations;

(2) a sophisticated browser for relations with icons;

(3) an editor for text data stored in relations;

(4) a map browser for geographical data.

The designers of TIMBER propose that each of the aforementioned applications virtually perform the same function and thus can be served by an Integrated Software System.

The basic concepts of TIMBER are: a window on a sophisticated graphics terminal; a relation in a database; and a cursor. The screen of the terminal can be split into several rectangular windows to which relations are bound. The tuples from the relations appear in the windows and can then be manipulated by TIMBER commands. The cursor on the screen is controlled by a bit pad and a mouse and many TIMBER commands affect the window in which the cursor is currently positioned.

The relations of TIMBER are of four types, each of which has a default screen format and coordinate system. The types are: normal fixed format, text, icon, and map. There are three ways in which the user can alter the contents of the windows on the screen. One can move the cursor, use the ZOOM feature, or use the six relational commands affecting the window.

TIMBER is composed of six major modules: Application Program, Intelligent Buffer, High Graphics, Low Graphics, Smart CRT, and INGRES. INGRES, however, does not support TIMBER without several added extensions. First, ICON must be added as a new data type. Second, the notion of ordered relations must be incorporated in INGRES in order to support the storage and manipulation of documents. Finally, there is a need to incorporate an efficient concurrency control system which will enable multiple concurrent browsers which perform updates.

In summary, the design of TIMBER, which is intended as a sophisticated two-dimensional graphical browser for text relations, fixed format relations, and relations containing icons or maps, provides several impressive features which support the use of the relational database model as the kernel of an ISS.

Several research projects including those mentioned below (section C.) have dealt with aspects of text editing and storage within the relational model, but the most extensive work in this area has been published by Stonebraker and various co-authors. They have considered the requirements of document processing and proposed enhancements to RDB management systems to support two text formats and a capable editor. [Ref. 5] Stonebraker's proposal is to provide facilities in a RDBMS which can support calls from a text editor running as an application program

The two schema for text format are:

(1) a binary relation with attributes line number and and a text line, and

(2) a ternary relation with attributes sentence, word number and text word of some maximum length.

Each scheme has an advantage for certain relational operations and the authors suggest that both are appropriate and that facilities are required to transfer from one to another. The first enhancement is the inclusion of variable length strings as table column types which may be stored as variable length fields or external to the database with pointers to their location. Since the order of these lines of text (and the order of the words in the case of (2) above) is important, Stonebraker describes the mechanism of ordered relations in which the database manager effects the ordering using unique tuple identifiers (TIDs) which are assigned by the system to each new tuple added to the relation. An ordered B-tree access structure is kept in which TIDs are at the leaves in the intended ordering on the tuples and each internal node holds the number of TIDS in each of its subtrees. The TIDs then act as keys into the relation containing the text, in which the text lines are unordered. Access to a specified line number is easily done by keeping count while traversing through internal nodes, selecting the appropriate subtree to find the correct leaf. Insertion and deletion is also easy, simply updating the internal nodes of the access structure and inserting or deleting the TID at the leaf, using standard B-tree algorithms. A generalization of this structure is described for format (2) above where the leaves contain a pointer to a second B-tree holding the ordered words within the ordered sentences.

Stonebraker introduces the idea of "extended wild card" substitutes for any character string, which may be used in qualification and target lists of replace and substitute commands. Essentially each extended wild card of form *i (for i in some integer range), once matched to a substring in an expression retains the value of that substring and so may be used in the target list.

To support other substring operations a simple substring operator is described which can select substrings between bounds set by position or content. A "break" operator is defined to perform the transformation of text in form (1) above to form (2), and a more generalized concatenation operator described for the reverse transformation.

Stonebraker carried out some relative performance evaluations of an editor running as an application program on the Relational Database Management System INGRES under UNIX, against the performance of the UNIX line editor ED. His results may be found in the reference, but they were sufficient to indicate that indeed a Relational Database System could be useful in the processing of text in ordered relations.

C. OTHER RDB MODEL APPLICATIONS

With the maturing of database technology, there has been a growing awareness of its usefulness in applications other than data processing. Areas such as Office Information

Systems, Engineering Design, Programming Environments, Operating Systems and recently, Artificial Intelligence, have been considered as areas where the Relational Model will be useful.

A recent approach to Automated Business Procedures has been the development of a "forms" oriented design of an integrated office, including word processing, mail and office communication tasks. [Refs. 6 and 7]

Essentially forms are stored in a Relational Database, each table representing a form type with each tuple an instance of a raised form. Included in tuples can be fields containing audit information, and of course integrity constraints can be applied to signature and special data fields. Each form type has its own display format which may include the details currently shown on printed forms, or be as simple as just one large text field for data entry for a text file. The display format for general users would not include audit information and is actually a view of the database. Forms can be used to display forms, enter data, perform word processing, and execute queries with the underlying RDB Management System providing the necessary facilities. Queries can be expressed by general users using the display views much like QBE, and more detailed queries may be framed on the conceptual database for audit and statistical purposes.

The increasing size and complexity of designs, in particular VLSI, and the problems of managing their data, has led to much research on the use of Relational Database Management Systems for these purposes. In doing so it is felt that extensions are required to the RDB model to suit the applications. Stonebraker, Rubenstein and Guttman [Ref. 8] discuss the inclusion of abstract data types (structure and operations) in INGRES and the use of extended indices for them. They propose that these extensions will help to improve the usefulness of database systems in Computer Aided Design applications.

Lorie and Plouffe [Ref. 9] discuss extensions to System R to make it more appropriate for engineering and design applications. They describe a method of storing complex objects, each spread across a number of different relations in which each component of an object is related to a hierarchical parent component in a tree structure. They also describe a mechanism for the sharing of design objects within a design environment which allows for the relatively slow transaction time in a design activity as compared to a straight forward data processing transaction. These ideas have also been developed by IBM's Yorktown Heights Computer Science Research Division [Ref. 10] and include designs for the storage and retrieval of arbitrarily long fields for both simple and complex data objects.

Representation of programs in the Relational Model and using that model to create, store, debug and manage program information is the primary goal of the OMEGA programming environment. [Ref. 11] An editor/query processor is designed on top of a Relational DBM system which gives a unified view of program and data and provides very powerful facilities for debugging and monitoring execution of programs. In this project they have found it necessary to add variable length strings and ordered relations to the RDB model, as well as three extensions which support the programming environment.

A new approach to Operating Systems has recently been suggested [Ref. 12] which uses an underlying RDB to present a unified conceptual view of the system to both user and systems programmer. An operating system can be implemented as a large integrated package using a Relational DBM system as a kernel. Interaction with such a system would be by access and manipulation of system relations using a command language based on a relational query language. The inclusion of a transaction mechanism to abbreviate commonly used commands would add a richness to the command language while retaining the conceptual view of the operations. The authors suggest that a graphical database query and update language like QBE would provide a flexible, easy, powerful and friendly user interface to such an operating system.

There has been recent interest and discussion [Ref. 13] on the possible uses of a relational DBM system as the model for Artificial Intelligence expert system knowledge bases. The similarity between stored data and "facts", the potential to include in a database an inference mechanism, and the possible use of some sort of trigger to emulate "rule firing" are questions which have been posed but not yet researched.

III. STRUCTURE AND USE OF ISS TABLES

The set of tables chosen for the design of this system is an extension and refinement of those described in Chapter II, including extra system tables for system use to provide a mechanism capable of supporting multilevel security. This full set of tables are those designed for the conceptual view of the system overall using the relational database model and need not necessarily be implemented exactly as described.

Within each of the five application areas there are four classes or types of tables: Data Tables, a Directory, Paragraph Classification Tables (PCTs) and Access Control Tables (ACTs). The data tables for each application are the tables created or used by the user while the others can be considered system tables for mostly system use. Although the data tables in each application area differ, the system tables have similar structure and function in each of the five areas. In addition to the above tables which make up the logical databases of the five applications, there are three table types which are used by the ISS kernel: (1) a Users Login Table, (2) a Capabilities List Table for each user and, (3) a Schema Table which describes every unique attribute or column of every table in the system. Of the above tables the Login, PCTs, ACTs and CLTs are used by the system to enforce security.

The sections of this chapter describe the structure and general use of the above table types, first dealing with the system tables and then each of the five application area data tables: text, forms, data, electronic mail and spreadsheet. The Schema Table is described first and then its format is used to describe the structure of every column of all fixed table types. The ISS must be installed with the schema rows describing the columns of system tables and the columns in the four standard table types for text, forms, spreadsheet and mail. Schema Table rows for previously undefined columns in Database Data Tables will be added as users create the new database tables.

Reference and retrieval of tables and tuples by the system is by name, with the advantage that this provides the most easily understood interpretation of pointers.

In this thesis we will not discuss the recursive problems encountered when those tables provided for multilevel security are, themselves, treated simply as data tables requiring coverage by further security controls. The special multilevel security tables (ACTs, PCTs and CLTs) have been included to provide a framework for future research and are not dealt with in later chapters on implementation strategies.

A. SCHEMA TABLE

The Schema Table is a single table which contains a row for each different column (attribute) name throughout all tables in the ISS. The system uses the schema in conjunction with a column listing in a directory to determine the structure of a table it is about to manipulate. The Schema Table columns are illustrated in figure 3.1 and are described by tuples from the table itself as shown in figure 3.2.

The ID column is a six digit integer which is a field simply representing the conceptual ordering of the rows in the data base and their display order as a table. The underlying physical system (implementation) need not store the relation in this fashion. It is attached to all tables in the system and corresponds to the record number in systems such as DBASE II. It will not be described again for other tables.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE

Figure 3.1 - Schema Table Schema

The NAME column is simply the textual name of a column which may appear in many tables of one or more types or a single table such as some database table. "ID" and "NAME" are typical values which may be found in this column. TYPE and WIDTH simply describe the physical data type and maximum size with the convention of "0" meaning "of varying length". SYNONYM is a column which gives the names of columns throughout the system which have the same characteristics and may be considered to carry compatible data under certain circumstances. TABLE gives a particular table name or type of table in which the column being described may be found. A simple literal denotes a particular table, a literal preceded by "-" indicates all of a particular table type or class. For example the "-ALL" in the TABLE column of figure 2.2 indicates the ID column is in all tables. Similarly -TEXT in a TABLE column would indicate the column being described by that row is in all Text Data Tables. Similar meanings are attributable for the other fixed structures, -FORM, -MAIL and -SPREAD.

The Schema Table is used by the system for retrieval of tables and by a user while creating a new Database Data Table. (The structure of tables in the other applications are predetermined and are described in sections G through K.) In the process of creating a new database table, its structure will be defined as the user describes each new attribute of the new table by appending a new row to the Schema Table.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE
	ID	INTEGER	6		-ALL
	NAME	CHAR	20		SCHEMA
	TYPE	CHAR	8		SCHEMA
	WIDTH	INTEGER	8		SCHEMA
	SYNONYM	CHAR	0		SCHEMA
	TABLE	CHAR	0		SCHEMA

Figure 3.2 - Self Describing Tuples in Schema Table

B. ISSUSERS TABLE

The ISSUSERS Table is used by the system to establish identities and security clearances of the user. The table has one row for every logical user of the ISS and should be accessible only to the database administrator (DBA). The table schema is illustrated in figure 3.3 and the columns are described by the appropriate rows from the Schema Table at figure 3.4.

The USER_NAME is simply the name of a logical user and need not be unique since the key to the table is the USERID which must be a unique alphanumeric string for each logical user. PASSWORD is an encoded password and each of the fields with application names is a logical indicator of a user's permission to access the directory and data tables in that application. CLEARANCE is the users security clearance and CC is a record of the user's compartment and caveat status in a multilevel security environment. ROLES is a field which may be used to denote special functions a user may have such as database administrator, or reviewer in a multilevel security system. CAPABILITIES is the name of the capabilities list table associated with that user and is described in section F of this chapter.

When a user invokes the ISS, the system retrieves the appropriate row from the ISSUSERS table and conducts a standard password procedure. Enhancements may enable the system to allow a limited number of password entry failures and then lock the device or initiate some alarm procedure. The users clearances, compartments, roles and access rights are retained as system variables throughout the ensuing session and can be used for access control during all transactions.

Insertion of new users may be done by the DBA by appending new tuples to the ISSUSERS relation. Similarly clearances, compartments and roles of the users may be changed by the DBA by modifying the appropriate values in the desired row. It may be desirable to include in the system a routine or view mechanism to allow users to change their own passwords, since a one way encoding step must first be applied to a password before storage.

ID	USER_NAME	USERID	PASSWORD	TEXT	DATABASE

FORMS	SPREAD-SHEET	MAIL	CLEAR-ANCE	CC	ROLES	CAPABILITIES

Figure 3.3 - ISSUSERS Table Schema

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE
	USER-NAME	CHAR	20		ISSUSERS
	USER-ID	CHAR	10		ISSUSERS
	PASSWORD	CHAR	20		ISSUSERS
	TEXT	BOOLEAN	1		ISSUSERS
	DATABASE	BOOLEAN	1		ISSUSERS
	FORMS	BOOLEAN	1		ISSUSERS
	SPREAD-SHEET	BOOLEAN	1		ISSUSERS
	MAIL	BOOLEAN	1		ISSUSERS
	CLEARANCE	CHAR	15		ISSUSERS
	CC	CHAR	0		ISSUSERS
	ROLES	CHAR	0		ISSUSERS
	CAPABIL-ITIES	CHAR	20	TABLE_NAME	ISSUSERS

Figure 3.4 - Schema Table Rows For ISSUSERS table

C. APPLICATION DIRECTORY TABLES

A Directory Table exists for each of the five application areas of the ISS with each row in the application directory describing exactly one data table. These directories can be used by users and by the system to find particular data tables which exist for the selected application. The schema for the Directory Tables is illustrated in figure 3.5 and each of the columns is described in figure 3.6 by the appropriate rows from the Schema Table.

ID	TABLE_NAME	COLUMNS	KEYS	O_CLEAR	O_CC

M_CLEAR	M_CC	ACT_NAME	PCT_NAME	REVIEWED	REV_REQD

VIRTUAL	CONDITION	GLOBALS	OWNER	DESCRI-PTION

Figure 3.5 - Directory Tables Schema

In each row of a directory, TABLE_NAME is the unique name of a data table which is being described by the row and is the pointer to that table in the underlying relational database management system. COLUMNS is a list of the columns in the data table, and with the Schema Table tuples for those columns, completely describes the structure of the data table. KEYS is a field containing the names of those columns which comprise the key to a table. O_CLEAR and O_CC are fields giving the overall security level, compartments and caveats of a table. M_CLEAR and M_CC give the minimal requirements to access some parts of the table, particularly Text Tables, where low security paragraphs in a high security document may be made accessible to those with the lower security clearance. PCT_POINTER and ACT_POINTER are the names of the two tables which control column and row wise access to every data table. REVIEWED and REV_REQD are logical values required by organizations practicing multilevel security. VIRTUAL is a logical field indicating if the table is composed from other ISS tables, and if true, then CONDITION is the description of the operations to be performed to realize the table. GLOBAL is a text string which may be used to contain print formatting or display mode data, or other parameters useful to the system. OWNER is simply the originator's userid and DESCRIPTION is a short textual description of a table.

The Directory Table may be used by a user to view the directory for any one application, and by the system to determine the structure of any retrieved table. The ID field is always a key field for any table. The freedom to select a key exists only for database data tables since other table structures are fixed. The setting up of access structures (indices) using these keys is of no consequence to the conceptual model and is not considered in this thesis. During display of a directory, information in the security fields may be used to filter the rows displayed so that users without sufficient clearances or need to know will not discover the existence of tables that exceed their clearances. During any call for a particular table the directory may be consulted to confirm eligibility for access to that table on the grounds of security clearances, compartments and caveats.

The GLOBALS field may contain application specific information in textual form, for example a text application may place in the GLOBALS field of the directory entry a page length for printed format by including .pl 60, or a page header .ph "NPS THESIS". A Mail Directory Table row can contain the name of the Mail Data Table owner, and possibly redirection instructions. In a spreadsheet directory GLOBALS may be used to store recalculation order. In any case, the information in GLOBALS is related specifically to the application so each application must have the intelligence to retrieve and use the tokens in this field.

Creation of new data tables is done conceptually by a user creating a new tuple in the directory or conversely creation of a table causes a new entry to be placed in the directory. It should be noticed that with the full multilevel security controls this would imply the creation also of the two access tables to be described in following sections. Figure 3.7 shows an example of a single row of a text directory.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE
	TABLE_NAME	CHAR	20		-DIRECTORY
	COLUMNS	CHAR	0		-DIRECTORY
	KEYS	CHAR	0	COLUMNS	-DIRECTORY
	O_CLEAR	CHAR	0	CLEARANCE, M_CLASS	-DIRECTORY
	O_CC	CHAR	0	CC,M_CC	-DIRECTORY
	M_CLEAR	CHAR	0	CLEARANCE, O_CLASS	-DIRECTORY
	M_CC	CHAR	0	CC,O_CC	-DIRECTORY
	PCT_NAME	CHAR	20	TABLE_NAME	-DIRECTORY
	ACT_NAME	CHAR	20	TABLE_NAME	-DIRECTORY
	REVIEWED	BOOLEAN	1		-DIRECTORY
	REV_REQD	BOOLEAN	1		-DIRECTORY
	VIRTUAL	BOOLEAN	1		-DIRECTORY
	CONDITION	CHAR	0		-DIRECTORY
	GLOBALS	CHAR	0		-DIRECTORY
	OWNER	CHAR	10	USERID	-DIRECTORY
	DESCRIPTION	CHAR	0	TEXT_BODY	-DIRECTORY

Figure 3.6 - Schema Table Rows For Directory Tables

ID	TABLE_NAME	COLUMNS	KEYS	O_CLEAR	O_CC
	RECIPE1	ID,TEXT_LINE	ID	UNCLAS	

M_CLEAR	M_CC	ACT_NAME	PCT_NAME	REVIEWED	REV_REQD
UNCLAS		ACT1234	PCT1234	TRUE	FALSE

VIRTUAL	CONDITION	GLOBALS	OWNER	DESCRIPTION
FALSE		.pl 60,.lm 10	2516P	Brownies

Figure 3.7 - Example Row From Text Directory

D. ACCESS CONTROL TABLES

Access Control Tables (ACTs) are tables which are associated with every data table in the ISS. To satisfy multilevel security and need to know arrangements the ACT for a particular data table lists explicitly the userids allowed to access that table and what rights each userid has concerning the table overall, the tuple level or on individual columns. The ACT schema is illustrated in figure 3.8, however the number of columns depend on the number of columns in the data table being described since the attribute column is repeated for each column in the data table. ACTs related to data tables within one of the text processing, form generation, spreadsheet or electronic mail applications will all be the same structure since data tables in one of these applications have a fixed number of columns.

ID	USERID	MODIFY	TUPLE	ATTRIBUTE	ATTRIBUTE	-
						-

Figure 3.8 - Access Control Tables Schema

Figure 3.9 describes the structure of the three as yet undescribed attributes of the Access Control Tables. The USERID has already been described in figure 2.5 except that the TABLE field for that row should now read "ISSUSERS ,-ACT". MODIFY is a boolean indicating a users right to modify the schema of the data table. This will generally be NO (or FALSE) since the structure of all application tables except Database Tables are essentially predetermined by the design of the system. TUPLE is a field containing up to four characters indicating the users right to insert(I), delete(D), read(R) or update(U) entire tuples. The repeating ATTRIBUTE columns may contain read(R) or update(U) controlling each individual's access to the columns of a data table in the order they are listed in the directory. Explicit TUPLE column values of read or update imply the value for all the individual attribute columns, for example a TUPLE update(U) implies a U in all of the attribute columns whether they are there or not.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE
	MODIFY	BOOLEAN	1		-ACT
	TUPLE	CHAR	4		-ACT
	ATTRIBUTE	CHAR	2		-ACT

Figure 3.9 - Schema Table Rows For Access Control Tables

The ACTs are useful to enforce views and multilevel security requirements. When accessing a data table via the directory the number of columns in the data table and the name of the associated ACT is retrieved. The system must then enforce the restrictions imposed by the ACT during use of the data table. If an all users (USERID = -ALL) entry is not in an ACT then users who are not explicitly on the list have no access to the data table with which the ACT is associated. An individual's rights to a file may be determined by more than one entry, for example -ALL may read tuples and 2516p in addition may update tuples. This mechanism may be used in multilevel security to enforce need to know lists of personnel for particular documents.

Although every table must have an associated ACT, there is no requirement for the ACT to be unique provided it describes the correct number of columns. For example within an application one ACT may be used as a simple owner write, all others read access table with which many data tables are associated. Figure 3.10 is an example of such a table which may be named in the ACT_PTR field of any text directory or forms directory row because it has two attribute columns. (One for ID and the other for TEXT-BODY or FORM-BODY.)

ID	USERID	MODIFY	TUPLE	ATTRIBUTE	ATTRIBUTE
	-OWNER	NO	I U R D		
	-ALL	NO	R		

Figure 3.10 - Example Access Control Table

In this example the hyphen is used to designate special userids such as owner and all users.

Discussion and *implementation strategies for* these tables are left for further research into multilevel security issues in an ISS.

E. PARAGRAPH CLASSIFICATION TABLES

The Paragraph Classification Tables (PCTs) are designed to provide access control, primarily of Text Data Tables, at a paragraph level as required for multilevel security systems. Since a paragraph may consist of a single line of a Text Table, essentially line by line control is required.

Figure 3.11 illustrates the schema of PCTs and figure 3.12 shows the Schema Table row for the OFFSET column which has not already been described in prior sections.

ID	OFFSET	CLEARANCE	CC

Figure 3.11 - Paragraph Classification Tables Schema

Each data table is covered by a PCT although as in the case of the Access Control Tables, the PCTs need not be unique and may cover many data tables. The Paragraph Control Table for each data table is named in the PCT_NAME field of the directory entry for that data table.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE
	OFFSET	INTEGER	6	ID	-PCT

Figure 3.12 - Schema Table Row For PCT Column OFFSET

The table is ordered on the offset field, the first and possibly only tuple being for an OFFSET of 1. The clearances, compartments and caveats for an offset represent those placed on the material in the associated data table at that ID and greater, until superceded by another row in the PCT. During retrieval and manipulation of tables, particularly text, only those rows of data which are less than or equal to the userid's security levels and clearances will be made accessible.

Figure 3.13 shows an example of a simple paragraph control table which will be pointed to by name from the directory entry for some text data table.

ID	OFFSET	CLEARANCE	CC
1	1	UNCLAS	
2	10	SECRET	
3	20	UNCLAS	

Figure 3.13 - Example Paragraph Classification Table

The PCT indicates that IDs (lines of a text document) 1 to 9 are unclassified, 10 to 19 are secret and 20 up to the end of the document are unclassified. Should the user be allowed access to the data table by the controls described in the previous section, and have a clearance less than secret, then he will be able to see the unclassified part of the document. As with the Access Control Tables, strategies for implementation and use of these PCTs to implement multilevel security measures are left for further research.

F. CAPABILITIES LIST TABLES

Capabilities List Tables (CLTs) are a multilevel security table associated with each userid in the ISS system and will be named in each row of the ISSUSERS table in the CAPABILITIES column. (See figures 3.3 and 3.4).

ID	TABLE_NAME	TYPE	OWNER

Figure 3.14 - Capabilities List Tables Schema

The information they contain is sufficient to determine immediately all tables to which any userid has access. It has been suggested [Ref. 2] that such tables are required in a multilevel security environment, although the data they contain is mostly redundant. One solution may be to provide the tables as virtual tables. Figure 3.14 illustrates the schema of the CLTs. All the Schema Table rows for the columns have already been described in previous figures except that now "-CLT" (all CLs) must be added to the TABLE and OWNER columns in the appropriate Schema Table rows to indicate that the columns are also used in the Capability List Tables.

For each userid in the ISSUSERS table the CLT is simply a list of tables to which he has access before security levels, compartments and caveats are token into account. The CLTs are a redundant method of enforcing need to know restrictions and requires positive input from the creator of any data table. Mechanisms to maintain this list would clearly be linked with the multilevel security precautions used to create and manipulate tables and are not discussed further in this thesis.

G. TEXT DATA TABLES

The Text Data Tables contain non-formatted textual information. This data can be used in a myriad of applications, including preparation of documents or computer programs, or as textual information to be combined with other application data tables by one or several of the ISS operators.

The Text Data Table contains two attributes, ID and TEXT_LINE. The schema for the ID has been previously defined. The schema for the TEXT_LINE attribute is shown in figure 3.15. The "0" in the width field indicate that the TEXT_LINE is of varying length. As depicted, the FORM_LINE attribute of the form data table and the BODY attribute of the MAIL data table can be aliased with the TEXT_LINE attribute of the Text Data Table.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE
	TEXT_LINE	CHAR	0	FORM_LINE BODY	-TEXT

Figure 3.15 - Schema Table Row For Text Data Table

Figure 3.16 shows an example of a Text Data Table. Each row is uniquely defined by the ID and TEXT_LINE. Any type information can be entered into the TEXT_LINE and all the data, (except the ID), will appear on output in the same format in which it was input. (Note that row 3 of figure 3.16 contains the characters ".sk 2". This character sequence will have no special meaning in the context of the ISS Text application. However, the Text Data Tables can be used as input to a TEXT FORMATTER APPLICATION SYSTEM in which these characters or similar syntax would have a special meaning, i.e. "skip two lines"). The key issue is that all user input into the Text Data Table will be operated on by ISS as pure textual or literal information.

ID	TEXT_LINE
1	What can be better
2	than this?
3	.sk 2
4	DON'T ASK!

Figure 3.16 - Text Data Table

H. FORM DATA TABLES

The Form Data Tables are a special set of Text Data Tables. They act as a window or view into the ISS Database

Data Tables and may be used for entry or extraction of data. Forms are generally used for repetitive tasks such as the printing of letters with the same or similar bodies but many different addressees. Forms may also be used in order to create facsimiles of documents on a CRT screen. These facsimiles of documents can then be used to facilitate easy insertion or retrieval into or from a Database Table. This will serve as a valuable tool for users who are more familiar with the order and relationships among entries on the documents than with the standard tabular formats used to manipulate data in a database system.

Form Data Tables are comprised of only two attributes, ID and FORM_LINE. Figure 3.17 illustrates the schema for the FORM_LINE. As shown, the TEXT_LINE attribute of a Text Data Table and the BODY attribute of a Mail Data Table can be used synonymously with the FORM_LINE attribute of the Form Data Table.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE
	FORM_LINE	CHAR	0	TEXT_LINE BODY	-FORM

Figure 3.17 - Schema Table Row For Form Data Table

Figures 3.18 and 3.19 show examples of two distinct types of Form Data Tables available in the ISS. In order to use either Form, the user must specify the name of the underlying Database Data Table to be invoked, hereafter referred to as the default Database Data Table.

Figure 3.18 is the simplest type form and can be used both for extraction and insertion of data from or into a single Database Data Table (file). In this type of form the user may operate in two distinct modes: insert or retrieve. In the retrieve mode a form with pre-printed attributes will appear on the screen with the attribute designated as a key enclosed in braces, { }. The user has a choice of specifying a unique tuple to be retrieved or requesting all tuples. If he wants a unique tuple he will enter the value of the key attribute for the desired tuple. The remaining data will be retrieved from the database by the system and inserted into the form. In figure 3.18, the user, operating in the retrieval mode, entered the value "123-45-6789" for the SS# database attribute. The ISS then used this value to search through the database for the matching tuple and automatically filled in the rest of the form with the attributes desired from this tuple.

The user may request that all tuples of the database be retrieved by using the KEY word , -ALL. If -ALL is used and there are 100 tuples in the associated Database Table, then there will be 100 forms filled in by the system. In all cases, the FORM_LINE(s) will appear in printed output exactly as it appears on the screen, including the data table attribute prompts. The ID of the FORM_LINE will not be printed.

In the insert mode the user will supply all the data as indicated by the blanks following each attribute. This data will be reformatted by the system to conform to the format of the underlying database data table before it is inserted into that table.

ID	FORM_LINE
1	NAME: G. Thompson {SS#}: 123-45-6789
2	ADDRESS: 2 Lane Rd , Somewhere, Fl. 22222
3	
4	OCCUPATION: Systems Analyst
5	SALARY: $100,000.00
6	

Figure 3.18 - Form Data Table

Forms of the type illustrated in figure 3.19 can be used only to retrieve data from one or several underlying Database Data Tables (files). In order to maintain database consistency, no insertions into the database are possible with these forms. Within forms of this type, text appearing in the form line can be of four distinct types. These types are distinguished by the presence or absence of special symbols within the text entered on the FORM_LINE.

If there are no special symbols within the text entered on a FORM_LINE, then the text is considered to be literal and will be printed or displayed as is. Text entered between two braces, { }, represent a key attribute of a given database data file and will determine which tuple(s) of the Database Data Table will be referenced for other retrievable data on the form. The format for this data type is {table.attribute} or {attribute}. The latter format implies that the default table is to be used. The user can enter a unique value to retrieve a single tuple, or he can enter the system variable -ALL to retrieve all tuples in the database.

The special characters, {}, can appear on a single form more than once but there must be a one to one correspondence between them and the distinct Database Data Tables referenced on the form. For example, row 1 of figure 3.19 contains the text, {NAME}. In this case, the user can enter a unique name to retrieve a single tuple from the default Database Data Table, or he can use a variable to retrieve all tuples from the default database. In row 7, the {B.SWEEPSTAKE} entry indicates that the SWEEPSTAKE attribute of database data table B will be retrieved. Table B must be distinct from the default table.

ID	FORM_LINE
1	{NAME}
2	[ADDRESS]
3	[CITY], [STATE], [ZIP]
4	
5	Dear [TITLE] [NAME],
6	You may already be a winner in our
7	{B.SWEEPSTAKE}. In order to find out
8	if you are an instant winner read the
9	following information.
10	/Sweepnum.Txt/
11	
12	If you are a winner, you may collect
13	your prize from [B.ADDRESS], [B.CITY],
14	[B.STATE].
15	
16	Sincerely,
17	
18	Sweepstakes Deluxe

Figure 3.19 - Form Data Table

Text entered between brackets, [ ], represent variables or attributes to be retrieved by the system from the corresponding tuples(s) retrieved previously using a "{ }"

FORM_LINE entry. The format for this type data is [table.attribute], or [attribute]. This type data is illustrated in rows 2,3,5,13 and 14 of figure 3.19. Note that in row 1 the FORM_LINE contains {NAME}, and in row 5 the FORM_LINE contains [NAME]. Even though they refer to the same field in the same database table, they differ in that {NAME} is used both to extract the value of the attribute, NAME, and to determine which tuple will be made the current tuple in the default Database Data Table. [NAME] simply extracts the value of the NAME attribute from the current tuple. In other words, {NAME} means "retrieve a new tuple with the given value in the name attribute, substitute its value for the symbols, `{NAME}', and retain the tuple as the current tuple of the default database data table". [NAME] simply means "substitute the symbols `[NAME]' with the value of the NAME attribute of the current tuple in the default Database Data Table.

Another type of text which can be entered on the FORM_LINE is data entered between slashes, / /. This indicates that the text to be displayed or inserted will be obtained from an existing Text Data Table (file). Its format is /filename.TXT/. In row 10 of figure 3.19, /Sweepnum.TXT/ indicates that the text file named Sweepnum will be inserted into the form upon view or print requests.

Forms of the types depicted in figures 3.18 and 3.19 are basic, but powerful enough to facilitate most applications involving Form Generation.

I. DATABASE DATA TABLES

The Database Data Tables differ slightly from the Text, Mail,Spreadsheet, and Form Data Tables in as much as there is no predetermined set of attributes that will apply to all Database Data Tables. More precisely, each Database Data Table may have one or several attributes in common with any other Database Data Table in the ISS. The attribute which must appear in all Database Data Tables is, of course, the ID. The ID is a key in each Database Data Table. Thus, as a user creates a new database data table the only schema for this table to appear initially in the Schema Table will be that of the ID. As the user defines the schema of each additional attribute of the given Database Data Table, these schemas will be automatically added to the Data Dictionary.

The Database Data Tables are used to form relationships among entity sets. Each Database Data Table defines one entity as described by the attributes or columns, and each row of the table represents a unique occurrence of the entity. Figure 3.20 illustrates a Database Data Table with an ID attribute and N user-defined attributes.

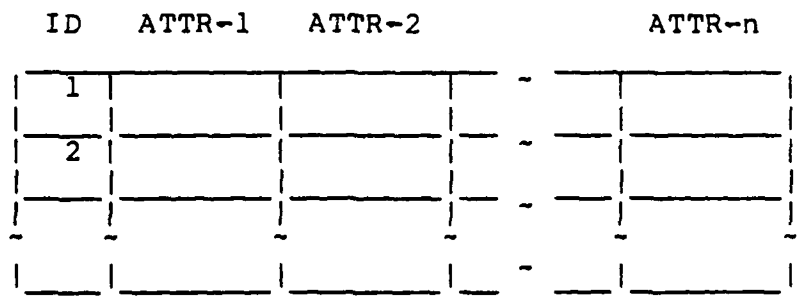


Figure 3.20 - Database Data Table

J. MAIL DATA TABLES

Electronic Mail is a utility which facilitates the exchange of textual messages among system users. In the ISS, each user with mail access rights will have a mail file which is essentially a set of messages, i.e. a Mail Data Table or set of Mail Data Tables.

The Schema Table rows for the Mail Data Table are shown in Figure 3.21. As indicated in the figure, the TEXT_LINE attribute of the Text Data Table and the FORM_LINE attribute of the Form Data Table can be used synonymously with the BODY attribute of the Mail Data Table.

The columns or attributes of the Mail Data Table are predefined by the ISS, and as shown in figure 3.22, they contain sufficient information to facilitate proper routing of mail to the ISS user. Each row of the Mail Data Table represent a complete message.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE
	VIEWED	BOOL	1	NONE	-MAIL
	FROM	CHAR	0	NONE	-MAIL
	TO	CHAR	0	NONE	-MAIL
	COPY_TO	CHAR	0	NONE	-MAIL
	DATE	CHAR	5	NONE	-MAIL
	SUBJECT	CHAR	0	NONE	_MAIL
	BODY	CHAR	0	TEXT_LINE FORM_LINE	-MAIL

Figure 3.21 - Schema Table Rows For Mail Data Table

The VIEWED attribute requires a "yes" or "no" data entry which is used to indicate whether the corresponding message has been read by the owner of the Mail Data Table. The FROM, TO, COPY_TO, DATE and SUBJECT attributes comprise the header of the message. The DATE attribute will provide the month and day that the message was originally created. The remainder of the header attributes are self-explanatory. All of the message header attributes can be used to form queries which operate on messages satisfying a given condition as posed in the queries.

The last attribute, BODY, contains the text of the message. This column will contain as much of the first line of the message as will fit one one line of varying length.

The remainder of the text will be retrievable by the user via a unique mail display operator.

If the length of the BODY column is insufficient to contain the full message, this fact will be denoted by the special characters, "=>", appearing at the end of the column as shown in row 1 of figure 3.22. Users also have the option of typing in the name and type of an ISS data file in the BODY column of the message. In this case, the name and type of file, preceded by a hyphen, "-", and followed by "=>" , will be displayed in the BODY attribute but no portion of the text of the file will be displayed. This type message is illustrated in row 3 of figure 3.22.

ID	VIEWED	FROM	TO	COPY_TO	DATE	SUBJECT	BODY
1	NO	SPOT	SAM	PETE	10-4	GAME	The Game =>
2	YES	DAN	MARY	LEE	10-3	DINNER	Cancel!
3	YES	IKE	TINA		10-1	DANCE	-Details.TXT =>

Figure 3.22 - Mail Data Table

In some cases the message body will be terse enough to fit, in its entirety, within the allotted columns. In these cases, the user will not be required to request further display modes for the message, and the special characters,

"=>", will not appear on the display. This type message is illustrated in row 2 of figure 3.22.

K. SPREADSHEET DATA TABLES

An electronic spreadsheet, in its traditional form, is a rectangular grid of rows and columns, like a large sheet of graph paper, to facilitate organizing numeric data for easy calculation and comprehension. It allows the user to model a variety of numerical problems in a standard tabular format on a computer with the elimination of the drudgery and errors encountered by using the scratchpad and pencil or hand calculator.

The schema for the columns of the Spreadsheet Data Table is illustrated in figure 3.23.

In the Electronic Spreadsheet application, the format of the Spreadsheet Data Table is quite different from the standard user's view of the spreadsheet. An example of a Spreadsheet Data Table is shown in figure 3.24. Figure 3.25 illustrates an example of the standard user's view of the Spreadsheet Data Table shown in figure 3.24.

In the view, the intersection of each row and column form one addressable unit or entry position. Each entry position in the view is described or defined by a row in the Spreadsheet Data Table. Although the ID is a key attribute in the Spreadsheet Data Table, it does not have any bearing on the order in which the entry positions in the view must appear in the data table. Blank entry positions in the view do not have to be described in the Spreadsheet Data Table. The X_COL attribute and Y_ROW attribute are the column position and row position, respectively, of a unique entry in the view. The FORMAT attribute of the Spreadsheet Data Table contains the information which determines how the data of the entry will be displayed in the view. This field contains three distinct types of data: the entry data type, right or left justification modes, and width of the column displayed in the view.

The entry data type can be of four types: character, integer,float, or monetary (dollars and cents number). To avoid inconsistencies and maintain readability, the user is prevented from declaring different column widths for entries in the same column. For example, if entry position A-1 has a column width of 15, then entry positions A-2 through A-n must have a column width of 15, even though the data types of the entry positions may differ.

The VALUE attribute contains either the display value of the FUNCTION attribute of the literal string for character entry types. The FUNCTION attribute contains either the keyword NONE, or an expression. The keyword, NONE, is used to indicate that the associated value of the entry is non-numerical and will be displayed in the view exactly as it was entered. The expressions in the FUNCTION column may be a numerical constant or an arithmetic expression. Numerical constants are entered by the user and appear in the view in the formats indicated by their data type. The arithmetic expressions may be composed of system built-in functions and operators or user defined functions or combinations of both. The operands of these expressions will be numerical constants and/or the values of other entry positions. Once defined, these arithmetic operations are automatically evaluated by the ISS and their values are displayed in the view. For instance, in the example, the data table entry for position B-5 is the function "B:2 * B:3". When the Spreadsheet Data Table is converted into the user's view, the value of the function is displayed, (in this case 12.00). If the user decides to change the value of `no. ordered' (B-2), or the value of `cost per item'(B-3), then the system will automatically reevaluate the function and immediately display the updated entry in the spreadsheet view in entry position B-5.

The user has the option of updating the Spreadsheet Data Table directly or indirectly via the view. However, if the view is chosen as a means for updating the data table, then only the non-calculatable entries can be updated, (i.e., literal strings and constant expressions).

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE
	X_COL	CHAR	1	NONE	-SPREAD
	Y_ROW	INT	3	NONE	-SPREAD
	FORMAT	CHAR	12	NONE	-SPREAD
	FUNCTION	CHAR	0	NONE	-SPREAD

Figure 3.23 - Schema Table Rows For Spreadsheet Data Table

ID	X_COL	Y_ROW	FORMAT	VALUE	FUNCTION
1	A	1	CHAR LEFT 15	********	NONE
2	A	2	CHAR LEFT 15	NO. ORDERED	NONE
3	A	3	CHAR LEFT 15	COST/ITEM	NONE
4	A	5	CHAR LEFT 15	TOTAL COST	NONE
5	B	1	CHAR RIGHT 10	STOCK1	NONE
6	B	2	INT RIGHT 10	3	3
7	B	3	MONEY RIGHT 10	4.00	4.00
8	B	5	MONEY RIGHT 10	12.00	B:2 * B:3
9	C	1	CHAR RIGHT 10	STOCK2	NONE

Figure 3.24 - Spread Sheet Data Table

	A	B	C
1	********	STOCK1	STOCK2
2	NO. ORDERED	3	
3	COST/ITEM	4.00	
4			
5	TOTAL COST	12.00	
6			

Figure 3.25 - Spread Sheet View

IV. CONCEPTUAL INTEGRATION

Conceptual integration of the five application areas of the Integrated Software System (ISS) has two components from a users perspective: a single data object, and a set of common operators which operate on those objects uniformly regardless of the application. In addition to these common operators, each application must provide its own set of specific operations to achieve its special functions. The single data object, the table, has already been discussed in chapter III.

The functional intersection of the five application areas is best understood by describing the common operations (as opposed to the actual primitive operators) which are required to manipulate data in and extract data from a file. The common operations can be performed by the eight basic ISS primitive operators, six of which are based on Relational Algebra, one which is an aggregate type of operator (sort) and one which is a proposed extension to Relational Algebra similar to Stonebraker's concatenation operator [Ref. 5] described in section B of chapter II. The primitive ISS operators are described in section A of this chapter and their mapping to the common operations and the utility of those operations are described in section B. The descriptions given are of the conceptual view of the operators and operations, not their form which is implementation detail. That is, their semantics and not syntax are described.

A special table called "ISSBLANK" is defined to be simply a blank row of any table. This convenience allows the primitive ISS operators to be set theoretic with both operands and results being tables.

The ID values in a table are the conceptual ordering of the relation in the system and thus is maintained by the system during operations on the table. Rows are always numbered contiguously from 1 to the number of tuples in the relation. Manipulation of the ID values in a table by a user is prohibited except indirectly via the re-ordering (and re-numbering) required after insertion and deletion of rows. New tables created by operators have their ordering imposed by the system as they are formed.

The effect of a primitive operator may be a change to the operand or the creation of a new table or both, depending on the operator. A created table can be assigned to (overwrite) an existing data table, which may or may not be one of the operands, or a new table. This is analogous to a conventional programming language variable assignment x := funct(x) except that the variables are now entire tables and the function is an ISS primitive operator. When operators are nested they operate on the result of the previous operator.

In addition to the primitive ISS operators provided for data manipulation in tables, there is a set of binary operators which operate between data tables. These complex ISS operators, which contribute to the extensibility of the ISS, may be applied to tables within and sometimes across application areas. In many cases, in view of the meaning of the structure of non-database data tables, inter-type operations are conceptually meaningless. The semantics of these operators and their utility is discussed in part C of this chapter.

A. PRIMITIVE ISS OPERATORS

The eight primitive ISS operators insert, modify, delete, project, select, union, sort and concatenate perform their functions on tables of any type in any application. Some operators are designed to change a target table while others are designed to extract data from a table creating, a new table for further use. The delete operator does both.

The modify, delete and select operators require a condition which specifies which rows of the data table are to be manipulated by the operator. The operands of the condition are literal or numeric constants, arithmetic expressions, column names or column values in the table. The literal expressions may contain pattern matching characters. The operators of the condition statement are the logical and arithmetic operators and the arithmetic relational operators, (less than, equal to, etc.). One common condition used will be specification of a line number or range by using the ID values. For example: (ID=10), or for a range: (ID>=3 AND ID<=14).

It should be mentioned here that although efficiency is not a concern of the ISS at this time, operations which cause rearrangement or re-ordering of rows in large data tables such as large Database Tables, will be very inefficient due to the amount of processing required.

1. Insert

Insert changes a target table by inserting into it a table at a specified location (ID value). If no ID is specified the new tuples are appended onto the end of the target table. The columns of the table to be inserted must contain a subset of those in the target table. The matching of column names is satisfied by equivalent attributes or synonymous attributes as defined in the Schema Table. Matching of synonyms is in the leftmost sense from both tables. Unmatched attributes in the target table have no value placed in the new rows. Effectively, the insert does row and column insertion, adding the extra structure required to the rows from the inserted table. The result of an insertion is the changed target table. Since the location of the insertion may be internal to the target table, the ID values of the inserted tuples and the subsequent tuples must be re-numbered by the system.

2. Modify

Modify alters a target table by changing the value of specified columns in a row or set of rows to new values. The rows to be modified are determined by their satisfying a given condition, all tuples which match the condition have their values changed. The new value to be placed in the rows chosen, may be specified absolutely or as a function of the old value to be replaced.

One special use of this operator is modification of the material under the curser during browsing or editing of a relation. The curser in this case has two values associated with it, the ID number (row) and the column name in which it displayed. Direct editing of the displayed old value becomes the new value. This use of the modify operator will be managed by the browsing software.

3. Delete

The delete operator changes a target table by deleting all rows which satisfy a given condition. In addition the delete operator creates a temporary new table of the same type and structure as the target table containing the deleted rows. It is the table of deleted rows which is returned by the operator when considering combined operators. The system manages the re-numbering of the ID column values in the target and deleted rows tables.

4. Project

The project operator creates a new table consisting of just those columns of the old table which are specified, in the order that they are specified. The appropriate columns of each row are copied into the new table, with rows in the same order as the original table. It is important to note that the resulting table will not in general be a table of the same type as the original table unless both are Database Tables, which have no specific schema like Text, Form, Mail and Spreadsheet Tables. (See chapter III for fixed schemas of non-database data tables.) In all cases except the trivial case of projection of an entire table, the projection will have a different structure than the operand table. For example the projection of a Mail Table on the FROM and TO columns gives a result with three columns (ID, FROM, TO) which is not a Mail Table or any of the other fixed schema type of tables.

5. Select

The select operator creates a new data table from all rows of a given table which satisfy a given condition. Its effect is similar to delete except the selected rows are not deleted from the given data table. The result of the operator is a newly created table of the same type and structure as the operand, however no assumption can be made about the order of the rows in the new table.

6. Union

Union operates on two data tables of possibly dissimilar types, producing a resultant data table. The two operand tables are of unequal status in that one, for the sake of a convention, will be referred to as the "left operand", dominates the other in that it determines the structure of the resultant table. The other, or "right" operand table of the union operator must have a set of columns (attribute names) which is a superset of the left operand. That is: each column name in the left operand (and hence result) must also be found in the right operand, or in other terms, an appropriate projection of the right operand could produce a table of the same structure as the left operand. Synonymous attribute names as defined in the Schema Table are considered equivalent for purposes of structure comparison. If the right operand table has a set of column names which contains two or more aliased names for a column in the resultant, the first column from the left will be used for the union.

The operator makes an identical copy of the dominant table and then appends to it copies of the shortened tuples from the other table. Like set union, a row from the second table is not added to the resultant if an equivalent row has already been included from the first table. If the resultant table is assigned to the left operand then union may be viewed as an appending of one tables rows onto the other.

Union is different from the insert operator in three major ways: it truncates the longer table, it performs theoretic set union on the set of rows and it always adds the right table to the end of the left table.

7. Sort

Sorting of a data table on the values in a specified set of columns creates a new table of the same type and structure with the values in the ID field corresponding to the increasing or decreasing sorted order of those columns. That is, the display order and conceptual ordering of the rows in the table are now sorted in increasing size of the values in the columns on which the sort was performed. If a column is numeric then the sort is done in numerical order, otherwise it is done in lexicographic order.

In some sense the sort operator is an aggregate operator in that it is the result of many basic manipulations which could be performed with much difficulty by using the other primitives.

8. Concatenate

Concatenation is a specialized operator which creates a new text data table from any other type of table. The values of specified columns of the operand table are concatenated into one single resultant table, row by row with each field in a row separated from the next by a space. The system provides the ID field for the resulting table.

B. REALIZATION OF ISS OPERATIONS

The desired operations on logical files of the ISS, regardless of the application, are easily realizable in terms of the primitive operators described above. These operations are applicable to all types of data tables (Text, Form, Mail, Spreadsheet and Database) and system tables (SCHEMA, ISSLOGIN, Directories and security tables), although the concern of this thesis is primarily the manipulation of data tables. The utility of the operations depend on the application, for example the moving of rows within a table is absolutely essential for Text Data Tables, but hardly useful at all in a database application. This section describes the operations in terms of the ISS primitives and their utility in the text, form, mail and spreadsheet applications. The usefulness of these operations on Database Data Tables depend entirely on the meaning of the tables. Since form creation is a text processing application, descriptions of operations on Text Data Tables always apply to Form Data Tables as well. The desired content of a form can be as general as any document.

It must be understood that when discussing the spreadsheet applications, there are two dimensions to the descriptions given. The underlying data table drives the presentation of the traditional view and rows and columns of the data table do not correspond to rows and columns of the view. Each row of the data table corresponds to a filled in position in the conventional view. The absence of a row in the data table for an entry position in the view simply means it is displayed as a blank. An empty Spreadsheet Data Table corresponds to a blank spreadsheet view. In the following descriptions an attempt is made to draw the distinction between view and data table manipulations.

It is not the intent to repeat the full analysis of possible uses for operators carried out by Nishimura [Ref. 1].

1. Removal Of Rows

Removal of rows in a data table is performed by applying the ISS delete operator with a condition to select the tuples for deletion. In a text or form application, removal of row corresponds to the deletion of a line of text, which may be selected by ID number or a pattern match on the lines of text. The table of deleted rows created by the delete operator can be used as part of an undo facility in the editing environment of the text application area.

Removal of a Mail Data Table row or set of rows, is simply the final disposal of a message or set of messages from a mail table. The table created by the delete operator may be used to transfer mail to another table, such as a dead letter box. The semantics of the delete operator requiring a condition to be given, allows for great flexibility in defining the set of messages to be deleted. Rows of a mail table may be deleted according to date of message, range of dates, whether viewed or not, range of date and whether viewed, individual names, pattern matches on sender and/or addressee names, subject pattern match and many other combinations given the mail table structure described in chapter III.

In the spreadsheet application, removal of a row or number of rows in a Spreadsheet Data Table corresponds to the blanking or setting to "empty" of an entry in the traditional spreadsheet view. Since the delete operator is specified with a condition on one or more of the Spreadsheet Data Table columns, data table rows and hence view, entries may be deleted according to a large range of criteria. This is in contrast to current traditional spreadsheets which can delete entries or "blank" by row, column or block only.

Deletion of a row or column in the spreadsheet view is somewhat more complex and must be done by a combination of operators. To remove a row in the view entries in that view row (Y_ROW = view row) must be removed (leaving a blank row), then all data table tuples for subsequent spreadsheet rows must have their Y_ROW values incremented by 1, forcing the view mechanism to now display them one row higher. A similar procedure is required for columns in the view.

2. Location Of Rows

The location of a particular row or a set of rows of a data table is simply an application of the select operator described in subsection B.5 with the desired condition imposed.

This operation may be used to display particular rows of a data table or to browse through a file. Browsing is affected simply by locating contiguous blocks of ID values. This type of browsing would be particularly useful for Text and Form Data Tables where the ordering of rows is meaningful. Using pattern matching, conditions this operator allows pattern search within a document.

Similarly, Mail Tables can be browsed using the select primitive operator, allowing a freedom to produce message summaries based on any criteria rather than being restricted by a predefined message summary.

In spreadsheet applications, with the traditional spreadsheet view in mind and an understanding of the underlying model (a Spreadsheet Data Table as described in section III), a single entry position can be located by selecting the row with the condition that X_COL and Y_ROW match the desired entry. In addition, the location of a set of rows in the data table correspond to search actions throughout an entire spreadsheet view. Traditional spreadsheet views generally reference only one entry position at a time whereas the ISS primitives select allows an entire set of entry positions to be located in a single operation. A set of rows (entry positions in the view) may be selected according to their position, function, value or even their format.

3. Addition Of Rows

Tuples may be added into data tables by applying the ISS primitive operator insert. The BLANK table may be inserted in the case of new tuples, or some other table for simple additions of existing tuples from another table. An empty row inserted would need subsequent modification to fill out its values. Appending rows to a table is simply insertion at the last ID. Insertion may also be done by using the ISS primitive operator union and assigning the result to the required table. The union operator will add rows onto the end of the table, adding only those rows from the other table which are not already in the resultant table. Generally insert will be more useful in the applications which deal with tables where row order is important (text and forms), and union in the others.

The general utility to be able to add rows to a table needs hardly be mentioned except to describe the effect in each application area. A Text or Form Table during creation or modification requires a facility to insert new lines at specified locations, followed perhaps by modification. During form processing the forms mechanism must dynamically insert entire tables where indicated, allowing standard blocks of text to be embedded in each of the output documents. Any of the fixed data table types may be inserted into a Database Data Table, providing new data tuples with some of the column values already entered.

In the mail application addition of a new row in a data table corresponds to creation of a new message. Addition via an insert or union command of a number of rows selected from another Mail Table would get copies of some other user's messages. Messages left in the care of another user's Mail Table could be collected by inserting the result of a deletion from that table (delete those for which addressee is the eventual owner) into the addressees mail table. A union of Mail Tables would create a list of unique messages received by a set of users.

In the spreadsheet application the addition of a row in the data table corresponds to the filling in of an entry position in the view. When using union and insert with Spreadsheet Data Tables care must be taken to ensure view entry positions are not duplicated, that is the X_COL and Y_ROW values together form a unique key. One of the spreadsheet application area specific functions will need to be the management of view row and column insertion procedures, manipulating the Spreadsheet Data Table to effect the changes required in the view. Insertion of a (blank) row or column in a view is done by modifying the

Spreadsheet Data Table and was discussed in the previous section.

4. Updating Of Rows

Updating of data table rows is performed by applying the ISS primitive operator modify with the appropriate condition to select the desired rows. In this way, data table values may be changed to new values explicitly given. The insertion of the blank table and subsequent updating completes the creation of new rows in a table.

Updating of text and form lines is clearly useful as this is merely part of the ordinary editing function. Global changes can be effected using pattern matching in the condition for modification. For normal text editing and the filling in of blank rows of other data tables, this raw use of the ISS primitive modify operator is very clumsy, and a control program such as mentioned in subsection A.2 to browse and edit is required. For example the deletion of a sentence spanning a number of lines would require updating of the lines where the sentence begins and ends and deletion of any intervening lines.

Updating a Mail Data Table is necessary after the BLANK insertion, to fill in the details of the message to be composed or to modify a message before "transmission". (Adding the message row to the addressees' Mail Tables.) Modification of the subject and message body of one Mail Data Table row and "retransmission" could ensure identical addressees for a sequence of messages, a function normally handled by routing lists. Similarly just the date and addressees could be altered to send the message to an amended distribution list. The mail application specific software could provide simple procedures to assist the user to effect these transactions.

Updating of Spreadsheet Data Table rows is essential to the use of these tables as a modeling tool. By thoughtful use of the conditions in the modify operator, a powerful set of changes can be realized.

Entries in the view are placed in another location by simply modifying the values of the X_COL and Y_ROW in the appropriate data table row, to the desired row and column. The values in entire rows in the view may be moved to another row by modifying Y_ROW in the data table tuples to the new value. That is: modify rows in which Y_ROW = R, changing the Y_ROW column value to R + 1. When the modified data table is mapped to the view all the entries which were in row R will be mapped instead to row R+1.

Formats for display in the view are changed easily by updating the format string in the FORMAT column of the data table. The format can be changed by single entry, by view rows or columns, globally or by any composite condition required.

Although some adjustment is required to think of a spreadsheet in terms of the ISS Spreadsheet Data Table, once accomplished, both the traditional view and the underlying conceptual model can be used to great advantage. For the model to be useful, of course, the spreadsheet application must have application specific operations to force spreadsheet related functions such as: recalculation of derived values after user changes to the entry position values; change of entry function expressions involving moved operands; increment or decrement X_COL and Y_ROW values when rows in the view have been moved, deleted or inserted; and, check on inconsistencies in the model such as duplicated X_COL,Y_ROW pairs.

5. Copying Of Rows

The copying of rows within a data table may be performed using the ISS primitive operator select followed by insert. Selection of the rows to be copied creates a new table which then may be inserted at the desired location. Similarly, copies of tuples from one data table may be copied into another table of the same structure by using a select followed by a union onto the desired table. Generally it would be preferable to use the union operator in non-text or non-form applications where order is unimportant and duplicated rows are undesirable.

The utility of copying lines in a text document or form is self evident, as it is a standard document preparation feature. Electronic mail and spreadsheet applications have little use for the simple copying of tuples per se, the utility being in the subsequent modification of the duplicated row.

The mail examples given in section 4 could be preceded by a copy operation, thus keeping a complete record of all messages or versions of a message sent to a group in the first example, and keeping a complete record of dispatch dates and addresses in the second.

In the spreadsheet application the copy operation would be useful more in filling out repetitive rows of the Spreadsheet Data Table, subsequently modifying the necessary columns.

6. Moving Of Rows

In the ISS, tuples may be moved within a data table by applying the delete operator to the desired rows identified by ID number or other condition, followed by an insert of the deleted rows at the required location. It is of benefit only in situations where ordering of rows is important.

In the text processing and form generation application areas, this operation is a generalization of the normal move operation allowed by editors. By the use of the condition in the delete operator it enables the moving of non-contiguous lines selected on content alone without the need to know line numbers.

In spreadsheet and mail applications the move operation is not particularly useful as the ordering of the rows in these data tables has little meaning. The operation may be performed to satisfy personal preference when browsing the entire Mail or Spreadsheet Table.

7. Sorting Of Table Rows

Sorting is done in the ISS by the operator sort which can be thought of as a series of selective moves to a new data table. It is useful to some extent in all application areas for producing hard copy listings of all rows for manual reference and personal preference when browsing a file. Although not discussed in this thesis, sorting a file on a key field will have considerable impact on retrieval efficiency in an enhanced system which is concerned with storage and retrieval efficiency.

In a text application a textual list could be deleted, sorted and re-inserted in its original location. A series of numbered one line points in a document or form may be sorted after haphazard insertion.

Mail Data Tables may be sorted according to the users desires. For example a user may like to regularly browse his mail listings sorted by addressee, sender or date. The data tables could be sorted and saved in that order or into a temporary file for display purposes only.

Spreadsheet Data Tables may be sorted on X_COL and Y_ROW values to maintain a listing the user is comfortable with. It is possible that users may become comfortable dealing with a Spreadsheet Data Table directly, bypassing the view during data entry and modification.

8. Merging Of Columns

The requirement to merge columns is essentially so that columns which are conceptually different fields may be merged into a single text field for display, or insertion into a document. The operation is performed using the ISS concatenate operator.

Uses of the column merge are to include part or all of some data table in a text document or form. For example, a letter detailing names and addresses of customers with overdue payments in an accounts database may be drafted. After drafting, a concatenated projection of the name and address fields of the rows selected from a customer database table on the condition of *negative balance*, can be inserted into the letter. Similar examples can easily be concocted for the use of merging columns in Spreadsheet and mail Tables.

9. Operator Combinations

The basic ISS operators alone and simply combined provide low level powerful data manipulation tools to effect the operations described above. Combinations of operators can provide much of the high level activities required by a user. Many of the examples in the preceding sections are ones which use two or more primitive operators to achieve their purpose.

One particularly important combination of operators is that used to modify the structure of existing Database Tables (the others are fixed) without the re-entry of existing data. This may be done by first using the ISS primitive insert to create a new Database Table by inserting a temporary name in the database directory and giving it the required columns. The old table may then be inserted into or unioned with the new, producing a result which may then be assigned to the new table. Insertion would be used when the new structure is a superset of the old and union when the new structure is a subset of the old, since insert will add columns when necessary and union will remove them. The old table may then be removed from the directory with delete operator, and the new table renamed to the old with the modify operator.

It is the function of the separate application areas to provide the mapping between the high level application specific operations and the low level primitives. Utilities in the kernel should provide convenience tasks such as the relations browser previously mentioned and multiple data entry routines, to repetitively apply the primitive operators. Utilities should also provide "walkthrough paths" for table creation where the system requires more to be done than just simple addition of a row to a Directory Table. This applies particularly in the database application where new attributes may have to be defined in the Schema Table and, where the full security provisions alluded to in previous chapters is applied. Although these functions are to be performed by the applications specific and utility software, the user can still retain the underlying Relational Database model in his thinking and understanding of the system, and is always able to revert to the primitive operators for data manipulation and retrieval regardless of the application area in which he is working.

C. EXTENSIBILITY OF ISS

The main thrust of the ISS is that it provides the capability to combine tables of different types to form new relationships and derive new information. This extension of the ISS is made possible by incorporating into the system some high level combining operators. Those chosen for this system are Union, Set Difference, Intersection, Join, and Natural Join. Using these operators on tables of the same or different types adds much depth to the scope of the ISS.

Haphazard use of these operators can result in a syntactic error or a new table which is semantically meaningless. For example, to apply an operator which requires two tables of the same structure to tables of different structure would be syntactically incorrect. Moreover, some combinations like a join of a Mail and Spreadsheet tables is syntactically correct but has no foreseeable meaning given the nature and design of the tables. If an attempt is made to apply an operator incorrectly, the ISS sets a trap which prohibits the user from performing the operation and displays an error message. In the cases where the application of the operator is deemed only semantically meaningless, the ISS trap will issue a warning to the user. If the warning trap is invoked, then the user has the option of aborting the operation or to continue executing it.

In the following parts of this section each of the combining operators are defined and described in general terms, examples are presented which illustrate how using the operators on tables of different types can result in some meaningful relationships, and in cases where combining tables using a given operator is meaningless or incorrect, the traps are discussed.

The semantic meaningfulness of combining tables of the same type using any of the operators discussed in this section is considered to be obvious and will not be discussed further in this thesis. An elaboration on this type table combinations (Intra-Type Combinations) can be found in [Ref. 1].

1. Union

The union operator has been described previously in subsection 4.B.6 of this thesis. Conventionally, the union operator requires tables to be of the same arity. However, the ISS union operator embeds a default projection (the mechanics of which have been discussed) which lifts the arity constraint from tables. It should be noted that the user can effectively override the default projection by first proceeding a union of two tables by a projection operator which projects out attributes of the user's choice from one or both of the tables. The user could then apply the union operator on the resultant table(s). Thus, the extension of the meaning of the conventional union operator is not a limiting factor, but to the contrary, it enables a wide variety of table combinations and new relationships to be achieved.

The following subsections illustrate how performing a union on tables of different types can be semantically meaningful. In the discussions the "+" is used to denote the union operator.

a. Text + Form

If A is a Text Data Table which contains the body of a letter which is to be sent to all employees of a given company, and B is a Form Data Table which extracts the name and addresses of all the employees of the company and places in letter head format, including a salutation, then performing the operation, A + B, would be very useful. It would yield a repetitive letter addressed to each employee and the resultant table would be a Form Table.

As another example, if A is a Form Data Table which contains intricate tax details for residents of a given state and B is a Text Data Table which contains a textual description or explanation of some of the columns on the tax form, then, B + A would yield a new relationship which contained both the explanation of the Form columns and the Form itself. The result would be a Text Data Table.

Since the FORM_LINE and TEXT_LINE can be aliased, and the two tables are of the same arity, no projection would be necessary. This is essentially the same as an Append.

b. Text + Database

If A is a Text Data Table and contains the names of the local stores which sold shoes and B is a Database Data Table which contains as one of its attributes the names of local stores which sold clothes, then A + B would be meaningful. First, ISS would project out all non-applicable attributes of table B. Then it would yield a Text Data Table which contained a list of stores that sold either shoes, clothes, or both. ISS would eliminate any duplicate tuples in table B.

If the order is reversed, i.e., B + A, then this is only meaningful if table B contains only two attributes, ID and STORES(or any column name which could be aliased with TEXT_LINE). Applications for tables of these types are numerous.

c. Text + Mail

This operation is semantically meaningful in only one direction. If A is a Text Data Table and B is a Mail Data Table, then the only feasible union is A + B. This would yield a resultant Text Data Table whose tuple contained the TEXT_LINEs of table A and the BODY of table B.

d. Form + Database

This combination has application similar to those described for TEXT + Database since the Form Data Table is essentially a "special" Text Data Table. Therefore, they both have the same semantic meaningfulness.

e. Form + Mail

This operation is syntactically correct, and thus semantically meaningful, in only one direction. For instance, if A is a Form Table and B is a Mail Table, then A + B is syntactically correct.

One application using this operator would be to have the BODY of a message in table B disseminated to all employees in letter form. To effect this, one could union table A, a form which yields the letter heading for each employee, to table B.

f. Spreadsheet + Database

This operation is only meaningful if the attributes of the Database table correspond to the schema of the Spreadsheet Data Table. If so, then this union is meaningful in both directions and virtually would be the same as unioning two Spreadsheet Tables or two Database Tables with identical attributes. It is recognized that this application would be very rare.

g. Database + Mail

If A is a Database Data Table which contains a subset of the attributes of a Mail Data Table then this application is semantically meaningful and syntactically correct. A + B would then result in a Database Data Table with tuples from the Mail Data Table (after necessary projections) or from the Database Table or both. In the other direction, i.e. B + A, this operation would only be possible if all the attributes of the Mail Data Table also appeared in the Database Data Table.

The following combinations of tables using the union operator are semantically meaningless or syntactically incorrect.

h. Text + Spreadsheet

If A is a text data and B is a Spreadsheet Data Table, then A + B and B + A are both syntactically incorrect. Neither order meets the requirements that the attributes of the right operand be a superset of the attributes of the left operand. Since this operation is incorrect, it will be prohibited by setting a "prohibit" trap condition.

i. Mail + Text

This is syntactically incorrect in one direction only. If A is a Mail Data Table and B is a Text Data Table, then A + B is incorrect due to the fact that both tables have predefined attributes and the text table can not be a superset of a Mail Table. This operation will be prohibited.

j. Form + Spreadsheet

This operation is essentially the same as that discussed in the Text + Spreadsheet operation.

k. Mail + Form

This operation is syntactically incorrect in one direction only. Due to their similarity, the discussion presented in the Mail + Text operation applies here as well.

l. Spreadsheet + Mail

This is not meaningful nor syntactically correct in either direction due to failure to meet the superset constraints. This operation will be prohibited.

2. Set Difference

Given two data tables, A and B, set difference is defined as the set of tuples in A that are not in B. This operator requires that tables A and B be of the same arity.

The following parts of this section illustrate how performing the set difference operator on data tables of different types can be semantically meaningful. The "-" is used in the discussions to denote set difference. The resultant table of each operation below will be of the same type as the table used as the left operand.

a. Text - Form

If A is a Text Data Table which contains the literal text of a corresponding form in a Form Data Table, B, then A - B or B - A would be very meaningful. For instance, if there are many tuples in tables A and B and table A has been slightly modified, then performing the operation, A - B or B - A, provides a quick and easy way of determining which tuples of the Text Table were modified without the user having to do a line by line comparison of the text.

b. Text - Database

If A is a Text Data Table and B is a Database Data Table, then A - B is meaningful only if table B contains exactly two attributes, the ID and another attribute , "b", of data type, character. In this case, set difference would yield the tuples in A that are not in B. Presumably, the TEXT_LINE attribute of table A and attribute "b" would range over the same domain.

c. Spreadsheet - Database

This operation is only meaningful in applications of a limited nature. For instance, if A is a Spreadsheet Data Table and B is a Database Data Table, then A - B is meaningful only in the cases where the attributes of table B are synonyms or aliases of those in table A. If table B meets those specifications then it could contain a subset of a spreadsheet defined and maintained in table A. Using set difference on these tables will yield entry positions which are defined in A but not in B, or it will reveal discrepancies in entry positions that are defined in both tables. For example, assume entry position G-1 was defined in both tables but the function attribute contained a value of " 10 * G:2 " in one table and a value of " 1000 * G:2 " in the other. Applying the set difference operator to these tables would help to maintain consistency within the tables.

d. Database - Mail

As with the Spreadsheet - Database operation, Database - Mail operation is only meaningful in the special cases where the attributes of the *Database Table are aliases* of the predefined attributes of the Mail Data Table. If B is a Database Data Table which meets those conditions and A is a Mail Data Table, then A - B would be useful in many applications, and would be essentially the same as performing the set difference operator on two Mail Data Tables.

The following combinations of tables using the set difference operator are either semantically meaningless, or syntactically incorrect.

e. Text - Spreadsheet

The Text and Spreadsheet Data Tables do not have the same arity and there is no correspondence, in general, among their attributes. Thus, this operation is both meaningless and syntactically incorrect and will generate a "prohibit trap".

f. Text - Mail

The same argument applies for performing set difference on the Text and Mail Data Tables as given in the Text - Spreadsheet discussion above.

g. Form - Spreadsheet

The same argument applies for performing set difference on the Form and Spreadsheet Data Tables as given in the Text - Spreadsheet discussion above.

h. Form - Mail

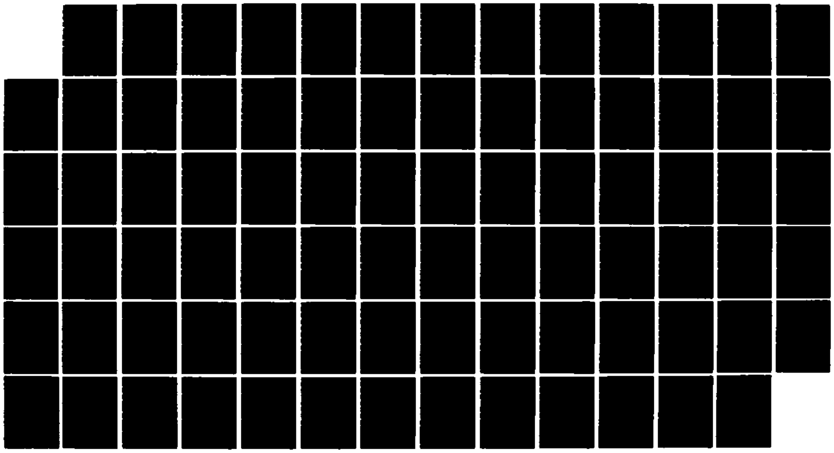
The same argument applies for performing set difference on the Form and Mail Data Tables as given in the Text - Spreadsheet discussion above.

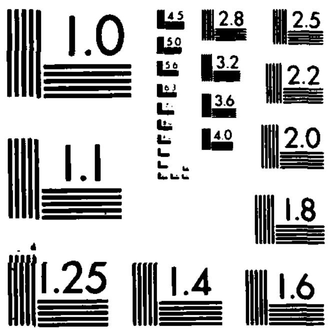
i. Text - Database

Applying set difference operator to a Text and Database Table is syntactically incorrect only if the arity of the Database Table is not the same as that of the Text table or the Database table attributes are not synonyms of those in the Text table. In these cases, the Text - Database operation will set a "prohibit" trap.

AD-A140 628 DESIGN OF AN INTEGRATED SOFTWARE SYSTEM BASED ON THE RELATIONAL DATA BASE MODEL(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P J HARRISON ET AL. DEC 83 2/2

UNCLASSIFIED F/G 5/1 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

j. Form - Mail

The same argument applies for performing set difference on the Form and Database Data Tables as given in the Text - Database discussion above.

k. Spreadsheet - Mail

The same argument applies for performing set difference on the Spreadsheet and Mail Data Tables as given in the Text - Spreadsheet discussion above.

l. Database - Mail

The same argument applies for performing set difference on the Database and Mail Data Tables as given in the Text - Database discussion above.

3. <u>Intersection</u>

Given two data tables, A and B, the intersection of A and B is the set of all tuples that are in both table A and table B. Both tables used as operands of the intersection operator must be of the same arity. In addition, the corresponding columns or attributes of the tables must be aliases of each other.

The following parts of this section illustrates how performing the intersection operator on data tables of different types can be semantically meaningful. The "&" is used in the discussions to denote intersection. The resultant table of each operation below will be of the same type as the table used as the left operand.

a. Text & Form

If A is a Text Data Table and B is a Form Data Table, then the operation, A & B, would be useful in applications to determine the commonality between the text in table A and the literal text in the form in table B.

b. Text & Database

If A is a Text Data Table and B is a Database Data Table, then as with the set difference operator, A & B is meaningful only if table B contains only two attributes, the ID and another attribute , "b", which can be aliased with the TEXT_LINE attribute of the Text Data Table. In this case, intersection would be meaningful in many operations. For instance, assume table A contained a list of students taking Trigonometry and table B contained in attribute "b" a set of all students taking Chemistry. Then, A & B would yield a set of tuples corresponding to the names of students enrolled in both courses.

c. Spreadsheet & Database

As with the set difference operator, the intersection of a Spreadsheet Data Table, A, and a Database Data Table , B, is only meaningful in limited applications and in the very specialized case where each of the Database table attributes is an alias for one and only one Spreadsheet table attribute. If table B meets these constraints then, A & B would be essentially the same as performing the intersection of two Spreadsheet Data Tables or two Database Data Tables with identical attributes.

d. Database & Mail

As with the Spreadsheet & Database operation, Database & Mail operation is only meaningful in the special cases where the attributes of the Database table are aliases of the predefined attributes of the Mail Data Table. If table A is a Database Data Table which meets those conditions and table B is a Mail Data Table, then A & B would be useful in many applications, and would be essentially the same as performing the intersection of two Mail data tables.

Using _he intersection operator is either syntactically meaningless or semantically incorrect on the same combinations of tables as using the set difference operator. For that reason and to avoid redundancy or duplication of the rationale, the reader is referred to subsection 4.C.2. (the discussion on the semantic meaningless and syntactically incorrect uses of the set difference operator on certain table combinations).

4. Join

A join of two data tables, A and B, is defined only if the two tables each have some attribute (column) which is defined over some common domain and the join is over those two corresponding columns. The result of the join would be a new table of the type database, which contains tuples qualified by the attributes of both tables. In other words, the structure of the resultant table would contain all the attributes of table A, followed by all the attributes of table B. Each row in the new table would be formed by adjoining tuples of table A with tuples of table B such that the new tuples in the resultant table satisfy the predicate (condition of the join) in the common columns.

In the ISS we expect that most joins will be formed by an equality predicate, i.e. a condition of "equality" between the values in the common attributes of table A and table B. This type join is called an equijoin. When the equijoin operator is used, the duplicate column is not eliminated, therefore, this operator can result in redundancy in the database unless followed by a projection operator to eliminate the common values. The ISS requires the common attributes to be identified in the Schema Table as aliases or synonyms of each other.

The semantic meaningfulness of combining tables of different types using the join operator is discussed below. There is a significant difference between the applications discused in Nishimura's analysis, [ref. 1], and those presented here. In that reference the applications described are based on the assumption that the user will be able to use the ID attribute of a given table in performing binary operations with the operators. Since that assumption is not made in this thesis, the applications presented by Nishimura are not valid herein.

The symbol "*" will be used in the discussions to denote the join operator.

a. Text * Mail

The join of a Text Table, A, and a Mail Table, B, is very practical. If the TEXT_LINE ranges over the same domain as the FROM, TO, SUBJECT, COPY_TO,VIEWED, and DATE attributes of the Mail Table, then A * B has many applications. As an example of its applicability, suppose table A contained a list of subjects considered high priority and assume each TEXT_LINE of table A contained only one of those subjects. Then, the equijoin of A * B would form a new relationship which groups messages in table B which are of the same subject and which are also considered high priority. Similar applications can be done using the other type joins and/or Mail table attributes.

b. Form * Mail

This is similar to the applications of Text * Mail. However, in most cases the Form Table will contain some FORM_LINES which are not actually values in the domain of the corresponding Mail Table attribute. For instance, using the same example as used in the Text * Mail application, more than likely some of the Form_LINES will have data that is not equivalent to a Subject in the Mail System.

c. Spreadsheet * Database

If table A is a Spreadsheet Data Table and table B is a Database Data Table, then the meaningfulness of performing A * B is dependent upon the context of table B. As an example, assume that table B contains the attributes, ID, X_COL, Y_ROW, and DESCRIPTION. Then, the equijoin of B * A would provide the user with a table that contains the subset of the spreadsheet whose entry positions were described in table B. The new relationship would give the user a clearer understanding of the entries in the original Spreadsheet Table.

It should be noted that this application can never have an effect on the spreadsheet view since the resultant table will always be a Database Data Table, not a Spreadsheet Data Table. Performing operations on Spreadsheet views are not discussed in this thesis.

d. Database * Mail

On page 68 of [Ref. 1], Nishimura presented a very detailed analysis of the applicability of joining database and Mail Data Tables. These operations will not be repeated here, but suffice it to say that with the exception of the join created using the ID attributes, the examples presented therein are considered feasible.

It should be pointed out though that the result of applying a join to a Database Table and a Mail Table would be a Database Data Table which contains all of the attributes of both tables. The resultant table, by no means, is ever a mail data table as alluded to by the examples. Further processing would be necessary to convert the resultant table into a Mail Data Table which conforms to the ISS Mail Data Table format.

The following subsections briefly discuss the combinations of tables that are viewed as semantically meaningless or syntactically incorrect using the join operator.

e. Text * Form

The join of a Text Table, A, and a Form Table B can be performed syntactically on the TEXT_LINE and FORM_LINE attributes of the corresponding tables. However, this operation is viewed as having little significance or usefulness. The join (equijoin), A * B would result in the intersection of the two tables that would just be two copies of the same text and would not contribute to any useful new relationship among the tuples of the resultant table. Furthermore, joins on these two attributes will usually result in TEXT_LINES or FORM_LINES which are much too long to be displayed or printed on the same line. In light of this observation, this operation would tend to be more annoying to the user than helpful to him. Any join involving this combination of table types would invoke a "warning" trap.

f. Text * Spreadsheet

The join of a Spreadsheet Table and a Text Table is syntactically incorrect. None of the attributes of the two tables are aliases of each other and rarely do they range over the same domain. This application would invoke a "prohibit" trap.

g. Text * Database

The join of a Text Table, A, and a Database Table, B, can only be performed on the TEXT_LINE attributes of table A and some attribute "b" of Table B. Joins of this type seem to serve little purpose in as much as no new relationship results, nor is there any outstanding inference made from the join. Since the join is done on a tuple basis, at best if the Text Table has only one TEXT_LINE it can be joined through a "not equal join" to produce a new relationship, but not a very useful one. Since there are so few realizable applications for this join, it too will invoke a "warning" trap.

h. Form * Spreadsheet

The join of a Form and Spreadsheet Data Table is similar to the Text * Spreadsheet application discussed above.

i. Form * Database

This combination will invoke a warning trap. See the Text * Database discussion for the rationale.

j. Spreadsheet * Mail

A join on tables of this type is syntactically incorrect. The attributes of the Spreadsheet Data Table in no way have anything in common with the attributes of the Mail Table. The schemas for both tables are predefined and there has been no provision to alias any of the attributes. This, then, supports the argument for the infeasibility of combining these type tables through a join.

In addition to syntactic infeasibility, this operation does not result in a useful new relationship since the semantics of the attributes of the spreadsheet and Mail Data Tables have nothing in common. Since this particular join is syntactically incorrect, it will set the "prohibit" trap.

5. Natural Join

The natural join of two data tables is a special case of the equijoin which was discussed in section E.4 above. Simplistically stated, the natural join is an equijoin followed by a projection. The result of this operator contains no duplication since the common column is eliminated by the projection.

The semantic meaningfulness of using the natural join to combine tables has been discussed implicitly in the join section with each reference to the equijoin, therefore it will not be elaborated on further.

6. Operators Summary

Figure 4.1 summarizes the information presented in this section. The matrix depicts the type trap that will be generated by the system when the user tries to use operators to combine tables whose combination would be either semantically meaningless or syntactically incorrect. If the table combination is only semantically meaningless then the "warning" trap is generated and a message advising the user of its potentially meaningless results is displayed. The warning trap may also be generated when the operation performed on the two tables is deemed to have only a limited number of cases where it can be syntactically correct. If using an operator to combine two tables is syntactically incorrect in all cases, then this operation generates a "prohibit" trap and a message is displayed to the user which informs him that the transaction has been prohibited.

In figure 4.1, the small letters denote the operators, i.e. "u" = UNION, "i" = Intersection, "s" = SET DIFFERENCE, "j" = JOIN, and "n" = NATURAL JOIN. If an operator does not appear in an entry in the matrix then this implies that using that operator to combine the indicated table types is both syntactically correct and semantically meaningful. The column and row headings indicate the types of the data tables used as operands.

	TEXT	FORM	MAIL	DBASE	SPREAD
TEXT		WARNING j n	PROHIBIT i s	WARNING i s j n	PROHIBIT u i s j n
FORM	WARNING j n		PROHIBIT i s	WARNING i s j n	PROHIBIT u i s j n
MAIL	PROHIBIT u i s	PROHIBIT u i s		WARNING u i s	PROHIBIT u i s j n
DBASE	WARNING u i s j n	WARNING u i s j n	WARNING u i s		WARNING u i s
SPRD	PROHIBIT u i s j n	PROHIBIT u i s j n	PROHIBIT u i s j n	WARNING u i s	

Figure 4.1 - Operators Usability

V. USER INTERFACE

This chapter focuses on the conceptual design of the user interface to the Integrated Software System. For any software system such as the ISS, it is paramount that the user interface is well designed. Two important design issues to be considered are the ease of use and simplicity. In other words, the system should be constructed in such a way that the user has only a few things to learn in order to use the system effectively. These thoughts were the driving forces behind the design of Graphics Prototype Interface (GPI), which is the ISS Data Manipulation Language (DML). GPI and its major features are thoroughly discussed in section A of this chapter.

Since the thrust of this thesis is the design of the integration of the selected applications, little mention is made of application specific user interfaces. However, some consideration was given to the issue of the user making transitions from ISS integrated modes to application unique modes and vice versa. This is discussed briefly in the GPI Command Mode commands section. Further application specific user interface design details are considered to be outside the scope of this thesis.

One other area that is not addressed in the user interface is access rights and authorization. These issues add another dimension to the User Interface design structure and are highly important. Their absence from this thesis is not an omission, nonetheless, since they are viewed as matters of system security which are not implemented herein. The framework for providing the user interface to effect security is, however, embedded within this chapter.

Section B of this chapter presents a brief overview of the implementation of the conceptual design of the user interface to the ISS.

A. GRAPHICS PROTOTYPE INTERFACE

Graphics Prototype Interface (GPI) is the Data Manipulation Language designed for ISS. It adopts some of the constructs of Query By Example (QBE), a domain calculus DML. Both GPI and QBE are designed for interactive terminal use in composing queries and provide graphic table displays to formulate user transaction requests and system responses. GPI is primarily a tuple based language, however, it also incorporates features from both relational algebra and domain calculus as well. In addition, it contains some additional features (such as insertion, deletion, modifications, arithmetic capabilities, printing capabilities, etc.) which are not part of the algebra or calculus but serve to increase the power and usefulness of the language.

The constructs or commands of GPI can be divided logically into four parts. First there are the High Level System commands which are used to control cursor movement and browsing of displays and to change the mode of the system. Second, there are Command Mode commands which perform functions such as "RUN" and can be issued only when the system is in the command mode. Third, there are Query Mode commands which can be issued only when the system is in the query mode, for example the "UNION" command. And fourth, there are Help Mode commands which can be issued only when the system is in the help mode. The commands which are specific to the Help Mode are not addressed in this version of GPI.

These command categories,(with the exception of the Help Mode commands), and their usage are discussed below. Also, a discussion on the general layout of the CRT display for queries is presented.

1. General CRT Layout

The general structure or layout of the CRT is depicted in figure 5.1. The figure illustrates the maximum amount of information that can be displayed on the screen at one time when the system is in the query mode (to be discussed later). This layout is based on the assumption that the viewing area is a 24 lines, 80 columns CRT screen. In order to prevent overcrowding the screen and baffling the user with too much information displayed at once, some necessary limitations are embedded in the GPI query display mechanism.

As shown, the screen is divided into four distinct areas: two data table skeleton areas, a condition box area, and a command/response area. For readability, each is separated by two blank lines. The different types of areas are discussed below.

a. Table Skeleton Areas

The table skeleton areas are reserved at all times during the query mode. Each table skeleton occupies exactly seven lines of screen area. The name and type of table will appear on the first line of the table area. Line two will be blank and on line three will be the data table attribute names, preceded by the special column heading, "COMMAND". The COMMAND column is used by the user in order to specify what command is to be performed on the selected tuples of the table. To distinguish this column heading from one of the attributes, it will be highlighted.

Lines five through seven of each table skeleton area are referred to as the query lines and are reserved for user query entries. The use of the query lines to formulate queries will become clear in later discussions.

GPI provides a mechanism for the user to change table skeletons. This display facility is described in detail in section 3.

b. Condition Box Area

The concept of a condition box was adopted from the QBE language. The condition box area is used to define conditions placed on attribute variables which are used in the query. It follows the second table skeleton area and occupies three lines on the CRT. The heading, "CONDITION BOX ", is displayed on the left side of the condition box area. The box area to the right side of the heading is reserved for the user to specify conditions placed on variables which represent attributes from the data tables. For example, if "X" is a variable name used in the query line of a table, a further condition can be on the value of the attribute which "X" represents by placing a statement such as " ?X > 10 " in the condition box (note that the question mark preceding the variable "X" in this example is part of the syntax and is not a part of the actual variable name). This statement would inform the system that the user wants to include in the query only those tuples which have a value greater than 10 in the attribute corresponding to variable "X".

Several unrelated conditions may be placed on the same line of the condition box, however they must be separated by a ";" character. The conventional arithmetic symbols are valid in writing GPI arithmetic statements, (i.e. +, -, *, /, (,)). In addition, the comparison operators, (such as >, <, >=, <=) can be used. The logical operator symbols used are: "&&" (and), "||" (or), and "~" (not).

c. Command/Response Area

The Command/Response area occupies only two lines. It is used by the system and by the user. When the user is operating in the command mode, all Command Mode commands ,i.e, RUN, CREATE, etc, must be entered in the Command/Response area. In addition, once a user runs a transaction the system issues a response in this area to inform the user immediately as to where the results of the transaction can be found or if further actions are necessary to complete the transaction. If a trap has occurred in an attempt to execute a transaction, that information is displayed in this area as well.

2. High Level System Commands

The High Level System commands are of two types; one type is for cursor control and the other type is for changing the mode of the system. The cursor control commands are effected by special keyboard keys and their function is to allow the user to easily browse or scan the display. These commands also enable the user to edit data entered on the table skeletons or condition boxes with ease and they can be used regardless to which mode the system is in (GPI operates in three distinct system modes: Command, Query, and Help. Each of the modes allow mode specific operations to be performed).

Each High Level System command is defined below. The first six are cursor control commands and the next three are mode change commands. Each of the Mode change commands are preceded by the symbol "^" which represents the control keyboard key. This is to indicate that the control key must be pressed simultaneous to the corresponding mode character in order to activate all mode change commands.

a. ^ - move the cursor up one line from current location

b. v - move the cursor down one line from current location

c. > - move the cursor one character position to the right

d. < - move the cursor one character position to the left

e. tab key - move the cursor to the beginning of the next field

f. carriage return - move the cursor to the beginning of the first field in the next line

g. ^C - change the system to the command mode

h. ^Q - change the system to the query mode

i. ^H - change the system to the help mode

In the next two sections the specific commands which can be run only in the command mode and the query mode are discussed. The commands which are unique to the Help mode are not presented since the Help Facility is not implemented in this version of GPI. The Help mode was mentioned here nonetheless, since its usefulness and importance is recognized.

3. Command Mode Commands

GPI provides several commands which are unique to the command mode. These commands are entered by the user on the Command/Result area of the CRT display and are initiated following a carriage return. In order to lessen the probability of the user inadvertently executing a command, a safety check is embedded in the commands which affect the current data table.

Each of the commands are described in detail below. In order to lift the burden of typing in long command words, the user may use the default and type only the first two letters of a given command. Also, in cases where the command requires one or more parameters, if the user types in only the command the system will prompt him for the parameter values.

a. Table

The TABLE command is used to draw a new table skeleton. The format for this command is : TABLE <table position> <table name> <table type> . Valid entries for the parameter, <table position>, are either "1" or "2". These numbers correspond to table skeleton area one or two as displayed on the CRT. The parameter, <table name> and <table type> refer to the name and type of the desired data table skeleton that the user wants displayed in the specified table area. Valid table types are TEXT, FORM, SPREAD (spreadsheet), MAIL, and DB (database).

If a table position of 1 is issued then the system will automatically erase the table skeleton already displayed in table area one and replace it with the table skeleton whose attributes correspond to the table name and table type specified. For example, if area one contained a text data table skeleton and the user issued the following command : TABLE 1 message mail , then the text data table skeleton would be erased and replaced with a mail data table skeleton. The mail data table named "message" would then become the current data table in table area one.

b. Erase

The ERASE command is used to erase user supplied information from the table skeleton areas or the condition box. The format for this command is : ERASE <position(s)>. Valid entries for <position(s)> are "1", "2", "3", or "*". Position number 1 or 2 correspond to the respective CRT table skeleton area. Position number 3 corresponds to the condition box area and a position entry of "*" refers to all three of these areas. For instance, if the command, "ERASE *", was entered, then all query information entered by the user in the data table areas and the condition box area would be erased. The current skeletons for the areas would be redrawn.

c. Run

The RUN command is issued in order to execute a transaction (query). The format for this command is simply : RUN. In cases where the execution of a transaction will change the contents of either of the current data tables (transactions involving INSERT, DELETE or MODIFY Query mode commands) the user will receive a "COMMIT? " response from the system. The user must then respond with a "yes" or "no" answer. If the answer is yes then the current data table will be updated after completion of the execution of the transaction. This extra step provides a safety feature which prevents the user from making irreversible mistakes.

d. Saveto

The SAVETO command is used to save the contents of a system defined table, ISSRESULT, into a table of the user's choice. The ISSRESULT table is a temporary system table which contains the results of the most recent successfully completed transaction. The table that is named as the recipient of the save must be the same type and structure and type as the ISSRESULT table. If the recipient table does not exist already, then the system will automatically create a table whose structure is the same as the ISSRESULT table and assign it the user specified name. The format for the command is : SAVETO <tablename> <table type>.

e. Undo

The UNDO command allows the user to reverse the effects of the most recently committed transaction. It takes no parameters and thus its format is just : UNDO .

f. Directory

This command will display the tuples of the specified Application Directory table. Its format is : DIRECTORY <application type> . Valid entries for the <application type> parameter are TEXT, DB,SPREAD, MAIL, and FORM.

Once the Directory command is issued, the system will automatically clear the screen and will display as many tuples of the specified directory as will fit on one screen. The user will be prompted by the system response "MORE?". A user response of "yes" to the question will initiate scrolling of the tuples in the Application Directory table. Once the user responds with "no" , the scrolling will cease and whatever contents were present on the CRT before the DIRECTORY command was issued will be redrawn on the terminal.

g. Application

This command enables the user to suspend any actions being performed and to enter one of the five ISS Application specific modes. The format used is : APPLICATION <application name> .

As an example, if a user desires to use the capabilities unique to the Spreadsheet Application Software, then she would enter the command " APPLICATION SPREAD ". The current contents of the screen would be erased and the user would be placed in the SPREADSHEET application. The user would then perform operations unique to the spreadsheet application such as displaying a spreadsheet view. Once the user terminates the application session, she may continue from where she left off prior to executing the APPLICATION command.

h. Create

The CREATE command is used to create new data tables. It has several embedded prompts or subcommands which make the task of creating new tables very elementary. Its format is simply : CREATE .

Once the command is issued the user will progress through a series of system prompts which will ultimately build the data table. The first system prompt is for the table name followed by a request for the table type. If the type is one of the predefined structures, ie TEXT, DB (Database), FORM, SPREAD,or MAIL, then the prompting session is over at this point and the system will add the new data table to the appropriate Application Directory table.

If the new table is a database data table then the prompting session continues and the user is asked to provide the attribute name, data type, character width.

Also the user is asked if the attribute is a key field. After the user has completely defined the structure of the new database data table and terminates the prompting session, the new table will be added to the Database Application Directory. Also, the system will then automatically update the Schema Table as neccessary.

Following the CREATE command, the system will issue a response on the Command/Response area to assure the user that the new data table has been created and added to the appropriate Application Directory table.

i. Delete

In the command mode, the DELETE command refers to the deletion of a table from the ISS database. The format is DELETE <table name> <table type>. Before executing this command the system will first echo the user's request and will ask the user if he is sure that he wants to execute the command. If the user response is no then the command is aborted. Otherwise, the specified data table is deleted from the system by erasing all references to the table in the application Directory table and other system tables.

j. Quit

The command, QUIT, enables the user to exit the ISS system. The format for this command is simply : QUIT.

4. Query Mode Commands

The Query mode commands of GPI provide the user with a very simple technique for writing both elementary and complex queries. A subset of the commands is based on the Relational Database Model and corresponds to the ISS primitives and combining operators which were discussed in chapter 4.

As mentioned earlier, each table skeleton display has a command column in addition to the data table attribute columns. It is in this command column that the user specifies query commands to be performed on selected tuples of the underlying data table.

Query mode commands can be used in conjunction with user specified variable names, literals, or order indicators (described below). The variables are preceded by a question mark. They are used as domain variables in cases where the user desires that the operation indicated by the command in the command column be performed only on tuples with certain conditions placed on attributes represented by a variable. Variables may also appear in the command column provided that they are preceded by a query command. Variables of this type are referred to as tuple variables since they represent the entire tuple as opposed to only one attribute. Variables which are used to express conditions other than equality are further defined in the condition box.

Any character string which is not preceded by an "@" or "?" and appears in an attribute column is considered to be a literal. If a literal appears under an attribute name then this implies that the query command only applies to tuples in the data table which have the same value as the literal in the specified attribute.

Order indicators can be used to specify attribute precedence or reordering of attributes. As an example, if a data table is to be sorted on two or more attributes, then the order indicator can be used to specify which field (attribute) is the primary sort field, secondary sort field, etc. The syntax for the order indicators is any integer preceded by an "@" sign, i.e. "@1", and if used they must be placed in the column of the attribute to which they apply. The lower number indicates higher precedence. Order indicators can also be used with the project, print or display command if the user desires to see the table attributes stored, printed or displayed in an order different from the way they appear in the data table.

The definition and usage of the individual Query mode commands are given below. The semantics of many of the commands have been defined in Chapter 4 and will not be repeated. However, many examples will be presented in order to show how the user can use the GPI language to effect the queries. In addition, the use of the variables, literals, and order indicators will be more clearly illustrated in the examples.

Each query command must be preceded by the symbol "!" and at least the first two letters of the command must be typed. The user may, however, choose to type the entire command.

a. Delete

The deletion operator is effected by the GPI command "DELETE". The tuples deleted by this command are stored in a system database data table named ISSDELETE. ISSDELETE automatically assumes the same structure of the table from which tuples were deleted. Figure 5.2 is an example of a GPI query to delete all the tuples whose ID is greater than 3 and less than 10. The command !DE is entered on the first row underneath the command column. The "?X" is a variable that represents the ID attribute and it is used to qualify which records of TableA are to be deleted. The qualification of the tuples is made complete with the condition statement in the condition box.

After executing this query tableA would no longer have the TEXT_LINES which corresponded to ID numbers 4 through 9 of TableA prior to running the query. However, the ISSDELETE table would contain the six deleted TEXT_LINEs and its structure would be the same as a text data table.

b. Insert

The Insert primitive operator corresponds to the command INSERT. It is used to insert a tuple or table of tuples into a target data table at a specified location (ID value). The command may be unary or binary, i.e. operate on one or two data tables. If it is used as a binary operator then the results of the command are stored in the table whose structure appears in the first table skeleton area. The contents of the second table will be unaltered by the execution of the INSERT command.

If the user desires that the tuple or table be inserted into the target table at a given location or ID value, he may indicate this by appending the value of the ID to the end of the INSERT command in the first table. For instance, figure 5.3, part A, gives an example of a query to insert TableB, a text data table into TableA, a form data table, at location 3.

Part B of the figure shows an example of the INSERT command used as a unary operation to insert a single tuple at location four of the target data table. The literals "48 Hours", "10", and "Eric" constitute the attribute values of the inserted tuple.

Part C illustrates a query to insert TableB, a smaller data table (one with fewer attributes) into TableA, a larger data table. Since the attributes of TableB are a subset of those in TableA, this is a valid insertion and is consistent with the discussion of the semantics of the Insert primitive operator given in chapter 4. Before making the insertion the system will add the REVIEWER attribute to TableB, but the Reviewer value for each tuple in the table will be blank. Since there is no ID value specified in the INSERT command, the tuples of TableB will be appended to those in TableA.

Figure 5.4 shows the contents of TableA after the execution of the query in part C of figure 5.3, assuming that each table had only one tuple. Note that the REVIEWER attribute of the "ET" tuple from TableB is empty.

c. Modify

The Modify primitive operator corresponds to the GPI command, MODIFY. The execution of the command will directly alter the contents of the underlying data table. It can be used in several ways. Part A of figure 5.5 represents GPI syntax to perform a query to increase the salary of all employees listed in TABLEB by ten percent. In this example a variable and a condition is used to effect this modification and all tuples are affected.

Part B represents the query " change G. Thompson's salary to $100,000 and JOBTITLE to manager ". Both literals and attribute variables are used to perform the query. The variable "Y" listed under the NAME attribute and qualified in the condition box by an equality statement lets the system know to only modify the tuple in tableA that has a value of "G. Thompson" in the name attribute. The literals "manager" and "100000" are the values that will replace whatever was in JOBTITLE and SALARY attributes of G. Thompson' tuple entry.

d. Project

The Projection primitive operator is effected by the command PROJECT. This command does not change the operand data table and thus is committed as soon as the RUN command is issued. PROJECT will create a new data table of the same number of tuples as the table in which the projection was performed. The attributes of the new table will consist of the system generated ID and the indicated subset of the attributes of the original data table. The new table will be stored in the ISSRESULT temporary data table. (Note that the ISSRESULT table can be copied into a permanent user data table by using the SAVETO command as discussed previously).

Figure 5.6 illustrates a query to project out the NAME and SALARY attributes of TABLEB. It also shows the structure of the ISSRESULT table after the projection query terminates.

e. Select

The SELECT command corresponds to the Select primitive operator. It is used to extract tuples of the current data tables which satisfy some specified condition and store them temporarily in the ISSRESULT table of the same structure. The SELECT command is quite versatile and is often used in the same query as other commands.

Figure 5.7, part A shows an example of a query to select all tuples whose NAME value is "P. Harrison" and JOBTITLE is "laborer". The literal "P. Harrison" in the name attribute and "laborer" in the JOBTITLE attribute determine which tuples are selected. In part b, the syntax expresses a query to find the tuples of all laborers in the data table whose salary is $9,000. It contains two literals, "laborer" and "9000" on the same query line and this implies an " AND" relationship. In other words, if a literal or variable appears in the columns of more than one attribute and on the same query line, then a "logical and" on the values of the attributes is done to determine which tuples are selected. Part c of the figure shows the same query as part b, but uses variables and the condition box in the syntax. Using the condition box adds greater query flexibility since it allows one to place any number of conditions on the variables. For instance, if there was a desire that only the laborers whose salary is greater than $10,000 be selected, then the entry in the condition box would be " ?Y > 10000.

f. Union

This command is used to perform the union primitive operator whose semantics has been discussed in chapter IV. Since union is a binary operator, the UNION command always operates on two different data tables simultaneously (this is the case with all commands which represent binary operators). The data table which appears in table area one is referred to as the dominant table (or left operand) and it determines the structure of the results from the union. The table that appears in table area two is called the subordinate table (or right operand) of the operation.

Consistent with the Union operator discussion in chapter IV, the UNION command may be performed on tables of different arity as long as the attributes of the dominant table are a subset of the subordinate data table. The syntax required to form a union query is very simple and is indicated in figure 5.8. All that is required is that the command !UN (or !UNION) be placed in the first row of the command column of each data table. Part a of figure 5.8 represents a query to form the union of a text data table and a form data table. TableA, the text data table is the dominant table in this query so the attributes of the ISSRESULT table will have the attribute names ID and TEXT_LINE after execution of the query. The union of these two tables is feasible since TEXT_LINE and FORM_LINE are aliases of each other.

Part B of the figure illustrates a query to perform the union of a text data table and a mail data table. The results of this union would also have the attributes only of the text data table. The contents of the ISSRESULT table would be all the TEXT_LINE's of TABLEA followed by the BODY of all tuples in TABLEC, the mail table, which were distinct from the TEXT_LINE's of TABLEA. The BODY attribute is used in the union since it is an alias of the TEXT_LINE attribute. The rest of the attributes of the mail data table would be ignored by the system.

g. Intersect

The INTERSECT command performs the intersection binary operator. It operates on two data tables of the same arity. In addition, each attribute in the dominant table must have one and only one attribute in the subordinate data table of which it is an alias or synonym and vice versa. The results of the INTERSECT is a set of tuples that are in both the dominant and subordinate data tables. Figure 5.9 shows a query that will take the intersection of a text data table and a form data table. Since IN is the short form of the INSERT command, IT is used as the short form of INTERSECT in order to distinguish the two commands.

h. Sdiff

The SDIFF command corresponds to the set difference operator. Is a binary command that requires both the dominant and subordinate data tables to be of the same arity. It poses the same alias constraints on the attributes as the INTERSECT command. SDIFF will compare the two data tables and store in the ISSRESULT table all tuples that are in the dominant table that are not in the subordinate table. Figure 5.10 illustrates a query to take the set difference of two mail data tables.

i. Join

JOIN is a binary command that operates on two data tables which may be of the same or different arity. The semantics of the command is discussed in chapter IV. It is only meaningful if the attributes on which the join is performed range over the same domain. Figure 5.11 illustrates the syntax to perform the query " join tuples of tableB to tuples of tableA if the subjects are the same ". This essentially is an equijoin on the SUBJ attributes

In the figure the variable "Q" is used to denote which of the attributes of the tables are used to form the join. The fact that the same variable appears in both tables under the appropriate attribute column implies an equijoin. (If the condition of the join is other than equality then two different variables must be used and the relation between them must be stated in the condition box). The ISSRESULT table will contain the results of the query and the structure will be all the attributes of TABLEA followed by all the attributes of TABLEB. Note that this is redundant since both of the SUBJECT attributes will appear in the results and will contain duplicate data.

j. Njoin

The NJOIN command is used to perform the natural join binary operation. By definition, natural join is an equijoin with the duplicate attributes removed or projected out of the resultant data table. Forming a query to perform a natural join of two data tables follows the same structure as a query to effect an equijoin (except of course the command entered would be !NJ or !NJOIN as opposed to !JO or !JOIN).

k. Concat

The CONCAT command is used to perform the ISS specialized operator, concatenate. This command is unary, i.e. it operates on only one data table at a time. It is used to concatenate the values of the attributes of a data table of arbitrary arity into a text data table structure. The values of the attributes are separated by one space. The results of the operation is stored in the ISSRESULT table and the operand data table is not altered.

Figure 5.12 shows an example of a query to convert a database data table into a text data table. Part b shows the results as they would appear in the ISSRESULT table.

l. Sort

The SORT command is not related to any ISS primitive or combining operator. It is included in the command set because of the recognition of its great utility and its enhancement to the ISS. It provides the capability for the user to sort a data table on any number of attributes in ascending or descending order. The default sort order is ascending.

The user designates the precedence of the sort field or attribute by placing an order indicator (an integer preceded by the "@" sign) in the column under the attribute name. The integers must be unique and the lower one takes the higher precedence. If descending sort order is desired on any field then the order indicator should be followed by the letter "D". Figure 5.13 shows an example of a query to sort a mail data table in descending order based on the date and within the each date sort on the TO attribute in ascending order. No variables or condition statements are required to effect a sort.

m. Display/Print

A Data Manipulation Language is not very useful from the user's viewpoint if it does not provide a facility for users to form queries that involve output. The commands DISPLAY and PRINT are included among the Query Mode commands for that reason. DISPLAY is used to direct output to the CRT and PRINT directs output to the line printer. Both commands operate on an entire data table which could either be the original current data table or the ISSRESULT table. In other words, if the PRINT or DISPLAY command is the only command in a query then all tuples of the original table will be printed (displayed), however if PRINT or DISPLAY is preceded by other commands in the same query, then all tuples of the ISSRESULT table will be printed (displayed). (Note that the use of attribute variables with a PRINT or DISPLAY command is simply ignored by the system since these commands operate on tuples only.)

The PRINT and DISPLAY commands can operate in two output formats. The default format is to output the attribute in table form with each row representing a tuple and each column representing an attribute name (the names of each attribute would appear at the top of each column). If the default is desired the user may simply type the command (PRINT or DISPLAY) or he may append a "1" to the end of the command. Figure 5.14 shows an example which uses format one, the default. As illustrated in the figure, if the user requires that the attributes be printed in different order, then the order indicators could be used to determine the printing sequence for the attributes. The results of the query are also shown. It should be noted that the generates temporary output which is not stored in the ISSRESULT table.

The other output method will display or print each of the attribute names on a separate line followed by the attribute value and it is invoked by appending a "2" to the end of the DISPLAY or PRINT command. This method limits the amount of information that can be displayed on the CRT as compared to the default format. Figure 5.15 illustrates a display query which invokes format method two.

5. Combining Query Mode Commands

Section 4 explained the query mode commands, defined the command syntax required for each of them, and illustrated their use by presenting various examples. In each example, only a single command was used in each query. This section addresses the issues of compound queries, i.e. those involving more than one query command in the same transaction.

Certain precedence rules apply when the user combines query mode commands. First of all, the order in which the commands appear in the table skeletons is important. Straying from conventional programming languages somewhat, if more than one command is listed in a table skeleton then ISS will execute the commands in a "bottom up" order and each subsequent command will operate on the data table results of the previous command in the query. Also, if two data tables are used in the query, then commands that apply only to table number two will be executed before any commands in table number one will be performed. Commands which apply to both data tables will be executed last and following the convention, they should be the appear on the first query line of both data table skeletons. This condition also imposes another important rule that must be followed when using multiple commands. The rule is that only one binary command is allowed in one transaction. This convention is necessary in order to keep the syntax simple and limit the amount of meaningless queries that can be formed by the user.

Another important issue to address is the use of domain variables in multiple command queries. In the queries where only one command was used this issue was not pressing. But, when there are several commands used either in the same data table or within both data tables a few extra rules are necessary. For example, if the same domain variable is used in an attribute of table one and also in an attribute of table two, the convention is that only the tuples of both tables that have the same value in those attributes will be operated on.

Tuple variables were mentioned briefly in the GPI introduction. These tuple variables, if used , must be preceded by some query mode command in the COMMAND column of the data table skeleton. They indicate that the command applies to the entire tuple and they are optional in cases where their absence presents no ambiguity. If a variable is used as a tuple variable in a transaction, it can not be used as a domain or attribute variable in that same transaction or query.

Tuple variables differ from attribute (domain) variables in yet another way. If the same tuple variable appears in several different commands of the same data table it implies that the subsequent command only applies to tuples that were qualified by the prior command. For Example, in figure 5.16 the tuple variable "Z" appears both with a SELECT command and a MODIFY command which operate on the given data table. In this case, the SELECT command will be performed first and the tuple that has a value of "WEEZY" in the name attribute will be selected and assigned to the variable "Z". The MODIFY command will be performed next but will only apply to tuples which are assigned to variable "Z". If no tuple variable was associated with the MODIFY command in this example, then execution of the query would update all addresses in the data table to "1 Popeye ST" instead of just modifying Weezy's address.

An example which illustrates most of the issues discussed in this section is shown in figure 5.17. The query represented operates on two data tables, Salary and Personnel. The query corresponds to "update all employees' salaries by twenty percent and join the Salary and Personnel tuples of all single male employees who work in either department A or department B and whose salary is $45,000 or more".

The SELECT command in table 2, Personnel, selects tuples from the data table whose SEX attribute equals "male" and whose marital status equals "single" and assigns them to tuple variable "Z". These tuples are the only ones from the Personnel table that will be utilized in the JOIN. This is indicated by the presence of the same tuple variable, "Z", following the JOIN command in the Personnel table skeleton.

Before the join is done, commands which pertain only to the Salary data table are executed. In this table, the MODIFY command will be performed first and will increase the SALARY attribute of all tuples in the Salary data table by twenty percent (all will be updated since no tuple variable is indicated). The SELECT command will be performed next and the two attribute variables, "S" and "D" will be used to qualify the desired tuples. The condition box is checked to determine the conditions placed on the attribute variables. Since the Select command is followed by the tuple variable, "X", all tuples which meet the criteria of the SELECT command will be assigned to this variable. These tuples will be the only ones from the Salary data table that will be used by the JOIN command.

Now that all the table specific commands have been performed, the system then executes the binary command, JOIN. As indicated by the "E" attribute variable, the join is performed on the EMP_NO and EMP_ID attributes of the corresponding tables, and only the previously qualified tuples of each table will be joined together. The join performed is an equijoin.

Although this example is not very complicated, it sufficiently demonstrates how one can utilize the capabilities to combine Query Mode commands in one transaction to compose queries of varying degrees of complexity. It also illustrates the flexibility and simplicity the user is afforded with the Graphics Prototype Interface Data Manipulation Language.

B. IMPLEMENTATION OVERVIEW

In the conceptual design of the user interface, several intricate design implementation issues and requirements arose. Due to the complexity of some of these issues and time constraints, realization of the GPI as described has been precluded. However, the intent of the design was to present a model system on which to base implementation. In order to test the feasibility of the design, the GPI commands described in section A of this chapter were implemented (prototyped) by mapping them into Shell programs written in the C and SHELL programming languages in the UNIX Operating System environment. This prototyped implementation, although not as elegant as the proposed system suggests, also serves to demonstrate the usefulness of GPI and ISS.

In the following chapter the implementation strategy used to effect the Prototype ISS is discussed, as well as the limitations of the strategy. Any limitation which gives rise to follow-on research is pointed out in chapter VII, the conclusion.

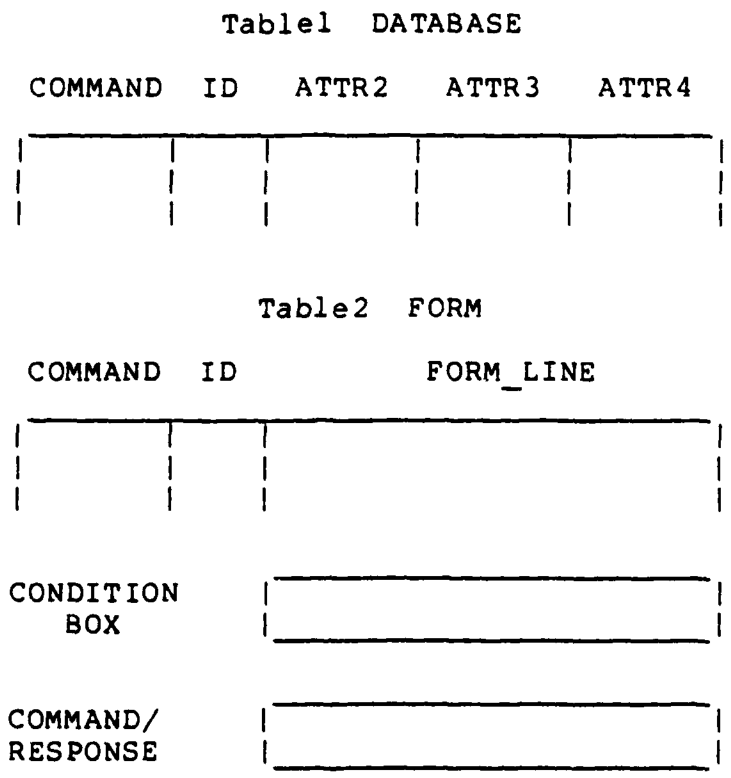


Figure 5.1 - General CRT Layout

TableA TEXT

COMMAND	ID	TEXT_LINE
!DE	?X	

CONDITION BOX	
	?X > 3 && ?X < 10

Figure 5.2 - Delete Query

TableA FORM

COMMAND	ID	FORM_LINE
!IN3		

TableB TEXT

COMMAND	ID	TEXT_LINE
!IN		

Part A

TABLEA DATABASE

COMMAND	ID	MOVIE	RATING	REVIEWER
!IN4		48 Hours	10	Eric

Part B

TABLEA DATABASE

COMMAND	ID	MOVIE	RATING	REVIEWER
!IN				

TABLEB DATABASE

COMMAND	ID	MOVIE	RATING
!IN			

Part C

Figure 5.3 - Insert Queries

TABLEA DATABASE

ID	MOVIE	RATING	REVIEWER
1	48 Hours	10	Eric
2	ET	9	

Figure 5.4 - Insert Query Results

TABLEB DATABASE

COMMAND	ID	NAME	JOBTITLE	SALARY
!MO				?X

CONDITION BOX

?X = 1.1 * ?X

Part A

TABLEA DATABASE

COMMAND	ID	NAME	JOBTITLE	SALARY
!MO		?Y	manager	100000

CONDITION BOX

?Y = G. Thompson

Part B

Figure 5.5 - Modify Queries

TABLEB DB

COMMAND	ID	NAME	JOBTITLE	SALARY
!PR		?A		?B

ISSRESULT DATABASE

ID	NAME	SALARY

Figure 5.6 - Project Query

TABLEA DATABASE

COMMAND	ID	NAME	JOBTITLE	SALARY
!SE		P. Harrison	laborer	

Part A

TABLEA DATABASE

COMMAND	ID	NAME	JOBTITLE	SALARY
!SE			laborer	9000

Part B

TABLEA DATABASE

COMMAND	ID	NAME	JOBTITLE	SALARY
!SE			?X	?Y

CONDITION BOX	
	?X = laborer && ?Y = 9000

Part C

Figure 5.7 - Select Queries

TABLEA TEXT

COMMAND	ID	TEXT_LINE
!UN		

TABLEB FORM

COMMAND	ID	FORM_LINE
!UN		

Part A

TABLEA TEXT

COMMAND	ID	TEXT_LINE
!UN		

TABLEC MAIL

COMMAND	ID	VIEWED	TO	COPY_TO	DATE	SUBJECT	BODY
!UN							

Part B

Figure 5.8 - Union Query

TABLEA TEXT

COMMAND	ID	TEXT_LINE
!IT		

TABLEB FORM

COMMAND	ID	FORM_LINE
!IT		

Figure 5.9 - Intersect Query

TABLEA MAIL

COMMAND	ID	VIEWED	TO	COPY_TO	DATE	SUBJECT	BODY
!SD							

TABLEB MAIL

COMMAND	ID	VIEWED	TO	COPY_TO	DATE	SUBJECT	BODY
!SD							

Figure 5.10 - Set Difference Query

TABLEA DATABASE

COMMAND	ID	SUBJECT	DATE	PRECEDENCE
!JO		?Q		

TABLEB MAIL

COMMAND	ID	VIEWED	TO	COPY_TO	DATE	SUBJECT	BODY
!JO						?Q	

Figure 5.11 - Join Query

TABLEA DATABASE

COMMAND	ID	COUNTRY	CURRENCY
!CO			

Part A

ISSRESULT

ID	TEXT_LINE
1	UNITED STATES DOLLAR
2	AUSTRALIA DOLLAR
3	ENGLAND POUND

Part B

Figure 5.12 - Concatenate Query

TABLEB MAIL

COMMAND	ID	VIEWED	TO	COPY_TO	DATE	SUBJECT	BODY
!SO					@1D	@2	

Figure 5.13 - Sort Query

```
                 TABLEA  DATABASE

COMMAND   ID      NAME            DEPT       SEX
 ______________________________________________________
| !PR1   | @1 | @2           |    @4     |    @3     |
|        |    |              |           |           |
|        |    |              |           |           |
```

```
        ID        NAME           SEX     DEPT

        1      S. Snodgrass       F        A
        2      P. Poopdeck        M        B
```

Figure 5.14 - Print Query

```
                    TABLEA   DATABASE

 COMMAND   ID      NAME            DEPT        SEX
 _____________________________________________________
| !DI2    |     |            |            |          |
|         |     |            |            |          |
|         |     |            |            |          |


ID:      1
NAME:    S. Snodgrass
DEPT:    A
SEX:     F

ID:      2
NAME:    P. Poopdeck
DEPT:    B
SEX:     M
```

Figure 5.15 - Display Query

```
                    TABLEA   DATABASE

 COMMAND   ID      NAME              ADDRESS
 _____________________________________________________
| !MO ?Z|     |            | 1 Popeye St.            |
| !SE ?Z|     |  Weezy     |                         |
|       |     |            |                         |
```

Figure 5.16 - Select and Modify Query

SALARY DATABASE

COMMAND	ID	EMP_NO	SALARY	DEPT
!JO ?X		?E		
!SE ?X			?S	?D
!MO			?M	

PERSONNEL DATABASE

COMMAND	ID	EMP_ID	EMP_NAME	ADDRESS	SEX	MARITAL
!JO ?Z		?E				
!SE ?Z					male	single

```
CONDITION |  ?D = A || ?D = B
   BOX    |  ?M = 1.2 * ?M    ;   ?S >= 45000
```

Figure 5.17 - Multiple Commands Query

VI. IMPLEMENTATION PROPOSALS

This chapter proposes a high level design for an ISS implementation and a strategy for a very simple prototype implementation using C Shell in the Unix environment. The prototype design is included to assist continuing research efforts to quickly implement a prototype system. When the basic system is working it may be evaluated, improved and expanded in future refinements of the overall design. This chapter is by no means a full design specification, as that task itself would cover much of the material for a separate masters thesis. The design proposal covers the ISS kernel main modules only, describing the overall hierarchy of the system. Subsequent refinements should include detailed module interfaces, file usage and data descriptions.

Although not stated explicitly, the ISS itself is not a database management system (DBMS) and so its design does not include those issues handled by a DBMS. These include rollback and recovery, concurrency control and retrieval strategies. It should be recognized that the lowest level modules in ISS, the primitive operators, will be highly dependent on the underlying DBMS or operating system.

A. SYSTEM DESIGN

The basic structure of ISS operates upon an underlying operating system, DBMS or file system for utilities, file manipulation and management services. Figure 6.1 shows this basic relationship.

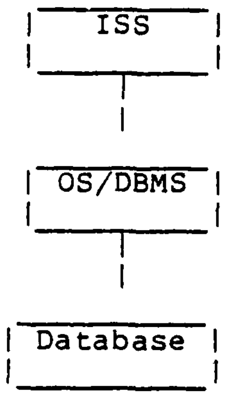


Figure 6.1 - Basic ISS Relationships

Within the ISS there are five major levels which are shown in figure 6.2 and correspond to levels of operation of the system: system entry; command mode interpretation, command, other mode and applications execution; graphics and run time sequences; and at the lowest level the ISS primitive operators. The basic operators then translate their calls into the appropriate calls on the operating system or DBMS.

The system database in use will include the Schema Table, the Application Directories, the ISS Users Table, the

ISSRESULT and ISSDELETE Tables as described in chapter three of this thesis. In addition, a number of tables will be required to hold error and trap conditions, warnings and error messages, graphics data and probably other run time requirements. A number of system variables will need to be maintained to hold directory entries of current tables, their types, system status, user and security information and, of course, run time temporary variables.

The following subsections briefly describe the major design modules and groups of modules in the system as depicted in figure 6.2. Groups of modules are shown in open boxes and represent separate equal level modules but are compressed into one group for in order to prevent overcrowding the diagram.

1. Entry Module

The Entry Module is responsible for invoking the ISS password procedure and identifying the prospective users. After a user is identified the entry module must perform the necessary initializations and then call the command module. If the password procedure fails after a predetermined number of times the system should take whatever action is determined appropriate.

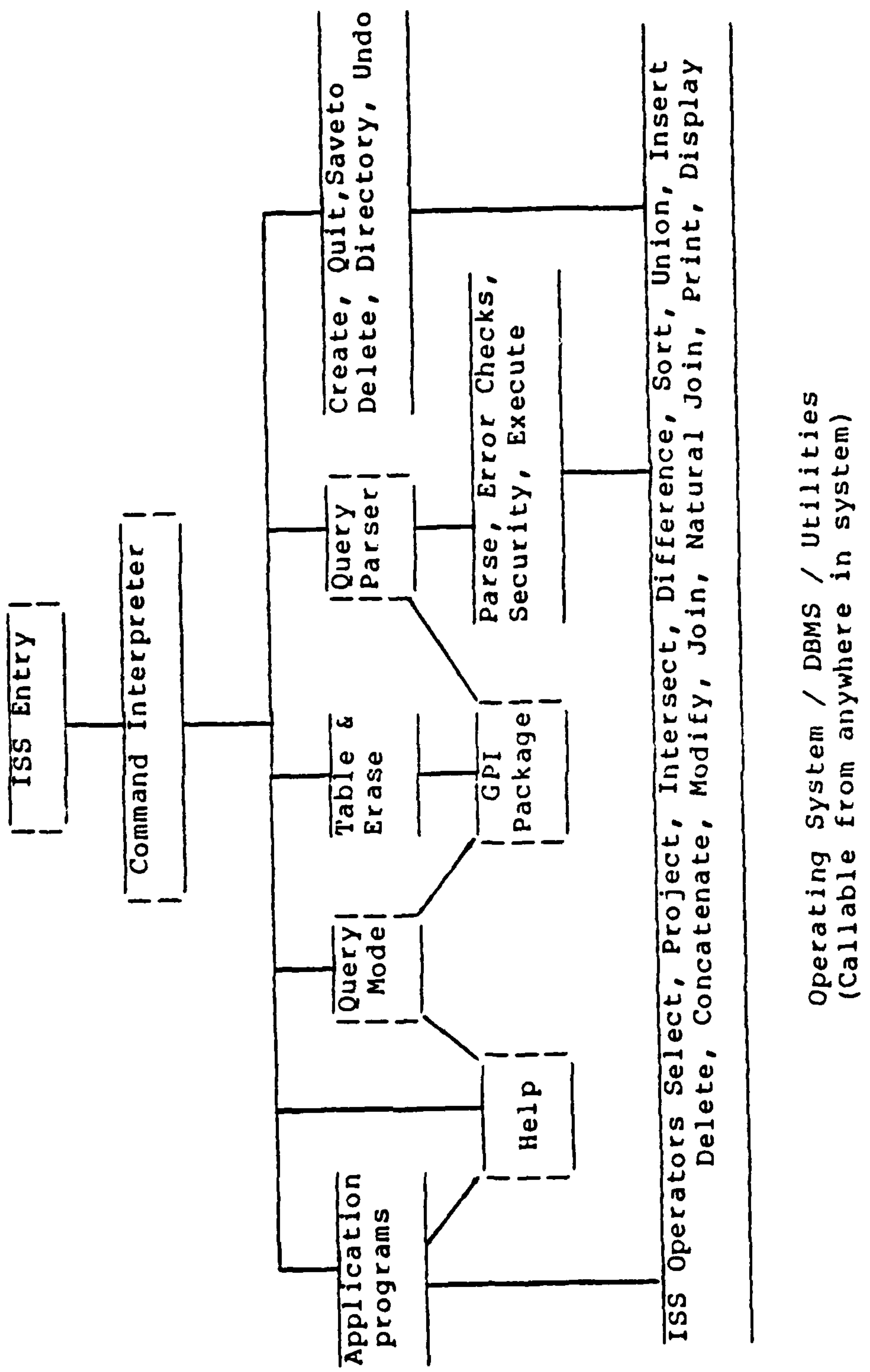

ISS Entry
Command Interpreter
Application programs
Help
Query Mode
GPI Package
Table & Erase
Query Parser
Parse, Error Checks, Security, Execute
Create, Quit, Saveto Delete, Directory, Undo
ISS Operators Select, Project, Intersect, Difference, Sort, Union, Insert Delete, Concatenate, Modify, Join, Natural Join, Print, Display
Operating System / DBMS / Utilities (Callable from anywhere in system)

Figure 6.2 - ISS Hierarchy Diagram

2. <u>Command Interpreter</u>

The Command Interpreter is a simple interpreter accepting commands or mode changes and calling the correct command execution module. After each command terminates the interpreter should reinstate the previous display. This may require direct calls to the GPI

3. <u>Help</u>

Help is a simple help facility which may be implemented in many different ways. Help is entered from any interactive environment by the entry of control H (^H) and returns control to the calling module.

4. <u>Query Mode</u>

The Query Mode is the module which allows entry of user queries into the skeletons described in chapter V. This modules draws heavily upon the services of the GPI package to allow on-screen editing of queries. The editing of the queries has no execution significance within the query mode, it is simply an entry function. The query mode is invoked by the entry of a control Q (^Q).

5. <u>GPI Package</u>

The Graphics Prototype Interface manages the graphics interface and storage of current queries. It is used by the Query Mode Module to change and enter queries and by the Query Parser to get the stored data. This module is highly dependent on the physical characteristics of the target system.

6. Query Parser

The Query Parser translates the queries in the skeleton form into a sequence of calls on the ISS operators and is invoked by the RUN command. It must retrieve the skeletons from the GPI Package, parse them, perform type checking for the issuance of warning of warnings and traps, check authorizations and privileges of the user, ask for verification on transactions which change the database, translate virtual tables to queries and, finally, execute the final transaction using a sequence of calls to the basic operators or return an appropriate error message.

7. Other Commands

Each of the other commands UNDO, SAVETO, DIRECTORY, TABLE, ERASE CREATE, DELETE and QUIT invokes a module to effect the command. The TABLE and ERASE commands must call on the GPI Package since they affect the current skeleton display of queries. The other commands may produce output and temporarily displace the current query image but are themselves overwritten by a redisplay of the current skeletons after execution. Some commands such as SAVETO will need to make direct calls to the underlying operating system or DBMS to perform functions.

8. Application Programs

Each of the logical applications is invoked from the command interpreter module and is enclosed in its own module. The applications may make calls to the operators and the operating system to achieve their objectives, but are in reality very much like a view on the database. It is the single underlying database with its set of operations which is at the heart of the integration of the system.

B. PROTOTYPE IMPLEMENTATION STRATEGY

In any system design a prototype implementation is often useful to determine unforeseen shortfalls and difficulties of a proposed system. This section is intended as a brief description of a strategy for implementing a simple ISS prototype kernel using C Shell running under Unix on a Vax 11/780 system currently operating at the Naval Postgraduate School, Monterey. As with many prototypes, efficiency, completeness and absolute adherence to the defined system is not intended, instead it represents the first step toward the Unix environment philosophy to "get something small working as soon as possible". [Ref. 14]

The target display is a simple 24 by 80 column display with Unix as the underlying operating system with its standard tools and file services. The following subsections describe the overall strategy, the database and a simple "graphics" implementation. The descriptions assume a knowledge of the C Shell interpreter and the Unix programming environment.

1. Strategy

Each module described in section A may be implemented as a C Shell script, calling on lower level scripts and Unix services when required. Tasks which are difficult to achieve using the C Shell may be written as C routines and then added to the standard set of Unix tools. One such example is the requirement for a utility to perform set difference on unsorted files. A simple C program may be written to perform this function, albeit inefficiently, and then called in the same way as normal Unix commands.

Many of the Unix commands act as filters, acting on standard input or specified files and send results to the standard output. Unix's ability to redirect the standard input and output provides powerful file manipulation facilities, and since the majority of tasks in the ISS kernel are file manipulations, this environment is ideal. Figure 6.3 shows a very simple minded example of a shell script for the DELETE operator using the Unix Awk pattern program and redirection of the results into the desired tables. Awk operates as a filter reading the whole file and performing the actions on each line in this case printing to the desired file depending on the value of the condition for the line. (The $1 and $2 in the Awk command lines are expanded to the parameters of the call to the delete script.)

Operations on files may be implemented by using filters to process input files into temporary files and result files. Processing can continue through a series of filters until the desired result is obtained.

```
# Simple ISS DELETE operator script using
# the Unix filter Awk.  The script is called
# with two parameters, the first a file name
# and the second a string which is a correct
# Awk field matching expression.  Awk sends
# its results to the standard output but this
# script redirects that output to the file
# ISSRESULT and then again to standard output.
# (The script calling this may redirect the
# result of this call to some other file!)
# Usage:    delete  file   'condition'

awk -e ' BEGIN {FS = ":"\
                OFS = F$\
        {\
        if ('$2') { print > "ISSDELETE" }\
        else { print }\
        }'     $1   > ISSRESULT\

cat ISSRESULT        # To standard output

# End script delete
```

Figure 6.3 - Sample Delete Script

Variables which need to be passed down to lower level scripts may be placed in the environment or passed as explicit parameters of the call. Values can be returned from lower levels by placing results in the standard output while the calling routine redirects standard output to a file or places it in a variable. This arrangement provides flexibility, since the calling routines have the freedom to place results, which may be entire files, wherever they choose. This essentially is the philosophy of the Unix filter, coupled with the freedom to redirect input and output. The set of ISS operators can be realized by using the Unix tools such as sed, Grep, Awk and Sort, and writing C routines to perform other low level routines.

2. Data Base

Unix files are simply sequences of characters organized in a hierarchical naming structure. Some Unix commands provide field discrimination by the identification of fields separators and thus for a first prototype it would be best to use this facility. System and user tables can be files with embedded file separators. The choice of separators should be defined as a variable and placed in the environment by the top level module. The use of field separators removes the need to be concerned about field lengths, therefore it is recommended that the prototype dispense with field length checks altogether, effectively making all fields of varying length.

The ID field should be explicitly included in every file, although it is not clear if this is necessary or whether, when IDs are required for a data manipulation, the builtin line counters of the Unix filters Awk and Sed are sufficient to meet all needs. This uncertainty applies particularly to the applications which use sorted files. In any case, care must be taken to ensure line IDs are removed before attempting line comparisons in the UNION, DIFFERENCE and INTERSECT operators, and resultant tables have all lines renumbered contiguously. The tasks to strip the ID field from a table and renumber a stripped table or result are ideal tasks for C programs written as filters.

The Unix files comprising the system should occupy their own set of sub-directories. It is suggested that the single top level entry module called "iss" occupy a directory as the only file with two sub-directories: "bin" to hold all the executable scripts and object code of the C programs in the system, and "data" to hold all the data files for the system. The top level entry module may then include the path to iss/bin in its path length and rehash to allow access to the commands therein while executing the iss and deeper level scripts. A third sub-directory may be used to store all source programs for the entire system.

3. A Simple GPI Package

A very simple query entry facility may be implemented using the Unix display editor Vi. It is stressed that this is not a true graphics implementation, but a simple measure for the first prototype. The essence is that the display is simply a file which is typed to the screen for display, or edited using Vi to enter queries. The screen file may be initially created by combining in sequence three template files holding the empty skeletons for the condition, and the selected tables. The templates may be made up for each of the fixed structure types and for every individual database data table. Since the copies of the templates are a fixed number of lines in the screen file, the screen file may be split up into the original components for parsing or for replacement of part of the file by a TABLE or ERASE command. Entry to the query mode is then a simple call to Vi to edit the screen file.

This query implementation is proposed because it is very simple and easy to implement. It does however, place a burden on the user to use Vi to enter his queries, not disturbing the number of lines in the file or the table skeleton structure.

Parsing can be done by searching for the strings in between the table column boundaries (the character |) line by line from the bottom up. Parsing should be relatively simple because of the simple rules of the GPI, that is: each operator operates on the current table, or if it is linked to a lower operator by a tuple variable it is applied to ISSRESULT.

The use of the method outlined will require the create command to produce the appropriate skeletons for new database tables and the command parser to produce the template for ISSRESULT each time it is created.

4. Limitations

An implementation as described in this section will provide a very basic ISS kernel which may be refined and improved in subsequent iterations of the project. The major limitation is the graphics interface which will be particularly error prone if editing is not done correctly. The skeleton templates will be the definition driving the field definition during parsing rather than the Schema as should be the case, hence if the tables are changed during query entry there will most likely be unpredictable behavior by the system.

Manipulation of files using filters will be very slow, especially on medium to long files. Although it is much easier to implement the basic operators this way, files will need to be filtered three or more times for some operations rather than the direct access methods available using a proper DBMS. This will involve a high overhead of processing as well as the overhead of the I/O involved in creating and using temporary files. Interpretation by the C Shell will further slow down execution, thus on the whole it will be unacceptably slow for any practical application.

The use of field separators and variable length fields will cause tables not to appear as "neat" tables but instead as untidy lines of uneven length.

True security will not be possible since the Unix Operating System is not itself secure.

Undoubtably, such an unsophisticated implementation will have many other limitations not described here, however the intention is to provide some guidance for future research and prototype implementation.

VII. CONCLUSION

The results of this thesis research more than sufficiently demonstrate that the Relational Database model can indeed be used as the underlying model to conceptually integrate the five application areas of interest: text processing, form generation,database, spreadsheet modeling, and electronic mail. The integration of the common functions of these applications into one ISS kernel which uses a relation expressed in table format as the single data object and which uses a small vocabulary of commands is not only feasible, but also of high utility.

The realization of all of the objectives stated in chapter one support the aforementioned claim and serve as the basis for the findings discussed in section A of this chapter. Section B presents a brief discussion of related research issues which are suggested for follow-on research.

A. FINDINGS

One of the primary objectives of this thesis was to perform a detailed evaluation and analysis of the conceptual level of the design proposed in Nishimura's thesis [Ref. 1]. This included a re-evaluation of the logical databases (tables) designed for the ISS and resulted in the design of supportive ISS system tables which could also be used to incorporate multilevel security in the system. Another significant change made in the design of the tables was the combination of the five application data table schema tables into one common Schema Table which is owned and maintained by the system. It was found that this method is more efficient than maintaining separate schema tables for each application and more truly reflects the idea of an integrated system.

Another major objective was to carefully select or revise the basic primitives needed to form the kernel command vocabulary. It was found that the primitive operators, Modify,Insert, Delete, Project, Select, Sort, and Concatenate, and the combining operators, Join, Union, Set Difference, Intersection, and Natural Join, form a complete set of operators necessary to perform all operations common and desirable to each specific application area. This set of primitive and combining operators coupled with the ability to combine data tables of different types prove to be quite useful in expanding the expressive power of the system.

In addition to refining the original thesis [Ref. 1], major efforts went towards expanding it to include the design of a User Interface to ISS and to propose an Implementation Strategy for the system. This extension resulted in the design of GPI (Graphics Prototype Interface), a Data Manipulation Language which enables the user to use QBE-like simple graphics to easily express queries of varying degrees of complexity. All of the primitive and combining operators are mapped into GPI commands and the details of their implementation are completely hidden from the user. Therefore, any subsequent change to the GPI Language can be made without having to completely redesign the system. Also, by abstracting the implementation details from the User Interface, GPI lends itself quite well to even the most naive user who has some basic background with database query languages.

The Implementation Strategy proposed herein suggests that the UNIX Operating System Environment (including the Shell and C programming languages) can be used to implement a prototype of the ISS. The feasibility of this suggestion was tested and proved by implementing crude prototypes of the basic primitives and combining operators.

Having revised and extended the proposal of Nishimura's thesis [Ref. 1] down to the level of Implementation Strategy, we feel that we have strongly demonstrated that using the Relational Database model in effecting the Integrated Software System is highly feasible and useful. Based on these findings it is suggested that furtherance of this research topic to the point of the completion of the implementation of the proposed design is desirable.

B. FOLLOW-ON RESEARCH

During the design of the ISS several issues arose which gave rise to follow-on research ideas. Some of these were viewed as shortfalls or limitations in the proposed design. Others, such as Database Management System issues were considered to be outside the scope of this thesis. These and other related follow-on research ideas are discussed in the subsequent paragraphs.

The current design of the ISS does not address the issue of efficiency since the main concern was to prove the feasibility of the proposal. Further iterations of this same topic should address efficiency of the design as it impacts on the physical implementation of the conceptual design.

This iteration of the research focuses only on the design of the kernel of the ISS, omitting efforts related to the design of the non-integratable features unique to the individual application areas. The conceptual design and implementation of the application specific commands and views need to be developed. It is recommended that the framework of the design not deviate highly from that suggested for the kernel. Also, in some cases a subset of the five applications may need to perform the same or similar functions. In other words, displaying a spreadsheet view is quite similar to displaying a view of a form. In order to continue with the premise that the total ISS command vocabulary be small, it is suggested that "sub-kernel" commands be developed to support those subsets of applications that have similar functions that are not applicable to all five applications.

The Implementation Strategy proposed addresses simple graphics issues, i.e., the drawing of tables on the CRT. However, it is recognized that a much more sophisticated Graphics Package is necessary to effect the High Level System Graphics commands proposed in the design of the User Interface. Such issues as widening columns during the insert mode to accommodate the input of variable length data attributes or displaying the complete field of a variable length attribute during the display mode should be incorporated into the design.

One further recommendation which is directly related to the current design is that the Physical Level of the design should be carefully studied. This will probably lead to a revision of some of the features presented in the conceptual design.

As stated before, the ISS system is based on the Relational Database model but is not, within itself, a Database Management System (DBMS), therefore, many of the issues that would be handled by the DBMS are not presented within this thesis. Nonetheless, it is recognized that in order for the ISS to be fully implemented and functional in a concurrent multi-user environment, the underlying DBMS must be designed. This would include the research and design of ISS Security. Although this version of ISS does not implement system security, it provides the basic framework for incorporating multilevel security. It is suggested that this framework be reviewed and modified as necessary to support actual implementation of a secure ISS package.

Another DBMS research effort which lends itself to the current proposal is that of transaction recovery. This issue was considered briefly with the inclusion of the UNDO system command which enables the user to reverse the effects of the last committed transaction. Of course this is a simplistic view of a highly complex problem and it is suggested that research in this area constitute a complete thesis topic.

Both the non-DBMS and the DBMS issues discussed herein are viewed as being important and necessary research follow-on items. Furtherance of research on the suggested topics would certainly provide tangible benefits and lead to the completion of the design and implementation of ISS, the Integrated Software System.

LIST OF REFERENCES

1. Nishimura R., Analysis of the Relational Data Base Model in Support of an Integrated Application Software System, Master of Science Thesis, Naval Postgraduate School, December 1982.

2. Wyatt R. W., Multilevel Security for the ISS Mail Application, Master of Science Thesis, Naval Postgraduate School, March 1984 (In preparation).

3. Meyrowitz N. and Van Dam A., "Interactive Editing Systems: Part 2", ACM Computing Surveys, Vol 14, No 3, pp. 353-416, September 1982.

4. Stonebraker M. and Kalash J., "Timber: A Sophisticated Relations Browser", Proceedings of the Eighth International Conference on Very Large Data Bases, pp. 1-10, Mexico City, September 1982.

5. Stonebraker, M., and others, "Document Processing in a Relational Database System", ACM Transactions on Office Automation, pp 143-158, Vol 1, No 2, April 1983.

6. IBM Research Laboratory Research Report RJ3050, Automating Business Procedures with Forms Processing, by V.Y. Lum and others, March 1981.

7. Tsichritzis D. C., "OFS: An Integrated Form Management System", Proceedings of the Sixth International Conference on Very Large Data Bases, pp 161-166, Montreal, October 1980.

8. Stonebraker M., Rubenstein B. and Guttman A., "Application of Abstract Data Types and Abstract Indices to CAD Databases", Proceedings of Annual Meeting, Database Week: Engineering Design Applications, pp. 107-114, May 1983.

9. Lorie R. and Plouffe W., "Complex Objects and Their Use in Design Transactions", Proceedings of Annual Meeting, Database Week: Engineering Design Applications, pp. 115-122, May 1983.

10. IBM Research Laboratory Research Report RJ3503, A Relational Representation of an Abstract Type System, by D.L. Weller, June 1982.

11. Powel M. L. and Linton M. A., "Database Support for Programming Environments", Proceedings of Annual Meeting, Database Week: Engineering Design Applications, pp. 63-72, May 1983.

12. Traiger I. L., "Virtual Memory Management for Data Base Systems", ACM Operating Systems Review, Vol 16, No 4, pp. 26-48, October 1982.

13. SIGMOD 83, Proceedings of Annual Meeting, Abstract of Session, p. 134, May 1983.

14. Bourne, S.R., The Unix System, Addison-Wesley, p. 5, 1983.

BIBLIOGRAPHY

Day K. R., "Relational DBMS Development: an incremental Approach", Proceedings of Annual Meeting, Database Week: Databases For Business and Office Applications, May, 1983.

Furuta R., Scofield J. and Shaw a., "Document Formatting Systems: Survey, Concepts and Issues", ACM Computing Surveys, Vol 14, No 3, pp. 417-472, September 1982.

Gates W., "The Future of Software Design", Byte Magazine, Vol 8, No 8, pp. 401-403, August 1983.

Hancock L. and Krieger M., The C Primer, page 5, McGraw Hill, 1982.

Henderson P. B., Sciore E. and Warren D. S., A Relational Model of Operating System Environments, Dept. of Computer Science, SUNY Stony Brook, NY 11794, undated.

IBM Research Laboratory Research Report RJ3070, Human Factors Studies of Database Query Languaguages: A Survey and Asessment, by P. Reisner, March 1981.

IBM Research Laboratory Research Report RJ3132, The Capabilities of Relational Database Management Systems, by E.F. Codd, May 1981.

IBM Research Laboratory Research Report RJ3182, On Extending the Functions of a Relational Database System, by R.L. Haskin and R.A. Lorie, November 1981.

Meyrowitz N. and Van Dam A., "Interactive Editing Systems: Part 1", ACM Computing Surveys, Vol 14, No 3, pp. 321-352, September 1982.

SIGMOD 83, Proceedings of Annual Meeting, "The Database language GEM", C. Zsniols, pp 207-218, May 1983.

Spewak S.H., "A Pragmatic Approach to Database Design", Proceedings of the Sixth International Conference on Very Large Data Bases, p 151, Montreal, October 1980.

Ullman J. D., Principles of Data Base Systems, Computer Science Press, 1982.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	Professor Dushan Z. Badal, Code 52ZD Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
4.	Professor Gordon H. Bradley, Code 52BZ Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
5.	Curricular Office, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943	1
6.	DCCS, JMOP Room F-3-47 Russell Offices Canberra, ACT 2600. Australia	1
7.	LT G. L. Thompson, USN 1491 S. Carolina Avenue Avon Park, Florida 33825	2
8.	LCDR P. J. Harrison, RAN c/- DCCS, JMOP Room F-3-47 Russell Offices Canberra, ACT 2600. Australia	1