A Relational Algebraic Retrieval System for Microcomputers

by

Gyeongja Hong

B.S., Korea University, Korea, 1978

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by:

Richard A McBride

Major Professor

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Overview

A recent trend in the research on the architecture of database management systems is concerned with providing data independent interfaces for non-specialist users. Data independence implies that users may view the contents of a database as constrained only by a logical data organization rather than by its implementation on physical storage.

The relational model proposed by E. F. Codd [COD70] is a simple logical formalism for describing the organization of data. In this model, all information contained in a database is presented to user as tables. The tables have a number of columns and rows, with the rows corresponding to records, and the columns representing fields within the records. Interactions of users with such database descriptions can be isolated from any physical representation, and user requests can be formulated in high-level query languages.

A highly important feature of relational query language is the ability to operate on multiple records at once, instead of dealing just with one record at a time. This particular kind of set processing, called relational processing, entails treating whole relations as operands. As a result, application programmers are not forced to think and code in terms of iterative loops that are often unnecessary.

The process of retrieval in the relational model involves a set of operations. The fundamental operations on relations include:

- selection (which creates a subset of all the rows in a table);
- projection (which creates a subset of the columns of a table);
- join (which combines two tables). These operations, along with others that will be discussed in Chapter 2, together constitute the relational algebra. Each operation of the relational algebra takes either one or two relations as its operand(s) and produces a new relation as its result.

Several relational systems, such as MacAIMS[GOL70,STR71] and PRTV[TOD76], provide a query language that is directly based on relational algebra. Since the relational algebra was first developed, however, a number of other languages have been designed on the basis of different categories of query languages for relational database systems. These newer classes of query languages include relational calculus, mapping-oriented languages, and graphic languages. Most relational systems support one of these newer types of query languages rather than the relational algebra. The main difference is that the algebra is a procedural system, while the other relational languages are more non-procedural. That is, an expression in relational algebra gives a set of operations on relations and an order in which to perform them. The other languages simply express what the result of the computation should be, but not how to carry out the computation. However, the algebra is still important even though it is less "user-friendly" than other non-procedural languages. The basic reason for its importance is that it provides a yardstick of relational completeness for other languages [COD79].

A language is said to be relationally complete, if the language has the capability of supporting the operations of the relational algebra.

This report describes an implementation involving the query operation which is based on relational algebra.

1.2 Purpose

The key goals established for the implemented system are the following.

(1) To provide a fairly high-level query system based on relational algebra.

(2) To provide a means of understanding how relational operators can be combined to generate responses to queries.

(3) To utilize the procedural nature of relational algebra. Because the query expression of relational algebra specifies the order of operations, the implementation is easier. The implementation programmers are not forced to capture the user's intent, that is, what the user is trying to do.

Since the relational algebra is basically a retrieval language, there must exist a framework which provides for the creation and manipulation of relational database. DBASE III has been chosen because it supports a relational database model and has powerful features needed for data processing.

However, the dBASE III programming language does not support recursive functions, which are essential for a query parser of the implemented system. Turbo Pascal, therefore, has been employed for the analysis of queries. Since both Turbo PASCAL and dBASE III can run on IBM PCs and compatibles, the implementation is designed to be run on these microcomputers.

1.3 Organization of The Report

Each of the next three chapters is devoted to a separate theme. Chapter 2 is concerned with a review of the literature. After a survey of relational query languages, relational algebra is discussed in detail, and then a brief survey of the relational algebra research work which has been reported in the literature is presented.

Chapter 3 is concerned with the implementation. An algorithm is given showing how the algebraic query is processed in the system.

Finally, chapter 4 presents the conclusions and some directions for future work.

In addition to the main body of the report, appendices are provided. The appendices contain a user's guide for operating the implemented system, syntax diagrams of language, error messages, and the entire source code listing of the system.

CHAPTER 2

REVIEW OF LITERATURE

2.1 Introduction

Until recently, the end user of a database was the receiver of a report or someone who could perform a few limited operations by running special programs from a terminal, perhaps by pressing function keys. It is relatively recently that interactive systems have been made simple enough that the end users have been able to perform their own general-purpose database manipulation. These systems are called query languages [BUN84]. A query language is usually at a high level, non-procedural, and intended for a more casual user.

Since the development of the relational concept, a number of query languages have been developed to be used with a relational database. The query, or retrieval of information from the database, is perhaps the aspect of relational languages which has received the most attention. This chapter presents the relational algebra, which is one of approaches to the design of relational languages for expressing queries, and stresses the fundamental nature of the relational algebra as a component of the relational model. Before the discussion of relational algebra in detail, the essential terminology is defined, and then a survey of other types of relational query languages is presented. In addition, some examples are given to illustrate how queries to a relational database can be processed in the relational algebraic system. Then, a survey of database systems which support implementations of relational algebra is presented. Finally, several research areas related to a relational algebra are mentioned.

2.2 Definitions and Terminology

This section defines the fundamental terminology which is used throughout the report.

2.2.1 Relation

A relation is a named two-dimensional table, with a fixed number of rows. Columns of a relation are referred as attributes and each row of the relation is called a tuple. If there are n columns or n attributes, the relation is said to be of degree n.

Each attribute has a domain, which is the set of values that can appear in the attribute. For example, the domain of a Sex attribute consists of two values, namely, male and female. The domain of an Age attribute is all possible integers less than, say, 100.

A relation then has the following properties:

(1) There is no duplication of rows (tuples),

(2) Row order is insignificant,

(3) All of attributes are explicitly named and their orders are insignificant.

Different terms are used interchangeably, in this report, for some of these formal terms. A relation may be referred to as a file, tuples as records, and columns as fields. These equivalents of formal names are used in business data processing environments. dBASE also uses these equivalent terms.

2.2.2 "Union-compatible"

This is an essential condition which must be satisfied for several operations ('difference', 'intersection', and 'union') in a relational algebra. The two operand relations for these operations must be of the same degree and corresponding attributes in the two relations must be defined on the same domain.

2.2.3 Relational Database, Base relation, and Derived relation

A relational database is a collection of time-varying tabular relations of assorted degrees defined on a given set of domains. "Time-varying relations" means that the set of tuples appearing in a given relation varies with time, that is, it changes as tuples are created, destroyed, and updated [COD79].

A base relation is a relation that has independent existence in the sense that no base relation is completely derivable from any other base relation(s). Each base relation is represented in storage by a distinct stored file. A base relation can be created at any time in dBASE by executing the command CREATE [ASH84].

Derived relations are the relations that do not have any existence in its own right, but can be completely derived from the base relations. It is this kind of relation which is normally produced by the evaluation of query expression [COD79].

2.3 Relational Query Languages

A query language is defined as a high-level computer language for the retrieval and modification of data held in databases or files[SAM81]. In this section, the four different approaches for relational query languages are described briefly.

2.3.1 Relational Calculus

Relational calculus languages constitute an applied first-order predicate calculus [COD79]. Relational calculus-based languages require the user to invent a variable to represent a tuple of relation, and to state a predicate which defines those tuples which are of interest in a particular query. Examples of such languages include DSL-ALPHA [COD71] and QUEL [HEL75]. This type of language is less procedural than relational algebra. For successful operation with these languages, however, the user must be proficient in predicate calculus in which the operation with quantifiers is particularly difficult. Query texts composed by one user may be incomprehensible to another user.

2.3.2 Mapping-oriented Language

The majority of query languages available today fall into this class. Some well known examples of these languages are SQUARE [BOY75] and SQL[CHA74]. Their most distinctive feature is that they use more English-like statements. The fundamental operation is called a mapping and has a definite syntax. The user describes the query by expressions based on "mappings" rather than by variables and quantifiers. Consequently, the queries are simpler and more concise than their equivalents in the relational calculus.

The mapping in SQL is represented syntactically as a SELECT-FROM-WHERE block. This is used to SELECT attributes FROM one or more relations WHERE the tuples of the relations satisfy certain conditions. In general, the result of a mapping may be used in the specification of another mapping. This process of "nesting" mappings inside each other makes it possible to express queries of great complexity.

2.3.3 Graphic Languages

In graphic or pictorial query languages, the user states his query not by a conventional linear syntax, but by making choices or filling in blanks on a graphic display. An example of this class of languages is Query-By-Example (QBE) [ZLO77] which is commercially available. In contrast to mapping-oriented languages such as SQL, QBE makes relations directly visible as objects (tables) to be manipulated on the screen, and user moves the cursor freely along the rows and columns of the tables.

2.3.4 Relational Algebra

Data manipulations in a relational algebraic language are carried out by executing algebraic operations on the relations. Actually, algebraic languages occupy an intermediate position between procedural and non-procedural query languages, since the user is required to specify the actual sequence of relational operations to be performed.

2.4 Operations of Relational Algebra

Relational algebra uses a set of operations defined on relations. Each operation takes one or more relations as its operand(s) and produces a new relation as its result. Since the result of a relational algebra operation is a relation, that relation in turn may be subjected to further algebraic operations. Operands of any given operation can thus be specified either as simple relation names or as expressions that evaluate to relations. In other words, relational algebraic expressions can be nested to any depth, with parentheses used, as needed, to remove ambiguities.

The algebra consists of two groups of operators: the traditional set operators union, intersection, difference, and Cartesian product; and special relational operators selection, projection, join and division[DAT81]. This section introduces these two groups of operators in subsequent subsections. For reference, a complete BNF syntax for the implemented version of the algebra is given in Appendix B.

2.4.1 Traditional Set Operations

Since relations are sets, the usual set operators such as UNION, INTERSECTION, and DIFFERENCE are applicable. However, they are constrained so that they are applied only to pairs of union-compatible relations (see p.7). This constraint guarantees that the result is a relation. CARTESIAN PRODUCT is applicable without this constraint.

UNION

The 'union' of two union-compatible relations A and B, denoted A UNION B, is a new relation in which all of its tuples belong to either A or B or both.

Example. Consider the two relations of Fig.2.1. The union of ARTCLASS and MATHCLASS will result in the relation in Fig.2.2(a). Note that the tuple, [Mary, 12, 7], which occurs in both relations, is not duplicated in the union.

INTERSECTION

The 'intersection' of two union-compatible relations A and B, denoted A INTERSECT B, is a new relation which has all of its tuples belonging to both A and B.

Example. Fig.2.2(b) shows the intersection of ARTCLASS and MATHCLASS relations.

Name	Room_No	Grade
AMY	12	7
JONES	15	8
MARY	12	7
ROBIN	14	8

(a) Relation ARTCLASS

Name	Room_No	Grade
DAVID	9	7
MARY	12	7
TOM	12	8

(b) Relation MATHCLASS

Fig. 2.1 Two relations

Name	Room_No	Grade
AMY	12	7
JONES	15	8
MARY	12	7
ROBIN	14	8
DAVID	9	7
TOM	12	8

(a) ARTCLASS UNION MATHCLASS

Name	Room_No	Grade
MARY	12	7

(b) ARTCLASS INTERSECT MATHCLASS

Name	Room_No	Grade
AMY	12	7
JONES	15	8
ROBIN	14	8

(c) ARTCLASS MINUS MATHCLASS

Fig. 2.2 Results of some relational operations

DIFFERENCE

The 'difference' between two union-compatible relations A and B, denoted A MINUS B, is a new relation which contains only those tuples which belong to A but not to B.

Example. The difference of ARTCLASS and MATHCLASS is shown in Fig.2.2(c).

Name	Room_No	Grade	Sex
AMY	12	7	F
DAVID	9	7	M
JONES	15	8	M
MARY	12	7	F
ROBIN	14	8	F

(a) Relation STUDENT

Sname	Class
AMY	ART
AMY	MATH
DAVID	MATH

(b) Relation ENROLLMENT

Fig. 2.3 STUDENT and ENROLLMENT relations

Name	Room_No	Grade	Sex	Sname	Class
AMY	12	7	F	AMY	ART
AMY	12	7	F	AMY	MATH
AMY	12	7	F	DAVID	MATH
DAVID	9	7	M	AMY	ART
DAVID	9	7	M	AMY	ART
DAVID	9	7	M	DAVID	MATH
JONES	15	8	M	AMY	ART
JONES	15	8	M	AMY	MATH
JONES	15	8	M	DAVID	MATH
MARY	12	7	F	AMY	ART
MARY	12	7	F	AMY	MATH
MARY	12	7	F	DAVID	MATH
ROBIN	14	8	F	AMY	ART
ROBIN	14	8	F	AMY	MATH
ROBIN	14	8	F	DAVID	MATH

STUDENT TIMES ENROLLMENT

Fig. 2.4 Example of Cartesian Product operation

CARTESIAN PRODUCT

The 'Cartesian product' of two relations A and B, denoted A TIMES B, is the concatenation of every tuple of A with every tuple of B. The Cartesian product of relation A, having m tuples, and relation B, having n tuples, contains m*n tuples.

Example. Consider the relations STUDENT and ENROLLMENT in Fig.2.3. STUDENT has five tuples and ENROLLMENT has three tuples. Therefore, the Cartesian product of two relations produces a new relation which has fifteen tuples. The result is shown in Fig.2.4.

2.4.2 Special Relational Operations

SELECTION

This operation returns a new relation by taking a horizontal subset of a relation, i.e., all of the tuples of the result relation which satisfy a given condition. Selection is denoted by specifying the relation name, followed by the keyword WHERE, followed by a condition involving attributes. Fig.2.5 gives some examples of selection. In Fig.2.5(a) all rows of STUDENT relation in which the grade is 7 are selected for inclusion in the newly created relation. In fig.2.5(b), only those tuples are selected from the STUDENT relation where Sex is female.

Name	Room_No	Grade	Sex
AMY	12	7	F
DAVID	9	7	M
MARY	12	7	F

(a) STUDENT WHERE (Grade = 7)

Name	Room_No	Grade	Sex
AMY	12	7	F
MARY	12	7	F
ROBIN	14	8	F

(b) STUDENT WHERE (Sex = 'F')

Fig. 2.5 Examples of Selection operation

PROJECTION

'Projection' forms a vertical subset of a relation by extracting specified attributes and removing any redundant duplicate tuples in the resulting relation. For example, the projection of STUDENT on Name and Grade attributes, denoted with brackets as STUDENT [Name, Grade], is shown in Fig.2.6(a). Another example of projection appears in Fig.2.6(b). Note that the redundant tuple [12, 7] was eliminated in Fig.2.6(b).

Name	Grade
AMY	7
DAVID	7
JONES	8
MARY	7
ROBIN	8

STUDENT [Name, Grade]

Room_No	Grade
12	7
9	7
15	8
14	8

STUDENT [Room_No, Grade]

Fig. 2.6 Examples of Projection operation

JOIN

The 'join' operation takes two relations A and B as operands. A new relation is formed by concatenating a tuple of A with a tuple of B wherever a given condition holds between them. The given condition must compare attributes from the two relations which arise from a common domain. There are many possible versions of a join, for example, an 'equijoin', a 'greater than join', a 'not equal join', and so on. The most common form of join is the 'equijoin', where the attribute values are compared for equality. The result of an equijoin always contains identical attributes. If all redundant attributes are removed, the join is referred as a 'natural join'. Fig.2.7 shows the natural join of relations STUDENT and ENROLLMENT over Name, denoted as STUDENT JOIN (Name = Sname) ENROLLMENT.

Name	Room_No	Grade	Sex	Class
AMY	12	7	F	ART
AMY	12	7	F	MATH
DAVID	9	7	M	MATH

STUDENT JOIN (Name = Sname) ENROLLMENT

Fig. 2.7. Result of a Natural Join

DIVISION

The 'division' operator divides a dividend relation A of degree m + n by a divisor relation B of degree n, and produces a relation of degree m. Consider the first m attributes of A as a single composite attribute X, and the last n as another, Y. A may then be thought of as a set of tuples <x,y>. Similarly, B may be thought of as a set of tuples, <y>. Then the result of dividing A by B, denoted A DIVIDEBY B, is the set of <x> tuples such that for all <y> tuples in B, the tuple <x,y> is in A.

Therefore, this operation is sometimes useful in expressing queries which contain the word "all". However, since it can be expressed in terms of the other algebraic operators, the division operator does not extend the logical power of the language. As an example, consider the relation R in Fig.2.8. It has Name and Class attributes. Each of the divisor relations D1, D2, and D3 have one attribute, namely, Class. The division of R by D1 yields just one tuple, MARY, because only Mary takes the three classes listed in D1. R divided by D2 yields both David and Mary because both of them take English and math classes. Since all students whose names are in R take English class, the division of R by D3 yields a relation with all of the names.

Name	Class
AMY	ART
AMY	ENGLISH
DAVID	MATH
DAVID	ENGLISH
JONES	ART
JONES	ENGLISH
MARY	ART
MARY	ENGLISH
MARY	MATH
ROBIN	ART
ROBIN	ENGLISH

Dividend relation: R

Class
ART
ENGLISH
MATH

D1

Class
ENGLISH
MATH

D2

Class
ENGLISH

D3

Divisor relations

Name
MARY

R DIVIDEBY D1

Name
DAVID
MARY

R DIVIDEBY D2

Name
AMY
DAVID
JONES
MARY
ROBIN

R DIVIDEBY D3

Result relations

Fig. 2.8 Examples of Division operation

2.5 Example Queries in Relational Algebra

Relational algebra is one means of representing operations to be performed on a database. The operations can be combined to act upon a database and generate responses to queries. In this section, several such applications of relational algebra are illustrated.

The illustrative examples used in this section and the next chapter are based on the following database relations:

EMP (Emp_id, Name, D_name, Salary)

DEPT (D_name, Floor)

USAGE (D_name, Item, Quantity)

SUPPLY (Item, Supplier)

The EMP relation has a tuple for every employee, giving a person's identification number, name, department, and salary. The DEPT relation gives the name and the floor of each department. The USAGE relation tells which items each department uses and what quantities of each item are consumed. The SUPPLY relation gives items and the companies which supplies them. Figures [2.9 - 2.12] show the sample data which will be assumed to populate in each of the above relations.

Emp_id	Name	D_name	Salary
11	HOOVER	ADMIN	30000
12	BUSH	SALES	23000
13	ELDER	PRODUC	21000
14	GIBSON	PRODUC	22000
15	COOPER	SALES	25000
16	FRANK	PRODUC	20000
17	DOLE	ADMIN	20500
18	JOHNSON	PRODUC	17000
19	ADAMS	SALES	19000
20	IRWIN	PRODUC	20000
21	FRANK	PRODUC	18000

Fig.2.9 Sample data for EMP relation

D_name	Floor
ADMIN	2
PRODUC	1
SALES	2

Fig. 2.10 Sample data for DEPT relation

D_name	Item	Quantity
SALES	COMPUTER	10
SALES	PRINTER	2
SALES	COPIER	2
ADMIN	COMPUTER	3
PRODUC	COMPUTER	20
PRODUC	PRINTER	5

Fig. 2.11 Sample data for USAGE relation

Item	Supplier
COMPUTER	IBM
COPIER	XEROX
PRINTER	ICL

Fig.2.12 Sample data for SUPPLY relation

Examples of queries that can be made on this data are given below. For each, the associated relational algebra expression is given immediately following the question and the relation which the query produces is also shown.

Q1. Find the names of all employees.

EMP [Name]

This is a simple projection which takes the value of the Name attribute of every tuple in the EMP relation. The result is:

Name
HOOVER
BUSH
ELDER
GIBSON
COOPER
FRANK
DOLE
JOHNSON
ADAMS
IRWIN

Q2. Find the id number and name of employees whose salary is greater than $22,000.

```
EMP WHERE (Salary > 22000) [Emp_id, Name]
```

This is a combination of two operations. The first operation selects the first, second, and third tuples, because their values in column 4 (Salary) is greater than $22,000. The projection operation then leaves only the first and second columns, Emp_id and Name, so the resulting table is:

Emp_id	Name
11	HOOVER
12	BUSH
15	COOPER

Q3. Find the department names for departments that use all items.

```
USAGE [D_name, ltem] DIVIDEBY (SUPPLY [ltem])
```

This expression performs a division in which the dividend is the projection of USAGE over department name and item; the divisor is the projection of SUPPLY over Item. This query yields the relation:

D_name
SALES

Q4. Find the names of those employees who work in some department on the second floor.

```
EMP JOIN (D_name = DEPT.D_name) DEPT WHERE (Floor = 2) [Name]
```

This is a more complicated type of query involves taking the natural join, then selecting tuples from this relation, and then projecting them onto the desired attribute. Joining is needed since the employee names and the department's location must come from different tables. The result is:

Name
HOOVER
BUSH
COOPER
DOLE
ADAMS

Note that the relation name DEPT was added as the prefix in the condition. When multiple relations possess identical attribute names, a prefix is required to remove any ambiguity in the reference to an attribute whose name is used in more than one relation.

Q5. Find the suppliers that supply all the items used by Gibson's department.

```
EMP WHERE (Name = 'GIBSON') JOIN (D_name = USAGE.D_name) USAGE
    JOIN (Item = SUPPLY.Item) SUPPLY  [Item, Supplier]
```

This expression involves the three relations and performs four operation steps. First, Gibson's tuple is selected from the EMP relation and is joined to the USAGE relation. Then the result is joined to the SUPPLY relation, and finally, the projection is taken to leave Item and Supplier columns. This produces the table:

Item	Supplier
COMPUTER	IBM
PRINTER	ICL

2.6 Implemented Relational Algebra Systems

A number of early projects in relational database management adopted the approach of implementing the relational algebra directly. Perhaps the earliest of these was the MacAIMS (Mac Advanced Interactive Management System), developed at MIT[GOL70]. The MacAIMS system was implemented on MULTICS (MULTIprogrammed Computer System) utilizing the large, directly addressable virtual memory and flexible access control capabilities of MULTICS [DEN65]. MacAIMS made a contribution to the field of data independence by enabling different relations to be stored in different forms and converted to a canonical form for comparison when necessary.

Another system virtually identical to MacAIMS, called RDMS (Relational Data Management System), was developed at MIT[STE74]. MIT's RDMS has been extensively used by many departments at MIT ever since it became operational in 1971. This system supports a language based on the relational algebra and provides extensive report generation and computational facilities.

Another early algebra-oriented system is the Relational Data Management System (RDMS) of General Motors [WHI72]. GM's RDMS is a prototype system which was intended to demonstrate the feasibility of developing a generalized information system. RDMS implements not only the operators of the relational algebra, but also a number of other set-oriented operators such as SORT, GRAPH, and HISTOGRAM. The current version of this system is called REGIS (RElational General Information System) [JOY76].

An important project in the implementation of relational algebra is located at the IBM Scientific Center in Peterlee, England. The Peterlee system was first called IS/1 (Information System 1), and later renamed PRTV (Peterlee Relational Test

Vehicle) [TOD76]. PRTV supports ISBL (Information System Base Language), which is a query language based on the relational algebra. However, ISBL doesn't support such user requirements as report generation and aggregate functions (i.e., max, min, avg, sum, count). PRTV allows the user to extend the capabilities of ISBL by writing PL/1 application programs and linking them to the base system. The most important feature of PRTV is perhaps its optimizer. First it transforms an ISBL expression into an equivalent expression which can be more efficiently evaluated [HAL74a]. Next it attempts to find an optimal set of access paths for evaluating the transformed expression. This is done by considering the estimated costs of various alternative access paths [VER76].

In 1975 a relational system named MAGNUM was implemented on a minicomputer, PDP-10. MAGNUM is a commercially available system which was developed by Tymshare, Inc., California [JOR75]. The MAGNUM query language is a variation of the relational algebra. It does not have the complete capabilities of relational algebra as defined in this chapter. For example, neither join nor division is possible. However, MAGNUM does provide extensive computational and report generation facilities.

A more recent project dealing with the implementation of relational algebra was located at McGill School of Computer Science, Canada. The ALDAT (ALgebraic DATa) project at McGill has produced several versions of the relational algebra system, MRDS. The latest version of MRDS was built on the U.C.S.D.P-system on an Apple II [MER83]. In addition to generalizing the relational algebra, Aldat introduces the domain algebra [MER76], a framework for algebraic operations on attributes which permits arithmetic operations and aggregate functions.

Since the earliest relational language proposed was relational algebra, it is natural that much of the earliest implementation work was directed toward this language. However, the relational algebra has a disadvantage from the end-user's point of view. For end-users, complex query formulations in relational algebra may be difficult. Because of this, today it is hard to find such systems that support relational algebraic query languages directly. Instead, in most of today's relational database systems, the operators of relational algebra are embedded in very high-level and friendly user interfaces.

2.7 Work Related to Relational Algebra

The problem of optimizing the execution of relational algebraic systems has attracted a great deal of interest. Hall and Todd have developed techniques for transforming a user request, expressed as a sequence of operations based on the relational algebra, into an equivalent expression which can be evaluated more efficiently [TOD74; HAL74a; HAL75]. This type of transformation is called an algebraic transformation. Hall has also developed techniques for identifying and removing redundant common subexpressions from a user's query [HAL74b]. Techniques for finding an optimal set of access paths for evaluating a user's query, on the basis of CPU time estimates of all possible access paths, were later developed by Verhofstad and Todd[VER76]. All of the above-mentioned techniques have largely been intergrated into PRTV system.

Smith and Chang have applied techniques of automatic programming to transform relational algebra expressions into equivalent sets of operators amenable to parallel processing [SMI75]. Gotlieb has published a study of various algorithms for implementing the join operator [GOT75]. Percherer described techniques for finding the optimal ordering of relations for computing products of relations [PEC76]. Yao and Dejong have developed a cost model for evaluating several techniques for computing joins of relations [YAO78].

Another area currently receiving much attention is the extension of the relational algebra to handle "null values", that is, entries in tuples that represent unknown, irrelevant, or inconsistent values. Enhancements to the algebra to deal with nulls have been proposed by many researchers[COD79; LAC76; VAS79; MAI80]. Codd, for example, suggests two variants of the join operator called 'maybe join' and 'outer join'. In the maybe join, tuples are joined, not on the basis of some condition being true, but rather on the basis of the condition having the unknown truth value. In the outer join, tuples in one relation having no counterpart in the other relation appear in the result concatenated with an all-null tuple. However, there are still problems with these approaches and there is no wholly satisfactory answer at this time.

2.8 Summary

Although, only query facilities are discussed in this chapter, most query languages provide a variety of facilities in addition to a query capability. The relational algebra is fairly limited when compared with other query languages. For example, it has no aggregate operators such as average or sum, and there are no facilities for the insertion, deletion, or modification of tuples. This is due to the fact that, as Codd has stated, "the relational algebra is not intended to be a standard language, to which all relational systems should adhere" [COD82,p.112]. Instead, it was proposed as a benchmark for comparatively evaluating query languages. That is, a language without at least the expressive power of relational algebra was deemed inadequate.

In practice, because of the aforementioned limitations, the system that supports a query language based on relational algebra should provide additional features. This is why the relational algebra is implemented on the top of the dBASE database management system in this project. For the operations other than retrieval, the user can interact directly with the dBASE system and use the available facilities which it provides.

This chapter provided an overview of relational algebra as the basis for a high-level query language. In the next chapter, one version of a relational algebraic query system on the microcomputers will be described. This system provides the capabilities of relational algebra which were presented in this chapter.

CHAPTER 3

SYSTEM IMPLEMENTATION

3.0 Introduction

This chapter describes the implementation of a relational algebraic retrieval system (RARS). RARS was implemented in Turbo Pascal and the dBASE III programming languages, and runs on IBM PCs and compatibles under DOS version 2.0 or higher. As a data base manager, the dBASE III is used because it contains all access and storage capabilities. Since part of RARS was written in Turbo Pascal, execution requires a minimum of 512K bytes of memory and a hard disk to avoid disk swappings.

The objective of RARS is to provide a facility for retrieval using relational algebra. The system offers the full range of relational algebraic operations which were defined in the previous chapter. This chapter presents the tools and algorithms used in the system.

3.1 Implementaion Tools

3.1.1 dBASE III

dBASE III [ASH84] is a relational database management system which is designed to be used on IBM PCs. As a relational database management system, dBASE III organizes data elements in a two-dimensional table consisting of rows and columns, where each row is a data record and each column is a data field. In dBASE III, information in the database can be processed in two ways. One way to handle the information in a database file is the method of interactive command processing. Information in the database may be manipulated interactively by commands entered from the keyboard. After each command is entered, results are displayed on an output device, such as a monitor or a printer. Another method for processing information in dBASE III is batch command processing. Processing tasks are defined in a set of command procedures. These commands are then executed as a batch. The collection of commands is stored in a command file, which is considered to be a computer program.

In dBASE III, various types of disk files can be used for holding different kinds of information. A database file, equivalent to the relation in a relational database, contains the data structure and all the data records. The data structure includes the number of fields, each field's name, data type, length or width, and the number of decimal places, if any. The number of fields and each one's name, data type, and size are established with the CREATE command, and can be changed with the MODIFY STRUCTURE command. The number of records is determined as they are added. Format files, label files, and report files are used to store the details needed for generating custom reports. A command file stores the collection of commands that are to be processed in the batch processing mode. The command file can be created by dBASE III's standard text editor or by any word processor that has a nondocument mode. An index file provides the necessary working spaces for an indexing operation. With an index file, a set of data can be used in a logical order rather than the order in which the records were entered in the database. A memory file stores the contents of the active memory variables.

Memory variables represent temporary memory locations that can hold computational results which may be used again for later processing. A text file can be used to save text that can be shared by other computer programs.

The dBASE III command set constitutes a full capability procedural language with many features found in modern high-level programming languages. It has 35 functions and more than 100 English-like commands with hundreds of variations. Its design encourages good structured programming practices. For example, dBASE III does not contain a GOTO statement which causes program control to jump to a different place in the program. Instead, there are commands for controlling the execution of command files, including DO WHILE...ENDDO, IF...ELSE...ENDIF, DO CASE... ENDCASE. In addition, command files may be called with parameters, making it possible to build up suites of procedures which can be used in different aspects of applications without duplicating effort. The commands fall into eight areas: program and data creation, data display and editing, record pointer positioning, file manipulation, memory variable manipulation, command file control, dBASE III system control parameter modification, and peripheral device control.

The functions fall into five areas: date and time operations, character manipulation, mathmatical operations, character and numeric conversions, and specialized tests.

dBase III can read or write a ASCII text file in fixed length or comma-delimited format. This facility provides the necessary links for information exchange between the dBASE III program and the Turbo Pascal program in this project. In addition, dBASE III can run any DOS command or any .COM, .EXE, or .BAT file as long as sufficient RAM is available. To take advantage of this ability, a compiled version of the Turbo Pascal program, which can have a file extention .COM, is employed within a dBASE III program to carry out a certain task in the project.

3.1.2 Turbo Pascal

Pascal is a block-oriented, structured programming language developed by Professor Niklaus Wirth as a tool for teaching good programming practices. The first Pascal compiler was made in 1970 and designed to work on a large mainframe computer. By the late seventies it had become extremely popular, first with university programmers and then with programmers in the business world. The language as Wirth defined it (which later became standard Pascal [JEN74]) was severely limited in many ways. It was not suitable for any kind of extensive interactive programming. It had very little file I/O and no provision for making calls to the operating system or otherwise operating computer peripherals. As a result, the vendors, who sold Pascal compilers commercially, began to expand the features of the compilers they sold to overcome these weaknesses.

Turbo Pascal [BOR85] is a compiler designed solely for use with a personal computer. In addition to converting Pascal programs into machine code, it acts as a user interface between edit and compilation, and includes a useful editor which is closely linked to the compiler. Turbo Pascal's most distinguishing feature is its speed. One of the reasons for this is that it has no link step to produce excutable code.

Turbo Pascal follows Standard Pascal closely, with only a few minor differences. The only significant variation from Standard Pascal is that the procedures GET and PUT are not implemented. Instead, READ and WRITE are extended to handle all types of files, not just text files.

While meeting most of the Standard Pascal criteria, Turbo Pascal also has many extensions; the outstanding extensions are in the declaration area. Constant, Type and Variable sections can be in mixed order, and constants can be assigned to be of any previousely defined type. This capability often saves time and code-space when initializing data structures.

Turbo Pascal offers a new basic type called string. A string is declared with its maximum length. Thus a string[80] can be less than or equal to 80 characters in length. There are built-in routines for searching, comparing, extracting and concatenating strings.

The filing facilities include random access to files. The way a file is referenced is by means of a file variable. The first operation on any file variable must be an ASSIGN, which binds the DOS name for a file to that file variable.

Turbo Pascal comes with some built-in screen control procedures. These include cursor positioning, insert line, delete line, clear screen, and normal/bright intensity control. These facilities make it easy to write screen-oriented programs.

Turbo Pascal contains a primitive facility for writing in-line machine code. The INLINE statement accepts a series of data elements and places them directly into the code being compiled. Using this feature, it is possible to write interrupt handlers or optimize specific routines for speed. This is again a non-standard but useful feature.

Some of Turbo Pascal's useful extensions involve bit-level operations on integer values which occupy two bytes in memory. Two new operators, SHL and SHR, allow shifting integer values left and right a specified number of bits. The procedures HI and LO return the upper and lower bytes of an integer, while SWAP exchanges the upper and lower bytes.

Turbo Pascal also includes a number of compiler directives that can be conveniently embedded directly into the program. A compiler directive is a special command that instructs the compiler to perform a task.

Overall, Turbo Pascal provides a complete and practical compiler for small to medium projects.

3.2 Query Processing in RARS

RARS starts the excution by displaying a simple menu. The user can query the database repeatedly until he chooses to exit from the menu. This menu provides two exits: one to dBASE III and the other to the operating system.

RARS can be broken down into two phases: a parsing phase for accepting a query expression and converting it to an intermediate form, and an evaluation phase for processing the intermediate

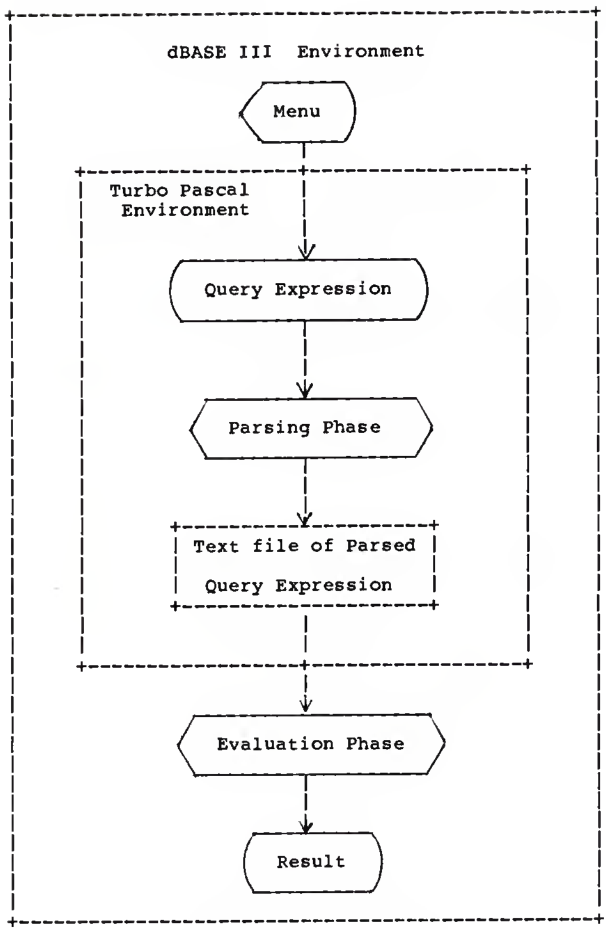

Fig. 3.1 System flow chart

form. The parsing phase is performed by a Turbo Pascal program which acts on the input query expression, and converts this expression into a parsed form. The expression is held, in its parsed form, in a text file. After processing is done, the parser passes this text file on to the evaluation phase. In block chart form, the overall processing flow appears as shown in Fig. 3.1.

3.2.1 Parsing Phase

The first of the two phases in processing a query is the parsing phase. This phase can be further divided into three modules: acceptance, parser, and conversion to Polish postfix form.

3.2.1.1 Acceptance module

The acceptance module is formulated to allow a query expression to be entered from the console. A <^Z> is recognized by the module as a special character that terminates a query text. The use of the <^Z>, instead of <CR>, allows the user to enter a query expression on more than one line.

The acceptance routine begins by prompting a user to enter the query. For reference, a skeleton of the syntax of a query expression is displayed on upper part of the screen. The routine then begins to accept typed characters and decides if they are acceptable. Acceptable characters are all of the printable ascii characters plus a few control characters. The <Enter> control character marks the end of the current line entry and moves cursor to the next line. <BS> backspaces over one character and deletes the character there. <CTRL Z> marks the end of the query and also the entry procedure. Any character that is not printable and not a recognized control character is just ignored.

If the input query is not an empty string, then it is passed to the next module, the parser.

3.2.1.2 Parser module

The parser module acts on a query expression and uses syntax equations to convert that expression into a parsed form. The algorithm for this module is based on a recursive chain of subtasks, since the syntax for a query expression is recursive.

A query expression is represented by a stream of lines of text and this stream is scanned for proper expression syntax. The result (legal or illegal) is indicated by setting an indicator. As the expression is scanned, words in the expression are matched against items in the syntax equations. If they are matched successfully, then new code is generated so that it can be used directly in the evaluation phase. For example, a Boolean operator 'AND' will be converted to '.AND.' since dBASE III does not recognize the former notation.

If a relation name is encountered, the module checks for the existence of that file. When a condition expression for a join or selection operation is scanned, the routine also constructs a special string which contains information about the condition expression. Thus, in the evaluation step, dBASE III can immediatly check for the validity of the condition expression without scanning the condition expression all over again. As an example, if the following condition expression is encountered:

(Age > 20 and Sex = 'M')

then, the generated special string will be:

(AGE:N)(SEX:C)

Here, 'N' denotes the number type constant and 'C' denotes the charater or character string type constant. Then dBASE III checks to see if the field type of AGE is 'N' and if the field type of SEX is 'C'.

If any syntax errors are detected during the parsing process, the module displays an error message and asks the interactive user if he wants to enter a query again. Otherwise, it passes an array of parsed expression items to the next module.

3.2.1.3 Conversion to Polish postfix form module

After a query expression is recognized as legal and is broken down into fragments, then the expression is converted into Polish postfix form. The purpose of this conversion is to eliminate problems associated with an expression which contains parentheses. When evaluating an expression that has been translated into Polish postfix form, there is no need to decide the order of evaluation.

The algorithm for converting a query to Polish postfix form uses the stack data structure. In this module, the parsed expression items are classified into three groups: operators, operands and parentheses. Each operator represents one of the eight relational algebra operations. Other expression items, with the exception of operators and parentheses, are considered as operands.

The following is a high-level description of the actions taken when the routine encouters each expression fragment (i.e., subexpression).

1. Append an operand to the output expression when it is encountered.
2. Push a '(' onto the stack.
3. When a ')' is encountered, pop operators off the stack, append them to the output expression until the matching '(' is hit. Then remove '(' from the stack.
4. When an operator is encountered and an operator is on the top of the stack, pop operators from the stack, append them to the output expression. Stop popping when '(' is hit. Then push the new operator onto the stack.
5. When the terminator ';' is encountered, pop the remaining contents of the stack and append to the output expression.

As an example, the reader can observe the action of the algorithm on the following query expression.

USAGE [D_name, Item] DIVIDEBY (SUPPLY [Item]).

The stored expression fragments after the parser module has parsed this expression and their corresponding types in the conversion module are:

index	expression item	item class
1	USAGE	operand
2	*P	operator (Projection)
3	D_NAME,ITEM	operand

4	*D	operator (Division)
5	(	left parenthesis
6	SUPPLY	operand
7	*P	operator (Projection)
8	ITEM	operand
9	)	right parenthesis
10	;	terminator

The stack and the output expression array are initially empty. The following is the series of actions on the above expression during the conversion process. For simplicity, S means a stack and PE means the output Polish expression.

input stream	stack content	output	action
USAGE	empty	USAGE	append operand to PE
*P	*P		push new operator onto S
D_NAME,ITEM	*P	D_NAME,ITEM	append operand to PE
*D	empty	*P	pop operator, append to PE
	*D		push new operator onto S
(	(, *D		push '(' onto S
SUPPLY	(, *D	SUPPLY	append operand to PE
*P	*P, (, *D		push new operator onto S
ITEM	*P, (, *D	ITEM	append operand to PE
)	(, *D	*P	pop operator, append to PE
	*D		remove '(' from S
;	empty	*D	pop remaining operators and append them to PE
	empty	;	append terminator to PE

The output array now contains the resulting Polish postfix form expression:

index	1	2	3	4	5	6	7	8
	USAGE	D_NAME,ITEM	*P	SUPPLY	ITEM	*P	*D	;

After the conversion to the Polish form is done, the module first outputs the original query text, entered by the user, to the text file for future use. It then appends the converted expression items to the text file. When the last part of the parsing phase is done, this text file is passed to the evaluation phase to produce the desired query result.

3.2.2 Evaluation Phase

After the Turbo Pascal program has finished the parsing process, control returns to the dBASE III command file, in order to evaluate the query expression.

There are three database files that always reside in the system. One file is needed for pulling the text file, which was produced by the Turbo Pascal program, into a database file.

Another file is needed to hold information about necessary changes of the attribute names during the evaluation process. By the definition of a relation, which is described in the previous chapter, a derived relation also must have attributes which are explicitly and uniquely named. After the union, intersection and difference operations, the same attribute names which appear in the first relation are generated for the derived relation. In this case, any name of an attribute, which belongs to the second relation and has a different name from the one in the first relation, is lost. The attribute names of the resulting relation now correspond to the attribute names used in the first relation. A problem arises when such a name is used later in the rest of the query expression.

For example, suppose the user entered the following query expression: STUDENT UNION TEACHER [ID_Num].
If STUDENT relation has attributes Number, Name and Major, and TEACHER relation has attributes ID_Num, Name and Class, then the relation resulting from the union operation will have the attribute name set Number, Name and Major. But the attribute name 'ID_Num', which no longer exists in the file derived from the union operation, was used in the projection list. In this case, the attribute 'ID_Num' must be considered as the attribute 'Number'.

The Cartesian product and join operations are only applicable when the two relations involved have no attributes with the same names. To allow these operations to be used as intended, the attribute names are checked before the processing. If the second relation has any attribute names which also appear in the first relation, these names will be changed automatically. The new names generated for these attributes will have a prefix which is the initial of the second relation name. For example, if the attribute name 'Item' in the second relation 'SUPPLY' is to be changed, the new name will be 'SItem'. In this case, again the system must know the name change has happened to avoid problems when the original name is used later in a query.

Attribute name changes can also happen during the equijoin operation. One of the two attributes appearing in the equijoin condition expression is not shown in the result relation, in order to avoid duplicate columns or fields. But it is possible for the user to state this attribute name after the equijoin operation. For example, if a query expression is:

STUDENT JOIN (Name = Teacher) STAFF [Number, Teacher].

Then, the relation resulting from the join operation would not have the Teacher column, since its contents would be a duplicate of the Name column. Therefore, the attribute name 'Teacher' appearing in projection list would be determined as 'Name'.

To take care of all of the above situations, a special database file is located in RARS. Whenever a change is made to the attribute name, the system reports it to this database file. Each record of the file has the original relation and attribute names and a new attribute name. This file then serves as a lookup table for the relation and attribute names which do not appear in the currently involved relations.

The last of the three files residing in the system is used by RARS for reporting an error to the user. Each record in this file has a description of each operation that has been done during the processing of a query. For example, if a user entered the query:

EMP WHERE (Salary > 2200) [Emp_id, Name]

then, after the evaluation is done the file would have following records:

record number	content
1	TEMP01 := EMP WHERE (Salary > 22000)
2	TEMP02 := TEMP01 [EMP_ID,NAME]

If an error is detected during the evaluation, the system displays an error message with the evaluation steps done so far. Thus, the user easily can locate the step at which the error occurred.

The evaluation phase starts by clearing the above three files. Then the text file of parsed expression items is read into a database file. With the existence of this file, the actual evaluation steps begin. Since the parsed query expression is in postfix form, a stack-based algorithm is chosen for evaluation. An algorithm that evaluates a parsed query expression is shown in Fig.3.2.

```
==============================================================
GET FIRST EXPRESSION ITEM

WHILE THE EXPRESSION ITEM IS NOT SEMICOLON DO
   IF THE EXPRESSION ITEM IS AN OPERATOR THEN
      POP NEEDED OPERANDS FROM THE STACK
      DO THE OPERATION
      PUSH THE RESULT ONTO THE STACK
   ELSE
      PUSH THE OPERAND ONTO THE STACK
   ENDIF

   GET NEXT EXPRESSION ITEM
ENDWHILE

IF ERROR OCCURRED THEN
   DISPLAY ERROR MESSAGE
ELSE
   IF UNION OR PROJECTION WERE PERFORMED DURING PROCESS THEN
      REMOVE DUPLICATE TUPLES IF ANY
   ENDIF
   DISPLAY THE RESULT
ENDIF

END OF STEPS FOR EVALUATION
==============================================================
```

Fig.3.2 Algorithm for Evaluation phase

The projection and union operations may introduce duplicate tuples as a side effect. To eliminate duplicate records in a file, the following method was used in the system. First, by indexing, the file or relation is sorted in order to bring all duplicate records together. Then, a sequential pass is made through the file comparing adjacent records and eliminating all but one of the duplicates. However, because of the expense of sorting, the system does not always eliminate duplicates when executing union or projection. Rather, the duplicates are kept and carried along in subsequent operations. After the final operation in a query is performed, the system checks the status of the flag which indicates whether there were union or projection operations during the process. If this flag shows it is true then the resultant relation is sorted and duplicates are eliminated.

After the evauation steps are completed, the system finally displays the result or an error message. The system then goes back to the entry menu and awaits the user's next action.

3.3 Summary

This chapter has described the implementation of a retrieval system based on the relational algebra. The system consists of approximately 2,500 lines of source code written in both Turbo Pascal and dBASE III programming languages. The executable code occupies about 57K bytes.

CHAPTER 4

CONCLUSION

This report has described a query language system for retrieving data in relational form. The system is based on relational algebra and is designed for use with the dBASE III database management system. This system satisfies the closure property of the relational algebra, in that the result of operating on relations is itself a relation. If one of the requirements that a query language is designed to meet, is to provide a means for a user to select subsets of data in the database, then this implementation also satisfies this.

This system is intended for interactive users who are not required to have any knowledge of computer programming. The language syntax is rather inelegant, but it is simple and powerful, in that any derivable relation can be retrieved using a single query expression. The system is also a single-user system, and so does not worry about concurrency or security.

Although the system is fairly reliable, there may be limitations in some cases. Most of these limitations stem from the nature of the relational algebra. While simple queries seem straightforward, more complex queries may require a good deal of expertise to put together the right combination of algebraic operations to arrive at the correct answers. This situation might be improved by defining language syntax in a more elegant way. Also, complex queries on a large volume of data may take more time than expected. This is due to the fact that the speed of the reply depends critically on the way the query is formulated by the user. This requires that the system should be extended to optimize the query processing.

There is another case where the system is still inadquate for many practical problems. Many queries that are very simple to state, such as "find the maximum salary" can not be formulated in the current version of system. This problem could be solved by adding some statistical facilities such as average, sum, max or min. These built-in functions are often found in most of the other query language systems.

Another limitation the system has, concerns the join operation. For the join operation to make sense, the two attributes in the condition expression must come from the same domain. To illustrate this point, if there is one domain of all possible ages, and one domain of all possible prices, it is not reasonable to join two relations on the basis of ages in one relation and prices in the other, even though they are compatible in types. The implemented system does not check such conditions, but leaves the user to determine what is a reasonable operation.

In conclusion, the system described here is a prototype system and further effort is needed to improve the efficiency and to fully develop other facilities.

BIBLIOGRAPHY

[ASH84] "dBASE III User's Manual," Ashton-Tate Inc., 1984.

[BOY75] R. F. Boyce, D. D. Chamberlin, W. F. King III, and M. M. Hammer, "Specifying Queries as Relational Expressions: SQUARE," CACM 18, No. 11, Nov. 1975.

[BUN84] P. Buneman, "Can We Reconcile Programming Languages and Databases?", In Databases - Role and Structure (ed. P. M. Stocker et al), University Press, England, 1984.

[CHA74] D. D. Chamberlin and R. F. Boyce, "SEQUEL: A Structured English Query Language," Proc. 1974 ACM SIGMOD Workshop on Data Description, Access and Control.

[COD70] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," Communications of the ACM 13, No.6, 377-387.

[COD71] E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM SIGFIDET workshop on Data Description, Access and Control.

[COD79] E. F. Codd, "Extending the Database Relational Model to Capture More Meaning," ACM Trans. on Database Systems 4, No. 3, pp. 397-434, 1979.

[DAT81] C. J. Date, An Introduction to Database Systems, 3rd Ed., Addison-Wesley Publishing company, Reading, MA, 1981.

[DEN65] J. B. Dennis, "Segmentation and the Design of Multi-programmed Computer Systems," Journal of ACM 12, No 4, Oct. 1965, pp. 589-602.

[GOL70] R. C. Goldstein and A. L. Strand, "The MacAIMS Data Management System," Proc. ACM SIGFIDET Workshop on Data Description and Access, Nov. 1970, pp. 201-229.

[GOT75] L. R. Gotlieb, "Computing Joins of Relations," Proc. Int. Conf. Management od Data (ACM), 1975, pp. 55-63.

[HAL74a] P. Hall and S. Todd, "Factorization of Algebraic Expressions," Report UKSC 0055, IBM UK Scientific Center, Peterlee, England, April 1974.

[HAL74b] P. Hall, "Common Subexpression Identification in General Algebraic Systems," Report UKSC 0060, IBM UK Scientific Center, Peterlee, England, Nov. 1974.

[HAL75] P. Hall, "Optimisation of Single Relational Expressions in a Relational Data Base System," Report UKSC0076, IBM UK Scientific Center, Peterlee, England, June 1975.

[HEL75] G. D. Held. M. R. Stonebaker, and E. Wong, "A Relational Data Base System," Proc. NCC 44, May 1975.

[JOR75] D. E. Jordan, "Implementing Production Systems with Relational Data Bases," Proc. ACM Pacific Regional Conf., April 1975, pp. 39-43.

[JOY76] J. D. Joyce and N. N. Oliver, "REGIS - A Relational Information System with Graphics and Statistics," Proc. AFIPS National Computer Conf., Vol.45, pp. 839-844, 1976.

[LAC76] M. Lacroix and A. Pirotte, "Generalized Joins," SIGMOD Record 8, No. 3, pp. 14-15.

[MAI80] D. Maier and J. D. Ullman, "Maximal Objects and the Semantics of Universal Relation Databases," TR-80-016, Dept. of C.S., SUNY, Stony Brook, N.Y.

[MCD75] N. Mcdonald and M. R. Stonebraker, "Cupid - The Friendly Query Language," Proc. ACM Pacific Regional Conference, San Fransisco, April 1975.

[MER76] T. H. Merrett, "MRDS - An Algebraic Relational Database System," Proc. Canadian Computer Conference, Montreal, May 1976, pp. 102-124.

[MER83] T. H. Merrett and G. K. W. Chiu, "MRDSA: Full Support of the Relational Algebra on an Apple II," Workshop on Relational DBMS Design / Implementation / Use on Micro-Computers, Toulouse, Feb. 1983, pp. 385-402.

[PEC76] R. M. Pecherer, "Efficient Exploration of Product Spaces" Proc. Int. Conf. Management of Data (ACM), 1976, pp. 169-177.

[SAM81] J. E. Samet, "Query languages - A unified approach," Report of the British Computer Society Query Languages Group, Heyden University Press, Cambridge, England, 1981.

[SMI75] J. M. Smith and P. Chang, "Optimizing the Performance of Relational Algebra Data Base Interface," Comm. ACM 18, No. 10, pp. 568-579, Oct. 1975.

[STE74] J. Steuert and J. Goldman, "The Relational Data Management System: A Perspective," ACM SIGMOD Workshop on Data Description, Access and Control, 1974, pp. 295-320.

[STR71] A. J. Strnad, "The Relational Approach to the Management of Data Bases," Proc. IFIP 1971 Congress, North-Holland, Amsterdam, pp. 901-904.

[TOD74] S. J. P. Todd, "Implementation of the Join operator in Relational Data Bases," Tech. Note 15, IBM UK Scientific Center, Peterlee, England, Nov. 1974.

[TOD76] S. J. P. Todd, "The Peterlee Relational Test Vehicle - A System Overview," IBM Systems Journal 15, No 4, pp. 285-307, 1976.

[VAS79] Y. Vassiliou, "Null values in Database Management - A Denotational Semantics Approach," ACM SIGMOD Int. Symp. on Management of Data, 1979, pp. 162-169.

[VER76] J. Verhofstad, "The PRTV Optimiser: the Current State," Report UKSC 0083, IBM UK Scientific Center, Peterlee, England, May 1976.

[WHI72] V. K. M. Whitney, "RDMS: A Relational Data Management System," Proc. Fourth International Symposium on Computer and Information Sciences, Dec. 1972.

[YAO78] S. B. Yao and D. Dejong, "Evaluation of Database Access Paths," Proc. Int. Conf. Management of Data (ACM), 1978, pp. 66-77.

[ZLO77] M. M. Zloof, "Query-by-Example : A Data Base Language," IBM Systems Journal 16, No. 4, 1977, pp. 324-343.

APPENDIX A

USER'S GUIDE

The Relational Algebraic Retrieval System is a query system that allows you to retrieve information from the database. This system requires little knowledge of dBASE III command syntax. However, a minimal knowledge of relational algebra is required to formulate a query on the screen.

To use system, you need the following equipment:

- IBM PCs or other 100% compatible personal computer
- minimum of 512K bytes memory
- MS-DOS or PC-DOS version 2.0 or higher
- one 360K floppy disk drive and a hard disk drive
- any printer with at least 80 column capacity.

This manual is intended to be used as a reference for users of the system, and it is organized that way. The manual will give you detailed instructions for using this query system.

A.1 Installing the System

Before you can begin to use the Relational Algebraic Retrieval System, you must copy the contents of the system disk onto your hard disk. To install the Relational Algebraic Retrieval System onto the hard disk, follow the steps below. These instructions assume that your floppy disk is called "Drive A:" and that your hard disk is "Drive C:". If this is not the case, you must substitute the correct disk drive names for those in the steps which follow.

1. Make drive C the current drive.

2. Switch to the subdirectory where the dBASE III program and database files are kept.

3. Place the Relational Algebaic Retrieval System disk into the floppy drive, and copy all of the files onto the hard disk by typing:

   COPY A:*.*

4. Put away the original system disk in a safe place. You are now ready to use the program.

A.2 Starting and Exiting from the System

To start the system, use the following procedure:

1. Make sure you are at the dBASE III subdirectory

2. Type "DBASE RARS" and press <RETURN> to start up the system. If you are already within the dBASE III environment, then type "DO RARS". Every time you run the system, the program will display the menu screen which is shown in Fig. A.1. The date displayed will be the current DOS system date.

```
                                    November 15, 1987
=========================================================
          Relational Algebraic Retrieval System
=========================================================

          [11]   Query to Database

          [22]   Exit to dBASE III Command Level

          [33]   Exit to Operating System Level

=========================================================

               Please enter your selection
```

Fig. A.1 The System Opening Menu Screen

3. Choose Option "11" if you want to retieve data from the database.
4. Exit from the menu with option "22" (to continue working in dBASE III at command level) or Option "33" (to return to the operating system).

A.3. Formulating Queries

This section will describe the structure of the language and general rules used for formulating queries.

A.3.1 Language Components

Query expressions are constructed using letters, digits, and some special characters that are presented on the keyboard. All these characters fall into the following five categories in this language system.

1. Reserved Words

Reserved words are the words that represent specific kinds of algebraic operations within the system. You cannot use them except to stand for those particular meanings. Figure A.2 list these words.

2. Identifiers

Identifiers are names of the relations or attributes. A relation name is a legal database file name without a file extension. An attribute name is the name of a field in the database file.

Reserved Word	Description
DIVIDEBY	represents division operation
INTER	represents intersection operation
MINUS	represents difference operation
JOIN	represents join operation
TIMES	represents product operation
UNION	represents union operation
WHERE	represents selection operation

Fig. A.2 Reserved Words

3. Operators

Operators are the symbols or words which are used to indicate how two items are to be related. These operators, except unary operators ('-' or '+'), are used to state conditions in the join and selection operations. All operators have a property called "precedence". Precedence provides a kind of priority evaluation system. If two operators have different precedence, the one with higher precedence is evaluated first. There are five degrees of precedence: 1 is the highest and 5 the lowest. Figure A.3 summarizes the operators implemented in this system.

4. Special Symbols

The special language symbols are listed in the Fig. A.4.

5. Constants

A number of constants such as the integer number 5000 and text string 'Smith' may be used in the expression. The available types of constants are described below.

An integer constant consists of a sequence of digits with or without a leading sign, and no decimal point, e.g.

+5, 500, -500

Real number constants consist of at least one leading digit followed by a decimal point, then some trailing digits. A leading sign may or may not be present, e.g.

5.0, -0.5, +3.14

Character string constants consist of a sequence of any characters which the computer can represent, enclosed in either single or double quotes. To insert a single quote in a string, enclose the string with double quotes instead of single quotes.

Operator	Description	Precedence
=	is equal to	2
<>	is not equal to	2
<	is less than	2
>	is greater than	2
<=	is less than or equal to	2
>=	is greater than or equal to	2
NOT	negation	3
AND	conjunction	4
OR	disjunction	5
-	sign for negative number	1
+	optional sign for positive number	1

Fig. A.3 Operators

A similar rule applies to including double quotes in a string. An example would be "Adam's apple" or 'Adam"s apple'.

Another available constant type is a date constant. A format for a date is mm/dd/yy, where mm, dd, and yy represent the numeric codes for the month, day, and year, respectively. For example, 10/15/87.

Symbol	Description
,	separates items in a list
' or "	delimits character and string literals
.	decimal point, or separates file name and field name
(	starts nested or condition expression
)	ends nested or condition expression
[	starts attribute list
]	ends attribute list

Fig. A.4 Special Symbols

A.3.2 Formats for Relational Algebra Operations

This section provides a format of each relational algebra operation within a query expression. For notation, uppercase is used for all reserved words. Other language components (or substitutions), that are to be filled by the user, are denoted in lowercase and enclosed in angle brakets. When you type a query, remember not to type the brackets themselves or the exact words between the brackets; instead, make an appropriate substition.

The following terms are used to describe the substitution items.

<relation>: a valid database file name. You must not include the file extension.

<attribute>: the name of a field in the database file. You can include the database file name at the begining of a field name separated by a period.

<list>: one or more items of the same type separated by commas.

<join-condition>: a Boolean expression for join operation with the value True or False. Comparisons must be done between one field and another field.

<sele-condition>: a Boolean expression for selection operation with the value True or False. Comparisons must be done between fields and constants.

The formats for relational algebra operations which are implemented in this system are as follows.

1. Union: <relation1> UNION <relation2>

   Ex. JUNIOR UNION SENIOR

2. Difference: <relation1> MINUS <relation2>

   Ex. STUDENT MINUS SENIOR

3. Intersection: <relation1> INTER <relation2>

   Ex. CLASS_A INTER CLASS_E

4. Division: <relation1> DIVIDEBY <relation2>

   Ex. FACULTY DIVIDEBY DEGREES

5. Cartesian Product: <relation1> TIMES <relation2>

   Ex. STUDENT TIMES CLASS

6. Projection: <relation> [<list of attributes>]

   Ex. STUDENT [Name, Grade]

7. Selection: <relation> WHERE (<sele-condition>)

   Ex. STUDENT WHERE (Major = 'ART')

8. Join: <relation1> JOIN (<join-condition>) <relation2>

   Ex. STUDENT JOIN (Major = Dept) FACULTY

A.3.3 General Rules for Formulating Queries

There are certain rules that must be followed to insure that query expressions are properly formulated:

1. Each relational algebra operation must conform to the format for that operation as described in the previous section.
2. The maximum query expression length is 254 characters.
3. The maximum number of query expression lines is 5.
4. The reserved words, identifiers, and three operators (AND, OR, NOT) must be separated by at least one blank space. The blank spaces and <ENTER> keys are counted in the 254 character limit.
5. Case differences are ignored. "A" is the same as "a" to the system. There is only one exception. Within the character string, which is enclosed by the quotes, upper and lowercase letters are treated as unique characters, so the string of "Smith" is not equal to "smith".
6. When single or double quotation marks(' or ") are used for character string, the same symbol must be used at both the begining and end.
7. Parentheses can be used for clarity or nesting.

A.4 Entering Queries

After you choose Option "11" (query to databases) from the opening menu, the screen will clear and you will be prompted to enter a query. The screen also displays formats as a reference during the query formulation process.

Before entering a query, you should plan exactly what data you want the system to retrieve for you. This step will help you get the desired result. When you are ready, enter a query by typing from the keyboard. You cannot use the function keys and the control keys during the entry process. However, the following control keys are available.

<BS>: This key moves a cursor one character to the left, and deletes the character there.

<RETURN> or <ENTER>: These keys move a cursor to the far left on the next line.

<CTRL><Z>: This key combination terminates the query entry process.

The backspace <BS> is one basic typing correction which you can do during the process of entering query lines. This is available only in the current line i.e. you cannot go back to the previous lines.

After you have entered the query expression, terminate the query expression by pressing <CTRL><Z>. If you made a syntax error, you will see the error message. If you want to try again, press Y in response to "Do you want to try again? (Y/N)". If you type N then you will be returned to the opening menu.

A.5 Getting Results

If you entered a query and made no syntax error, the system will display the message:

"Query is processing -- Please do not interrupt".

While you are waiting, the system processes your query. If an error is detected during the process, the system will display the error message. Figure A.5 shows an example of the error message screen.

After processing is done, the system will ask you where the output is to be directed. Depending on your choice, the system will display the contents of the result file on either the screen or printer. If the result is displayed on the screen, the display process will pause when a screen is full (22 lines). The system prompts with: "More records -- Press any key to continue".

```
+------------------------------------------------------------------+
|                                                                  |
|  *** ERROR occurred during evaluation                            |
|                                                                  |
|  Your Query is:                                                  |
|                                                                  |
|   STUDENT [Name, Status]  JOIN (Major = Dept) FACULTY            |
|                                                                  |
|  The Evaluation steps so far:                                    |
|                                                                  |
|   TEMP01 :=  STUDENT [Name, Staus]                               |
|   TEMP02 :=  TEMP01 JOIN (Major = Dept) FACULTY                  |
|                                                                  |
|   ** Field Major does not exist in file TEMP02 **                |
|                                                                  |
|                                                                  |
|                                                                  |
|     Press any key to return to the main menu ...                 |
|                                                                  |
+------------------------------------------------------------------+
```

Fig. A.5 An Example of Error Message Screen

ID_NUMBER	NAME	MAJOR	BIRTH_DATE
415568962	T. Baker	History	10/12/61
568310121	K. Chapman	Economics	08/04/66
109442593	A. Zeller	Art	07/22/68
723402157	J. Smith	English	07/04/67
374590212	W. Lee	Chemistry	12/24/59
547635642	D. Gregory	History	05/14/59
387024663	T. Hanson	Music	02/15/64
287409826	C. Duff	Economics	07/04/65
739264874	H. Nelson	English	06/18/60
389461232	M. Thompson	History	09/20/64

Fig. A.6 The Example Output in Tabular Format

Record#	Fieldname	Content
1	NAME	Thomas T. Hanson
	HOME_ADDR	35 Fountain St., Elgin, IL 60102
	PHONE_NO	415-567-8967
	HOBBIES	swimming; sailing; stamp collecting
2	NAME	Kirk D. Chapman
	HOME_ADDR	879 Wiltshire Rd., Lowell, MA 01835
	PHONE_NO	617-625-7845
	HOBBIES	ice skating; fishing; spectator sports
3	NAME	Tina B. Baker
	HOME_ADDR	23 Antlers Dr., Lake Bluff, IL 60044
	PHONE_NO	309-456-9873
	HOBBIES	water sports; sculpting (clay); pottery
4	NAME	Mary W. Thompson
	HOME_ADDR	110 Summer St., Los Angeles, CA 90057
	PHONE_NO	213-432-6782
	HOBBIES	painting; classical piano; reading
5	NAME	Albert K. Zeller
	HOME_ADDR	6440 Oregon St., Ft. Myers, FL 33901
	PHONE_NO	813-457-9801
	HOBBIES	chess; theater; reading; bridge; racket ball
6	NAME	Gerald L. Maurer
	HOME_ADDR	78 Doyle St., Trenton, NJ 08607
	PHONE_NO	609-242-9003
	HOBBIES	tennis; golf; travel; swimming; reading

Fig. A.7 The Example Output in Linear Format

There are two kinds of format for the output: tabular format and linear format. When the length of single record (including separators between fields) fits in one line(80 or 132 columns), the tabular format is used. Otherwise, the other format will be used. Figures A.6 and A.7 show examples of the outputs in these two formats.

If you want to save the result file, just type Y (or y) to the system's message:

">> Do you wish to save the result file (Y/N)?".

The system will then ask you for a new file name which will be assigned to the result file.

Now, the processing of a single query is done, and you will find yourself seeing the opening menu again.

A.6 Error Messages

The following is the alphabetical listing of error messages you may get from the system. Many error messages are totally self-explanatory, but some need a little explanation as provided in the following.

** "(" is expected

** ")" is expected

** "," is expected in attribute list

Attribute names must be separated by a comma.

** Attribute name is expected

In the attribute list, no attribute name is specified after the comma is issued.

** Closing quote is expected for character string constant.

** Comparison operator is expected

One of "<", ">", "=". "<>", "<=" and ">=" is expected.

** Constant value is expected

In a condition expression for selection operation, the right hand side of the comparison operator must be a constant string of number, character or date.

** Data type mismatch between <field name 1> and <field name 2>

Data types are not matched when the two fields in a condition expression for join operation, are compared.

** Divisor relation is empty

** Divisor relation is same as Dividend relation

** Divisor relation has attribute(s) not belonging to dividend relation

** Divisor relation has too many attributes

For division relation, the fields of divisor relation must be a subset (not a same set) of the ones in dividend relation.

** Empty attribute list

No attribute names are specified in the square brakets for projection operation.

** Empty query expression: cannot be processed

A query expression is terminated before it is filled, or it only contains blank spaces.

** Field <field name> does not exist in current file

A field name that does not exist in the current file in use, has been specified in the condition expression or atttibute list.

** File <file name> does not exist

A specified file or relation name is not found in the current directory.

** Field name is too long

The length of specified field name is greater than 10.

** File name is too long

The length of specified file name is greater than 8.

** Illegal data format for date value

The format for date value must be "mm/dd/yy".

** Illegal field name: Field name must start with a letter

** Illegal file name: File name must start with a letter

** Illegal symbol in date constant

A symbol other than the digits or slash was used in a date.

** Illegal symbol in field name

** Illegal symbol in file name

Only letters, digits and underscore can be used for field or file name.

** Illegal symbol in number constant

A symbol other than a digit, decimal point, or sign appeared in the number constant string.

** Invalid date constant

Impossible date value was specified in date string.

** Mismatched data types: <field name> is not a numeric field

** Mismatched data types: <field name> is not a character field

** Mismatched data types: <field name> is not a date field

In a condition expression for selection operation, the contents of a field is compared to different type of data value.

** Missing "]" for attribute list

** Query expression is in too many lines

Total number of query lines exceeds 5.

** Query expression is too long

An attempt was made to enter a query expression more than 254 characters long.

** Two relations are not union compatible

For the union, intersection and difference operations, the two involved relations must have the same number of fields and each set of corresponding fields must have same types.

** Unexpected end, missing condition expression

A query expression is ended when a condition expression is expected for the join or selection operations.

** Unknown symbol or syntax error in condition expression

A condition expression embeds an unknown symbol or is constructed incorrectly.

** Unknown symbol or syntax error

A query expression embeds an unknown symbol or is constructed incorrectly.

APPENDIX B

SYNTAX OF RARS IMPLEMENTATION

The syntax of the Relational Algebraic Rietrival System language is presented here using the formalism known as Backus-Naur Form. The following meta-symbols of BNF are used:

::= meaning "is defined as"

| meaning "or"

{ } specifies syntax items to be repeated zero or more times

< > angle brackets used to surround category names.

The angle brackets distinguish category names from terminal symbols, which are written exactly as they are to be represented.

```
<alg-exp>    ::= <infix> | <projection> | <selection> | <join>
<attr-name> ::= <identifier>
<attr-spec> ::= <attr-name> | <rel-name>.<attr-name>
<attr-list> ::= <attr-spec> { , <attr-spec> }
<comp-op>    ::=  <  |  =  |  <=  |  <>  |  >=  |  >
<const-val> ::= <number> | <string> | <date>
<date>       ::= <month>/<day>/<year>
<identifier> ::= <letter> { <letter-or-digit> }
<infix>      ::= <primitive> <infix-op> <primitive>
<infix-op>   ::= UNION | INTER | MINUS | TIMES | DIVIDEBY
<join>       ::= <primitive> JOIN ( <join-cond-exp> ) <primitive>
<join-cond-exp> ::= <join-cond-term> { OR <join-cond-term> }
<join-cond-term> ::= <join-cond-factor> { AND <join-cond-factor>}
<join-cond-factor> ::= <join-cond-primitive> |
                       NOT <join-cond-primitive>
```

```
<join-cond-primitive> ::= <attr-spec> <comp-op> <attr-spec> |
                          ( <join-cond-exp> )
<letter-or-digit>  ::= <letter> | <digit>
<number>     ::= <integer> | <real> |
                <sign> <integer> | <sign> <real>
<primitive> ::= <rel-name> | ( <alg-exp> )
<projection> ::= <primitive> [ <attr-list> ]
<rel-name>    ::= <identifier>
<selection>   ::= <primitive> WHERE ( <sele-cond-exp> )
<sele-cond-exp> ::= <sele-cond-term> { OR <sele-cond-term> }
<sele-cond-term> ::= <sele-cond-factor> { AND <sele-cond-factor>}
<sele-cond-factor> ::= <sele-cond-primitive> |
                       NOT <sele-cond-primitive>
<sele-cond-primitive> ::= <attr-spec> |
                          <attr-spec> <comp-op> <const-val> |
                          ( <sele-cond-exp> )
<sign>     ::=  +  |  -
<string>   ::= '{ <character> }' | "{ <character> }"
```

APPENDIX C

PROGRAM LISTING

This section contains the full listing of RARS implementation. The list of routines contained in the RARS is as follows.

dBASE III routines

Turbo Pascal routines

```
*****************************************************************
*                          RARS.PRG                             *
* This program is a main module of the system and serves as the *
* entry and exit point of the system.  The module first displays*
* a menu for a user to select either, a query to database or    *
* exits from the system.  Depending on the user's choice, the   *
* module calls a subroutine to process the query or exit from   *
* the system.                                                   *
*****************************************************************

CLEAR ALL
SET TALK OFF
SET ECHO OFF
SET SAFETY OFF
SET DELETED ON
SET INTENSITY ON
SET COLOR TO W, W+
SET BELL OFF
DO WHILE .T.
   CLEAR
   @ 5, 44 SAY CMONTH(DATE()) + " " + STR(DAY(DATE()),2) +;
               ", " + STR(YEAR(DATE()),4)
   TEXT
      ========================================================
               Relational Algebraic Retrieval System
      ========================================================

               [11]   Query to Databases

               [22]   Exit to dBASE III Command Level

               [33]   Exit to Operating System Level

      ========================================================
   ENDTEXT
   STORE "  " TO Mselect
   @ 22, 20 SAY "Please enter your selection  " GET Mselect ;
         PICTURE "99"
   READ
   DO CASE
      CASE Mselect = "11"
         CLEAR
         RUN RAPARSER
         IF FILE("SCANOUT.DAT")
            DO RAEVAL
         ENDIF
      CASE Mselect = "22"
         CLEAR
         SET TALK ON
         SET SAFETY ON
         SET DELETED OFF
         SET BELL ON
         CANCEL
```

```
      CASE Mselect = "33"
         CLEAR
         QUIT
      OTHERWISE
         Mmessage = "  Not a valid selection -- Press any key " ;
                  + "to try again "
         SET COLOR TO /W, U+
         @ 22, 5 SAY Mmessage
         WAIT ""
         SET COLOR TO W, W+
   ENDCASE
ENDDO   [ .T. ]
RETURN


*****************************************************************
*                          RAEVAL.PRG                           *
* This program is the second level program, which is called by  *
* a main module to evaluate a user's query.  The program first  *
* reads a parsed query expression, which is held in the text    *
* file, into the database file.  Then the program starts calling*
* a series of subroutines to perform the evaluation task.       *
*****************************************************************

@ 22, 7 SAY "Query is processing -- Please do not interrupt"
STORE " " TO EvalStack, Result
ErrMsg = ""
NoError = .T.
MaybeDup = .F.
TempFile = "TEMP00"
TempidNo = "00"
AssignMark = " := "
USE RASTEPS
ZAP
USE RAFNAMES
ZAP
USE RADATA
ZAP
APPEND FROM SCANOUT.DAT SDF
DELETE FILE SCANOUT.DAT
USE RADATA
DO WHILE CONTENT <> ";"
   SKIP
ENDDO
SKIP
Mcontent = TRIM(CONTENT)
SET PROCEDURE TO RAPROCES
DO WHILE Mcontent <> ";"
   IF SUBSTR(Mcontent, 1, 1) = "*"
      * Save record number in order to return after processing.
      ReturnPos = RECNO() + 1
      OpCode = SUBSTR(Mcontent, 2, 1)
      DO PROCESS WITH OpCode, EvalStack, NoError
      IF NoError
         USE RADATA
```

```
         GO ReturnPos
      ELSE
         EXIT
      ENDIF
   ELSE
      EvalStack = STR(RECNO(), 2) + ";" + EvalStack
      SKIP
   ENDIF
   Mcontent = TRIM(CONTENT)
ENDDO
IF NoError
   DO POP WITH EvalStack, Result
   IF MaybeDup
      DO REMOVDUP WITH Result
   ENDIF
   SET PROCEDURE TO RARESULT
   DO FINISH WITH Result
ELSE
   DO DISP_ERR WITH ErrMsg
ENDIF
CLOSE PROCEDURE
CLEAR ALL
RETURN


*****************************************************************
*                PROCEDURE FILE :  RAPROCES.PRG                 *
* This procedure file contains 32 procedures and remains open   *
* until the last operation in the query expression is done, or  *
* an error is encountered.                                      *
*****************************************************************

*****************************************************************
* This procedure adds a single record, field by field.          *
*****************************************************************

PROCEDURE ADDREC
PARAMETERS R1Fields, R2Fields

   Fields1 = R1Fields
   Fields2 = R2Fields
   APPEND BLANK
   DO WHILE AT(",", Fields1) > 0
      Pos1 = AT(",", Fields1)
      Pos2 = AT(",", Fields2)
      Mreplace = SUBSTR(Fields1,1,Pos1-1) + " WITH B->" + ;
                 SUBSTR(Fields2,1,Pos2-1)
      REPLACE &Mreplace
      Fields1 = SUBSTR(Fields1, Pos1+1)
      Fields2 = SUBSTR(Fields2, Pos2+1)
   ENDDO
   Mreplace = Fields1 + " WITH B->" + Fields2
   REPLACE &Mreplace
RETURN
```

```
*********************************************************************
* This procedure adds records in one file to the other file.       *
* Before each record is added, the record is checked to see if     *
* it already exists in the receiving file.                         *
*********************************************************************

PROCEDURE ADDUNIQ
PARAMETERS ToFile, FromFile, FldList1, FldList2

   Mcond = ""
   DO GETCOND WITH Mcond, FldList1, FldList2
   SELECT 1
   USE &ToFile
   SELECT 2
   USE &FromFile
   DO WHILE .NOT. EOF()
      SELECT 1
      LOCATE FOR &Mcond
      IF EOF()
         DO ADDREC WITH FldList1, FldList2
      ENDIF
      SELECT 2
      SKIP
   ENDDO
RETURN


*********************************************************************
* This procedure checks if the two relations are union compati-    *
* ble by examining the number and type of fields in them.          *
*********************************************************************

PROCEDURE CHK_COMP
PARAMETERS REL1, REL2, Compatible, R1Fields, R2Fields

   USE &REL2
   COPY TO R2_STRUC STRUCTURE EXTENDED
   USE R2_STRUC
   GO BOTTOM
   NumFields2 = RECNO()
   USE &REL1
   COPY TO R1_STRUC STRUCTURE EXTENDED
   USE R1_STRUC
   GO BOTTOM
   NumFields1 = RECNO()
   IF NumFields1 <> NumFields2
      Compatible = .F.
      RETURN
   ENDIF
   SELECT 2
   USE R2_STRUC
   SELECT 1
   USE R1_STRUC
   SET RELATION TO RECNO() INTO R2_STRUC
```

```
   SELECT 1
   DO WHILE .NOT. EOF()
      IF FIELD_TYPE <> B->FIELD_TYPE
         Compatible = .F.
         EXIT
      ELSE
         R1Fields = R1Fields - "," - FIELD_NAME
         R2Fields = R2Fields - "," - B->FIELD_NAME
         SKIP
      ENDIF
   ENDDO
   IF Compatible
      R1Fields = SUBSTR( TRIM(R1Fields), 2)
      R2Fields = SUBSTR( TRIM(R2Fields), 2)
   ENDIF
   SET RELATION
RETURN


*****************************************************************
* This procedure first checks if a field exists in the given    *
* file.  If not found, then the procedure checks if the field   *
* name is changed.  If this is the case, the existence checking *
* is done again.  If found, the procedure calls another proce-  *
* dure to replace the field name appearing in the expression,   *
* with a changed one.  When the field is found in the file, the *
* procedure returns the field type.                             *
*****************************************************************

PROCEDURE CHKFIELD
PARAMETERS REL, FldName, FldType, Exp

   IF AT(".", FldName) > 1
      OldField = FldName
      Fname = SUBSTR(OldField, 1, AT(".",OldField)-1)
      FldName = SUBSTR(OldField, AT(".",OldField)+1)
      IF Fname <> REL
         USE RAFNAMES
         LOCATE FOR FILENAME = Fname .AND. OLDFLDNAME = FldName
         IF .NOT. EOF()
            FldName = NEWFLDNAME
         ENDIF
      ENDIF
      USE &REL
      FldType = TYPE(FldName)
      IF FldType = "U"
         FldName = SUBSTR(OldField, AT(".",OldField)+1)
      ELSE
         DO NEWEXP WITH Exp, OldField, FldName
      ENDIF
   ELSE
      USE &REL
      FldType = TYPE(FldName)
      IF FldType = "U"
         USE RAFNAMES
```

```
         LOCATE FOR OLDFLDNAME = FldName
         IF .NOT. EOF()
            NewField = NEWFLDNAME
            USE &REL
            FldType = TYPE(NewField)
            IF FldType <> "U"
               DO NEWEXP WITH Exp, FldName, NewField
               FldName = NewField
            ENDIF
         ENDIF  [ .NOT. EOF() ]
      ENDIF  [ FldType = "U" ]
   ENDIF  [ AT(".", FldName) > 1 ]
RETURN


*****************************************************************
* This procedure checks if the second relation has any field    *
* names that appear in the first relation. If this is true, this*
* procedure assigns new names for those fields, and reports the *
* name changes to the name change holding file.                 *
*****************************************************************

PROCEDURE CHKFLDS1
PARAMETERS REL1, REL2, FldNameDup

   Prefix = SUBSTR(REL2,I,I)
   USE &REL1
   COPY TO R1_STRUC STRUCTURE EXTENDED
   USE &REL2
   COPY TO R2_STRUC STRUCTURE EXTENDED
   SELECT 3
   USE RAFNAMES
   SELECT 2
   USE R1_STRUC
   INDEX ON FIELD_NAME TO RI_STRUC
   USE R1_STRUC INDEX R1_STRUC
   SELECT 1
   USE R2_STRUC
   DO WHILE .NOT. EOF()
      Mfldname = FIELD_NAME
      SELECT 2
      SEEK Mfldname
      IF .NOT. EOF()
         FldNameDup = .T.
         NewName = TRIM(SUBSTR(Prefix+Mfldname, 1, IO))
         SEEK NewName
         DO WHILE .NOT. EOF()
            NewName = TRIM(SUBSTR(Prefix+NewName, 1, 10))
            SEEK NewName
         ENDDO
         SELECT 3
         DO REPORTC WITH REL2, Mfldname, NewName
         SELECT 1
         REPLACE FIELD_NAME WITH NewName
      ENDIF
```

```
      SELECT 1
      SKIP
   ENDDO
RETURN

******************************************************************
* This procedure checks if the divisor relation has a subset of *
* fields appearing in the dividend relation.  At the same time, *
* the procedure builds a list of fields which will appear in the*
* result file after the division operation.                      *
******************************************************************

PROCEDURE CHKFLDS2
PARAMETERS REL1, REL2, DivError1, DivError2, R2Fields, QuotFlds

   USE &REL1
   COPY TO R1_STRUC STRUCTURE EXTENDED
   USE R1_STRUC
   GO BOTTOM
   NumFields1 = RECNO()
   USE &REL2
   COPY TO R2_STRUC STRUCTURE EXTENDED
   USE R2_STRUC
   GO BOTTOM
   NumFields2 = RECNO()
   IF NumFields1 <= NumFields2
      DivError1 = .T.
      RETURN
   ENDIF
   SELECT 1
   USE R1_STRUC
   INDEX ON FIELD_NAME TO R1_STRUC
   USE R1_STRUC INDEX R1_STRUC
   SELECT 2
   USE R2_STRUC
   DO WHILE .NOT. EOF()
      STORE FIELD_NAME TO MfldName
      SELECT 1
      SEEK MfldName
      IF EOF()
         DivError2 = .T.
         RETURN
      ELSE
         DELETE
      ENDIF
      SELECT 2
      R2Fields = R2Fields - "," - FIELD_NAME
      SKIP
   ENDDO
   SET DELETED OFF
   SELECT 1
   USE R1_STRUC
   GO TOP
   DO WHILE .NOT. EOF()
```

```
      IF .NOT. DELETED()
         QuotFlds = QuotFlds - "," - FIELD_NAME
      ENDIF
      SKIP
   ENDDO
   SET DELETED ON
   R2Fields = SUBSTR( TRIM(R2Fields), 2)
   QuotFlds = SUBSTR( TRIM(QuotFlds), 2)
RETURN

*****************************************************************
* This procedure takes a field name from the attribute list, one*
* at a time, and passes it to the other procedure to check its  *
* existence in the given file.                                  *
*****************************************************************

PROCEDURE CHKFLIST
PARAMETERS REL, AttrList, AtrListErr

   FldType = " "
   FldList = AttrList
   DO WHILE AT(",", FldList) > 0
      FldName = SUBSTR(FldList, 1, AT(",",FldList)-1)
      DO CHKFIELD WITH REL, FldName, FldType, AttrList
      IF FldType = "U"
         AtrListErr = .T.
         EXIT
      ELSE
         FldList = SUBSTR(FldList, AT(",",FldList)+1)
      ENDIF
   ENDDO
   IF .NOT. AtrListErr
      FldName = FldList
      DO CHKFIELD WITH REL, FldName, FldType, AttrList
      IF FldType = "U"
         AtrListErr = .T.
      ENDIF
   ENDIF
   IF AtrListErr
      ErrMsg = "Field " + FldName + " does not exist " + ;
               "in file " + REL
   ENDIF
RETURN

*****************************************************************
* This procedure compares the contents of two adjacent records  *
* to see if their contents are same.                            *
*****************************************************************

PROCEDURE COMPARE1
PARAMETERS ListExp, PTR1, PTR2, Duplicate

   FldList = ListExp
```

```
   DO WHILE AT(",", FldList) > 0 .AND. Duplicate
      Fld = SUBSTR(FldList, 1, AT(",",FldList)-1)
      GO PTR1
      FVAL1 = &Fld
      GO PTR2
      FVAL2 = &Fld
      IF TYPE(Fld) = "L"
         IF (FVAL1 .OR. FVAL2) .AND. .NOT. (FVAL1 .AND. FVAL2)
            Duplicate = .F.
            EXIT
         ENDIF
      ELSE
         IF FVAL1 <> FVAL2
            Duplicate = .F.
            EXIT
         ENDIF
      ENDIF
      FldList = SUBSTR(FldList, AT(",",FldList)+1)
   ENDDO
   IF Duplicate
      Fld = FldList
      GO PTR1
      FVAL1 = &Fld
      GO PTR2
      FVAL2 = &Fld
      IF TYPE(Fld) = "L"
         IF (FVAL1 .OR. FVAL2) .AND. .NOT. (FVAL1 .AND. FVAL2)
            Duplicate = .F.
         ENDIF
      ELSE
         IF FVAL1 <> FVAL2
            Duplicate = .F.
         ENDIF
      ENDIF
   ENDIF
RETURN


*****************************************************************
* This procedure compares the contents of two records that come *
* from the different files.                                     *
*****************************************************************

PROCEDURE COMPARE2
PARAMETERS ListExp1, ListExp2, Duplicate

   FldList1 = ListExp1
   FldList2 = ListExp2
   DO WHILE AT(",", FldList1) > 0 .AND. Duplicate
      Fld1 = SUBSTR(FldList1, 1, AT(",",FldList1)-1)
      Fld2 = "B->" +SUBSTR(FldList2, 1, AT(",",FldList2)-1)
      IF TYPE(Fld1) = "L"
         IF (&Fld1 .OR. &Fld2) .AND. .NOT. (&Fld1 .AND. &Fld2)
            Duplicate = .F.
            EXIT
```

```
         ENDIF
      ELSE
         IF &Fld1 <> &Fld2
            Duplicate = .F.
            EXIT
         ENDIF
      ENDIF
      FldList1 = SUBSTR(FldList1, AT(",",FldList1)+1)
      FldList2 = SUBSTR(FldList2, AT(",",FldList2)+1)
   ENDDO
   IF Duplicate
      Fld1 = FldList1
      Fld2 = "B->" + FldList2
      IF TYPE(Fld1) = "L"
         IF (&Fld1 .OR. &Fld2) .AND. .NOT. (&Fld1 .AND. &Fld2)
            Duplicate = .F.
         ENDIF
      ELSE
         IF &Fld1 <> &Fld2
            Duplicate = .F.
         ENDIF
      ENDIF
   ENDIF
RETURN


*****************************************************************
* This procedure deletes records, which appear in the second    *
* relation, from the first relation.  Searching is performed    *
* sequentially.                                                 *
*****************************************************************

PROCEDURE DELBYSEQ
PARAMETERS FirstFile, SecondFile

   Mcond = ""
   DO GETCOND WITH Mcond, R1Fields, R2Fields
   SELECT 1
   USE &FirstFile
   SELECT 2
   USE &SecondFile
   DO WHILE .NOT. EOF()
      SELECT 1
      LOCATE FOR &Mcond
      IF .NOT. EOF()
         * This record occurs in the second relation.
         DELETE
      ENDIF
      SELECT 2
      SKIP
   ENDDO
RETURN
```

```
***********************************************************************
* This procedure searches the first file to find out the records*
* that also occur in the second relation.  If found, then those *
* records will be deleted from the first file.  Search is a     *
* random search onto the indexed first file.                    *
***********************************************************************

PROCEDURE DELBYRAN
PARAMETERS FirstFile, SecondFile

   SELECT 2
   USE &SecondFile
   SELECT 1
   USE &FirstFile
   INDEX ON &KeyExp1 TO &FirstFile
   USE &FirstFile INDEX &FirstFile
   SELECT 2
   DO WHILE .NOT. EOF()
      STORE &KeyExp2 TO Mkey2
      SELECT 1
      SEEK Mkey2
      IF .NOT. EOF()
         STORE &KeyExp1 TO Mkey1
         DO WHILE Mkey1 = Mkey2 .AND. ( .NOT. EOF() )
            IF AllFields
               * This record occurs in the second relation.
               DELETE
            ELSE
               Duplicate = .T.
               DO COMPARE2 WITH NonKey1, NonKey2, Duplicate
               IF Duplicate
                  DELETE
               ENDIF
            ENDIF
            SKIP
            STORE &KeyExp1 TO Mkey1
         ENDDO
      ENDIF
      SELECT 2
      SKIP
   ENDDO
RETURN


***********************************************************************
* This procedure deletes the remaining temporary database and   *
* index files after the process.                                *
***********************************************************************

PROCEDURE DELTEMPS
PRIVATE DelFile

   IF "TEMP" $ REL
      DelFile = REL + ".DBF"
      DELETE FILE &DelFile
```

```
   ELSE
      IF "TEMP" $ REL1
         DelFile = REL1 + ".DBF"
         DELETE FILE &DelFile
      ENDIF
      IF "TEMP" $ REL2
         DelFile = REL2 + ".DBF"
         DELETE FILE &DelFile
      ENDIF
   ENDIF
   DelFile = Result + ".NDX"
   IF FILE(DelFile)
      DELETE FILE &DelFile
   ENDIF
RETURN


*******************************************************************
* This procedure displays an error message with a user's origi-  *
* nal query expression and the evaluation steps that have been   *
* done so far.                                                     *
*******************************************************************

PROCEDURE DISP_ERR
PARAMETERS ErrMsgLine

   CLEAR
   @ 1, 0 SAY "*** ERROR occurred during evaluation"
   @ 3, 0 SAY "Your Query is : "
   ?
   USE RADATA
   DO WHILE CONTENT <> ";"
      Mcontent = TRIM(CONTENT)
      ? Mcontent
      SKIP
   ENDDO
   ?
   ? "The Evauation Steps so far : "
   ?
   StepNo = 1
   USE RASTEPS
   DO WHILE .NOT. EOF() .AND. StepNo < 10
      ? " Step", STR(StepNo,1), " ", TRIM(EVALSTEP)
      StepNo = StepNo + 1
      SKIP
   ENDDO
   ?
   ? "    ** ", ErrMsgLine, " **"
   USE
   Response = " "
   SET COLOR TO /W, U+
   @ 22, 2 SAY " Press any key to return to the main menu ... " ;
           GET Response PICTURE "!"
   SET COLOR TO W, W+
   READ
```

```
   IF VAL(TempidNo) > 1
      RUN ERASE TEMP??.*
   ENDIF
RETURN


*******************************************************************
* This procedure performs the division operation of relational   *
* algebra.                                                        *
*******************************************************************

PROCEDURE DIVISION
PARAMETERS REL1, REL2, Result, Mevalstep

   MevalStep = REL1 + " DIVIDEBY " + REL2
   IF REL1 = REL2
      ErrMsg = "Divisor relation is same as Dividend relation"
      RETURN
   ENDIF
   USE &REL2
   IF EOF()
      ErrMsg = "Divisor relation is empty"
      RETURN
   ENDIF
   STORE .F. TO DivError1, DivError2
   STORE "" TO R2Fields, QuotFlds
   DO CHKFLDS2 WITH REL1, REL2, DivError1, DivError2, ;
                    R2Fields, QuotFlds
   IF DivError1
      ErrMsg = "Divisor relation has too many attributes"
   ELSE
      IF DivError2
         ErrMsg = "Divisor relation has attribute(s) not " ;
                  + "belonging to Dividend relation"
      ELSE
         DO GENTEMP WITH TempFile, TempidNo
         USE &REL1
         IF EOF()
            COPY TO &TempFile FIELDS &QuotFlds
         ELSE
            Mcond = ""
            CommonFlds = R2Fields
            DO GETCOND WITH Mcond, CommonFlds, R2Fields
            SELECT 2
            USE &REL2
            GO BOTTOM
            R2NumRec = RECNO()
            GO TOP
            SELECT 1
            USE &REL1
            JOIN WITH &REL2 TO &TempFile FOR &Mcond ;
                 FIELDS &QuotFlds
            USE &TempFile
            IF .NOT. EOF()
               IF R2NumRec > 1
```

```
            COPY TO STRUFILE STRUCTURE EXTENDED
            HoldFile = TempFile
            DO GENTEMP WITH TempFile, TempidNo
            CREATE &TempFile FROM STRUFILE
            STORE "" TO KeyExp, NonKeyFlds
            AllFields = .T.
            DO GETKEY1 WITH KeyExp, NonKeyFlds, AllFields
            ResultFlds = QuotFlds
            SELECT 1
            USE &TempFile
            SELECT 2
            USE &HoldFile
            INDEX ON &KeyExp TO &HoldFile
            USE &HoldFile INDEX &HoldFile
            STORE &KeyExp TO Mkey2
            DO WHILE .NOT. EOF()
               Mkey1 = Mkey2
               PTR1 = RECNO()
               SKIP
               STORE &KeyExp TO Mkey2
               NumDupRec = 1
               DO WHILE Mkey1 = Mkey2
                  IF AllFields
                     NumDupRec = NumDupRec + 1
                  ELSE
                     PTR2 = RECNO()
                     Duplicate = .T.
                     DO COMPARE1 WITH NonKeyFlds, PTR1, ;
                                      PTR2, Duplicate
                     IF Duplicate
                        NumDupRec = NumDupRec + 1
                     ENDIF
                  ENDIF
                  SKIP
                  STORE &KeyExp TO Mkey2
               ENDDO
               IF NumDupRec >= R2NumRec
                  SKIP -1
                  SELECT 1
                  DO ADDREC WITH ResultFlds, QuotFlds
                  SELECT 2
                  SKIP
               ENDIF
            ENDDO
            CLOSE DATABASES
            DelFile1 = HoldFile + ".DBF"
            DelFile2 = HoldFile + ".NDX"
            DELETE FILE &DelFile1
            DELETE FILE &DelFile2
            DELETE FILE STRUFILE.DBF
         ENDIF
      ENDIF
   ENDIF
   Result = TempFile
ENDIF
```

```
   ENDIF
   CLOSE DATABASES
   RUN ERASE R?_STRUC.*
RETURN


*******************************************************************
* This procedure generates a file name, which is created during *
* the evaluation process, for the temporary database file.      *
*******************************************************************

PROCEDURE GENTEMP
PARAMETERS TempFile, TempidNo

   TempidNo = SUBSTR( STR( &TempidNo+101,3), 2,2)
   TempFile = "TEMP" + TempidNo
RETURN


*******************************************************************
* This procedure generates a condition expression that will be  *
* used to compare the contents of two records.                  *
*******************************************************************

PROCEDURE GETCOND
PARAMETERS Mcond, ListExp1, ListExp2

   Fields1 = ListExp1
   Fields2 = ListExp2
   DO WHILE AT(",",Fields1) > 0
      Pos1 = AT(",",Fields1)
      Pos2 = AT(",",Fields2)
      Mcond = Mcond + " .AND. " + SUBSTR(Fields1,1,Pos1-1) + ;
              "=B->" + SUBSTR(Fields2,1,Pos2-1)
      Fields1 = SUBSTR(Fields1, Pos1+1)
      Fields2 = SUBSTR(Fields2, Pos2+1)
   ENDDO
   Mcond = Mcond + " .AND. " + Fields1 + "=B->" + Fields2
   Mcond = SUBSTR(Mcond, 8)
RETURN


*******************************************************************
* This procedure builds a list of field names, which will appear*
* in the result relation after the join operation.              *
*******************************************************************

PROCEDURE GETFLDS
PARAMETERS REL1, REL2, FldNameDup, DelFields, FieldList

   Prefix = SUBSTR(REL2,1,1)
   STORE "" TO FldList1, FldList2
   USE &REL1
   COPY TO R1_STRUC STRUCTURE EXTENDED
   USE R1_STRUC
```

```
   DO WHILE .NOT. EOF()
      FldList1 = FldList1 + "," + TRIM(FIELD_NAME)
      SKIP
   ENDDO
   USE &REL2
   COPY TO R2_STRUC STRUCTURE EXTENDED
   SELECT 3
   USE RAFNAMES
   SELECT 1
   USE R1_STRUC
   INDEX ON FIELD_NAME TO R1_STRUC
   USE R1_STRUC INDEX R1_STRUC
   SELECT 2
   USE R2_STRUC
   DO WHILE .NOT. EOF()
      Mname = TRIM(FIELD_NAME)
      IF .NOT. (","+Mname+"," $ DelFields)
         SELECT 1
         SEEK Mname
         IF .NOT. EOF()
            FldNameDup = .T.
            NewName = TRIM(SUBSTR(Prefix + FIELD_NAME, 1, 10))
            SEEK NewName
            DO WHILE .NOT. EOF()
               NewName = TRIM(SUBSTR(Prefix + NewName, 1, 10))
               SEEK NewName
            ENDDO
            FldList2 = FldList2 + "," + NewName
            SELECT 2
            REPLACE FIELD_NAME WITH NewName
            SELECT 3
            DO REPORTC WITH REL2, Mname, NewName
            DO NEWEXP WITH CondExp, "B->"+Mname, "B->"+NewName
         ELSE
            SELECT 2
            FldList2 = FldList2 + "," + TRIM(FIELD_NAME)
         ENDIF
      ENDIF
      SELECT 2
      SKIP
   ENDDO
   FieldList = SUBSTR(FldList1,2) + FldList2
RETURN


*******************************************************************
* This procedure generates a key expression that will be used to*
* index a database file.                                          *
*******************************************************************

PROCEDURE GETKEY1
PARAMETERS KeyExp, NonKeyFlds, AllFields

   KeyLen = 0
   USE STRUFILE
```

```
    * Index key cannot be more than 100 characters long.
    DO WHILE .NOT. EOF() .AND. KeyLen <= 100
       * Logical field cannot be an index key.
       IF FIELD_TYPE <> "L" .AND. KeyLen + FIELD_LEN <= 100
          * Index key expression should be a character type.
          * So, convert other types into character types.
          DO CASE
             CASE FIELD_TYPE = "C"
                KeyExp = KeyExp - "+" - FIELD_NAME
             CASE FIELD_TYPE ="N"
                IF FIELD_DEC = 0
                   KeyExp = KeyExp - "+STR(" - FIELD_NAME - ;
                              "," - STR(FIELD_LEN,2) - ")"
                ELSE
                   KeyExp = KeyExp - "+STR(" - FIELD_NAME -"," - ;
                      STR(FIELD_LEN,2) - "," -STR(FIELD_DEC,1)-")"
                ENDIF
             CASE FIELD_TYPE = "D"
                KeyExp = KeyExp - "+DTOC(" - FIELD_NAME - ")"
          ENDCASE
          KeyExp = TRIM(KeyExp)
          KeyLen = KeyLen + FIELD_LEN
       ELSE
          NonKeyFlds = TRIM(NonKeyFlds + "," + FIELD_NAME)
       ENDIF
       SKIP
    ENDDO
    IF "" <> KeyExp
       KeyExp = SUBSTR(KeyExp,2)
    ENDIF
    IF "" <> NonKeyFlds
       NonKeyFlds = SUBSTR(NonKeyFlds,2)
       AllFields = .F.
    ENDIF
RETURN


*****************************************************************
* This procedure generates the key expression for indexing a    *
* database file.  At the same time, the corresponding key       *
* expression is generated for the second relation.              *
*****************************************************************

PROCEDURE GETKEY2
PARAMETERS KeyExp1, KeyExp2, AllFields, NonKey1, NonKey2

    KeyLen = 0
    SELECT 2
    USE R2_STRUC
    SELECT 1
    USE R1_STRUC
    SET RELATION TO RECNO() INTO R2_STRUC
    SELECT 1
    DO WHILE .NOT. EOF() .AND. KeyLen <= 100
       IF FIELD_TYPE <> "L" .AND. KeyLen + FIELD_LEN <= 100
```

```
         DO CASE
            CASE FIELD_TYPE = "C"
               KeyExp1 = KeyExp1 - "+" - FIELD_NAME
               KeyExp2 = KeyExp2 - "+" - B->FIELD_NAME
            CASE FIELD_TYPE ="N"
               IF FIELD_LEN = B->FIELD_LEN
                  IF FIELD_DEC = 0
                     KeyExp1 = KeyExp1 - "+STR(" - FIELD_NAME - ;
                               "," - STR(FIELD_LEN,2) - ")"
                     KeyExp2 = KeyExp2 - "+STR(" - B->FIELD_NAME;
                             - "," - STR(B->FIELD_LEN,2) - ")"
                  ELSE
                     KeyExp1 = KeyExp1 - "+STR(" -FIELD_NAME - ;
                               "," - STR(FIELD_LEN,2) - "," ;
                               - STR(FIELD_DEC,1)-")"
                     KeyExp2 = KeyExp2 - "+STR(" - B->FIELD_NAME;
                             - "," - STR(B->FIELD_LEN,2) - "," ;
                             - STR(B->FIELD_DEC,1) - ")"
                  ENDIF
               ELSE
                  AllFields = .F.
               ENDIF
            CASE FIELD_TYPE = "D"
               KeyExp1 = KeyExp1 - "+DTOC(" - FIELD_NAME - ")"
               KeyExp2 = KeyExp2 - "+DTOC(" - B->FIELD_NAME - ")"
         ENDCASE
         KeyExp1 = TRIM(KeyExp1)
         KeyExp2 = TRIM(KeyExp2)
         KeyLen = KeyLen + FIELD_LEN
      ELSE
         AllFields = .F.
         NonKey1 = TRIM(NonKey1 + "," + FIELD_NAME)
         NonKey2 = TRIM(NonKey2 + "," + B->FIELD_NAME)
      ENDIF
      SKIP
   ENDDO
   IF "" <> KeyExp1
      KeyExp1 = SUBSTR(KeyExp1,2)
      KeyExp2 = SUBSTR(KeyExp2,2)
   ENDIF
   IF "" <> NonKey1
      NonKey1 = SUBSTR(NonKey1,2)
      NonKey2 = SUBSTR(NonKey2,2)
   ENDIF
   SET RELATION
RETURN


*****************************************************************
* This procedure performs the intersection operation, which will*
* produce a file of common records of the first and second files*
*****************************************************************

PROCEDURE INTER
PARAMETERS REL1, REL2, Result, Mevalstep
```

```
Mevalstep = REL1 + " INTER " + REL2
IF REL1 = REL2
   * Result is same as the first relation.
   DO GENTEMP WITH TempFile, TempidNo
   USE &REL1
   COPY TO &TempFile
   USE
   Result = TempFile
   RETURN
ENDIF
Compatible = .T.
STORE "" TO R1Fields, R2Fields
DO CHK_COMP WITH REL1, REL2, Compatible, R1Fields, R2Fields
IF .NOT. Compatible
   ErrMsg = "Two relations are not union compatible"
ELSE
   DO GENTEMP WITH TempFile, TempidNo
   USE &REL1
   IF EOF()
      * Result is an empty file.
      CREATE &TempFile FROM R1_STRUC
   ELSE
      USE &REL2
      IF EOF()
         * Result is an empty relation.
         CREATE &TempFile FROM R1_STRUC
      ELSE
         USE &REL1
         COPY TO &TempFile
         SET DELETED OFF
         STORE "" TO KeyExp1, KeyExp2
         AllFields = .T.
         STORE "" TO NonKey1, NonKey2
         DO GETKEY2 WITH KeyExp1, KeyExp2, AllFields, ;
                         NonKey1, NonKey2
         STORE TempFile TO DelFile1, DelFile2
         IF LEN(KeyExp1) = 0
            DO DELBYSEQ WITH TempFile, REL2
         ELSE
            DO DELBYRAN WITH TempFile, REL2
            DelFile2 = DelFile2 + ".NDX"
         ENDIF
         DelFile1 = TempFile + ".DBF"
         DO GENTEMP WITH TempFile, TempidNo
         SELECT 1
         COPY TO &TempFile FOR DELETED()
         USE &TempFile
         RECALL ALL
         CLOSE DATABASES
         DELETE FILE &DelFile1
         IF FILE(DelFile2)
            DELETE FILE &DelFile2
         ENDIF
         SET DELETED ON
```

```
            ENDIF
         ENDIF
         USE RAFNAMES
         DO REPORTC WITH REL2, R2Fields, R1Fields
      ENDIF
      CLOSE DATABASES
      DELETE FILE R1_STRUC.DBF
      DELETE FILE R2_STRUC.DBF
      Result = TempFile
   RETURN


   *********************************************************************
   * This procedure performs join operations including natural-join*
   *********************************************************************

   PROCEDURE JOINN
   PARAMETERS REL1, REL2, CondExp, CondData, Result, Mevalstep

      OldCondExp = SUBSTR(CondExp, 1, AT("B->",CondExp)-1) + ;
                   SUBSTR(CondExp, AT("B->",CondExp)+3)
      Mevalstep = REL1 + " JOIN " + OldCondExp + " " + REL2
      STORE .F. TO  EquiJoin
      DelFields = ","
      STORE " " TO FldType1, FldType2
      CondData = CondData + " "
      DO WHILE AT(")", CondData) > 0
         FldPair = SUBSTR(CondData, 2, AT(")",CondData)-2)
         IF SUBSTR(FldPair,1,1) = "="
            EquiJoin = .T.
            FldPair = SUBSTR(FldPair, 2)
         ENDIF
         Field1 = SUBSTR(FldPair, 1, AT(":",FldPair)-1)
         Field2 = SUBSTR(FldPair, AT(":",FldPair)+1)
         DO CHKFIELD WITH REL1, Field1, FldType1, CondExp
         IF FldType1 = "U"
            ErrMsg = "Field " + Field1 + " does not exist";
                   + " in file " + REL1
            EXIT
         ENDIF
         DO CHKFIELD WITH REL2, Field2, FldType2, CondExp
         IF FldType2 = "U"
            ErrMsg = "Field " + Field2 + " does not exist " ;
                   + "in File " + REL2
            EXIT
         ENDIF
         IF FldType1 <> FldType2
            ErrMsg = "Data type mismatch between " + Field1 ;
                   + " and " + Field2
            EXIT
         ELSE
            IF EquiJoin
               DelFields = DelFields + Field2 + ","
               USE RAFNAMES
               DO REPORTC WITH REL2, Field2, Field1
```

```
         ENDIF
      ENDIF
      CondData = SUBSTR(CondData, AT(")",CondData)+1)
   ENDDO
   IF "" = ErrMsg
      FldNameDup = .F.
      FldList = ""
      DO GETFLDS WITH REL1,REL2, FldNameDup, DelFields, FldList
      DO GENTEMP WITH TempFile, TempidNo
      IF .NOT. FldNameDup
         SELECT 2
         USE &REL2
         SELECT 1
         USE &REL1
         JOIN WITH &REL2 TO &TempFile FOR &CondExp ;
              FIELDS &FldList
      ELSE
         DO NEWCOPY WITH REL2, "NEWREL2", "R2_STRUC"
         SELECT 2
         USE NEWREL2
         SELECT 1
         USE &REL1
         JOIN WITH NEWREL2 TO &TempFile FOR &CondExp ;
              FIELDS &FldList
      ENDIF
      CLOSE DATABASES
      RUN ERASE R?_STRUC.*
      IF FILE("NEWREL2.DBF")
         DELETE FILE NEWREL2.DBF
      ENDIF
      Result = TempFile
   ENDIF
RETURN


*****************************************************************
* This procedure performs the difference operation and produces *
* a result file.  The result file consists of records appearing *
* only in the first file.                                       *
*****************************************************************

PROCEDURE MINUS
PARAMETERS REL1, REL2, Result, Mevalstep

   Mevalstep = REL1 + " MINUS " + REL2
   IF REL1 = REL2
      * Result is an empty file. So, copy the first file's
        structure and leave it empty.
      DO GENTEMP WITH TempFile, TempidNo
      USE &REL1
      COPY TO R1_STRUC STRUCTURE EXTENDED
      CREATE &TempFile FROM R1_STRUC
      DELETE FILE R1_STRUC.DBF
      Result = TempFile
      RETURN
```

```
   ENDIF
   Compatible = .T.
   STORE "" TO R1Fields, R2Fields
   DO CHK_COMP WITH REL1, REL2, Compatible, R1Fields, R2Fields
   IF .NOT. Compatible
      ErrMsg = "Two relations are not union compatible"
   ELSE
      DO GENTEMP WITH TempFile, TempidNo
      USE &REL1
      IF EOF()
         * Result is an empty file.
         CREATE &TempFile FROM R1_STRUC
      ELSE
         COPY TO &TempFile
         USE &REL2
         IF .NOT. EOF()
            STORE "" TO KeyExp1, KeyExp2
            AllFields = .T.
            STORE "" TO NonKey1, NonKey2
            DO GETKEY2 WITH KeyExp1, KeyExp2, AllFields, ;
                            NonKey1, NonKey2
            IF LEN(KeyExp1) = 0
               DO DELBYSEQ WITH TempFile, REL2
            ELSE
               DO DELBYRAN WITH TempFile, REL2
            ENDIF
            SET DELETED OFF
            SELECT 1
            LOCATE FOR DELETED()
            IF .NOT. EOF()
               PACK
            ENDIF
            SET DELETED ON
         ENDIF
      ENDIF
      USE RAFNAMES
      DO REPORTC WITH REL2, R2Fields, R1Fields
   ENDIF
   CLOSE DATABASES
   DELETE FILE R1_STRUC.DBF
   DELETE FILE R2_STRUC.DBF
   Result = TempFile
RETURN


*****************************************************************
* This procedure creates a new file.  The new file is a copy of *
* one file, but has different field names.                      *
*****************************************************************

PROCEDURE NEWCOPY
PARAMETERS OldFile, NewFile, FileStruc

   CREATE &NewFile FROM &FileStruc
   USE &OldFile
```

```
   COPY TO TEMPTEXT DELIMITED
   USE &NewFile
   APPEND FROM TEMPTEXT DELIMITED
   USE
   DELETE FILE TEMPTEXT.TXT
RETURN


*****************************************************************
* This procedure replaces a field name, within the expression,  *
* with a changed name.                                          *
*****************************************************************

PROCEDURE NEWEXP
PARAMETERS Exp, OldField, NewField

   DO WHILE AT(OldField, Exp) > 0
      Exp = SUBSTR(Exp, 1, AT(OldField,Exp)-1) + TRIM(NewField) ;
            + SUBSTR(Exp, AT(OldField,Exp)+LEN(OldField))
   ENDDO
RETURN


*****************************************************************
* This procedure pops the top item from the evaluation stack.   *
* If the top item is a record number then the procedure gets the*
* contents of that record.                                      *
*****************************************************************

PROCEDURE POP
PARAMETERS StackLine, TopItem

   IF AT(";", StackLine) > 0
      TopItem = SUBSTR(StackLine, 1, AT(";",StackLine)-1)
      IF VAL(TopItem) <> 0
         GO VAL(TopItem)
         TopItem = TRIM(CONTENT)
      ENDIF
      StackLine = SUBSTR(StackLine, AT(";",StackLine)+1)
   ENDIF
RETURN


*****************************************************************
* This procedure calls another appropriate procedure, depending *
* on the operation code, to perform one of the relational       *
* algebra operations.                                           *
*****************************************************************

PROCEDURE PROCESS
PARAMETERS OpCode, EvalStack, NoError

   STORE "" TO ErrMsg, MevalStep
   STORE " " TO REL,REL1,REL2, AttrList, CondExp,CondData, Result
   IF OpCode = "U" .OR. OpCode = "M" .OR. OpCode = "I" .OR. ;
```

```
      OpCode = "T" .OR. OpCode = "D"
      DO POP WITH EvalStack, REL2
      DO POP WITH EvalStack, REL1
   ENDIF
   DO CASE
      CASE OpCode = "U"
         DO UNION WITH REL1, REL2, Result, Mevalstep
      CASE OpCode = "M"
         DO MINUS WITH REL1, REL2, Result, Mevalstep
      CASE OpCode = "I"
         DO INTER WITH REL1, REL2, Result, Mevalstep
      CASE OpCode = "T"
         DO TIMES WITH REL1, REL2, Result, Mevalstep
      CASE OpCode = "D"
         DO DIVISION WITH REL1, REL2, Result, Mevalstep
      CASE OpCode = "P"
         DO POP WITH EvalStack, AttrList
         DO POP WITH EvalStack, REL
         DO PROJECT WITH REL, AttrList, Result, Mevalstep
      CASE OpCode = "S"
         DO POP WITH EvalStack, CondData
         DO POP WITH EvalStack, CondExp
         DO POP WITH EvalStack, REL
         DO SELEC WITH REL, CondExp, CondData, Result, Mevalstep
      CASE OpCode = "J"
         DO POP WITH EvalStack, REL2
         DO POP WITH EvalStack, CondData
         DO POP WITH EvalStack, CondExp
         DO POP WITH EvalStack, REL1
         DO JOINN WITH REL1, REL2, CondExp, CondData, ;
                          Result,Mevalstep
   ENDCASE
   IF "" <> ErrMsg THEN
      NoError = .F.
      DO GENTEMP WITH TempFile, TempidNo
      Mevalstep = TempFile + AssignMark + Mevalstep
   ELSE
      Mevalstep = Result + AssignMark + Mevalstep
      EvalStack = Result + ";" + EvalStack
      DO DELTEMPS
   ENDIF
   USE RASTEPS
   APPEND BLANK
   REPLACE EVALSTEP WITH Mevalstep
   CLOSE DATABASES
RETURN


*****************************************************************
* This procedure performs the projection operation to produce   *
* the result file.  The result file is a vertical subset of the *
* given file.                                                   *
*****************************************************************

PROCEDURE PROJECT
```

```
PARAMETERS REL, AttrList, Result, Mevalstep

   Mevalstep = REL + " [" + AttrList + "]"
   AtrListErr = .F.
   DO CHKFLIST WITH REL, AttrList, AtrListErr
   IF .NOT. AtrListErr
      DO GENTEMP WITH TempFile, TempidNo
      USE &REL
      IF .NOT. EOF()
         MaybeDup = .T.
      ENDIF
      COPY TO &TempFile FIELDS &AttrList
      USE
      Result = TempFile
   ENDIF
RETURN


*****************************************************************
* This procedure removes duplicate records from the file.       *
*****************************************************************

PROCEDURE REMOVDUP
PARAMETERS Result
   InFile = Result
   USE &InFile
   GO BOTTOM
   IF EOF() .OR. RECNO() = 1
      USE
      RETURN
   ENDIF
   COPY TO STRUFILE STRUCTURE EXTENDED
   STORE "" TO KeyExp, NonKeyFlds, FldList
   AllFields = .T.
   DO GETKEY1 WITH KeyExp, NonKeyFlds, AllFields
   IF LEN(KeyExp) = 0
      DO GENTEMP WITH TempFile, TempidNo
      CREATE &TempFile FROM STRUFILE
      ResultFlds = NonKeyFlds
      DO ADDUNIQ WITH TempFile, InFile, ResultFlds, NonKeyFlds
      Result = TempFile
   ELSE
      SET DELETED OFF
      USE &Result
      INDEX ON &KeyExp TO &Result
      USE &Result INDEX &Result
      STORE &KeyExp TO Mkey2
      DO WHILE .NOT. EOF()
         Mkey1 = Mkey2
         PTR1 = RECNO()
         SKIP
         STORE &KeyExp TO Mkey2
         DO WHILE Mkey1 = Mkey2
            IF AllFields
               DELETE
```

```
            ELSE
               PTR2 = RECNO()
               Duplicate = .T.
               DO COMPARE1 WITH NonKeyFlds, PTR1, PTR2, Duplicate
               IF Duplicate
                  DELETE
               ENDIF
            ENDIF
            SKIP
            STORE &KeyExp TO Mkey2
         ENDDO
      ENDDO
      LOCATE FOR DELETED()
      IF .NOT. EOF()
         PACK
      ENDIF
   ENDIF
   CLOSE DATABASES
   DELETE FILE STRUFILE.DBF
   IF Result <> InFile
      DelFile = InFile + ".DBF"
   ELSE
      DelFile = Result + ".NDX"
   ENDIF
   DELETE FILE &DelFile
   SET DELETED ON
RETURN


*****************************************************************
* This procedure adds new records to the file, in which informa-*
* tion about a field name change is kept as a record.           *
*****************************************************************

PROCEDURE REPORTC
PARAMETERS RelName, OldFields, NewFields

   DO WHILE AT(",", OldFields) > 0
      OldField = SUBSTR(OldFields, 1, AT(",",OldFields)-1)
      NewField = SUBSTR(NewFields, 1, AT(",",NewFields)-1)
      APPEND BLANK
      REPLACE FILENAME WITH RelName, OLDFLDNAME WITH OldField, ;
              NEWFLDNAME WITH NewField
      OldFields = SUBSTR(OldFields, AT(",",OldFields)+1)
      NewFields = SUBSTR(NewFields, AT(",",NewFields)+1)
   ENDDO
   APPEND BLANK
   REPLACE FILENAME WITH RelName, OLDFLDNAME WITH OldFields, ;
           NEWFLDNAME WITH NewFields
RETURN

*****************************************************************
* This procedure performs the selection operation, which will   *
* produce a file of horizontal subset of the given file.        *
*****************************************************************
```

```
PROCEDURE SELEC
PARAMETERS REL, CondExp, CondData, Result, Mevalstep

   Mevalstep = REL + " WHERE " + CondExp
   FldType = " "
   CondData = CondData + " "
   DO WHILE AT(")", CondData) > 0
      CondFactor = SUBSTR(CondData, 2, AT(")",CondData)-2)
      FldName = SUBSTR(CondFactor, 1, AT(":",CondFactor)-1)
      FldValType = SUBSTR(CondFactor, AT(":",CondFactor)+1)
      DO CHKFIELD WITH REL, FldName, FldType, CondExp
      IF FldType = "U"
         ErrMsg = "Field " + FldName + " does not exist " + ;
                  "in file " + REL
         EXIT
      ELSE
         IF FldType <> FldValType
            DO CASE
               CASE FldValType = "C"
                  TypeString = "Character"
               CASE FldValType = "N"
                  TypeString = "Number"
               CASE FldValType = "L"
                  TypeString = "Logical"
               CASE FldValType = "D"
                  TypeString = "Date"
            ENDCASE
            ErrMsg="Mismatched data types: " + FldName ;
                   + " is not " + TypeString + " type"
            EXIT
         ENDIF
      ENDIF
      CondData = SUBSTR(CondData, AT(")",CondData)+1)
   ENDDO
   IF "" = ErrMsg
      DO GENTEMP WITH TempFile, TempidNo
      USE &REL
      COPY TO &TempFile FOR &CondExp
      USE
      Result = TempFile
   ENDIF
RETURN


*****************************************************************
* This procedure performs the Cartesian product operation.      *
* Before the merging of two files, the procedure checks the     *
* field name duplication, and generate new names for those dup- *
* licate field names.                                           *
*****************************************************************

PROCEDURE TIMES
PARAMETERS REL1, REL2, Result, Mevalstep
```

```
   Mevalstep = REL1 + " TIMES " + REL2
   FldNameDup = .F.
   DO CHKFLDS1 WITH REL1, REL2, FldNameDup
   DO GENTEMP WITH TempFile, TempidNo
   IF .NOT. FldNameDup
      SELECT 2
      USE &REL2
      SELECT 1
      USE &REL1
      JOIN WITH &REL2 TO &TempFile FOR .T.
   ELSE
      DO NEWCOPY WITH REL2, "NEWREL2", "R2_STRUC"
      SELECT 2
      USE NEWREL2
      SELECT 1
      USE &REL1
      JOIN WITH NEWREL2 TO &TempFile FOR .T.
   ENDIF
   CLOSE DATABASES
   RUN ERASE R?_STRUC.*
   IF FILE("NEWREL2.DBF")
      DELETE FILE NEWREL2.DBF
   ENDIF
   Result = TempFile
RETURN


*****************************************************************
* This procedure performs the union operation by adding all the *
* records of the first file to the second file.                 *
*****************************************************************

PROCEDURE UNION
PARAMETERS REL1, REL2, Result, Mevalstep

   Mevalstep = REL1 + " UNION " + REL2
   IF REL1 = REL2
      DO GENTEMP WITH TempFile, TempidNo
      USE &REL1
      COPY TO &TempFile
      USE
      Result = TempFile
      RETURN
   ENDIF
   Compatible = .T.
   STORE "" TO R1Fields, R2Fields
   DO CHK_COMP WITH REL1, REL2, Compatible, R1Fields, R2Fields
   IF .NOT. Compatible
      ErrMsg = "Two relations are not union compatible"
   ELSE
      DO GENTEMP WITH TempFile, TempidNo
      USE &REL1
      COPY TO &TempFile
      STORE .F. TO R1Empty, R2Empty
      SELECT 1
```

```
      USE &TempFile
      IF EOF()
         R1Empty = .T.
      ENDIF
      SELECT 2
      USE &REL2
      IF EOF()
         R2Empty = .T.
      ELSE
         DO WHILE .NOT. EOF()
            SELECT 1
            DO ADDREC WITH R1Fields, R2Fields
            SELECT 2
            SKIP
         ENDDO
      ENDIF
      IF .NOT. (R1Empty .OR. R2Empty)
         MaybeDup = .T.
      ENDIF
      USE RAFNAMES
      DO REPORTC WITH REL2, R2Fields, R1Fields
   ENDIF   [ .NOT. Compatible or ELSE ]
   CLOSE DATABASES
   DELETE FILE R1_STRUC.DBF
   DELETE FILE R2_STRUC.DBF
   Result = TempFile
RETURN


*****************************************************************
*                 PROCEDURE FILE: RARESULT.PRG                  *
* This procedure file contains 7 procedures which will be used  *
* to display, print, and save the results.                      *
*****************************************************************

*****************************************************************
* This procedure displays or prints the contents of the result  *
* file in a table format.                                       *
*****************************************************************

PROCEDURE DISPLAY1
PARAMETERS ColWidths

   Heading = "|"
   Underlines = "+"
   DO GENHEAD WITH Heading, UnderLines
   @ RowNo, 0 SAY Heading
   RowNo = RowNo + 1
   @ RowNo, 0 SAY Underlines
   SELECT 1
   USE STRUFILE
   SELECT 2
   USE &Result
   DO WHILE .NOT. EOF()
```

```
      RowNo = RowNo + 1
      IF RowNo > PageLen
         IF Choice = "S"
            @ 23, 4 SAY ">> More Records -- Press any key " + ;
                     "to continue"
            WAIT ""
            CLEAR
         ELSE
            EJECT
         ENDIF
         @ 1, 0  SAY Heading
         @ 2, 0  SAY Underlines
         RowNo = 3
      ENDIF
      @ RowNo, 0 SAY "|"
      ColNo = 1
      WidthList = ColWidths
      SELECT 1
      GO TOP
      DO WHILE .NOT. EOF()
         FldName = FIELD_NAME
         ColWidth= VAL( SUBSTR( WidthList,1,AT(",",WidthList)-1))
         Spaces = INT((ColWidth - FIELD_LEN)/2)
         SELECT 2
         @ RowNo, ColNo+Spaces  SAY &FldName
         ColNo = ColNo + ColWidth
         @ RowNo, ColNo  SAY "|"
         ColNo = ColNo + 1
         WidthList = SUBSTR(WidthList, AT(",",WidthList)+1)
         SELECT 1
         SKIP
      ENDDO
      SELECT 2
      SKIP
   ENDDO
   CLOSE DATABASES
   RELEASE ALL
RETURN


*****************************************************************
* This procedure displays or prints the contents of the result  *
* file in linear format.                                        *
*****************************************************************

PROCEDURE DISPLAY2

   USE &Result
   COPY TO STRUFILE STRUCTURE EXTENDED
   SELECT 1
   USE STRUFILE
   SELECT 2
   USE &Result
   DO WHILE .NOT. EOF()
      RecordNo = RECNO()
```

```
      IF RowNo > PageLen - 3
         IF Choice = "P"
            EJECT
         ELSE
            @ 23, 4 SAY ">> More Records -- Press any key " + ;
                     "to continue"
            WAIT ""
            CLEAR
         ENDIF
         RowNo = 1
      ENDIF
      @ RowNo, 1 SAY "Record# Fieldname  Content"
      @ RowNo+1, 1 SAY "------- ---------  --------"
      RowNo = RowNo + 2
      @ RowNo, 1 SAY STR(RecordNo,6)
      SELECT 1
      GO TOP
      DO WHILE .NOT. EOF()
         FldName = FIELD_NAME
         SELECT 2
         @ RowNo, 9 SAY FldName
         @ RowNo, 20 SAY &FldName
         RowNo = RowNo + 1
         IF RowNo > PageLen
            IF Choice = "P"
               EJECT
            ELSE
               @ 23, 4 SAY ">> More Records -- Press any key" + ;
                        " to continue"
               WAIT ""
               CLEAR
            ENDIF
            RowNo = 1
         ENDIF
         SELECT 1
         SKIP
      ENDDO
      IF RowNo < PageLen
         RowNo = RowNo + 1
      ENDIF
      SELECT 2
      SKIP
   ENDDO
   CLOSE DATABASES
   RELEASE ALL
RETURN


*****************************************************************
* This procedure determines how and where the result is to be   *
* directed, by interacting with the user.                       *
*****************************************************************

PROCEDURE DISP_RES
```

```
Choice = " "
DO GETWHERE WITH Choice
CLEAR
@ 1, 0 SAY "Your Query is:"
RowNo = 2
USE RADATA
DO WHILE CONTENT <> ";"
   RowNo = RowNo + 1
   @ RowNo, 0  SAY TRIM(CONTENT)
   SKIP
ENDDO
RowNo = RowNo + 2
USE &Result
GO BOTTOM
IF EOF()
   @ RowNo, 3 SAY "*** No qualified records for this Query **"
   USE
   RETURN
ENDIF
PageWidth = 80
IF Choice = "P"
   Answer = " "
   DO WHILE .NOT. ( Answer $ "AaBb" )
      STORE " " TO Answer
      @ RowNo, 1 SAY ">> What is your printer's page width,"+;
                 " (A) 80 or (B) 132 ? " GET Answer PICTURE "!"
      READ
      CLEAR GETS
   ENDDO
   IF UPPER(Answer) = "B"
      PageWidth = 132
   ENDIF
ENDIF
STORE "" TO ColWidths
USE &Result
COPY TO STRUFILE STRUCTURE EXTENDED
USE STRUFILE
GO BOTTOM
TotalWidth = RECNO() + 1
GO TOP
DO WHILE .NOT. EOF() .AND. TotalWidth < PageWidth
   IF LEN(TRIM (FIELD_NAME)) > FIELD_LEN
      ColWidth = LEN(TRIM(FIELD_NAME))
   ELSE
      ColWidth = FIELD_LEN
   ENDIF
   TotalWidth = TotalWidth + ColWidth
   ColWidths = ColWidths + STR(ColWidth,3) + ","
   SKIP
ENDDO
IF Choice = "P"
   Answer = " "
   @ 21, 1 SAY "*** Please turn on printer and make sure" + ;
               " 'on-line' lamp is lit"
   @ 22, 1 SAY ">> Press any key when ready...";
```

```
            GET Answer PICTURE "!"
    READ
    PageLen = 58
    SET DEVICE TO PRINT
    EJECT
    RowNo = 1
 ELSE
    PageLen = 22
    @ RowNo, 0 SAY "The Result of Your Query is:"
    RowNo = RowNo + 2
 ENDIF
 IF EOF() .AND. TotalWidth <= PageWidth
    ColWidths = ColWidths + " "
    DO DISP1 WITH ColWidths
 ELSE
    DO DISP2
 ENDIF
 IF Choice = "P"
    SET DEVICE TO SCREEN
 ENDIF
 DELETE FILE STRUFILE.DBF
 RELEASE ALL
RETURN


*******************************************************************
* This procedure completes the processing of a user's query, by *
* displaying and saving the result based on the user's wish.    *
*******************************************************************

PROCEDURE FINISH
PARAMETERS Result

   DO DISP_RES
   IF "TEMP" $ Result
      @ 23, 0
      STORE " " TO Answer
      @ 23, 2 SAY ">> Press any key to continue...";
              GET Answer PICTURE "!"
      READ
      CLEAR
      STORE " " TO Answer
      DO WHILE .NOT. (Answer $ "YyNn")
         STORE " " TO Answer
         @ 3,1 SAY ">> Do you wish to save the result file (Y/N)? ";
                GET Answer PICTURE "!"
         READ
         CLEAR GETS
      ENDDO
      Result = Result + ".DBF"
      IF UPPER(Answer) = "N"
         DELETE FILE &Result
         RETURN
      ENDIF
      DO SAVE_RES
```

```
   ENDIF
   @ 23, 0
   Answer = " "
   @ 23, 2 SAY ">> Press any key to return to the main menu..." ;
           GET Answer PICTURE "!"
   READ
   RELEASE ALL
RETURN

**********************************************************************
* This procedure generates a heading, which will be used when       *
* the contents of result file is displayed or printed in a table*
* format.                                                            *
**********************************************************************

PROCEDURE GENHEAD
PARAMETERS Heading, Underlines

   WidthList = ColWidths
   Underline = "------------------------------------";
             + "------------------------------------"
   USE STRUFILE
   DO WHILE .NOT. EOF()
      ColWidth = VAL( SUBSTR( WidthList, 1, AT(",",WidthList)-1))
      Blanks = SPACE(ColWidth - LEN(TRIM(FIELD_NAME)))
      Heading = Heading + TRIM(FIELD_NAME) + Blanks + "|"
      Underlines= Underlines + SUBSTR(Underline,1,ColWidth) + "+"
      WidthList = SUBSTR(WidthList, AT(",",WidthList)+1)
      SKIP
   ENDDO
RETURN

**********************************************************************
* This procedure asks the user where he wants to see the            *
* results, on the screen or printer.                                 *
**********************************************************************

PROCEDURE GETWHERE
PARAMETERS Choice

   DO WHILE .NOT. ( Choice $ "PpSs" )
      STORE " " TO Choice
      @ 22, 1 SAY ">> Where do you want to see the result," +;
                " (S)creen or (P)rinter? " GET Choice PICTURE "!"
      READ
      CLEAR GETS
   ENDDO
   Choice = UPPER(Choice)
RETURN
```

```
**********************************************************************
* This procedure asks the user for a new file name which will be*
* assigned to the result file.  If a file name is syntactically *
* correct and is not an existing file name, then this name will *
* replace the result file's old name which was temporary one.   *
**********************************************************************

PROCEDURE SAVE_RES

   STORE "ABCDEFGHIJKLMNOPQRSTUVWXYZ" TO Alphas
   NameChars = Alphas + "0123456789_"
   STORE "        " TO NewName
   DO WHILE NewName = "        "
      @ 5, 1 SAY ">> Enter new file name without extension " + ;
          "(or </> to exit) -->" GET NewName PICTURE "!!!!!!!!"
      READ
      CLEAR GETS
   ENDDO
   IF NewName = "/"
      DELETE FILE &Result
      RETURN
   ENDIF
   RowNo = 5
   NewName = UPPER(TRIM(NewName))
   DO WHILE .T.
      RowNo = RowNo + 2
      NewFile = NewName + ".DBF"
      IF FILE(NewFile)
         @ RowNo,1 SAY "*** Data file " +NewName+" already exist"
      ELSE
         IF .NOT. ( SUBSTR(NewName,1,1) $ Alphas )
            @ RowNo, 1 SAY "*** Illegal file name"
         ELSE
            CharPos = 2
            DO WHILE CharPos <= LEN(NewName)
               IF .NOT. (SUBSTR(NewName,CharPos,1) $ NameChars)
                  @ RowNo, 1 SAY "*** Illegal file name"
                  EXIT
               ENDIF
               CharPos = CharPos + 1
            ENDDO
            IF CharPos > LEN(NewName)
               EXIT
            ENDIF
         ENDIF
      ENDIF
      RowNo = RowNo + 2
      @ RowNo, 1
      STORE "        " TO NewName
      DO WHILE NewName = "        "
         @ RowNo, 1 SAY ">> Re-enter file name (or </> to " + ;
                "exit) --> " GET NewName PICTURE "!!!!!!!!"
         READ
         CLEAR GETS
      ENDDO
```

```
      IF NewName = "/"
         DELETE FILE &Result
         RETURN
      ENDIF
      NewName = UPPER(TRIM(NewName))
   ENDDO
   RENAME &Result TO &NewFile
   RELEASE ALL
RETURN
```

```
{*****************************************************************}
{* This Pascal program is called by the system's main module    *}
{* which is a dBASE III program.  The program accepts a user's *}
{* quary, parses it, and then outputs the parsed expression to *}
{* a text file called 'SCANOUT.DAT'.  The origianl user's query*}
{* expression is also written to this file for future use.      *}
{*****************************************************************}

PROGRAM RAPARSER;

CONST AttrNameLen = 10;
      MaxExpElements = 30;
      MaxQLines = 5;
      RelNameLen = 8;
      WordStrLen = 63;
      BackSpace = #8;
      Enter = #13;
      Esc = #27;
      LineFeed = #10;
      Blank = ' ';
      Comma = ',';
      LeftParen = '(';
      RightParen = ')';
      Marker = '?';
      Null = '';
      Semicolon = ';';
      OutFileName = 'SCANOUT.DAT';
      OpWords =' UNION MINUS INTER TIMES DIVIDEBY WHERE JOIN [ ';

TYPE CompOpStr = STRING[2];
     ExpStr = STRING[255];
     LineStr = STRING[127];
     OpCodeStr = STRING[2];
     PartExpStr = STRING[127];
     RelNameStr = STRING[RelNameLen];
     VarKindStr = STRING[10];
     WordStr = STRING[WordStrLen];

VAR AttrList, CondExp : PartExpStr;
    Alphas, Digits, VarNameChars : SET OF CHAR;
    Delimiters : SET OF CHAR;
    ErrMsg : LineStr;
    ErrPlace : CHAR;
    PosA, PosC, PosQ : integer;
    IndexNew, i : BYTE;
    CondExpLen : BYTE;
    NextWord, NextCondWord : wordstr;
    NewQueryExp : Array[1..MaxExpElements] OF PartExpStr;
    NoError : BOOLEAN;
    ParseDone, TryAgain : BOOLEAN;
    QLineNo : BYTE;
    OutFile : TEXT;
    QueryExp : ExpStr;
    QueryLines : ARRAY [1..MaxQLines] OF LineStr;
```

```
{*****************************************************************}
{* This procedure resets an error flag, and clears the string   *}
{* and array of string variables before the query acceptance.   *}
{*****************************************************************}

PROCEDURE Initialize;

VAR i : BYTE;

  BEGIN
    NoError := TRUE;
    ErrMsg := '';
    QueryExp := '';
    FOR i := 1 TO MaxQLines DO  QueryLines[i] := '';
    FOR i := 1 TO MaxExpElements DO  NewQueryExp[i ] := '';
  END;  { procedure Initialize }


{*****************************************************************}
{* This procedure checks if the file, which is specified in a   *}
{* user's query expression, exists in the current directory.    *}
{*****************************************************************}

PROCEDURE CheckExist(RelName:RelNameStr);

VAR DBFile : FILE;
    Exist : BOOLEAN;

  BEGIN
    ASSIGN(DBFile, RelName+'.DBF');
    {$I-}
    RESET(DBFile);
    {$I+}
    Exist := (IORESULT = 0);
    IF Exist THEN
      CLOSE(DBFile)
    ELSE
    BEGIN
      ErrMsg := 'File ' + RelName + ' does not exist';
      NoError := FALSE
    END;
  END;  { procedure CheckExist }


{*****************************************************************}
{* This procedure reports an error to the user.  It displays    *}
{* an error message and a user's query expression with a marker*}
{* underneath where the error has been found.                   *}
{*****************************************************************}

PROCEDURE ReportError(ErrMsg:LineStr; ErrPlace:CHAR);

VAR ErrLineNo, ErrPos, i, NumChar : BYTE;
```

```
 BEGIN
   WRITELN;
   WRITELN;
   CASE ErrPlace OF
     'A': ErrPos := PosQ - LENGTH(AttrList) + PosA - 1;
     'C': ErrPos := PosQ - CondExpLen + PosC - 1;
     'Q': ErrPos := PosQ;
   END;  { case }
   IF ErrPos = LENGTH(QueryExp) THEN
   BEGIN
     ErrLineNo := QLineNo;
     ErrPos := LENGTH(QueryLines[ErrLineNo]);
   END
   ELSE
   BEGIN
     i := 1;
     NumChar := LENGTH(QueryLines[1]);
     WHILE NumChar < ErrPos DO
     BEGIN
       i := SUCC(i);
       NumChar := NumChar + LENGTH(QueryLines[i]);
     END;
     ErrPos := ErrPos - (NumChar - LENGTH(QueryLines[i]));
     ErrLineNo := i;
   END;  { else }
   WRITELN('*** Error in Query line ',ErrLineNo, ':');
   WRITELN;
   WRITELN(' ',QueryLines[ErrLineNo]);
   WRITELN(' ',Marker:ErrPos);
   WRITELN;
   WRITELN('** ':5, ErrMsg, ' **');
 END;  { procedure ReportError }


{*****************************************************************}
{*  This proceudre prints a brief syntax of relational algebra *}
{*  operations on the upper part of the screen.                  *}
{*****************************************************************}

PROCEDURE PrintSyntax;
 BEGIN
   LowVideo;
   WRITELN;
   WRITELN(' <Union>          rel1 UNION rel2', '|':4,
           '  <Join>      rel1 JOIN ( condition ) rel2');
   WRITELN(' <Difference>    rel1 MINUS rel2', '|':4,
           '  <Projection>  rel [ list of attributes ]');
   WRITELN(' <Production>    rel1 TIMES rel2', '|':4,
           '  <Selection>   rel WHERE ( condition )');
   WRITELN(' <Intersection> rel1 INTER rel2', '|':4,
           '  <Division>    rel1 DIVIDEBY rel2');
   WRITELN('-------------------------------------------------',
           '------------------------------');
   NormVideo;
 END;    { procedure PrnitSyntax }
```

```
{*****************************************************************}
{* This procedure prompts a user to decide whether he wants to *}
{* try again or not, and gets the user's response.             *}
{*****************************************************************}

PROCEDURE GetResponse(VAR Yes:BOOLEAN);

  VAR Response : CHAR;

  BEGIN
    GOTOXY(1, 24);
    TextBackground(1);
    WRITE('  Do you want to try again? (Y/N) ');
    REPEAT
      READ(KBD, Response);
    UNTIL Response IN ['Y','y','N','n'];
    WRITE(Response);
    Yes := Response IN ['Y','y'];
    TextBackground(Black);
  END;  { procedure GetResponse }


{*****************************************************************}
{*  This function converts the words, that represent specific  *}
{*  relational algebra operations, into the internal codes.    *}
{*****************************************************************}

FUNCTION OpCode(NextWord: WordStr): OpCodeStr;

  BEGIN
    IF NextWord = 'UNION' THEN
      OpCode := '*U'
    ELSE
    IF NextWord = 'MINUS' THEN
      OpCode := '*M'
    ELSE
    IF NextWord = 'INTER' THEN
      OpCode := '*I'
    ELSE
    IF NextWord = 'TIMES' THEN
      OpCode := '*T'
    ELSE
    IF NextWord = 'DIVIDEBY' THEN
      OpCode := '*D'
    ELSE
    IF NextWord = 'JOIN' THEN
      OpCode := '*J'
    ELSE
    IF NextWord = '[' THEN
      OpCode := '*P'
    ELSE
    IF NextWord = 'WHERE' THEN
      OpCode := '*S';
  END;  { function OpCode }
```

```
{*****************************************************************}
{*  This procedure stores parsed query expression items into    *}
{*  the designated array.                                        *}
{*****************************************************************}

PROCEDURE StoreExp(ExpElement:PartExpStr);

  BEGIN
    IndexNew := SUCC(IndexNew);
    NewQueryExp[IndexNew] := ExpElement;
  END;  { procedure StoreExp }


{*****************************************************************}
{* This procedure accepts a user's query, character by charac- *}
{* ter, from the keyboard and then stores it to an array.       *}
{*****************************************************************}

PROCEDURE GetQuery;

  VAR i, NumChar : INTEGER;
      InChar : CHAR;
      Escape, FuncKey : BOOLEAN;
      Printables : SET OF CHAR;

  BEGIN
    CLRSCR;
    PrintSyntax;
    GOTOXY(1,8);
    TextBackground(1);
    WRITELN('>> Enter Query.  Press <Ctrl> Z, ',
            'or <Esc> to abort');
    TextBackGround(Black);
    WRITELN;
    Escape := FALSE;
    Printables := [' '..'}'];
    NumChar := 0;
    QLineNo := 1;
    REPEAT
      REPEAT
      UNTIL KeyPressed;
      READ(KBD,InChar);
      FuncKey := FALSE;
      IF InChar = Esc THEN
      BEGIN
        IF KeyPressed THEN
        BEGIN
          READ(KBD,InChar);
          FuncKey := TRUE;
        END
        ELSE
          Escape := TRUE;
      END;
      IF (InChar <> ^Z) AND (NOT Escape) AND (NOT FuncKey) THEN
```

```
  BEGIN
    IF InChar IN Printables THEN
    BEGIN
      WRITE(InChar);
      NumChar := SUCC(NumChar);
      QueryLines[QLineNo] := QueryLines[QLineNo] + InChar;
    END
    ELSE
      IF InChar = Enter THEN
      BEGIN
        WRITE(InChar);
        WRITE(LineFeed);
        NumChar := SUCC(NumChar);
        QueryLines[QLineNo] := QueryLines[QLineNo] + ' ';
        QLineNo := SUCC(QLineNo);
      END
      ELSE
        IF InChar = BackSpace THEN
        BEGIN
          WRITE(InChar);
          WRITE(Blank);
          WRITE(InChar);
          IF LENGTH(QueryLines[QLineNo]) > 0 THEN
          BEGIN
            DELETE(QueryLines[QLineNo],
                   LENGTH(QueryLines[QLineNo]), 1);
            NumChar := NumChar - 1;
          END;
        END;
  END;  { if InChar <> ctrl-Z }
UNTIL (InChar = ^Z) OR  Escape  OR
      (NumChar = 254) OR (QlineNo > MaxQLines);
IF  Escape THEN
  ParseDone := TRUE
ELSE
  IF NumChar >= 254 THEN
    ErrMsg := 'Error -- Query expression is too long';
  IF QLineNo > MaxQLines THEN
    ErrMsg:='Error -- Query expression is in too many lines';
  IF QueryLines[QLineNo] = Null THEN
  BEGIN
    QLineNo := PRED(QLineNo);
    WHILE QueryLines[QLineNo] = Blank DO
      QLineNo := PRED(QLineNo);
  END;
  IF QLineNo = 0 THEN
    ErrMsg := 'ERROR -- Empty query: cannot be processed'
  ELSE
  BEGIN
    FOR i := 1 TO QLineNo DO
      QueryExp := QueryExp + QueryLines[i];
    QueryExp := QueryExp + Semicolon;
    i := 1;
    WHILE QueryExp[i] = Blank DO
      i := SUCC(i);
```

```
        IF QueryExp[i] = Semicolon THEN
          ErrMsg := 'ERROR -- Empty query: cannot be processed';
      END;  { else }
      IF LENGTH(ErrMsg) > 1 THEN
      BEGIN
        WRITELN;
        WRITELN('    ** ', ErrMsg);
        NoError := FALSE;
      END;
    END;  { inchar <> Esc }
  END;  { procedure GetQuery }

{****************************************************************}
{* This procedure reads an attribute list until the end of the *}
{* list is encountered, and copies it into a designated string *}
{* variable.                                                   *}
{****************************************************************}

PROCEDURE GetAttrList;

  BEGIN
    AttrList := '[';
    REPEAT
      AttrList := AttrList + QueryExp[PosQ];
      PosQ := SUCC(PosQ);
    UNTIL (QueryExp[PosQ] = ']') OR (QueryExp[PosQ] = ';');
    IF QueryExp[PosQ] = Semicolon THEN
    BEGIN
      ErrMsg := 'Missing "]" for attribute list';
      NoError := FALSE;
    END
    ELSE
    BEGIN
      AttrList := AttrList + ']';
      PosQ := SUCC(PosQ);
    End; { else }
  END; { procedure GetAttrList }

{****************************************************************}
{* This procedure reads a condition expression until its end   *}
{* is encountered, and copies it into the string variable.     *}
{****************************************************************}

PROCEDURE GetCondExp;

  VAR Pcount : BYTE;
      Matched : boolean;

  BEGIN
    WHILE QueryExp[PosQ] = BLANK DO
      PosQ := SUCC(PosQ);
    IF QueryExp[PosQ] = Semicolon THEN
      ErrMsg := 'Unexpected end, missing condition expression'
```

```
    ELSE
      IF QueryExp[PosQ] <> LeftParen THEN
        ErrMsg:= 'Parenthesized condition expression is expected'
      ELSE
      BEGIN
        Pcount := 0;
        Matched := FALSE;
        CondExp := '';
        REPEAT
          IF QueryExp[PosQ] = '(' THEN
            Pcount := Pcount + 1
          ELSE
            IF QueryExp[PosQ] = ')' THEN
              Pcount := Pcount - 1;
          IF Pcount = 0 THEN
            Matched := TRUE;
          CondExp := CondExp + QueryExp[PosQ];
          PosQ := SUCC(PosQ)
        UNTIL Matched OR (QueryExp[PosQ] = Semicolon);
        IF NOT Matched THEN
          ErrMsg := '")" is expected';
      END;  { else }
    IF LENGTH(ErrMsg) > 1 THEN
      NoError := FALSE;
  END; { procedure get cond exp}


{*****************************************************************}
{* This procedure scans the relation or attribute names.        *}
{*****************************************************************}

PROCEDURE ScanVarName(VarName:WordStr; VarNameLen:INTEGER;
                      VarKind:VarKindStr);
VAR NameLen, i : BYTE;

  BEGIN
    IF NoError THEN
    BEGIN
      IF NOT (VarName[1] IN Alphas) THEN
        ErrMsg := 'Illegal ' + VarKind + ' name: must start '
                  + 'with a letter'
      ELSE
      BEGIN
        NameLen := LENGTH(VarName);
        IF NameLen > VarNameLen THEN
          ErrMsg := VarKind + ' name is too long.'
        ELSE
        BEGIN
          i := 1;
          WHILE (VarName[i] IN VarNameChars) AND (i<=NameLen) DO
            i := SUCC(i);
          IF i <= NameLen THEN
            ErrMsg := 'Illegal symbol in ' + VarKind + ' name';
        END;  { else }
        IF LENGTH(ErrMsg) > 1 THEN
```

```
          NoError := FALSE;
        END;  { else }
      END;  { if NoError }
    END;  { procedure ScanVarName }

{**************************************************************}
{* This procedure analyzes an attribute name.  If an attribute *}
{* name has a relation name as a prefix, then the procedure   *}
{* passes that realation name to other procedure to check its *}
{* existence.                                                 *}
{**************************************************************}

PROCEDURE ScanAttrName(AttrName:WordStr);

VAR LastChPos, DotPos : BYTE;
    RelName : WordStr;

  BEGIN
    LastChPos := LENGTH(AttrName);
    DotPos := POS('.',AttrName);
    IF (DotPos > 0) AND (DotPos < LastChPos) THEN
    BEGIN
      RelName := COPY(AttrName, 1, DotPos-1);
      AttrName := COPY(AttrName, DotPos+1, LastChPos-DotPos);
      ScanVarName(RelName, RelNameLen, 'Relation');
      IF NoError THEN
        CheckExist(RelName);
    END;  { DotPos > 0 }
    ScanVarName(AttrName, AttrNameLen, 'Attribute');
  END;  { procedure ScanAttrName }

{**************************************************************}
{* This procedure checks if an attribute list is constructed  *}
{* syntactically correct, and separates each attribute name in *}
{* the list.  Then the procedure passes it to other procedure  *}
{* to check its syntactical correctness.                      *}
{**************************************************************}

PROCEDURE ScanAttrList;

VAR AttrName : WordStr;
    NewAttrList : PartExpStr;

  BEGIN
    PosA := 2;
    WHILE AttrList[PosA] = BLANK DO
      PosA := SUCC(PosA);
    IF AttrList[PosA] = ']' THEN
    BEGIN
      ErrMsg := 'Empty attribute list';
      NoError := FALSE;
    END
    ELSE
```

```
BEGIN
  NewAttrList := '';
  WHILE NoError AND (AttrList[PosA] <> ']') DO
  BEGIN
    AttrName := '';
    WHILE NOT (AttrList[PosA] IN [Blank,Comma,']']) DO
    BEGIN
      AttrName := AttrName + UPCASE(AttrList[PosA]);
      PosA := SUCC(PosA);
    END;
    IF AttrName <> Null THEN
    BEGIN
      ScanAttrName(AttrName);
      IF NoError THEN
      BEGIN
        NewAttrList := NewAttrList + Comma + attrName;
        WHILE AttrList[PosA] = BLANK DO
          PosA := SUCC(PosA);
        IF AttrList[PosA] = Comma THEN
          PosA := SUCC(PosA)
        ELSE
          IF AttrList[PosA] <> ']' THEN
          BEGIN
            ErrMsg := '",," is expected in attribute list';
            NoError := FALSE;
          END;
        WHILE AttrList[PosA] = BLANK DO
          PosA := SUCC(PosA);
      END;  { if NoError }
    END   { if AttrName <> Null }
    ELSE  { AttrName = Null }
    BEGIN
      ErrMsg := 'Attribute name is expected';
      NoError := FALSE;
    END;
  END;  { while }
 END;  { else }
 IF NOT NoError THEN
   ErrPlace := 'A'
 ELSE
 BEGIN
   DELETE(NewAttrList, 1, 1);
   StoreExp(NewAttrList);
 END;
END;  { procedure ScanAttrList }


{*****************************************************************}
{* This procedure identifies the single- or double-charactered *}
{* comparison opertors.                                         *}
{*****************************************************************}

PROCEDURE ScanCompOps(VAR CompOp:CompOpStr);

VAR NextChar : CHAR;
```

```
  BEGIN
    CompOp := NextCondWord[1];
    IF CompOp[1] IN ['>','<'] THEN
    BEGIN
      NextChar := CondExp[PosC];
      IF (CompOp[1] = '<') AND (NextChar IN ['=','>']) THEN
      BEGIN
        CompOp := CompOp + NextChar;
        PosC := SUCC(PosC);
      END
      ELSE
        IF (CompOp[1] = '>') AND (NextChar = '=') THEN
        BEGIN
          CompOp := CompOp + NextChar;
          PosC := SUCC(PosC);
        END;
    END;  { if }
  END;  { procedure ScanCompOps }


{*****************************************************************}
{* This procedure scans a date type constant and checks if an   *}
{* impossible date is specified in it.                           *}
{*****************************************************************}

PROCEDURE ScanDate(DateStr:WordStr);

VAR DateStrLen, SlashPos, LastDay : BYTE;
    RemainStr : STRING[8];
    StatusM, StatusD, StatusY : INTEGER;
    Month, Day, Year : STRING[2];
    MonthVal, DayVal, YearVal : INTEGER;
    LeapYear : BOOLEAN;

  BEGIN
    DateStrLen := LENGTH(DateStr);
    IF (DateStrLen < 7) OR (DateStrLen > 8) THEN
      ErrMsg := 'Date constant is too short or too long'
    ELSE
    BEGIN
      Month := COPY(DateStr, 1, POS('/',DateStr)-1);
      RemainStr := COPY(DateStr,POS('/',DateStr)+1,5);
      SlashPos := POS('/',RemainStr);
      IF SlashPos <> 3 THEN
        ErrMsg := 'Illegal data format for date constant'
      ELSE
      BEGIN
        Day := COPY(RemainStr,1,2);
        Year := COPY(RemainStr,4,2);
        VAL(Month, MonthVal, StatusM);
        VAL(Day, DayVal, StatusD);
        VAL(Year, YearVal, StatusY);
        IF (StatusM > 0) OR (StatusD > 0) OR (StatusY > 0) THEN
          ErrMsg := 'Illegal symbol in date constant'
```

```
      ELSE
        IF (MonthVal < 1) OR (MonthVal > 12) THEN
          ErrMsg := 'Invalid date constant'
        ELSE
          IF YearVal < 0 THEN
            ErrMsg := 'Invalid date constant'
          ELSE
          BEGIN
            IF DayVal > 0 THEN
              IF MonthVal = 2 THEN
              BEGIN
                LeapYear := (YearVal MOD 4) = 0;
                IF LeapYear THEN
                  LastDay := 29
                ELSE
                  LastDay := 28;
              END  { if MonthVal = 2 }
              ELSE
                IF MonthVal IN [1,3,5,7,8,10,12] THEN
                  LastDay := 31
                ELSE
                  LastDay := 30;
            IF (DayVal < 1) OR (DayVal > LastDay) THEN
              ErrMsg := 'Invalid date constant';
          END;  { else }
    END;  { else }
  END;  { else }
  IF LENGTH(ErrMsg) > 1 THEN
     NoError := FALSE;
END;  { procedure ScanDate }


{***************************************************************}
{* This procedure reads a literal string until the closing     *}
{* quote is found.  The procedure also identifies an imbedded  *}
{* quote.                                                      *}
{***************************************************************}

PROCEDURE ScanLiteral(Quote:CHAR; VAR LiteralStr:WordStr);

  BEGIN
    LiteralStr := Quote;
    WHILE (CondExp[PosC] <> Quote) AND (PosC <= CondExpLen) DO
    BEGIN
      LiteralStr := LiteralStr + CondExp[PosC];
      PosC := SUCC(PosC);
    END;
    IF PosC > CondExpLen THEN
    BEGIN
      ErrMsg := 'Closing quote is expected for string constant';
      NoError := FALSE
    END
    ELSE
    BEGIN
      LiteralStr := LiteralStr + Quote;
```

```
      PosC := SUCC(PosC);
    END;
  END;  { procedure ScanLiteral }


{*****************************************************************}
{* This procedure scans a number type constant, which may be an*}
{* integer or a real number, with or without a sign.           *}
{*****************************************************************}

PROCEDURE ScanNumber(NumberStr:WordStr);

VAR i, NumStrLen : BYTE;

  BEGIN
    NumStrLen := LENGTH(NumberStr);
    i := 1;
    IF (NumberStr[1] IN ['+','-']) THEN
      i := 2;
    WHILE (NumberStr[i] IN Digits) AND (i <= NumStrLen) DO
      i := SUCC(i);
    IF (i < NumStrLen) AND (NumberStr[i] = '.') THEN
    BEGIN
      i := SUCC(i);
      WHILE (NumberStr[i] IN Digits) AND (i <= NumStrLen) DO
        i := SUCC(i);
    END;
    IF i <= NumStrLen THEN
    BEGIN
      ErrMsg := 'Illegal symbol in number constant';
      NoError := FALSE
    END;
  END;  { procedure ScanNumber }


{*****************************************************************}
{* This procedure classifies a contant string based on its first}
{* character and calls an appropriate procedure to scan it.    *}
{*****************************************************************}

PROCEDURE ScanConst(VAR ConstStr:WordStr; VAR ConstType:CHAR);

VAR FirstChar : CHAR;

  BEGIN
    FirstChar := ConstStr[1];
    IF (FirstChar IN Digits) AND (POS('/',ConstStr) > 0) THEN
    BEGIN
      ConstType := 'D';
      ScanDate(ConstStr);
    END
    ELSE
      IF FirstChar IN Digits+['+','-'] THEN
      BEGIN
        ConstType := 'N';
```

```
      ScanNumber(ConstStr);
    END
    ELSE
      IF FirstChar IN ['"',''''] THEN
      BEGIN
        ConstType := 'C';
        ScanLiteral(FirstChar, ConstStr);
      END
      ELSE
      BEGIN
        ErrMsg := 'Constant value is expected';
        NoError := FALSE
      END;
  END;  { procedure ScanConst }


{*****************************************************************}
{* This procedure scans a condition expression for the join or *}
{* selection operation.  The procedure performs differently on *}
{* the expression depending on what operation the condition    *}
{* expression is for.                                           *}
{* The analysis is based on the recursive definitions:         *}
{*   <cond-exp> = <cond-term> | <cond-term> OR <cond-term>     *}
{*   <cond-term> = <cond-factor> |                              *}
{*                 <cond-factor> AND <cond-factor>             *}
{*   <cond-factor> = <cond-atom> | NOT <cond-atom>             *}
{*   <cond-atom> for selection                                  *}
{*              = <attr-name> | ( <cond-exp> ) |               *}
{*                <attr-name> <comp-op> <const-value>          *}
{*   <cond-atom> for join                                       *}
{*        = <attr-name> <comp-op> <attr-name> | ( <cond-exp> ) *}
{*****************************************************************}

PROCEDURE ScanCond(CondExpKind:CHAR);

CONST CondDelimiters :
        SET OF CHAR = ['(',')',' ','<','>','=','+','-','"',''''];

VAR ConstStr : WordStr;
    ConstType : CHAR;
    CompOp : CompOpStr;
    CompOps : SET OF CHAR;
    CondExpItem : WordStr;
    CondExpData, NewCondExp : PartExpStr;


{*****************************************************************}
{* This procedure reads the next component from the condition  *}
{* expression.                                                  *}
{*****************************************************************}

  PROCEDURE GetCondWord;

  BEGIN
```

```
    IF NoError AND (PosC <= CondExpLen) THEN
    BEGIN
      WHILE CondExp[PosC] = ' ' DO
        PosC := SUCC(PosC);
      IF CondExp[PosC] IN CondDelimiters THEN
      BEGIN
        NextCondWord := CondExp[PosC];
        PosC := SUCC(PosC);
      END
      ELSE
      BEGIN
        NextCondWord := '';
        WHILE NOT (CondExp[PosC] IN CondDelimiters) AND
              (PosC <= CondExpLen) DO
        BEGIN
          NextCondWord := NextCondWord + UPCASE(CondExp[PosC]);
          PosC := SUCC(PosC);
        END;  { while }
      END;  { else }
    END;  { if NoError AND PosC <= CondExpLen }
  END;  { procedure GetCondWord }


{*****************************************************************}
{* This proceudre is called by the procedure ScanCondFactor to *}
{* scan the atomic structure of condition expression.          *}
{*****************************************************************}

  PROCEDURE ScanCondAtom;

  BEGIN
    ScanAttrName(NextCondWord);
    IF NoError THEN
    BEGIN
      CondExpItem := '(' + NextCondWord;
      NewCondExp := NewCondExp + Blank + NextCondWord;
      GetCondWord;
      IF NOT (NextCondWord[1] IN CompOps) THEN
        IF CondExpKind = 'J' THEN
          ErrMsg := 'Comparison operator is expected'
        ELSE
        BEGIN
          ConstType := 'L';
          CondExpItem:=CondExpItem + ':' +ConstType + ')';
        END
      ELSE
      BEGIN
        ScanCompOps(CompOp);
        NewCondExp := NewCondExp + BLANK + CompOp;
        GetCondWord;
        IF CondExpKind = 'J' THEN
        BEGIN
          IF CompOp = '=' THEN   { indicate equijoin }
            INSERT('=', CondExpItem, 2);
            ScanAttrName(NextCondWord);
```

```
          IF NoError THEN
          BEGIN
            CondExpItem:= CondExpItem+ ':' +NextCondWord + ')';
            NewCondExp := NewCondExp + ' B->' + NextCondWord;
            GetCondWord;
          END;
      END   { if CondExpKind = 'J' }
      ELSE  { CondExpKind = 'S' }
      BEGIN
        ConstStr := NextCondWord;
        ScanConst(ConstStr, ConstType);
        IF NoError THEN
        BEGIN
          CondExpItem := CondExpItem + ':' + ConstType + ')';
          NewCondExp := NewCondExp + Blank + ConstStr;
          GetCondWord;
        END;    { if NoError }
      END;    { CondExpKind = 'S' }
    END;    { else }
  END;  { if NoError }
  IF NoError THEN
    CondExpData := CondExpData + CondExpItem;
END;  { procedure ScanCondFactor1 }


{*****************************************************************}
{* This procedure analyzes a condition expression by using the *}
{* locally defined recursive procedures.  At the same time the *}
{* procedure constructs a special string which contains infor- *}
{* mation about the data types used in a condition expression. *}
{*****************************************************************}

PROCEDURE ScanCondExp;

  PROCEDURE ScanCondTerm;

    PROCEDURE ScanCondFactor;

    BEGIN  { ScanCondFactor }
      IF NoError THEN
      BEGIN
        IF (NextCondWord = 'NOT') THEN
        BEGIN
          NewCondExp := NewCondExp + ' .NOT.';
          GetCondWord;
          ScanCondFactor;
        END
        ELSE
          IF NextCondWord[1] IN Alphas THEN
            ScanCondAtom
          ELSE
            IF NextCondWord = '(' THEN
            BEGIN
              NewCondExp := NewCondExp + Blank + LeftParen;
              GetCondWord;
```

```
                ScanCondExp;
                IF NoError THEN
                  IF NextCondWord = RightParen THEN
                  BEGIN
                    NewCondExp := NewCondExp + RightParen;
                    GetCondWord;
                  END
                  ELSE
                    ErrMsg:='")" is expected';
              END  { if NextCondWord = '(' }
              ELSE
                ErrMsg :='Unknown symbol or syntax error'
                          + ' in condition expression';
          IF LENGTH(ErrMsg) > 1 THEN
            NoError := FALSE;
        END;  { if NoError }
      END;  { procedure ScanCondFactor }

    BEGIN  { ScanCondTerm}
      ScanCondFactor;
      WHILE (NextCondWord = 'AND') AND NoError DO
      BEGIN
        NewCondExp := NewCondExp + ' .AND.';
        GetCondWord;
        ScanCondFactor;
      END;
    END;  { procedure ScanCondTerm }

  BEGIN  { ScanCondExp }
    ScanCondTerm;
    WHILE (NextCondWord = 'OR') AND NoError DO
    BEGIN
      NewCondExp := NewCondExp + ' .OR.';
      GetCondWord;
      ScanCondTerm;
    END;
  END;  { procedure ScanCondExp }

BEGIN  { ScanCond }
  CompOps := ['<','>','='];
  NewCondExp := '%';
  CondExpData := '%';
  PosC := 1;
  CondExpLen := LENGTH(CondExp);
  GetCondWord;
  ScanCondExp;
  IF (PosC < CondExpLen) AND NoError THEN
  BEGIN
    ErrMsg := 'Syntax error in condition expression';
    NoError := FALSE;
  END;
  IF NoError THEN
  BEGIN
    StoreExp(NewCondExp);
    StoreExp(CondExpData);
```

```
  END
  ELSE
    ErrPlace := 'C';
END;  { procedure ScanCond }


{*****************************************************************}
{* This procedure parses a query expression based on a recur-  *}
{* sive chain of subtasks.                                      *}
{*****************************************************************}

PROCEDURE ScanQuery;
CONST Delimiters : SET OF CHAR = ['[','(',')',' ',';'];



{*****************************************************************}
{* This procedure reads the next element of a query expression.*}
{*****************************************************************}

  PROCEDURE GetWord;

  BEGIN
    IF NoError THEN
    BEGIN
      WHILE QueryExp[PosQ] = ' ' DO
        PosQ := SUCC(PosQ);
      IF QueryExp[PosQ] IN Delimiters THEN
      BEGIN
        NextWord := QueryExp[PosQ];
        PosQ := SUCC(PosQ);
      END
      ELSE
      BEGIN
        NextWord := '';
        WHILE NOT (QueryExp[PosQ] IN Delimiters) AND
              (QueryExp[PosQ] IN VarNameChars) AND
              (LENGTH(NextWord) < WordStrLen) DO
        BEGIN
          NextWord := NextWord + UPCASE(QueryExp[PosQ]);
          PosQ := SUCC(PosQ);
        END; { while }
        IF NOT (QueryExp[PosQ] IN Delimiters) THEN
        BEGIN
          IF LENGTH(NextWord) >= WordStrLen THEN
            ErrMsg := 'Identifier is too long'
          ELSE
            ErrMsg := 'Illegal symbol in Query expression';
          NoError := FALSE;
        END;
      END; { else }
    END;  { if NoError }
  END;  { procedure GetWord }
```

```
{*******************************************************************}
{* This routine uses the locally defined recursive procedures     *}
{* to parse a query expression.                                    *}
{*******************************************************************}

  PROCEDURE ScanQueryExp;

  VAR CondExpKind : CHAR;

    PROCEDURE ScanQueryFactor;

    VAR RelName : RelNameStr;

    BEGIN
      IF NoError THEN
      BEGIN
        IF (NextWord[1] IN Alphas) THEN
        BEGIN
          ScanVarName(NextWord, RelNameLen, 'Relation');
          IF NoError THEN
          BEGIN
            RelName := NextWord;
            CheckExist(NextWord);
            IF NoError THEN  StoreExp(RelName);
          END;
        END  { if POS(' '+NextWord+' ', OpWords) = 0 }
        ELSE
          IF NextWord = '(' THEN
          BEGIN
            StoreExp(LeftParen);
            GetWord;
            ScanQueryExp;
            IF NoError THEN
            BEGIN
              IF NextWord = RightParen THEN
                StoreExp(RightParen)
              ELSE
                ErrMsg := '")" is expected';
            END;
          END   { if NextWord = '(' }
          ELSE
            ErrMsg := 'Unknown symbol or syntax error';
        IF LENGTH(ErrMsg) > 1 THEN
          NoError := FALSE;
      END;  { if NoError }
    END; { ScanQueryFactor }

  BEGIN { ScanQueryExp }
    ScanQueryFactor;
    GetWord;
    WHILE (POS(' '+NextWord+' ', OpWords) > 0) AND NoError DO
    BEGIN
      StoreExp(OpCode(NextWord));
      IF NextWord = '[' then
      BEGIN
```

```
        GetAttrList;
        IF NoError THEN
          ScanAttrList;
      END
      ELSE
      IF NextWord = 'WHERE' THEN
      BEGIN
        GetCondExp;
        CondExpKind := 'S';
        IF NoError THEN
          ScanCond(CondExpKind);
      END
      ELSE
        IF NextWord = 'JOIN' THEN
        BEGIN
          GetCondExp;
          IF NoError THEN
          BEGIN
            CondExpKind := 'J';
            ScanCond(CondExpKind);
            GetWord;
            ScanQueryFactor;
          END;  { if NoError }
        END
        ELSE
        BEGIN
          GetWord;
          ScanQueryFactor;
        END;
      GetWord;
    END;  { while }

  END;  { ScanQueryExp }

BEGIN { ScanQuery }
  Alphas := ['A'..'Z','a'..'z'];
  Digits := ['0'..'9'];
  VarNameChars := Alphas + Digits + ['_'];
  PosQ := 1;
  IndexNew := 0;
  ErrPlace := 'Q';
  GetWord;
  ScanQueryExp;
  IF NOT NoError THEN
    NextWord := Semicolon;
  IF NextWord <> Semicolon THEN
  BEGIN
    ErrMsg := 'Unknown symbol or syntax error';
    NoError := FALSE;
  END;
  IF NoError THEN
    StoreExp(Semicolon)
  ELSE
    ReportError(ErrMsg, ErrPlace);
END;   { procedure ScanQuery }
```

```
{*************************************************************}
{* This procedure converts a scanned query expression to a    *}
{* postfix form expression using a stack. At the same time, the*}
{* procedure outputs the converted expression to a text file. *}
{*************************************************************}

PROCEDURE PostConvert;

CONST MaxStackItems = 15;

TYPE ExpStack = RECORD
              ExpItems : ARRAY [1..MaxStackItems] of PartExpStr;
              Top : 0..MaxStackItems
            END;
     ExpItemTypes = (Operand, Operator, Open, Close);

VAR S: ExpStack;
    Done, Success : boolean;
    ExpItem, TopExpItem : PartExpStr;
    i :BYTE;
    TextLine : PartExpStr;

{*************************************************************}
{* This function determines if stack S is empty.              *}
{*************************************************************}

FUNCTION IsEmpty(S:ExpStack) : boolean;
  BEGIN
    IsEmpty := (S.Top < 1)
  END;


{*************************************************************}
{* This procedure adds newitem to stack.  The operation fails *}
{* if the stack is full.  The flag Success indicates whether  *}
{* the operation succeeded.                                   *}
{*************************************************************}

PROCEDURE Push(Var S:ExpStack; NewItem:PartExpStr;
               VAR Success:boolean);

  BEGIN
    IF S.Top = MaxStackItems THEN
      Success := FALSE
    ELSE
    BEGIN
      S.Top := S.Top + 1;
      S.ExpItems[S.Top] := NewItem;
      Success := TRUE;
    END;
  END; { push }
```

```
{*****************************************************************}
{* This procedure removes from a stack, the item that was most *}
{* recently added.  The operation fails if the stack is empty. *}
{*****************************************************************}

PROCEDURE Pop(var s:ExpStack; VAR TopItem: PartExpStr;
              var success: boolean);

  BEGIN
    IF S.Top < 1 THEN
      Success := FALSE
    ELSE
    BEGIN
      TopItem := S.ExpItems[S.Top];
      S.Top := S.Top - 1;
      Success := TRUE;
    END;
  END;  { pop }


{****************************************************************}
{* This function determines the type of an expression item.   *}
{****************************************************************}

FUNCTION ExpItemType(ExpItem: PartExpStr): ExpItemTypes;

  BEGIN
    IF ExpItem[1] ='*' THEN
      ExpItemType := Operator
    ELSE IF ExpItem = LeftParen THEN
      ExpItemtype := Open
    ELSE IF ExpItem = RightParen THEN
      ExpItemtype := Close
    ELSE ExpItemtype := Operand;
  END;  { itemtype }


BEGIN { postconvert }

  S.Top := 0;
  ASSIGN(OutFile, OutFileName);
  REWRITE(OutFile);
  For i := 1 TO QLineNo DO
    WRITELN(OutFile, QueryLines[i]);
  WRITELN(OutFile, Semicolon);
  i := 1;
  WHILE NewQueryExp[i] <> ';' DO
  BEGIN
    ExpItem := NewQueryExp[i];
    CASE ExpItemType(ExpItem) OF
      Operand: BEGIN
                 IF ExpItem[1] = '%' THEN
                   DELETE (ExpItem, 1, 1);
                 WRITELN(OutFile, ExpItem);
               END;
```

```
      Open : Push(S, ExpItem, Success);
      Close: BEGIN
               Pop(s, TopExpItem, Success);
               While TopExpItem <> LeftParen DO
               BEGIN
                 WRITELN(OutFile, TopExpItem);
                 Pop(S, TopExpItem, Success)
               END;
             END;
      Operator: BEGIN
                  Done := FALSE;
                  WHILE (NOT IsEmpty(S)) AND (NOT Done) DO
                  BEGIN
                    Pop(S, TopExpItem, Success);
                    IF TopExpItem = LeftParen THEN
                    BEGIN
                      Done := TRUE;
                      Push(S, TopExpItem, Success);
                    END
                    ELSE
                        WRITELN(OutFile, TopExpItem);
                  END;  { while }
                  Push(S, ExpItem, Success);
                END;  { operator }
      END;  { case }
      i := i + 1;
    END;  { while }

    { Move the rest of the stack to the output file. }
    WHILE NOT IsEmpty(S) DO
    BEGIN
      Pop(S, TopExpItem, Success);
      WRITELN(OutFile, TopExpItem);
    END;
    WRITELN(OutFile, Semicolon);
    CLOSE(OutFile);
  END; { procedure PostConvert }

{*****************************************************************}
{* This is the main body of the program RAPARSER.               *}
{*****************************************************************}

BEGIN { main program }

  ParseDone := FALSE;
  WHILE NOT ParseDone DO
  BEGIN
    Initialize;
    GetQuery;
    IF NoError AND (NOT ParseDone) THEN
    BEGIN
      ScanQuery;
      IF NoError THEN
      BEGIN
        PostConvert;
```

```
        ParseDone := TRUE;
      END;
    END;
    IF NOT NoError THEN
    BEGIN
      GetResponse(TryAgain);
      IF NOT TryAgain THEN
        ParseDone := TRUE;
    END;  { if not NoError }
  END;  { while }
END.
```

A Relational Algebraic Retrieval System for Microcomputers

by

Gyeongja Hong

B.S., Korea University, Korea, 1978

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

ABSTRACT

The relational model for databases provides for high-level query languages in which queries can be posed simply and succinctly. The basis of one of such query language is relational algebra. Relational algebra presents a set of operations that can be formulated independently of the physical representation of data in the database. The relational operators are applied to one or more relations, and return a relation as a result.

In this report, the implementation of the Relational Algebraic Retrieval System (RARS) has been described. RARS is a query system based on relational algebra which offers the full range of relational operations. The operations implemented in the RARS include the union, difference, intersection, division, Cartesian product, projection, join, and selection operations. A strict syntax was provided for the user to formulate a query expression using these primitive operations. Simple to highly complex queries may be expressed in a clear manner using English-like keywords and symbols.

RARS was designed for use with the dBASE III database management system, and to run on IBM PCs and compatibles computers.