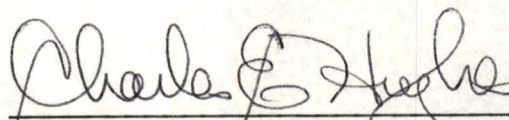


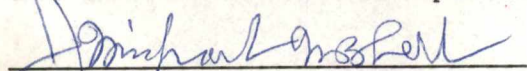
To the Graduate Council:

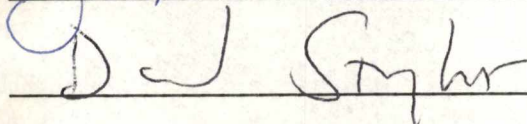
I am submitting herewith a thesis written by Elliott Evans entitled "A Microcomputer Oriented Pascal Based Relational Algebra System." I recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

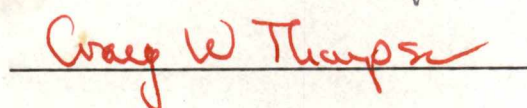


Charles E. Hughes, Major Professor

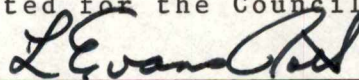
We have read this thesis
and recommend its acceptance







Accepted for the Council:



Vice Chancellor
Graduate Studies and Research

A MICROCOMPUTER ORIENTED
PASCAL BASED RELATIONAL
ALGEBRA SYSTEM

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Elliott Evans
August 1980

3044882

Copyright by Elliott L. Evans 1980
All Rights Reserved

ABSTRACT

A relational database is a time varying collection of relations; a relational algebra system is a programming system which allows a user to create and modify relational databases. Here is described a purely interactive relational algebra system designed for use in the microcomputer environment.

This microcomputer relational algebra system, named MICRAS, is written entirely in the PASCAL language and resides in the environment of the UCSD PASCAL system designed for mini and micro computers. MICRAS keeps created databases on floppy disk but automatically makes core resident that part of any subject database the user is working with. MICRAS provides a wide range of commands allowing not only standard database manipulation but also direct user control over the operating system like maintenance features.

It is believed that MICRAS and its descendants will prove to be useful and interesting to microcomputer users exploring the relational approach to databasing and its limitations.

TABLE OF CONTENTS

CHAPTER	PAGE
I. RELATIONAL DATABASES AND MICROPROCESSORS.	1
Introduction.	1
The Relational Model.	5
Micro Environment Limitations, Decisions.	11
II. THE MICRAS USER MANUAL.	13
Overview.	13
Storage Aspects.	13
Running MICRAS.	22
The Commands: Creation, Insertion, Destruction . . .	26
The Commands: Printing a Table.	32
The Commands: Sorting.	34
The Commands: Status Changing.	39
ALGEBRA: Computation of New Relations.	43
III. IMPLEMENTATION OF MICRAS.	58
Core Storage: Internal Tables and Arrays.	59
Core Storage: Nodes and Linked Structures.	61
Disk Storage.	63
Core to Disk and Back.	64
Allocation and Recovering: Evolution of RESET. . . .	65
Implementation of Algebraic Operations.	66
Structure of the MICRAS Program.	67
IV. THE FUTURE OF MICRAS.	69
Problems.	69
Bigger Relations.	70

Views.	72
More Balanced Trees.	73
Division, Retrieval.	73
LIST OF REFERENCES.	75
VITA.	77

CHAPTER I

RELATIONAL DATABASES AND MICROPROCESSORS

1.0 Introduction

The work presented here is based on the historical convergence of two phenomena. One is the emergence of the relational model among the views of database management systems as preeminent in ease of use and conceptual simplicity. The other is the growth in availability and consequent popularity of microprocessor based computers with the more recent appearance of structured high level languages on these machines. It is natural then to expect experiments involving the implementation of relational systems in the micro environment which capitalize on the higher level capabilities. This is one such experiment.

The relational model first appeared theoretically in a series of papers of Codd [1,2,3]; these papers not only established firmly the theoretical framework of the model but also contained some rather specific implementation suggestions. In the few years since then many implementations of the relational model have appeared, all heavily influenced by Codd's work. Among these and important in activities here are SYSTEM R

(discussed extensively in Date[4]) and INGRES (discussed in Ullman[8]); the latter is of particular interest in that it is implemented in the minicomputer environment. Generally the implementations of the relational model fall into two categories: those whose querying of the database is calculus based versus those using explicit algebraic computation of relations for such interrogation. Without going into the distinction (see Date [4] for a discussion of this) we point out that the implementation here is algebra based. Despite the many variations all the implementations adhere to the view of the database as a collection of relations and all provide the standard facilities for creation, modification, and destruction of relations.

This writer's initiation to the relational model came in the implementation of a student project oriented single user relational database design, SUR, developed at the University of Tennessee by Craig Thompson (design description of SUR [6] is as yet unpublished and remains in technical report form). The SUR design blends features of INGRES and SYSTEM R but adds to them several nice original features (such as uniqueness in the storage of raw data strings). SUR is a prototype relational algebra system; it is designed however for incore storage of the entire database in a spatially liberal

environment and some difficulty is encountered transferring the SUR approach to a smaller system. The work described here owes much to the SUR design, it attempts to preserve as many of SUR's nice features as can survive the large to small transition and are consonant with the other objectives of this experiment. Necessarily there are dramatic modifications of and departures from the SUR design.

The possibilities of implementing a relational algebra system on a microcomputer have been greatly enhanced by the recent improvements in availability and capability of this kind of machine. Additionally the arrival of certain high level language facilities and their use in the construction of system software, as most notably in the UCSD PASCAL system (see [7]), provide background systems in the micro world into which a relational algebra system might blend well. UCSD PASCAL provides more than just the high level language (PASCAL); it gives the micro user an operating system like control program, a file handling system, a screen oriented text editor and other facilities all based in PASCAL. The UCSD system is extremely popular and available on an increasing variety of micros one of which is the 64K APPLE II microcomputer. It was decided to experimentally implement a microcomputer based relational algebra system

called MICRAS using the UCSD PASCAL system on the APPLE II. But more than that the goal was to build a MICRAS which blended in as much as possible with the UCSD background and which used as many facilities supplied by that system as reasonable.

In the rest of this chapter we lay out the theoretical description of the relational model and discuss some limiting factors involved in its micro implementation. Chapter II is a user's manual for MICRAS. Chapter III discusses some of the implementation ideas used. Finally chapter IV presents extensions and goals for the next step in the experiment of relational databasing on micros.

1.1 The Relational Model

We present a semiformal description of the relational model. An attribute set is any finite linearly ordered set. Using A to denote an attribute set, we write it $A = (a(1), \dots, a(N))$ with A 's N elements in their linear order. A relation scheme with attribute set A is a system of sets $(S(a): a \text{ in } A)$ called domains (an assignment of a set to each element of A). We use the pair notation (S, A) to denote the scheme and its attribute set and write $S(i)$ for $S(a(i))$. Call $a(i)$ the i -th attribute and $S(i)$ its associated domain set. A relation R on the scheme (S, A) is a time varying subset of the N -fold cartesian product $S(1) \times \dots \times S(N)$. Thus the elements of the relation R at time t are N -tuples $r = (r(1), \dots, r(N))$ where for each i , the component $r(i)$ is any element of $S(i)$. Refer to $r(i)$ as the component of r associated with the attribute $a(i)$. A relational database is a time varying finite collection of relation schemes along with relations over each of these schemes.

It is natural to think of a relation as a table, the attributes of its scheme are the names of the columns of the table and the domain set associated schematically with a given attribute is the set of all objects which might legally appear as entries in that column. The rows of the table are the current tuples of the relation.

Consider for instance the scheme (S,A) where

A=(Name, Age)

and with

S(1) = set of all human names

S(2) = set of integers from 1 to 200 inclusive.

We might have a relation R defined on this scheme so that currently the content of R consists of the tuples (Mary,19), (Jimmy,53) and (Billy,48). This situation is nicely summed up in the table below.

RELATION : R

NAME	AGE
Mary	19
Jimmy	53
Billy	48

Thus a relational database can be thought of as a collection of table schemes (table formats) each with a current tuple content.

Besides its conceptual simplicity the relational model has a great deal of computational power due to the presence of certain algebraic operations which one can perform on the relations of the database. These are SELECT, PROJECT and JOIN. In our discussion of these we use the descriptive language of tables and descend occasionally to low levels of informality. Assume (D,A) and (E,B) are relation schemes, $A = (a(1), \dots, a(N))$, $B =$

$(b(1), \dots, b(M))$, and that R and S are relations over $(D, A), (E, B)$ respectively.

The PROJECT operation is based on the notion of taking a relation, selecting certain of its attributes and producing a new relation whose attribute set is a permutation of the selected set. One collapses each tuple of the original relation by taking just those components whose attribute has been selected. Essentially one is collapsing the table by removing columns for unselected attributes, striking out new duplicate rows, and permuting the remaining columns to reflect the new ordering. For the abstract relation R above a PROJECT operation can be formally described as follows. Let k be an integer $1 \leq k \leq N$. Let P be any 1-1 function of the set $1, 2, \dots, k$ into $1, 2, \dots, N$ (P is choosing in a particular order k integers from $1 \dots N$). Note that $(D(P), A(P))$ is a relation scheme where $D(P)(i) := D(P(i))$ and $A(P)(i) := A(P(i))$. Given any tuple r as above in R we can restrict it to the new attribute list.

$$(r(1), \dots, r(N)) \text{ -----> } (r(P(1)), \dots, r(P(k)))$$

Gather up all such restrictions and identify duplications to build a new relation over the scheme $(D(P), A(P))$ which

is denoted by

PROJECT R OVER [A(P)].

Suppose $F(X(1), \dots, X(N))$ is a first order formula in the language of relational algebra in the N free variables $X(1) \dots X(N)$. In human language, F is a property which an N -tuple may or may not possess. One can create a new relation T also over the scheme (D, A) containing precisely those tuples of R satisfying F . That is,

r is in T $\iff$ r is in R and $F(r)$ holds.

This operation is called SELECT and the specific relation computed above is here denoted

SELECT R WHERE [F].

We are merely building a new table by choosing as its rows the rows of an existing table which satisfy some condition.

The JOIN operation is most complicated and there are many kinds of JOINS. We discuss and implement only the equi-join. For this, choose i as some integer among $1 \dots N$ and j as one of $1 \dots M$. In effect we are choosing one attribute from each relation : $A(i)$ from R and $B(j)$ from S . The idea of the equi-join of R and S over these attributes is to build a new relation whose tuples are the concatenations of tuples from R with tuples from S . But we only concatenate the tuples if the tuple from R has i -th component equal to the j -th component of the tuple from S . The new relation will have scheme (F,C) where

$$C(k) := A(k) \text{ and } F(k) := D(k) \text{ for } k=1, \dots, N$$

$$C(N+k) := B(K) \text{ and } F(N+k) := E(K) \text{ for } k=1, \dots, M-1$$

$$K=k \text{ if } k < j$$

$$K=k+1 \text{ if } k \geq j$$

Given any tuples r of R and s of S we concatenate them, r on the left, dropping out the j -th entry of s to form an $N+M-1$ tuple, if and only if $r(i)=s(j)$. The relation consisting of all tuples so formed is denoted

JOIN R and S OVER $A(i)$ AND $B(j)$.

JOIN gives one a convenient way of combining two tables which might contain different information about common entries.

Abstract though these operations may seem, they provide a great deal of processing power. Using them one builds new relations out of already existing ones, the new relations presumably providing more specific and useful information than the originals. Used in combination they provide query capability across a large set of tables. Readers who feel this is all a bit dry are urged to turn to the concrete examples given in the user manual (chapter II) and try some computations.

The phrase relational algebra system as used here refers to a system allowing a user to create and manipulate a relational database. Such a system should provide facilities for

- 1.creation/destruction of relation schemes
- 2.insertion/deletion of tuples into/from relations
- 3.sorting tuples of relations on selected keys
- 4.displaying relations and sorts in easily
readable tabular form
- 5.computing JOINS,SELECTs and PROJECTs alone and
in combination.

The small relational algebra system described here (MICRAS) provides all these.

1.2 Micro Environment Limitations, Decisions

Recent enhancements and UCSD PASCAL notwithstanding, the microprocessor environment is still plagued with various problems. Performance reliability and disk drive problems are annoying but most painful and important is the problem of small core size. In light of the above mentioned aim of building MICRAS atop UCSD PASCAL, it appears that things might get quite crowded in APPLE II's 64k of core since MICRAS is sharing space not only with the data of the database currently being worked on but also with the UCSD system. Partial remedies to some of these problems were available, but some serious tradeoffs had to be made. Obviously the database would have to be kept all on disk but with a proper subset an incore target. It was decided to use a relation-wise swapping system with whole relations residing in core. Obviously this provides a sharp upper limit to the size of relations and demands a fairly efficient swapping system since disk IO is sometimes slow. Further it assumes that all relations should be treated (almost) equally, an axiom of varying truth in reality.

To capitalize on what UCSD provides it was decided to keep relations on disk as TEXT files (readable with the system editor, see [7]) with records read and written as STRINGS [7] with PASCAL's readln and writeln. For

core storage of raw data it was decided to use STRINGS of a fixed upper-limit length called here SHORTSTRINGS, thus allowing the full use of UCSD'S STRING intrinsics for string manipulation. An obvious drawback to this involves allocation of these with wasted space increasing if the limiting length is poorly chosen (since the system allocates the maximum number of bytes).

The MICRAS program itself could not be too large and yet it must provide the full RAS facilities as well as disk file manipulation abilities. The UCSD SEGMENT mechanism is used to break the MICRAS program into swappable pieces of executable code. Unfortunately the current UCSD version allows only seven such segments (actually there are more but the system uses them) and this presents tight limits on the size of the overall MICRAS program. MICRAS uses all seven segments.

As to the common dangers of crash and disk IO disasters it is evident that MICRAS must provide mechanisms by which the disk version of the database can easily and frequently be updated so as to reflect recent transactions.

CHAPTER II

THE MICRAS USER MANUAL

2.0 Overview

MICRAS described here is designed to allow creation of new as well as modification of existing relational databases in an environment compatible with UCSD PASCAL. Some understanding of the general principles of the relational model and some familiarity with the UCSD system are helpful for the optimal use of MICRAS. Remember that relations are for all practical purposes just tables whose column names are the relation's attributes and whose rows are the tuples of the relation. MICRAS provides a system for creating, interrogating and viewing any collection of data tables one might want to construct.

2.1 Storage Aspects

2.1.0 Database on Disk

MICRAS as currently implemented operates on a two disk drive system with the MICRAS control program on the boot disk (#4) and the database under consideration being on the alternate disk (#5) stored as a collection of UCSD

EDITOR readable TEXT files. For each database created and made permanent by MICRAS there is kept a master TEXT file; in this master file are listed the names and schematic descriptions of all relations now in the database. The database itself has a user-given name and this name doubles as the name of the disk #5 master file. For each current relation of the database there is a disk #5 file, whose name is the same as the relation's name, wherein are stored in condensed string form the various tuples currently in the relation. One can at any time use the EDITOR to examine these files. It is strongly recommended that only one database be kept per floppy disk.

2.1.1 Database in Core, Saturation

While MICRAS is running actively on a given database the master file is held internally as a table of relations (this table is not treated like a relation itself); also certain of the relations (presumably those the user is currently working on) are kept entirely in core with the balance of the relations remaining on disk. MICRAS treats relations equally (although some favoritism can be established) acting like they are of approximately equal size. It automatically brings into core relations

you want to do something with (though you can manually bring relations in yourself with the BRINGIN command). For relations in core the components of the tuples are uniquely stored as strings (thus though Harry Smith and Joseph Smith may both be in an incore relation with firstname and lastname fields the string Smith appears only once in core). There is a system fixed upper limit on the length of these strings; they are called by MICRAS SHORTSTRINGS. Tuples are realized in core as lists of pointers to the actual SHORTSTRINGS. Tuples, relations and indices on relations appear in core as various data structures built from nodes of a standard type and standard type hashtables.

Knowing how all this is done is not necessary; but one must be aware of certain consequences of the internal storage strategy. All storage is done using the three standard storage unit types: SHORTSTRINGS, NODEs and HASHTABLEs. At MICRAS startup time the user is allocated certain fixed numbers of each of these and as he progresses they are used up. The SHORTSTRING storage space units are not dynamically recovered by MICRAS; although the nodes and tables are reclaimed they can still be eaten up fast (as when one sorts a fairly large relation). Each time a new raw data string enters the core part of the database a new SHORTSTRING is used up;

each time a relation is internally constructed or sorted more nodes and hashtables are used up. The user must inevitably face a situation which we will call saturation where with the current configuration of relations in core there are dangerously short supplies of the needed standard storage units. See section 2.2.2 for more on the saturation problem.

2.1.2 Data in MICRAS

All raw data, regardless of its conceptual type, is stored in string form. But at a higher level MICRAS does recognize three data types: string, integer and real, the latter two called numeric types. By string type data we mean simple character strings with no embedded blanks. The collection of strings (actually SHORTSTRINGS in MICRAS) is ordered lexicographically based on the ASCII sequence of individual characters. This boils down to an alphabetical type ordering with all digits preceding all letters (each in the usual order).

Numeric data, though stored in string form (exactly as the user types it in), is converted to its appropriate type whenever comparisons or computations are to be made. Thus if we consider the two strings 0101 and 100 it is clear that as strings $0101 < 100$ but when compared as

integer type data items one obtains $100 < 0101$. The strings would be temporarily converted to integers at the moment when the comparison is needed. For all numeric data a + sign is assumed unless - is present. Plus signs should never be typed.

For real type data the following points should be kept in mind.

(1) Never type real numbers in scientific notation. MICRAS currently has no facilities for handling this.

(2) Each real number must have a decimal point which must be preceded and followed by at least one digit. Thus .04 and 2. are illegal in MICRAS and must be replaced with 0.04 and 2.0

(3) It is the user's responsibility to maintain uniformity in the number of decimal digits in real data. One should fix and use a constant number of decimal digits throughout all real number work. MICRAS returns strings from computations with three decimal digits

(4) There is a system parameter EPSILON (a fairly small positive number) used as a tolerance for real number equality. This feature is added to improve computation of SELECTs for equality on real keys.

When real numbers are to be compared MICRAS first checks for EPSILON equality.

2.1.3 CORELIM and Swapping

The system parameter CORELIM is the maximum number of relations which can be in core at any given time. CORELIM can be either 3,4 or 5 and the user chooses which. Since MICRAS is set up such that needed relations are automatically brought into core it inevitably occurs that there are CORELIM relations in core. At this point bringing another relation into core requires selecting and removing a core resident relation. This process, called swapping, is done by MICRAS as follows.

Each relation has associated with it what is called a reference switch. Each time an operation is performed on a relation (like printing it or sorting it or using it in a computation) this reference switch is turned on; this is called referencing the relation. Thus each actual use of a relation results in a referencing of the relation; conversely if a relation is in core and is never used it is never referenced. Now when the need arises to swap and to choose an incore relation to remove it is appropriate to find one whose reference switch is off. This is what MICRAS does, throwing out of core the first unreferenced relation it finds. But MICRAS must

insure that there will always be a core resident relation with its reference switch off. Whenever MICRAS finds (in the act of referencing one relation) that it is about to reach a condition of all incore relation's reference switches being on, it deliberately goes about cutting all reference switches off and lets things start over from there. In this way, at least approximately speaking, the relation that gets thrown out is that least recently used (see [5]). This swapping is done automatically by MICRAS, but there are methods available by which the user can more directly control which relations go out and which stay in.

2.1.4 Core Preference

A relation can be declared CORE PREFERRED by the user and this status can be removed from it at any time at his discretion. CORE PREFERRED status is defined by the following features.

(1) Whenever a CORE PREFERRED relation is brought into core (whether by the system or manually by the user) its reference switch is turned on. Thus even if the relation is not thereafter immediately used (referenced) the BRINGIN and RESET commands (see below) as well as system startup cause the reference switch to be turned

on; merely bringing into core a nonpreferred relation does not cause such action. The relation must be used in some way to get referenced.

(2) At startup time and at each RESET (see section on RESET command below) only CORE PREFERRED relations are automatically brought into core. If m denotes the minimum of the number of CORE PREFERRED relations and the CORELIM value, then the first m CORE PREFERRED relations are automatically brought in.

(3) At relation creation time a CORE PREFERRED relation is automatically brought into core whereas nonpreferred relations are only written on disk.

Core preference is not a strong lock-in type mechanism but rather is designed to give certain relations a slight edge in their chances of getting into and staying in core. The idea is that the core preference configuration is to be constantly modified, as the user enters each new phase of his work he declares as CORE PREFERRED those relations he is about to be working with. He declares all others nonpreferred. The next reset he performs will insure that all and only relevant relations will be in core.

2.1.5 System Parameters

Certain system/user set parameters provide limiting values in all MICRAS activities. Below we list the most important of these as well some other system values one should be familiar with.

SHORTSTRING length: currently set at 14, can be tuned and increased at the sacrifice of total NODES available

LIMATTS: limit to the number of attributes a relation can have, currently set at nine.

LIMDICES: limit to the number of indices a relation can have, currently set at three.

LIMDAT: limit to the number of raw data SHORTSTRINGs that can be in core at any time, currently this is 400.

AVDATA: SORTSTRING storage spaces are numbered from one up to LIMDAT. AVDATA is the number of the first unused (empty) one into which a new raw data string can be put when brought into core. Starts at one and increases. A saturation state (DANGER!!!) exists if this value nears the LIMDAT value. AVDATA is displayed at each WHAT command and the system

warns you if it gets within 75 of LIMDAT. AVDATA-1 is the number of SHORTSTRINGS currently being tied up.

NODESLEFT: tells how many general purpose nodes you have left to work with. Current configuration starts you with 800 of these; RESEting will help to get this value back up. The closer this is to zero, the more danger you are in. This value is always displayed in the command mode.

TABLESLEFT: tells the number of hashtables you have left. You start with 5. This value is displayed with the menu.

2.2 Running MICRAS

2.2.0 Starting up the System

Type R while in UCSD's command mode to start MICRAS. In this startup phase remember to follow all responses with a carriage return (we indicate this with <CR>). MICRAS now asks if you want 3 or 4 or 5 as the CORELIM value. If the relations will tend to be small you might choose 5, 3 is safer for larger relations. MICRAS tends to protect itself by choosing 3 as the default value (so if you type anything other than 3, 4 or 5 MICRAS uses 3).

Next you are asked whether you will be creating a new database or modifying an old one. The response here is either NEW or OLD (or you may try to fool MICRAS typing something else but MICRAS will take anything else to mean OLD). If your response is NEW, a new database will be created by MICRAS and MICRAS immediately asks you to name the database. The name must be a SHORTSTRING of 10 or fewer characters unique among disk #5 filenames. After you have named the new database, MICRAS takes you immediately to its command mode (see below). There are at this time no relations in the new database, you will create them later.

If modification of an existing database is needed type OLD <CR> and MICRAS will then ask you for the name of the old database. You must give exactly the same name you gave when you first created the old database. The old database must be on disk #5; upon receiving the name MICRAS searches that disk for the master file. Failure to find it produces an error message and MICRAS termination. Successful location of the master file on the disk causes that file to be read in by MICRAS into the core relation table. Then as many CORE PREFERRED relations as possible are read into core from disk #5. Needless to say, these relations associated TEXT files must be present on #5, but MICRAS will not crash if they

are not, it will only treat such relations as empty (no tuples).

2.2.1 The Menu and the Command Line

By the "menu" is meant the list of command initials displayed across the top of the TV screen. The currently implemented menu looks like:

```
COMMANDS:C(REATE),D(ESTROY),I(NSERT)
L(INK),S(ORT),M(ODIFY CORE PREFERENCE)
P(RINT),H(ALT),W(HAT IS CONDITION OF DATABASE?)
R(ESET),B(RINGIN),T(AKEOUT),A(LGEBRA)
```

When the screen is displaying this menu the system is said to be in (MICRAS) command mode. When MICRAS is in this mode, you can invoke any of its commands (by typing the appropriate first letter indicated).

2.2.2 Coping with Saturation

MICRAS keeps you constantly posted on the state of AVDATA, NODESLEFT, and TABLESLEFT. MICRAS also "beeps" the user if any of these values nears a danger point. Presumably the user is aware if and when saturation is near. The principal mechanism in MICRAS for dealing with saturation is the use of the MODIFY and RESET commands in combination.

When saturation occurs or is near one first changes the configuration of CORE PREFERRED relations using the MODIFY command. Relations not immediately subsequently needed are made nonpreferred and just those absolutely needed are declared preferred. Then the RESET command is invoked; this causes all relations to be core removed and disk written with all nodes, tables and strings they are tying up being freed; this action sets AVDATA back to 1 and NODESLEFT and TABLESLEFT to be set back to maximal values. Then only the CORE PREFERRED relations are internalized.

2.3 The Commands: Creation, Insertion, Destruction

2.3.0 CREATE

This command allows creation a new relation to add to the current database. This entails three phases: (1) uniquely naming the new relation, (2) giving its relation scheme and (3) optionally inputting tuples into the new relation. Invoke the CREATE command by typing C while in the command mode. Then follows the prompt:

NAME OF RELATION?--->

Respond with the desired name for the new relation followed by <CR>. For instance to name the relation CLASSES:

NAME OF RELATION?---> CLASSES <CR>

The name you give should be from 1-10 alphanumeric or punctuation characters, the first of these being alphabetic. It should be unique among relation and index names for the current database as well as among names for TEXT files on the disk you have in #5. Violation of this uniqueness will lead to variable but generally tragic results.

Next the relational scheme is declared. The user is prompted to type in the attributes for the relation as well as their data types and width limits. For this a set of three successive prompts is used repeatedly.

```

ATTRIBUTE NAME--->
KIND?...S(TRING),I(NTEGER),R(EAL)-->
WIDTH--->

```

This prompt set will be presented no more than LIMATTS times. If the relation has fewer than LIMATTS attributes, then, after typing in all the ones desired respond to the ATTRIBUTE NAME---> prompt with *
<CR>. Thus * cannot be an attribute name. Follow responses to all these prompts with <CR>. All responses to these prompts are SHORTSTRINGS. Attribute names can be any shortstrings uniquely named within the set of attributes for the given relation (but not unique across all the relations since it is quite alright for several different relations to have attribute names in common). The response to the KIND prompt should be one of the letters S,I,R; any other response is interpreted as S. These initials represent the three data types MICRAS recognizes (see 2.1.2 above). The WIDTH prompt response should be a positive integer N no greater than the SHORTSTRING length limit (see 2.1.5 above). The width value given is critical because no string of length longer than this value will be allowed in the given domain; longer strings will be truncated on the left end if numeric and on the right otherwise. Once all the attributes have been inputted MICRAS knows fully the scheme for the new relation (it now has enough knowledge to, for instance, format the tabular printing of the relation, see 2.4).

EXAMPLE. To create a relation named ONE with two attributes: NAME of type string and width 10 and IQ of type integer and width 3 one uses the following response sequence.

```
NAME OF RELATION--->ONE <CR>
ATTRIBUTE NAME-->NAME <CR>
KIND?..S(TRING),I(NTEGER),R(EAL)-->S <CR>
WIDTH-->10 <CR>
ATTRIBUTE NAME-->IQ <CR>
KIND?..S(TRING),I(NTEGER),R(EAL)-->I <CR>
WIDTH-->3 <CR>
ATTRIBUTE NAME-->* <CR>
```

Granted that the scheme was successfully input, MICRAS now asks

```
RELATION PREFERRED FOR CORE?--->(YES OR NO)
```

Respond with the appropriate choice (anything else is interpreted as NO). Thus the core preference condition of a relation is set from creation time (see 2.1.4 and the discussion of MODIFY below).

Finally comes the input stage of the creation act. You are now prompted tuple by tuple for the rows of the table you are creating. MICRAS prompts for a tuple by naming one by one the attributes given above and waiting, after prompting with that name, for you to type the component of the tuple corresponding to the named attribute. When MICRAS has recieved all the components of a tuple it says END OF TUPLE..... and then prompts for another tuple in exactly the same way. To terminate

this process type <CR> in response to the prompt for the first component of a tuple. Thus to input two tuples (MELVIN,48) and (CANDI,157) into the relation ONE whose scheme we gave in the above example we proceed as follows.

```

NAME :  MELVIN <CR>
IQ :   48 <CR>
END OF TUPLE.....
NAME :  CANDI <CR>
IQ :   157 <CR>
END OF TUPLE.....
NAME :   <CR>

```

The last response gets us out of the input mode.

Behind the scenes MICRAS is writing these inputted values to a disk file, first checking each string's length against the defined width for the given attribute (truncating if necessary) then writing the string directly into the disk file. In the input phase there is no check for duplicate tuples. At the end of the input phase this file is closed and the relation is brought into core if CORE PREFERRED.

If you do not want to input tuples at creation time use the INSERT command later to enter tuples; in this case leave the input mode with <CR> in response to the first component prompt for the very first tuple.

Note that the width values you assign for numeric type attributes should include room for a possible minus sign and an obligatory decimal point.

2.3.1 INSERT

The INSERT command allows the inserting of tuples into already existing relations. Type I while in the command mode to invoke INSERT. Then you will be prompted

NAME OF HOST RELATION-->

MICRAS wants to know which relation you want to add tuples to. Type the relation's name then <CR>.

Immediately you are prompted for the tuple you want to insert. As in the input phase of the CREATE command, the prompting for the tuple occurs by MICRAS naming a line at a time the attributes in the order of their definition. MICRAS waits for you to interactively type in the components of the tuple one at a time in response to the attribute name prompts (each component response is followed by <CR>). When MICRAS has received all the components it gives you one more chance to look at what has been typed in and perhaps abort the insertion. Hence the prompt:

DO YOU WANT TO CONFIRM THIS INSERTION?

Y(ES) OR N(O)-->

Here no carriage return is needed since merely typing Y will actualize the insertion and typing N will abort it. Should you type anything else MICRAS assumes Y. Regardless of the fate of the first insertion you are

always asked if you care to make further insertions into this relation. Thus the prompt is:

DO YOU WANT TO MAKE ANOTHER INSERTION?

Y(ES) OR N(O)-->

Only a single Y or N is needed (no <CR>) and the default is Y; only N will get you out of the insertion mode. The INSERT command checks for duplication of tuples; if the user types in as an insertion an already existent tuple it will after all only appear once in the relation.

EXAMPLE. To insert the one tuple (JIMMY,87) into the relation ONE, created above (see 2.3.0) the following works.

```
NAME OF HOST RELATION-->ONE <CR>
NAME : JIMMY <CR>
IQ : 87 <CR>
DO YOU WANT TO CONFIRM THIS INSERTION?
Y(ES) OR N(O)-->Y
DO YOU WANT TO MAKE ANOTHER INSERTION?
Y(ES) OR N(O)-->N
```

2.3.2 DESTROY

This command allows you to destroy any relation or to destroy a secondary index on a relation without destroying the relation itself (see 2.5 on SORT). Type D in the command mode to invoke DESTROY. You get the prompt below.

RELATION OR INDEX TO DESTROY?-->

You then type the name of the relation or index you want destroyed. Granted that such a relation or index does exist, MICRAS then totally removes all vestiges of it from the database freeing all nodes it is tying up and removing it from the relation (or index) table. Note that the disk file associated with the relation is scratched from #5 but that the master file is NOT updated until the next RESET (see 2.6). Of course if a relation is destroyed all secondary indices on it are destroyed too. Return to the command mode is automatic.

2.4 The Commands: Printing a Table

The PRINT command allows viewing of any relation or secondary index on the TV screen. It also can be used to create a disk TEXT file copy of a relation or index in tabular form which can be later sent to a printer using the UCSD FILER. The tabular display with column attributes left to right in their created order is shown on the screen one screensworth at a time (the size of this can be varied). Type P from the command mode to invoke PRINT.

There is an initial prompt for the name of the relation or index to be printed. Respond with the relation (index) name <CR>. The relevant relation is brought into core if necessary and referenced. The next

prompt asks if a paper copy is needed. Only a single letter N or Y is the response; if it is Y then a text file is left behind on #5 whose content is the formatted tabular printout. This file is named by MICRAS PX where X denotes the name of the relation or index being printed.

Then MICRAS displays all of or at least the first screen of the relation (index). If more than one screen is needed, the first screen is held up until a K is typed. Typing the K causes MICRAS to show the rest of the relation or the next screen whichever comes first. If there are even more screens needed to see the relation the same typing of K mechanism is used to hold up each screen until the user is ready and then to display the next screen of it. Finally after the relation has all been displayed MICRAS asks if the user wants to see it again. A single letter response of N here sends MICRAS back to command mode; any other single letter (preferably Y) repeats the entire printing of the relation.

The table is formatted so that each attribute gets two extra spaces in addition to its defined width. String data is left justified in larger fields while numeric data is right justified. An attempt is made by MICRAS to fit the table into the 80 character width of

the TV screen. If it does not fit each attribute field loses one space in tabular width and MICRAS tries again to fit. This action is repeated until a width fit is achieved. String data is truncated if needed on the right end, numeric data on the left end. Within the limits of screen width what ultimately governs the table format is the set of width values assigned by the user at creation time (current MICRAS has no way to modify these widths but later versions will include this easy to add feature).

2.5 The Commands: Sorting

2.5.0 SORT

MICRAS keeps the tuples of a relation in no particular order. It is necessary on occasion to require the accessing of tuples in a particular order grouped or sorted according to value on some attribute(s). These particular orders or groupings, called secondary indices, can be imposed (and later destroyed) on relations without altering the actual storage of the relation. The SORT command allows one to create secondary indices on any existing relation. Such an index is not a new relation; rather it is either a (tree) sorting or a (hash) grouping of the tuples of a relation according to some key, a key being an attribute or ordered set of attributes. Type S

from the command mode to invoke SORT.

MICRAS first asks which relation the index is to be defined on. After typing the relations name (and <CR>) the user is asked to name the index. Here the name follows the same rules as relation names (see 2.3.0). The name of the index is important in that all references for printing, computation and so on to the index are through its name not that of its associated relation. Thus try to remember the name or invoke the WHAT command to recall it (see 2.6 section on WHAT).

After the naming of the index, MICRAS asks the user whether the index is to be tree type or hash type. Use a tree type if you want to get an ascending order sort on your key; use hash type if just a grouping by key values is needed (as the latter is faster). Type T or H as you wish and a <CR>; here it is crucial to type no other letter as the system will produce confused results.

Now MICRAS wants to know if the relation is to be in core or not. A relation need not be in core for an index to be created on it. If it is not, the appropriate entry will be made in the index table but the actual tree or hashtable construction will not occur until the relation is brought in core. But even if a relation is core resident at index creation time, these actual constructions can be deferred until the next time the

relation is taken out of then back in to core.

Responding NO causes deferred construction; a YES response causes the relation to immediately be brought in, referenced and the needed structures built.

Finally, MICRAS asks for the list of attributes, in order of importance for the sort, primary sort attribute first, secondary sort attribute next and so on. When the user has typed in all the attributes making up the sort, he responds with *`<CR>` to the next prompt for an attribute. The key can be made up of at most LIMATTS attributes.

Suppose we have a relation R with attributes NAME, AGE and WEIGHT and want its tuples sorted in increasing order according to AGE, with each set of tuples having common AGE value being sorted according to increasing WEIGHT. It must be understood that the tuples of R will always remain in the unordered heap form; what the SORT command allows us to do is to create a new entity, the secondary index which we name (some name other than that of the relation) such that reference to the tuples of R by this index name will give us those tuples ordered according to (AGE,WEIGHT). We decide to name our index AGE-WT. It is of tree type.

```
RELATION TO BE INDEXED-->R <CR>
NAME OF INDEX--->AGE-WT <CR>
H(ASH) OR T(REE)-->T <CR>
WANT INDEX IN CORE NOW?
```

```

YES OR NO--->YES <CR>
GIVE THE COMBINATION OF ATTRIBUTES
FORMING THE KEY
ATTRIBUTE : AGE <CR>
ATTRIBUTE : WEIGHT <CR>
ATTRIBUTE : * <CR>

```

Now for instance, though PRINTing the relation R will still display only the unordered tuples, a request to PRINT the index AGE-WT will display the tuples in the desired order. Had we only wanted the tuples grouped according to AGE WEIGHT combination values we would have requested a hash type SORT, responding to the third prompt above with H rather than T.

From a storage point of view note that the creation of tree type indices consumes NODES (recall the standard type nodes from which most structures here are built) and thus diminishes the NODELEFT total. Core formation of hash type indices consumes one HASHTABLE each time, dropping the TABLESLEFT total by one. Be aware of the implications of indexing relative to the saturation issue.

Secondary indices are dynamically updated to reflect any changes in the underlying relation.

2.5.1 The Link Command

Closely related to SORT is the LINK command which allows one to build a tree-hash left-right pair of indices on a pair of relations. The LINK mechanism

speeds up significantly the computation of the appropriate join of the two relations (see section on ALGEBRA) and is dynamically updated as the relations involved are modified. Type L from the command mode to invoke LINK.

The LINK involves two relations, the left relation LR and the right relation RR. Also needed are attributes, the left attribute AL of RL and the right attribute AR of RR such that RL and RR can be joined over AL and AR (same type attributes). The link is given a name of at most nine characters (adhering to the general rules about names for indices) which here we abstractly symbolize with X. Formation of the link involves construction of a secondary index named LX on RL, which is a tree sort of RL on the key AL, and the formation of an index named RX which is a hash grouping of RR according to AR values. Once this link is formed, the join of RL and RR can be fairly quickly computed.

Each half of the link LX and RX is treated just as any secondary index on a relation, with dynamic updating being automatic; the linkhalves can for instance be viewed individually by applying PRINT to their names LX and RX. The link halves however are paired together for life, the destruction of either half (DESTROY) resulting in the destruction of its partner. Note that the use of

the DESTROY command applied either to the left or to the right half of the link is the only way to destroy a link without damaging the underlying relations.

2.6 The Commands, Status Changing

2.6.0 BRINGIN, TAKEOUT

As mentioned earlier MICRAS allows the user to selectively bring into core and remove from core any relation he wants. The command BRINGIN, invoked from the command mode by typing B, allows the user to "manually" (that is without the use of the automatic swapping system) bring a relation into core. BRINGIN prompts the user to name the relation to be brought in. The relation is completely entered into core, with a swap performed if there are CORELIM relations internal. The BRINGIN causes construction of all trees/hashtables needed for all secondary indices currently defined on that relation. But BRINGIN does not cause an automatic referencing of the relation.

TAKEOUT, invoked by typing T, causes the reverse action for a named relation with the relation's current tuples being written in that relations disk TEXT file and with all internal storage space for that relation released. This effectively lowers the count of incore

relation by one (no new relation is brought in).

Both of these commands allow the user to override MICRAS's swapping system in order to make his own decisions about which relations are to be in core.

2.6.1 MODIFY

Invoked by typing M from the command mode, the MODIFY command is used to (1) ascertain the current core preference status of a relation and (2) to change that status if desired. The prompts here are self explanatory.

2.6.2 RESET

The wise user protects his database from the twin dangers of saturation and unexpected crash. The goal is not to lose the modifications made to the database in the current work session. As to saturation the user is regularly informed of the current values of the relevant system parameters. Based on this the user should occasionally invoke the RESET command (type R in command mode) to reconfigure the core database and record permanent master file changes.

The RESET command causes three things to happen.
(1) All currently in core relations are written out to disk with all NODEs, HASHTABLEs and SHORTSTRINGs for them

being released. The AVDATA pointer is set to one. (2) The master file of the database is rewritten to reflect the current condition of the internal relation table (the latter lists most recent information about what relations are in the database). (3) Up to CORELIM CORE PREFERRED relations are brought into core.

The only time the master disk file of the database is updated (other than HALT time) is when the RESET command is invoked. Thus, during a long work session, one should frequently issue the RESET command. We repeat here the strategy for coping with saturation. Use the MODIFY command to change the CORE PREFERRED configuration and follow this with RESET.

2.6.3 HALT

Invoke the HALT command (type H) when ready to end the current work session. MICRAS will ask if the database is to be made permanent or not. Only the response NO <CR> will cause nonpermanence, with all disk vestiges of the database scratched. Any other response (<CR> needed) leads to a full writing of the database onto disk, with master file and individual relation files being written. In any case the HALT terminates MICRAS and brings you back to UCSD's command mode.

2.6.4 WHAT

The WHAT command, short for WHAT IS THE STATUS OF THE DATABASE?, is invoked by typing W. It causes a display of the current number of relations in the database, the number of them now in core, the number referenced and so forth. After this, WHAT displays a list of all relations in the database, with their names and schematic descriptions (attributes are named, as is their type and width) and a list, for each relation, of all secondary indices now defined on it. It is told for each relation whether it is currently in core. [The format of this display is of varying quality in the versions of MICRAS.]

2.7 ALGEBRA: Computation of new Relations

2.7.0 Working in the ALGEBRA Mode

The ALGEBRA command allows the user to enter the computational submode ALGEBRA. In this mode one can use the algebraic operations of SELECT, PROJECT, JOIN and DELETE individually, and in combination, to build new relations out of already existing ones. Invoke the ALGEBRA command by typing A from the command mode.

The computational submode has only two prompts, given one right after the other as shown below.

NEW RELATION NAME-->

GIVE DEFINING EXPRESSION BELOW

The general idea is that the user is to provide an algebraic expression which tells how to compute a new relation, using SELECTs, PROJECTs, JOINs and DELETEs applied to existing relations. The newly computed relation is given a name by the user and becomes one of the permanent relations of the database (just as if it has been created). Rather than by directly querying the database, in this algebraic approach, information is extracted by the computation of new relations which offer closeups of the data.

The proper response to the first of the two prompts above is the name of the new relation to be built (followed by <CR>), this name should not be the name of any relation or index currently in the database. MICRAS will loop, waiting until a brand new name is given.

Then the second prompt appears; in the line(s) following the prompt the user types the algebraic expression describing the computation of the new relation. The expression may be extended over several lines but must closely adhere to the syntax for algebraic expressions (see 2.7.7); in general the expression must be well formed in the operators (SELECT, PROJECT, DELETE, JOIN) with all tokens separated by blank or <CR>. Violation of the ALGEBRA syntax rules causes immediate departure from the computational submode (return to MICRAS command mode) with all computations done so far for the offending expression being released, i.e. temporary relations created as intermediate results are destroyed and the new named relation is not created. However relations created by earlier successful computations in the same ALGEBRA session do survive.

The above pair of prompts is repeated over and over as long as the syntax rules are obeyed and new relations are being computed. To exit the ALGEBRA submode type STOP <CR> in response to the first prompt.

For purposes of the discussion here, assume our database possesses among its relations the two described below.

RELATION : FOLKS

FIRSTNM	LASTNM	AGE
BETTYLOU	SMITH	18
PABLO	GONZALES	39
GUISEPPE	MARCONI	116
SUELLEN	SMITH	40
ROY	SMITH	18
CLANCY	O'HARA	42
BILLY	O'HARA	10

RELATION : ETHNIC

NAME	HERITAGE
O'HARA	IRISH
SCHMIDT	GERMAN
SMITH	ANGLO-SAXON
GONZALEZ	SPANISH

We now describe the various operations available in ALGEBRA.

2.7.1 SELECT,DELETE

Given an existing relation R and a formula P, the SELECT operation allows the creation of a new relation S schematically identical to R containing precisely those tuples of R which satisfy the formula P. The syntax of SELECT expressions is given formally in 2.7.7 but can be

loosely described as: `SELECT <relation> WHERE [ <formula> ]`. Here the tokens `SELECT,WHERE,[,]` are necessary, `<relation>` denotes the name of the relation from which the SELECTION is to be made and `<formula>` denotes the condition a tuple must satisfy to be selected. The `<formula>` can be a simple comparison like `AGE = 45` or `IQ >= 175` (with an attribute name on the left and a constant on the right) or an "andformula", which is simply several simple comparisons connected by `AND` (the condition demands that they all be true for selection), or can be several andformulas connected by `OR` (the selection is made if at least one of the component andformulas is true). MICRAS always assumes the formula is in this disjunctive normal form and does a left to right analysis where `AND` has higher precedence than `OR`.

EXAMPLE. Suppose from the FOLKS relation above a new relation MIDDLE is to be created containing precisely those tuples whose AGE value is between 18 and 40 inclusive. The prompt/response sequence would be as follows.

```
NEW RELATION NAME-->MIDDLE <CR>
GIVE DEFINING EXPRESSION BELOW
SELECT FOLKS WHERE [ AGE >= 18 and AGE <= 40 ] <CR>
```

Also legal would be the following expression.

```
SELECT <CR>
FOLKS <CR>
WHERE <CR>
[ AGE >= 18    AND <CR>
  AGE <= 40    ] <CR>
```

Note that the algebraic expression can be spread out over many lines but that tokens (like the word SELECT or the [) must be separated by at least one blank or a <CR> and that the tokens themselves cannot be broken up across more than one line. The expression above creates the relation that follows.

RELATION : MIDDLE

FIRSTNM	LASTNM	AGE

BETTYLOU	SMITH	18
PABLO	GONZALES	39
SUELLEN	SMITH	40
ROY	SMITH	18

The DELETE operation is for MICRAS a complementary SELECT operation. Given a relation R and a formula P as above the DELETE operation can be used to create a new relation S containing precisely those tuples of R not satisfying P. For instance creating a new relation EXTREMES with the expression:

```
DELETE FOLKS WHERE [ AGE >= 18 AND AGE <= 40 ]
```

would form the relation below.

RELATION : EXTREMES

FIRSTNM	LASTNM	AGE
GUISEPPE	MARCONI	116
CLANCY	O'HARA	42
BILLY	O'HARA	10

The syntax of the DELETE algebraic expression is the same as that for SELECT expressions.

2.7.2 PROJECT

Use the PROJECT operation to collapse a table by reducing it to certain selected columns. These columns can be all the columns and can be rearranged in any chosen order. Loosely put, the syntax of PROJECT is:

PROJECT <relation> OVER [<attribute list>]

where <relation> is the name of the subject relation and <attribute list> is the collection of column names over which the PROJECTION is to be done (in their desired order). Suppose we are interested in just the (AGE, LASTNM) combination values from the FOLKS relation. We want to choose just the AGE and LASTNM columns in that order and reduce the FOLKS table to produce a new table called FOLKLAST. The expression below would compute the table required, also given below.

PROJECT FOLKS OVER [AGE LASTNM]

RELATION : FOLKLAST

AGE	LASTNM
18	SMITH
39	GONZALEZ
116	MARCONI
40	SMITH
42	O'HARA
10	O'HARA

Note that the relation FOLKLAST has six tuples where FOLKS had seven, this being due to the fact that the ROY SMITH and the BETTYLOU SMITH tuples had the same (AGE, LASTNM) value. Note also the rearrangement of the column order in FOLKLAST. Thus the operation

PROJECT FOLKS OVER [AGE LASTNM FIRST]

would, for example, create a new relation carrying precisely the same information as the original but with its columns in reverse order. Technically in MICRAS the two relations are different.

2.7.3 JOIN

The JOIN operation is used to combine the information of two tables which have a common type field. JOIN requires two relations RL and RR as operands and attributes AL and AR selected (from RL and RR respectively) of the same data type. The JOIN operation connects each tuple of RL with each tuple of RR whose AR

value agrees with its AL value. The newly formed connected tuples are gathered up to form a new relation. The syntax of JOIN is:

JOIN RL AND RR OVER AL AND AR *

this being only an approximate description.

Suppose it is required to combine the tuples of the relation FOLKS with the tuples of ETHNIC for all those people whose lastnames appear in both. The resulting combined tuples are to go into the new relation COMB. The expression below would be used.

JOIN FOLKS AND ETHNIC OVER LASTNM AND NAME *

The table below would be created.

RELATION : COMB

FIRSTNM	LASTNM	AGE	HERITAGE
PABLO	GONZALES	39	SPANISH
CLANCY	O'HARA	42	IRISH
BILLY	O'HARA	10	IRISH
BETTYLOU	SMITH	18	ANGLO-SAXON
SUELLEN	SMITH	40	ANGLO-SAXON
ROY	SMITH	18	ANGLO-SAXON

Notice that the attributes of the new relation are first those of the left operand (FOLKS) followed by those of the right operand (ETHNIC) but with the right join attribute (NAME) removed since all these tuples would on

that attribute only repeat the LASTNM value. Note that each tuple from FOLKS with LASTNM value SMITH has been connected to the tuple of ETHNIC with NAME value SMITH; had there been another tuple in ETHNIC with NAME value SMITH it too would have been connected to all the SMITH tuples of FOLKS. Note also that the tuples of the JOIN (COMB) appear ordered according to the left join attribute (LASTNM which is presumably string type so they are alphabetized); this is a by-product of the way MICRAS computes JOIN.

The attributes of the new JOIN computed relation can be renamed but only sequentially in left to right order. In the above syntax the * is replaced by the GIVING clause whose form is

GIVING [<attribute list>]

The <attribute list> names in left to right order the new names for the columns of the JOIN relation. Thus

JOIN FOLKS AND ETHNIC OVER LASTNM AND NAME GIVING [FIRST
LAST]

would produce exactly the same table as above but with the columns named FIRST LAST AGE HERITAGE. Note the renaming is sequential, so to rename just the second column LAST and leave the first named FIRSTNM one uses

GIVING [FIRSTNM LAST].

If the giving clause is not needed replace it with *.

MICRAS computes the join by establishing a temporary link between the two JOINed relations. Thus the user can speed up the JOIN computation by pre-establishing the link himself with the LINK command (see 2.5.1) or by establishing independent indices a tree type sort on the left operand and a hash grouping on the right.

Currently MICRAS computes only EQUI-JOINS on simple (noncomposite) keys.

2.7.4 Nested Operations

One may create more complicated algebraic expressions by nesting computations. Suppose we require the FIRSTNM and HERITAGE values for all those whose lastname appears in both FOLKS and ETHNIC and whose age is less than 40. The following sequence would build a relation FINAL with the needed information.

```
NEW RELATION NAME-->TEMP1 <CR>
GIVE DEFINING EXPRESSION BELOW
SELECT FOLKS WHERE [ AGE < 40 ] <CR>
NEW RELATION NAME-->TEMP2 <CR>
GIVE DEFINING EXPRESSION BELOW
JOIN TEMP1 AND ETHNIC OVER LASTNM AND NAME * <CR>
NEW RELATION NAME-->FINAL <CR>
GIVE DEFINING EXPRESSION BELOW
PROJECT TEMP2 OVER [ FIRSTNM HERITAGE ] <CR>
```

But the following nested computation gives the same result.

```

NEW RELATION NAME-->FINAL <CR>
GIVE DEFINING EXPRESSION BELOW
PROJECT JOIN SELECT FOLKS WHERE [ AGE 40 ] <CR>
      AND ETHNIC OVER LASTNM   AND NAME * <CR>
      OVER [ FIRSTNM HERITAGE ] <CR>

```

Note that for convenience in keeping track of this more complicated syntax we have spread the expression across several lines. See 2.7.7 for syntax descriptions which should be carefully followed in nested computations.

2.7.5 Renaming a Relation or Index

An algebraic expression can be a single relation or index name. If the expression is just a relation name then the effect is to rename the relation.

```

NEW RELATION NAME-->PEOPLE
GIVE DEFINING EXPRESSION BELOW
FOLKS <CR>

```

The above would cause the relation FOLKS to hereafter go by the name PEOPLE with the name FOLKS no longer a relation name.

If the response to the DEFINING EXPRESSION prompt is the name of a secondary index I on an existing relation R, then the new relation created by ALGEBRA has precisely the same tuples as R but only ordered in the new heap as they appear according to the index. It is important to realize that the new relation is a full-fledged relation not just an appendage-like indexing of R. It thereafter has its own independent life.

2.7.6 Range SELECT and DELETE

The SELECT and DELETE operations are performed by a full linear scan of all the tuples in the subject relation. But if these tuples happen to be sorted in the heap according to a given attribute and if the SELECT (DELETE) formula is just a specification of a range of values for that attribute then the computation of the SELECT (DELETE) can be significantly speeded up. If in the algebraic expression we precede the word SELECT (DELETE) with the word RANGE and type the rest as usual, MICRAS will be tipped off that this is a RANGE SELECT (DELETE), it will assume the tuples are ordered as it does its computation and it will perform the needed computation in a quite different way from the usual. A faster computation is obtained.

By a RANGE formula is meant a formula involving one attribute and specifying a range of values of that attribute. Each of the following is a RANGE formula.

```
AGE >= 14 AND AGE < 44
AGE > 67
AGE <= 33
AGE = 33
```

The syntax is covered in 2.7.7 but generally RANGE formulas specify some lower bound value and/or some upper bound value but it is required that the lower bound value, if there is one, be given first.

Recall the select example done in 2.7.1. Its defining formula is certainly a RANGE formula. Suppose that AGES is the name of a tree type sort we have defined on FOLKS keyed on the single attribute AGE. The following algebraic expression would result in the same tuple collection as the above SELECT (only the tuples would be in AGE order).

```
RANGE SELECT AGES WHERE [ AGE >= 18 AND AGE <= 40 ]
```

Observe that the index name AGES, not the relation name FOLKS, serves as the operand of the SELECT. In general indices can always be operands of the ALGEBRA operations.

The RANGE SELECT is preferred if the subject relation is large but of course one pays a price due to the necessity of presorting.

2.7.7 Syntax of Algebraic Expressions

Now we present, as far as possible, the BNF form grammar for algebraic expressions. All tokens must be separated by at least one blank or <CR>. Here we use parentheses () to denote the repetition of the enclosed symbols zero or more times. Lowercase alphabetic names refer to nonterminals.

General

```

<shortstr> ::= (<letter> | <digit> | <decimal pt>)
<identifier> ::= <letter>(<letter>|<digit>|
                    <punctuation mark>) limit length 10
<relname> ::= <identifier>
<indname> ::= <identifier>

```

Formulas

```

<relop> ::= < | <= | > | >= | = | <>
<cons> ::= <shortstr>
<attname> ::= <shortstr>

<atomfm> ::= <attname> <relop> <cons>
<andfm> ::= <atomfm> | <atomfm> AND <andfm>
<formula> ::= <andfm> | <andfm> OR <formula>

```

Algebraic expression

$\langle \text{algexp} \rangle ::= \langle \text{relname} \rangle \mid \langle \text{indname} \rangle \mid \langle \text{joinexp} \rangle \mid$
 $\langle \text{projexp} \rangle \mid \langle \text{sdexp} \rangle \mid \langle \text{rangexp} \rangle$

SELECT, DELETE

$\langle \text{whereclause} \rangle ::= \text{WHERE} [\langle \text{formula} \rangle]$
 $\langle \text{sd} \rangle ::= \text{SELECT} \mid \text{DELETE}$
 $\langle \text{sdexp} \rangle ::= \langle \text{sd} \rangle \langle \text{algexp} \rangle \langle \text{whereclause} \rangle$

RANGE

$\langle \text{hi} \rangle ::= > \mid \geq \mid =$
 $\langle \text{lo} \rangle ::= < \mid \leq$
 $\langle \text{lower} \rangle ::= \langle \text{attname} \rangle \langle \text{hi} \rangle \langle \text{cons} \rangle$
 $\langle \text{upper} \rangle ::= \langle \text{attname} \rangle \langle \text{lo} \rangle \langle \text{cons} \rangle$
 $\langle \text{rangfm} \rangle ::= \langle \text{lower} \rangle \mid \langle \text{upper} \rangle \mid \langle \text{lower} \rangle \langle \text{upper} \rangle$
 where in the latter case only one $\langle \text{attname} \rangle$ present
 $\langle \text{rangexp} \rangle ::= \text{RANGE} \langle \text{sd} \rangle \langle \text{algexp} \rangle \text{WHERE} [\langle \text{rangfm} \rangle]$

PROJECT

$\langle \text{attlist} \rangle ::= [\langle \text{attname} \rangle (\langle \text{attname} \rangle)]$
 $\langle \text{projexp} \rangle ::= \text{PROJECT} \langle \text{algexp} \rangle \text{OVER} \langle \text{attlist} \rangle$

JOIN

$\langle \text{givingclause} \rangle ::= * \mid \text{GIVING} \langle \text{attlist} \rangle$
 $\langle \text{joinexp} \rangle ::= \text{JOIN} \langle \text{algexp} \rangle \text{AND} \langle \text{algexp} \rangle \text{OVER} \langle \text{attname} \rangle$
 $\text{AND} \langle \text{attname} \rangle \langle \text{givingclause} \rangle$

CHAPTER III

IMPLEMENTATION OF MICRAS

With the objective in mind of providing a full relational algebra system within the constraints of the micro environment certain design objectives were initially identified for the implementation of MICRAS. We list these goals below.

Achieve maximal use of the UCSD system; avoid duplication of UCSD features. MICRAS should be behaviorally similar to UCSD PASCAL, blending in as much as possible with the UCSD system.

Make MICRAS a purely interactive system with an efficient set of quick prompts with fast response time.

Provide a combination disk/core storage system for the database which automatically assures that the core resident part of the database is precisely that part required for current user activity.

Provide an automatic swapping system.

Provide the option of direct user control of the disk/core configuration of the database. The

knowledgeable MICRAS user can optimize MICRAS performance if given the ability to directly bring relations into core, take them out and selectively prefer some relations for core storage. However the automatic swap system should remain effective if the user defaults on this.

The overall MICRAS control program should be fairly small and structured in such a way that it detracts minimally from the core space available to the subject database.

In the sections below we discuss certain implementation features of MICRAS describing some of the design decisions resulting from these goals.

3.0 Core Storage: Internal Tables and Arrays

There is maintained a table of relations currently in the database. This is a PASCAL array of record type variables with fields to describe the various features of the relation. Included are: the name of the relation, the names, types and widths of its attributes, pointers into the index table (for secondary indices on the relation) and a collection of status defining Boolean variables. The status variables indicate whether there is a copy of the relation currently on disk, if the

relation is currently in core, if it is referenced, if it has been modified since last written to disk and whether the relation is CORE PREFERRED. Also maintained is the table of indices, quite similar to the relation table, carrying descriptive information about the secondary indices currently defined on the relations.

Rather than designing MICRAS to have its own string manipulation facilities, it was decided to have character string storage in the form of UCSD STRING type variables. This was to capitalize on that system's available STRING handling intrinsics as well as the rather efficient way UCSD writes STRINGS onto disk TEXT files (see [7]). The data type SHORTSTRING was defined by putting an upper limit on the length of admissible strings; all actual raw data items of the subject database are stored as SHORTSTRINGS. There is an array DATA of type SHORTSTRING which is sequentially filled up as each new data item is entered into core. The items appear uniquely in the DATA array, $DATA[i]=DATA[j]$ iff $i=j$. They are entered into the DATA array with a combination hashing collision overflow handling routine developed in SUR [6]. For purposes of the discussion below let H denote the function whose value assignment is the result of this procedure. That is, if x denotes a string which is a component of some tuple of a relation now in core, then

$H(x)$ denotes number of the DATA array entry storing x ; thus $DATA[H(x)] = x$. AVDATA, which points to the first unused DATA entry, starts at one and increases.

It was decided that recovering these DATA entries dynamically was too expensive in terms of code and system response time cost. Thus the elements are not dynamically freed and reused (at least not between RESETs, see below). So if the string x occurs only once in all the in core relations, the process of removing from core that one relation which contains the string x does not free $DATA[H(x)]$ for reuse.

3.1 Core Storage: Nodes and Linked Structures

At the lowest level all construction of linked lists, heaps, and trees uses a common type of two word node. It is the remaining number of these that the user is constantly being informed of (NODESLEFT). At a higher level though there are (built from the simple nodes) three conceptual types of nodes. Entry nodes, called ENTs, have two fields: NEXT, a pointer to another ENT, and VAL, an integer field. Leader nodes, called LEADs, also have two fields: NEXT, a pointer to another leader node, and ENTRY, which points to an entry node. Finally there are tree nodes, called TREENDs, having four fields: LEFT, RIGHT, both TREEND pointers, KEY, an entry pointer,

and TUPLES, a pointer to a leader node. Recall that in PASCAL a pointer variable has the value NIL if it is currently pointing to no other core storage unit. Use the PASCAL symbolism of $P^{\wedge}$ to denote the storage unit pointed to by the pointer variable P .

We describe some notation useful below. Let P denote a pointer to either an ENT or a LEAD. We define the address sequence beginning at P by the following.

$$A(P)(0) := P$$

and

$$A(P)(i+1) := A(P)(i)^{\wedge}.next \quad \text{if } A(P)(i) \text{ not NIL}$$

with

$$A(P)(i+1) := \text{NIL} \quad \text{otherwise}$$

The length of this sequence is defined as the least positive integer k such that $A(P)(k) = \text{NIL}$. We define the linked list starting at (address) P to be the collection of nodes $(L(P)(i) \mid i=1 \dots \text{length}(A(P)))$ where $L(P)(i) := A(P)(i-1)^{\wedge}$.

A tuple $r = (r(1), \dots, r(n))$ is realized by MICRAS internally as the linked list of ENTs starting at address $P = P(r)$ of length n where $L(P)(i).VAL = H(r(i))$ for $i = 1, \dots, n$. The heaped collection of tuples of a relation R , $(\text{tup}(1), \dots, \text{tup}(m))$, is internally stored (in the

order the tuples were inputted by the user) as the linked list of LEADs starting at the address $Q = \text{TOP}(R)$ where $L(Q)(i).\text{ENTRY} = P(\text{tup}(i))$ for $i = 1, \dots, m$.

Secondary indices are constructed on the lines of SUR model. Hash indices use a common size standard hash table with collisions allowed. Pointers to the tuples hashing to a particular bucket are attached to that bucket. Binary tree type indices are built from TREENDs, the nodes of the constructed tree being in one to one correspondence with the current key values existing in the subject relation and with the tree structured to reflect an INORDER sort of the key values. Located at the nodes is the collection of pointers to the tuples having the key value associated with that node. The secondary indices are dynamically updated.

3.2 Disk Storage

The database is kept on disk as a collection of TEXT (see [7]) files with a master file for the whole database and one file each per relation. The master file contains names and schemes of all relations in the database as well as the names and descriptive data on all secondary indices currently defined. The master file appears in the UCSD FILER directory under the name D.TEXT where D is the user given name of the database. For each relation R

of the database the file R.TEXT holds the tuples of R. All these files have one string per record.

3.3 Core to Disk and Back

Up to RELLIM relations are allowed in the database but only CORELIM of these are allowed in core at any time. An automatic swapping system based on the least recently used approximation algorithm (see Madnick and Donavan [5] for a description) is used to swap whole relations in and out of core.

The condition of core preference is designed to allow preferential treatment of certain relations (see 2.1.4). This feature interacts with the automatic swapping mechanism allowing some user influence over swapping. The core preference condition of a relation is controlled by the user through the MODIFY command.

As MICRAS developed, it was decided to allow the user to fully overcome the swapping system and directly bring relations into core and back out. This led to the addition of the BRINGIN and TAKEOUT commands which provide these facilities. The swap system remains active by default.

3.4 Allocation And Recovery: Evolution Of Reset

When a relation is written to disk all nodes and tables used in its internal realization are freed. However, as mentioned above, the DATA entries which it is tying up are not released. Inevitably after some database activity these DATA entries get filled up and a storage crisis nears. Principally to handle this situation the RESET command was designed. As initially constructed the RESET command released all DATA entries, setting AVDATA to 1, by removing all relations from core and writing them to disk, effectively starting things over. This would also cause release of all nodes and tables. But as the core preference idea developed it was realized that by adding to RESET the action of automatically bringing in to core all CORE PREFERRED relations the user could be provided with a facility to reduce the number of incore relations and the amount of internal space being used to reflect his needs.

Originally the only time the database master file was rewritten (updated) was at the HALT time. It was later decided to add master file updating to the RESET command. Thus the RESET algorithm attained its current form which we repeat here. (1) Write all incore relations out to disk. (2) Reclaim all string (DATA), node and table storage space. (3) Rewrite the master

file if database modification has occurred since the last RESET. (4) Bring into core all possible CORE PREFERRED relations.

3.5 Implementation of Algebraic Operations

Early on it was decided to create a separate submode which could be entered by a simple command and wherein one could compute new relations using the operations of SELECT, PROJECT and JOIN. Thus was born the ALGEBRA command. Within this mode temporary relations and indices are created for the computation of the operations and are later destroyed automatically when no longer needed. Only the final result of the computation is made permanent (receiving a user given name) counting against the user's CORELIM total. The submode repeatedly prompts the user to compute new relations until he types STOP. The submode handles nested computation of the operations with temporary results being destroyed as the work proceeds (with the computation going on interactively as the user types the defining expression).

PROJECTs and SELECTs are computed by linear scan. JOINs are computed by construction of what MICRAS calls a link (the user can build these himself with the LINK command). This simply involves forming a tree index on the left relation operand over the join attribute key and

forming a hash index on the right relation operand over its join attribute. The actual computation preceeds via a scan of the tree; as each node is traversed the key value for that node is hashed to the right operand hash table (thus finding all the tuples of the right operand having that same key value) then connecting all tuples at the given tree node to all tuples at the given hash bucket.

3.6 Structure of the MICRAS Program

The MICRAS control program is broken into seven UCSD SEGMENTs and two separately compiled UCSD UNITs. The UNITs reside in the system library and are invoked by the UCSD PASCAL USES statement. They contain all the lowest level routines such as those for bringing relations into core and realizing tuples internally. These UNITs remain core resident throughout the running of MICRAS. The SEGMENTs are swapped in and out by the UCSD system as MICRAS is executed with an individual segment being brought in only when it needs to be run. The handling of the various commands is broken up across the segments. Notably the ALGEBRA command handling routine, being quite large, is one whole segment which itself is broken into two inner nested segments.

See [7] for details on the UCSD system to whose current configuration the MICRAS control program structure is tailored.

CHAPTER IV

THE FUTURE OF MICRAS

4.0 Problems

As this first version of the experiment of databasing in the micro environment neared completion irritation with the drawbacks of MICRAS combined with the pleasures of working with it to create an eager anticipation for the next step. Indeed several future versions of MICRAS are planned.

As for the present version many of the drawbacks are minor and can easily be circumvented. For instance the current JOIN only allows single attribute JOIN keys. A small extension to the code will allow all later versions of JOIN to handle multiple attribute JOIN keys.

Somewhat more challenging is the problem of MICRAS being restricted to databases with fairly small relations. The current MICRAS operates with whole relations in core and in the current microcomputer world where most systems provide at most 64k of core to work with this approach demands that relation size be kept small. Touching on other problem areas, note that MICRAS makes no attempt to balance the trees it uses in its secondary indices. Unfortunately as processing goes on for a long while these trees sometimes get quite out of

balance and access to the sorted tuples slows down. Also the way MICRAS does DELETES does not appear to be particularly advantageous as the current action of DELETE creates a new relation which lacks the secondary indices of the original. Finally MICRAS currently offers no facilities for aggregate computations.

On the plus side we modestly point out that MICRAS does at least offer all the facilities of a relational algebra system in the micro environment, that its prompt system is fairly fast and that there is some enjoyment to be derived from working with it. There will be future versions, the current one being only a first try. We now discuss some of the additions and modifications planned for the later versions. Most involve an effort to deal with the problems mentioned above.

4.1 Bigger Relations

There are two basic directions MICRAS will go in dealing with the small relations problem. In one subsequent version, designed for micros offering no more than 64k, relations will no longer appear all together in core but rather will be paged. The swapping algorithm, the CORELIM value and so forth will all be page rather than relation oriented. Still relations cannot be allowed that are too big and it is anticipated that the

addition of paging will cause a change in the MICRAS internal storage form of a relation. No longer will the tuples be kept as an unordered heap, rather there will be an automatic sort order over all attributes as a combined primary key.

It appears that the number of relations allowed in core is evolving downwards. In future versions there will no doubt be a two case core limit situation. Where unary action on one relation is being performed that relation, all pages of it, and only that relation will be in core. In the binary situation (as when JOIN is computed), the pages of the two subject relations, and only pages from these, will be in core. These limits refer only to permanent database relations, not the temporary ones created by the system as intermediate steps in computations.

But it is anticipated that the next generation of micros will be endowed with greater core and a version of MICRAS is planned for these. This version will be much like the current one with the swap mechanism not encumbered by paging. Though this version will doubtless lower the CORELIM to two, it will otherwise let the availability of more core solve the small relation problem.

4.2 Views

In the current version of MICRAS, when one creates a new relation B via an algebraic computation on a relation A, the relation B thereafter has its own independent life. As an algebraic product of A, B is in some sense a slice of A; but in current MICRAS it is not a dynamic slice of A. Rather B is like a snapshot (or copy) of A taken at the time the computation is done. But even this analogy is not true for long, as B itself could undergo extensive modification after its birth.

Mainly, subsequent changes in A do not induce changes in B and B is allowed to change in ways independent of A. It would be useful to be able to form dynamic algebraic slices of a relation. These would not really be independent relations in their own right since modifications to them would occur only as reflections of modifications of the parent relation(s). Subsequent versions of MICRAS will provide the facility for constructing and activating dynamic algebraic views of relations or sets of relations using arbitrary algebraic expressions applied to the relations.

However the algorithms anticipated for doing this will add significantly to the code of the MICRAS control program. It therefore appears that such additions will not be implemented under the current UCSD system with its

seven segment limit. Later versions of UCSD will grant the user many more segments creating an environment where significant extension is possible.

4.3 More Balanced Trees

The balancing of trees significantly improves access performance and subsequent versions of MICRAS will attempt to achieve balancing. It is expected that the current form of trees used in secondary indexing will give way to the B-tree (see Ullman [8] for a discussion of this) but such a change it now appears will require use of additional segments.

4.4 Division, Retrieval

In terms of algebraic capability it was originally planned that MICRAS provide some version of the division operation. This operation has been discussed initially by Codd [1] and is given a fairly simplified discussion in Date [4]. We mention a simple case of it here. Given the scheme (D, A) where $A = (a(1), a(2))$ and $D = (D(1), D(2))$, let R be a relation over the scheme (D, A) and let S be a relation over (E, B) where $E = (D(1))$ and $B = (A(1))$. Then the quotient of R/S is defined as the set of all 1-tuples $q = (q(2))$ over $D(2)$ such that for every tuple $s = (s(1))$ of S the pair $(s(1), q(2))$ is an element of R . This can

easily be generalized to higher arity relations; but it is clear that this operation provides something of a quantificational facility connecting a pair of relations. Future versions of MICRAS operating in the greater than seven segment environment will have an experimental version of this.

Later versions of MICRAS will have some aggregate function capability with frequency, mean, max and min computations available on multiple attribute keys. Finally it is anticipated that a RETRIEVE command, computationally based in ALGEBRA but with a calculus looking language, will be implemented to allow temporary gathering of constructed tuples in an easily viewable workspace.

LIST OF REFERENCES

LIST OF REFERENCES

1. Codd, E.F., "A Relational Model of Data for Large Shared Data Banks." CACM 13, No. 6 (June 1970).
2. Codd, E.F., "Further Normalization of the Database Relational Model." In "Data Base Systems", Courant Computer Science Series, Vol. 6. Englewood Cliffs, N.J.: Prentice Hall (1972).
3. Codd, E.F., "Understanding Relations." FDT 7:3-4, pp. 23-28, New York: ACM (1973).
4. Date, C.J., "An Introduction to Database Systems." Reading, Mass.: Addison Wesley (1977).
5. Madnick, S.E. and J.J. Donavan, "Operating Systems." New York: McGraw Hill (1974).
6. Thompson, C.W., "SUR: A Single User Relational DBMS" CS-80-45 Technical Report, Computer Science Department, The University of Tennessee, Knoxville (August 1980).
7. University of California, San Diego, Institute for Information Systems, "UCSD (Mini-Micro Computer) PASCAL, Version II.0." La Jolla, Cal.: Institute for Information Systems (1979).
8. Ullman, J.D., "Principles of Database Systems." Potomac, Md.: Computer Science Press (1980).

VITA

Elliott Evans grew up in Houston, Texas. He attended the University of Houston and received a Bachelor of Arts degree in Mathematics in August 1969. Remaining at Houston for graduate work in Mathematics, he received a Master of Science degree in that field in May 1973 and Doctor of Philosophy degree (Mathematics) in December 1975.

After three years as Assistant Professor of Mathematics at the University of Tennessee, Knoxville, he decided to depart Mathematics and enter the field of Computer Science. Entering the graduate program in Computer Science at Tennessee in 1978, he completed the work for the Master of Science degree in summer of 1980.

Dr. Evans, who maintains mathematical contact with membership in the American Mathematical Society as well as several other professional organizations, has undetermined plans for the future.