MRDSC Project Report

Submitted As An M. Sc. (Applied) Double Project

by

Charles Snow

(7611565)

22 Decemeber, 1986

CONTENTS

toycol

toy	colour
ball	blue
ball	green
ball	red
ball	white
pail	red
pail	yellow
shovel	blue
shovel	green
shovel	red
shovel	white
pail	yellow
shovel	blue
shovel	green
shovel	red
shovel	white

amercol

colour
blue
red
white

amertoy

toy
ball
shovel

Figure 1.2 Example of Relation Division

Name	Function	Description
$\supseteq$-join	division	L includes R; output relation is a vector of attribute values from the non-join domains of L and R which appear with all values of the vector R in the join domain
$\subseteq$-join		R includes L; output relation is a vector of attribute values from the non-join domains of L and R which are entirely contained in the vector R
$\circledcirc$-join		L and R do not intersect; output relation is a vector of attribute values from the non-join domains of L taken from tuples which have no values from vector R in the join domain
$\supset$-join		L is a subset of R; output relation is a vector of attribute values from the non-join domain of L which appear with more than all values of the vector R in the join domain

$\not\supset$-join		L is not a subset of R; (the complement of the above) output relation is a vector of attribute values from the non-join domains of L which appear with all or fewer of the values of the vector R in the join domain
=-join	equality	L is R; output relation is a vector of attribute values from the non- join domains of L which appear with exactly (no more and no fewer) the values of the vector R in the join domain

The θ-joins provide the means to actually 'join' tuples of two source relations to produce 'wider' output relations. Output tuples are formed by concatenation of source tuples (the join domain is not replicated) subject to the join constraint which requires the values of the attributes in the join domain to be equal to, less than or greater than each other. The table below summarizes this family for the example L θ-join R:

Name	Function	Description
equi-join	equality	output relation contains tuples which result from concatenation of tuples in L with those in R whose attribute value in the join domain equals the attribute value in L in the join domain
less-than-join	less-than	output relation contains tuples which result from concatenation of tuples in L with those in R whose attribute value in the join domain is less than the attribute value in L in the join domain
greater-than-join	greater-than	output relation contains tuples which result from concatenation of tuples in L with those in R whose attribute value in the join domain is greater than the attribute value in L in the join domain

User requests directed at a relational databases ultimately are performed as a sequence of the relational operations described above. Typically users are provided with some high-level language with which to express requests to retrieve or modify the database. The user interface they see supports the translation of these requests into a sequence of basic operations which, applied to the relations in the database, provide the desired results. Much effort has been spent in the design of these interfaces and optimization of the translation from user request to database action (see, for example, [Blas77]).

1.4 Relational Calculus

The *relational calculus*, also proposed by Codd [Codd71b], provides the same power for manipulating relations as does the relational algebra, but does so in a different fashion. As a result of its origins, the essence of the relational calculus is non-procedurality. That is, a 'pure' expression of a query in the relational calculus contains no indication in terms of the abovementioned relational operations as to how the query is to be satisfied. Producing from the relational calculus a usable query language required a limited re-introduction of procedurality, at least to the point of introducing verbs such as get, put, *etc.*

Queries expressed in the relational calculus can be very elaborate while at the same time being very terse. This expressive power arises from *quantifiers*, *quotas*, Boolean operators (and, or, not) and comparative operators (=, <, >). Some examples illustrate the expressive power of the relational calculus compared to the relational algebra:

(1) On relation toycol, find red toys.

The relational calculus expression is

{(toycol.toy) : toycol.colour = 'red'}

stating that a set, {}, an output relation, is to be made from tuples in toycol on domain colour subject to the predicate (that following the colon) that the attribute in toycol's toy domain be equal to red.

Codd's ALPHA data sublanguage, with syntax used by Date[Date77], would express the query as:

get *W* (toycol.toy): toycol.colour = red

where the verb, get, and named destination, *w*, are the only differences from the above set definition. *w* denotes the user's workspace into which the relation satisfying the predicate is to be placed.

The corresponding relational algebra expression of the query would be:

redtoys ← *select* toycol colour = 'red'
result← *project* redtoys *on* toy

(2) Find blue toys other than balls.

get *w* (toycol.toy): toycol.colour = 'blue' Λ toycol.toy ≠ 'ball'

where the Boolean and (Λ) is used to concatenate predicates. As a relational algebra expression,

blutoys ← *select* toycol *where* colour = blue
result ← *select* blutoys *where* toy != ball

(3) Find any one yellow toy.

get w (1) (toycol.toy): toycol.colour = yellow

The (1) imposes a *quota* on the number of tuples to be accepted; *i.e.*, stop 'getting' as soon as any yellow toy is found.

1.5 Databases and Data Organization

Databases typically manage an underlying collection of files which can be very large (several hundreds or thousands of megabytes). While the processing of operations on the database could be made fast if the entire database were kept in memory, this is almost never possible. The files of the database reside on direct access storage devices, and relevant portions of them are loaded into main memory as required. The cost of processing database operations increases with the database size because (1) more disk operations are required in order to provide (2) more data to be manipulated. Shortcuts have to be taken.

An obvious shortcut is to reduce the amount of data which must be stored using compression techniques: *e.g.*, front/end compression, multiple character suppression. Such compression usually provides a saving only when applied to character strings (often the majority of the database's contents), but is not particularly advantageous for binary fixed or floating point representations.

Other shortcuts aim at reducing the number of costly disk look-ups performed. These disk operations pose the greatest expense to database performance. Disks are expensive because they are slow compared to the speed of main memory in access time. The slowness arises from the mechanics of disk storage technology, *i.e.*, an access arm must be moved from one place to another over the surface of the disk, and await the correct portion of that surface to be rotated to a point where it is positioned directly beneath the arm. The final accepting of data to be written or providing of sought data being read may involve multiple arm movements and rotational delays owing to [OS, not database] index lookups, block remapping, *etc.* Each such delay is typically 35 milliseconds on a reasonably fast disk, with the result that disk bandwidth can be as small as 20 Kbytes/sec on a medium sized minicomputer [McKu83]. An example to use in comparisons: for a moderately sized relation of 10 mbytes, a sequential search which, on average, would require reading half the relation, would require four and one half minutes to complete.

There are several ways to reduce this unacceptable sequential processing time. The simplest is to ensure that the data is ordered and a binary search can then reduce accesses to nearly $\log n$. Such an ordering is advantageous so long as most accesses are made via values of the key attribute. Queries which are concerned with attribute values other than that of the key do not benefit from the ordering.

An ordering scheme which does not favour one attribute over another and provides full advantage of ordering to all attributes equally was proposed by Orenstein [Oren82]. By cyclically interleaving the bits representing the attribute values one remaps the tuple to an n-attribute space in an ordering, z-order. The interleaving of bits represents, effectively, a decision as to which half (the 0 or 1 half) of the remaining unpartitioned space to pursue, thereby further partitioning the space. For example, the tuple from toycol which is "ball red" becomes, in z-order†, 320C30233070105000 in hexadecimal. Searches for individual tuples or entire ranges, based on any attribute or combination of attributes, are performed by constructing a z-ordered 'mask' and examining the subset of z-ordered space it isolates.

A logarithmic organization introduced by Bayer and McCreight [Baye72]is the B-tree, consisting of nodes of the form:

† The bit numbering and byte ordering within a word produce machine dependent z-orderings. The one shown above is for a DEC VAX11 computer. The attribute strings are terminated with null bytes.

$$p_0, (x_1,\alpha_1,p_1), \ldots, (x_l,\alpha_l,p_l)$$

where p_i are pointers to nodes lower in the tree, x_i are keys, and α_i are the indexed items. Searches through the tree involve comparison of a search value with the x_i on a page until some x_i is greater than or equal to the search value. If equal, the search stops, otherwise the pointer p_{i-1} is followed to another node and the search continues there. Thus searching the tree requires at most h accesses. The authors mention that a tree of height four accommodates over 200 million entries.

An early modification to the B-tree design was the B^*-tree wherein all data (the α_i of the B-tree) is moved to the leaves of the tree, and branch nodes take the form:

$$p_0, (x_1,p_1), \ldots, (x_l,p_l)$$

This has the desirable effect of making the tree flatter by permitting more indexing entries per branch node, and facilitating sequential access to the data.

A way to further increase the flatness of the B^*-tree was presented by Bayer and Unterauer [Baye77]. By replacing a key with a *separator*, a string which distinguishes the previous from the current entry, the entire key value need not appear in the branch nodes (now denoted by (s_i,p_i)). The separator will be the most significant portion of the key which distinguishes two entries, *i.e.*, a *prefix*, and thus is the minimal length string capable of distinguishing the entries. A B^*-tree making use of prefix separators in this way is known as a *simple prefix B^*-tree.*

The simple prefix B^*-tree can provide a fast access index for a relation at the [relatively cheap and tunable] price of maintenance overhead. When insertions of data entries cause a leaf node to overflow, a new leaf node is generated, and the entries in the original leaf distributed evenly across the two leaves, an event called a *split.* The split also requires a separator/pointer pair to be inserted into a parent branch node to index the new leaf. This insertion may cause the branch node to overflow, and it is similarly split, propagating an (s_i,p_i) insertion into its parent, *etc.* In the worst case the insertion causes the root to be split with the result that the tree grows in height.

An enhancement to the split algorithm suggests that upon overflow, before performing a split with its attendant overhead, we should attempt to redistribute node entries between the overflowed node and one of its siblings.

The B^*-tree, while providing quick access to entries, suffers from the same favoritism for which simple ordering of tuples in a tabular organization was criticized. To provide equally rapid access on any attribute requires a separate B^*-tree index for each such attribute, or a full inversion index as suggested by Wedekind [Wede74]. To solve this, one could build the B^*-tree using the tuples' z-order representation.

All of the preceding deals with ways to reduce disk accesses at the level of the database programs. A lower level strategy can be employed when the hosting environment of the database system allows the database to physically organize its disk files in a way better suited to its needs than is the more general purpose organization used by the hosting system for its work. Database systems which implement this strategy tend to offer better performance, albeit at the expense of more difficult implementation [Blas81, Ston80]

Chapter 2

2: The McGill Relational Database System

2.1 Overview

The McGill Relational Database System, MRDS, implements a relational database, supports an ever growing family of relational algebra operators, and provides batched and interactive access. Originally implemented in PL/I for MVS, it has evolved to versions in Pascal for MVS and the Apple II microcomputer (Stanford and UCSD Pascals respectively). A partial implementation in C to run under UNIX is described in chapter 3 of this report.

Access to a relational database is provided from a high level language (PL/I or Pascal) via calls to a library of MRDS user callable routines. In addition, a high level query language, ALDAT [Merr77], is supported.

Relations contain fixed length tuples of fixed length attributes, and only the character data type is supported. Users receive workspaces into which they may write new relations, either the output of transactions or copies of existing relations. The entire database (called in MRDS vernacular the *permanent* database) resides in a separate space and is unmodifiable by users, thereby simplifying concurrency problems. Relations created in the user's workspace may not be modified or deleted during sessional or program use. Workspaces may be deleted at session or program termination.

2.2 Design and Implementation

The permanent database is stored as a single file (as a direct access data set member in MVS versions) divided into *pages*. Relations must be page aligned and tuples may not cross page boundaries. MRDS performs its own page management using a fixed sized set of page frames and least recently used page replacement. No indexes or file maps are maintained. All user and control data are kept as relations.

A database is managed by means of its three system relations:

rel holds data on each relation in the database (including, of course, the system relations): name of relation, tuple width, upper limit on size of relation, pointers (tuple numbers) reflecting current read and write positions within the relation, and page number where relation starts,

dom holds data on all domains known in the database: domain name, and length of attributes in this domain,

rd has tuples which, for each relation, indicated by order of appearance, list the sequence of domains within a tuple of the relation, and the byte offset from the start of a tuple at which the attribute from the domain appears.

Tight upper limits are imposed on the system relations so that they can be kept resident in main memory while the database is active, occupying the first two to four pages of memory.

Only upon termination of activity, normal or otherwise, on the database are the memory resident relations flushed to disk.

Relations are of two types: *general* and *constant.* The general relation is what one usually associates with the term relation. A constant relation is a vector (unary relation) of small size which is to be kept memory resident. Constant relations are invariant following creation and are kept in separate pages from general relations. Typically, constant relations find use in σ-joins.

Early MRDS implementations used external sort/merge programs (SYCSORT in the MVS versions). As families of join became more clearly defined it became more attractive to have a sort/merge resource internal to MRDS since the μ-, σ-, families can each be implemented by a 'clever' sort/merge.

Error handling is performed by a central handler which understands four severities of error from warning to "catastrophe", and issues an error message, an indication of the severity, and an additional string passed to it by whatever code detected the error. The handler provides MRDS with a reasonably fault tolerant attitude; control is usually passed back to the error handler's caller, and MRDS execution resumed. A parameter in the handler keeps a severity weighted tally on the number of errors, and causes the handler to abend MRDS execution upon exceeding a preset threshold.

Exception handling is essentially non-existent. About the only run time exception recognized is an excess number of I/O operations, resulting in forced termination.

MRDS provides a library of user callable routines which may be invoked from any programming language whose object modules can be linked with MRDS, *i.e.*, that support the parameter passing mechanism understood by MRDS. The advantage of this style of interface is that it gives database access to programmers without their having to become acquainted with a query language.

At present, the user callable routine library contains the following routines:

conrel(name,on_domain,size,value)
: create the constant relation name on the domain on_domain whose absolute size is size. value is either the address of a list containing the tuples of is the list itself

merjoin(rname1,rname2,domlist1,domlist2,sizeof_domlists,outranme,mu)
: perform a merge join (μ-join) joining relations rname1 and rname2 on their respective domains domlist1 and domlist2, each domain list being of length sizeof_domlists, creating the new relation outrname. The last parameter, mu, identifies which member of the μ-join family is desired

project(rname,on_domains,sizeof_domlist,outrname)
: project relation rname on the domain list of length sizeof_domlist contained in on_domains producing new relation outrname

printrel(rname,title,dest)
: produce a 'formatted' listing of the relation rname's contents, printing the string title at the top of every page. dest is a parameter which can be used to specify an output device or file to which the listing is to be directed. Printrel makes no particularly insightful effort at formatting, and tuples exceeding line length are simply truncated.

setup(dbname,disp)
: always called before any other MRDS routine to initialize the database dbname; disp identifies whether the database is old (already exists) or new (being created).

sigjoin(rname1,rname2,domlist1,domlist2,sizeof_domlists,outrname,sigma)
: as merjoin, but with sigma specifying which of the σ-joins to perform. The former divide function is absorbed by sigjoin.

qtexpr(rname,attrib_val,qt_expr,num_of_syms,funtion,qt_pred,qt_count,outrname)
: produce the new relation outrname from rname by evaluating qt_expr on tuples in rname. In addition, qt_count will return the quantity (or proportion, depending upon the quantifier) of tuples satisfying the qt selector.

save(rname_list,sizeof_list)
: provides way for user to add relations named in rname_list, currently in user workspace, to the permanent database. The system relations are also updated to reflect new additions; this routine is meant to be called last in an MRDS run.

MRDS is an evolving system, hence the above list, while being correct at some point in time, cannot be expected to remain so long.

The above user callable functions rely upon a lower level collection of routines in the MRDS internal library. Routines at this level provide relations and tuples as output using database and lower level objects.

2.3 System Design: Features and Goals

The current system began as a straight translation of the MRDS implementation written in PL/I and run under MVS, and hence reflected its two layered design of user callable and internal, system level, routines. This has remained the dominant organization, despite a potentially more attractive one which surfaced recently (see chapter 6). The layering, as in the original version, assures that users see only relational entities.

MRDSc does not at present support the permanent database and temporary workspace organization of MRDS. Any relations created by a user during a run are kept in the database's directory. However, and unfortunately, MRDSc is true to the original in providing no way to delete a relation. This introduces the possibility of concurrency conflicts when more than one person uses a database. Generally speaking, MRDSc is *not* a multi-user database system: it has no facility for concurrency read or write control.

MRDSc is implemented under UNIX and is organized as shown in Figure 3.2. The choice of UNIX as a hosting operating system brought about several changes. Firstly, individual relations were separated out into individual files. A database now consists of several files, one per relation, and is contained within a single directory.

A second change occurred as a result of the way the UNIX I/O system performs file buffering. For all that UNIX is touted as having a byte-stream I/O system, disk reads and writes are inherently blocked at sizes ranging from 512 to 8192 bytes per transfer. UNIX will read the entire disk block containing so little as a single user requested byte, first loading the block into a system buffer, then satisfying the user request with a copy into user memory. Newer versions of UNIX have more clever strategies than this, *e.g.*, Masscomp's RTU bypasses the system data space buffering, reading from disk directly into user memory. Because individual relations are now held in separate files and because of the "double buffering" still in widespread use, the buffer frame with LRU page replacement subsystem of MRDS is not implemented in MRDSc; it was judged more efficient to let UNIX look after the buffering of its disk pages, rather than have both

UNIX and MRDSc doing the same work.

UNIX does not support multiple storage volume files or directories (yet). This limits the size of a database to that which can physically be stuffed onto a particular disk type. This effectively constricts MRDSc to databases smaller than 600 Mbytes. By relaxing the constraint that all database files must reside within a directory, one could bypass this restriction at the cost of some performance.

This implementation contains some new features not found in previous implementations.

The original version supported only character string data; MRDSc supports arbitrary data-types: six primitive built-in types (character: string and individual byte, floating point [single precision], and various sized integers), plus user definable ones. Character strings may be of any length, but within an attribute, must always be the same length (as in MRDS).

Another new feature is the introduction of different data organizations. Relations may be kept in the traditional tabular form (ordered or unordered), or with tuples in z-order, or as simple prefix B^* trees, or any [sensible] combination of these. The actual data organization is transparent to the user level relations (beyond, of course, any effect that organization may have on performance). Further, different relations may have different data organizations, and, in the case of z-ordering, not all attributes in a tuple need participate in the ordering.

Beyond the specific objectives of supporting multiple data types and organizations, there were four underlying goals. Foremost was *speed*, bought at the price of expending memory. In tradeoff decisions, speed almost always won out over saving memory. Secondly was the view that 'memory is cheap', and most computers now have it (both virtual and real) in abundance. Tables, buffers, *etc.*, are allocated with little regard for memory consumption. As much as possible, allowance for machines living in the 16-bit dark ages (*e.g.*, PDP11's) was made: both compile time and dynamically allocated structures can be downsized to assure fitting within 64K segments. Third was portability of code. As mentioned, data structures can be scaled to fit within 16 bit architectures, and operands are machine independent. Both portability and the fourth design goal, open endedness, are served by extensive use of the C preprocessor's macro facility to reduce the appearance of 'magic numbers' in the code to nil.

Chapter 3

3: Implementation Manual

This chapter describes how each major component of MRDSc has been assembled, beginning with an overview of the implementation, synopsis of its relational system elements, followed by a detailed discussion of how each component works. The information here, with references to the source code, should enable any systems programmer to modify, extend, or correct the code.

3.1 Overview

MRDSc is an implementation of a subset of the full MRDS described previously. It is written in C for use under the UNIX operating system. Target machines for which this implementation is intended should have virtual memory as well as at least 1 Mbyte of directly addressable real user memory: no attempt has been made to make the system or its run-time memory usage small. Implemented is a subset of the system layer of 'standard' MRDS, hereinafter called MRDSc, with the introduction of support for simple prefix B^*-trees and z-ordered data organizations, and multiple data types for relational data. Figure 3.1 shows the relationship of MRDSc to its hosting UNIX system.

The reader is assumed to be familiar with systems programming under UNIX in C. Should this not be the case, two of the references in the bibliography are recommended (particularly Horspool): [Hors86] and [Kern84].

3.2 MRDSc System Elements

3.2.1 Data and Domains

Data stored in MRDSc relations is uncompressed, unaligned, and is one of the following types:

Type	Name	Length	Description
string	DT_STRING	var	a string of ASCII valued bytes of undetermined length, terminated by a null byte
integer	DT_INT	2/4	a fixed point binary integer, 16 or 32 bits long depending on host machine. (In reality, this datatype is merely an alias for whichever of short or long is appropriate)
char	DT_CHAR	1	a single ASCII valued byte
float	DT_FLOAT	4	a single precision floating point number
short	DT_SHORT	2	a 16 bit fixed point binary integer

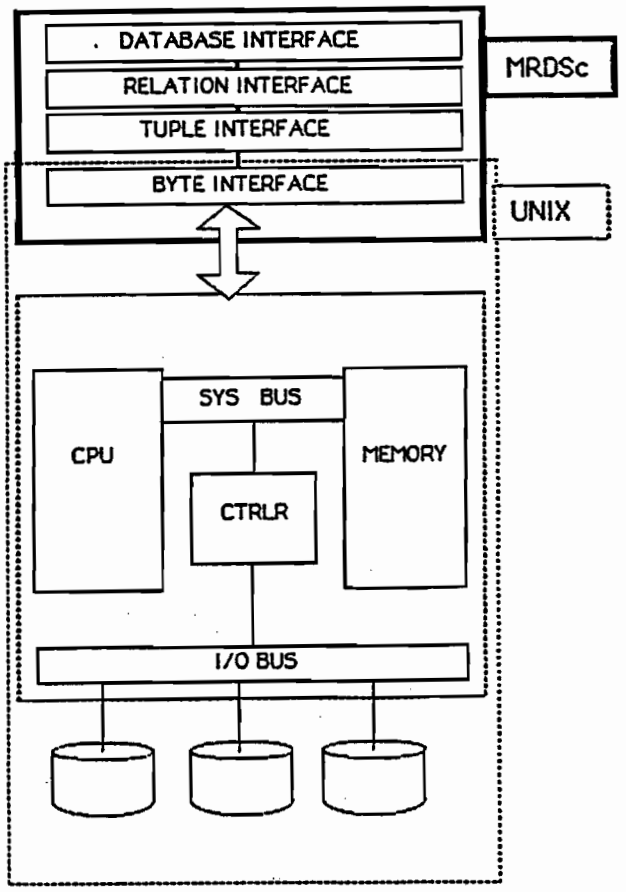


Figure 3.1: How MRDSc Fits Into UNIX

long DT_LONG 4 a 32 bit fixed point binary integer

or is a user defined data type which is some simple combination of the above types. For example, a data type 'course number' would, at McGill, be described as being of type DT_STRING The 'name' appearing in the table for these datatypes is the name by which they are known to MRDSc source code (as defined in the header file mrds.h).

All data in relations takes the form of attribute values drawn from some domain. The information needed to manage all domains is contained in the system relation dom, whose tuples are of the form:

domname	X	domtype	length

where:

Attribute	Length	Description
domname	11	a 10 byte domain name plus terminating null byte
×	1	a spare byte allocated for alignment of the following attribute values; it is at present unused
width	2	a short integer identifying the data type (0 to 5, or, DT_STRING to DT_LONG in the table above) of the domain
length	2	an unsigned short integer specifying the length in bytes of values in this domain (thus a maximum length of 65,535 bytes)

The current implementation allows 128 distinct domains per database (committing 2K of memory: 128 tuples × 16 bytes per tuple = 2048 bytes).

3.2.2 Relations

Relations consist of fixed width attribute values concatenated to compose tuples. There are no delimiters demarking the position of attributes within a tuple, nor are there tuple delimiters. A tuple may have up to 32 attributes in any combination of data types.

Each relation is stored as a separate UNIX file having the same name as the relation. All relation files pertaining to a particular database must reside within one UNIX directory. Most current versions of UNIX restrict individual files to be less than 4 Gbytes in length (the limit of a 32 bit address), hence no one relation may exceed this size. In practice, since UNIX also requires files to reside entirely within one physical storage volume, a more realistic upper limit on relation size becomes 500 Mbytes. MRDSc will allow relations up to the maximum file size supported by the host UNIX system.

The system relations rel and rd control the relations. rel is concerned with the particulars of a relation and rd with the organization of tuples therein. Tuples in rel are structured as:

relname	mode	width	fd	Zmap	cursize	maxsize	rindx	windx

where:

Attribute	Length	Description
relname	11	a 10 byte relation name plus terminating null byte
mode	1	identifies the type of relation: constant, general, flat, ordered, B^*-tree organized, z-ordered

domtype	2	unsigned short integer giving width of tuple in bytes (thus maximum tuple width is 65,535)
fd	2	short integer holding the UNIX file descriptor for the relation file; is -1 if relation file is not currently open
Zmap	4	bit map of attributes which are *z*-ordered (thus limit of 32 distinct attributes per tuple)
cursize	4	current size, in bytes, of the relation. If relation is organized as a B^*-tree then cursize represents current size in tuples.
maxsize	4	maximum size, in bytes (tuples for a B^*-tree organized relation), relation is allowed to become
rindx	4	current position within the relation of the *read* pointer, implemented as byte offset from top of file
windx	4	as for rindx, but for *write* pointer

A maximum of 85 different relations (including the three system relations) may be defined per database, committing another 3K of memory (85 tuples × 36 bytes per tuple = 3060 bytes).

Tuples in rd are structured as:

relname	domname	pos

where:

Attribute	Length	Description
relname	11	a 10 byte relation name plus terminating null byte
domname	11	a 10 byte domain name plus terminating null byte
pos	2	unsigned short integer offset within a tuple at which attribute from domain 'domname' starts, *counting from 0*

A maximum of 340 distinct rd entries are allowed per database, committing a further 8K of memory (340 tuples × 24 bytes per tuple = 8160 bytes).

Just as the system relations appear as entries in rel, so too are their corresponding entries present in dom and rd. Thus no database can have less than the following minimal content:

rel:

rel	8	36	-1	0L	108	3060	0L	108L
dom	8	16	-1	0L	208	2048	0L	208L
rd	8	24	-1	0L	360	8160	0L	360L

dom:

relname	×	0	11
mode	×	2	1
width	×	4	2
fd	×	4	2
Zmap	×	5	4
cursize	×	5	4
maxsize	×	5	4
rindx	×	5	4
windx	×	5	4
domname	×	0	11
domtype	×	4	2
len	×	4	2
pos	×	4	2

rd:

rel	relname	0
rel	mode	11
rel	width	12
rel	fd	14
rel	Zmap	16
rel	cursize	20
rel	maxsize	24
rel	rindx	28
rel	windx	32
dom	domname	0
dom	domtype	11
dom	len	13
rd	relname	0
rd	domname	11
rd	pos	22

The table below compares total allocated space (in tuples) with space available for user relations following installation of the system relations:

Relation Name	Total Allocated Space	Available Space
dom	128	115
rd	340	325
rel	85	82

3.2.3 Databases

MRDSc relations forming a database are grouped together in one UNIX filesystem directory and are given a *database name.* The actual name of the directory is independent of the database name, unlike the relationship between files and the relations they contain. A database is further characterized by having an *owner,* who must be an authorized UNIX system user.

MRDSc keeps track of all the databases under its control by means of a special relation hidden in a directory reserved for MRDSc administration. The term 'relation' applies loosely; entries in the file are not true tuples in the MRDSc sense and the file is more a table than a relation.

The 'relation' is named dblist (called DBLIST in the source code) and is structured as:

dbs_owner	dbs_ident	dbs_stat	dbs_dfltmode	dbs_name	dbs_homedir

where:

Attribute	Length	Description
dbs_owner	2	UNIX system's user ident for the owner of the database
dbs_ident	2	a unique identifying unsigned short integer associated with the database (possibly its 'tuple' number within dblist)
dbs_stat	1	current state of the database (see below)
dbs_dfltmode	1	flag bit map of default modes used to set up the database for use (see below)
dbs_name	16	name of the database plus terminating null byte
dbs_homedir	255	null terminated *full* path specification to the directory holding the database. The last character of the path *must* be a '/'

The dbs_stat flag is provided so that a pass through dblist will, at any time, provide an overall picture of MRDSc activity on each of its databases. Currently defined values for dbs_stat mark a database as *inactive* (not in use and not known to be corrupted), *active* (in use), *new* (being created), *check* (undergoing a consistency check before being made available for active use) and *corrupted* (failed the consistency check). A database remains corrupted until either

restored to health under MRDSc control (for which there is at present no implemented support) or fixed by hand.

The dbs dfltmode allows a database which is being set up for active use to be initialized to a particular state depending on the flag bits. At present no use is made of this value.

3.3 General Implementation Notes

3.3.1 Speed

Summarized here are some seemingly 'obvious' but often missed aspects of programming. Underlying all considerations of speed, at some point, is a price to be paid in memory usage; MRDSc nearly always trades memory for speed.

The basic programming building block used in constructing systems is the procedure, function or subroutine. The organizational advantages to the program of breaking a complex operation into many interconnected simpler pieces are manifest. Such 'proceduralization' exacts a noticable toll on performance, however (see Appendix 1 for a striking example). Parameters are passed by value in C, with the merciful inconsistency that arrays are always passed by reference. The overhead of the actual context switch is not insignificant, particularly if the invoked procedure has elaborate storage requirements in terms of automatically allocated variables. Generally, and especially in short procedures (which tend to be called often), emphasis should be placed on (1) minimizing storage requirements and (2) maximizing the use of register variables. This latter action reduces allocation/deallocation of automatic storage and provides greater speed outright. Since the number of registers available is highly limited (6 on a VAX), one must 'rank' register declarations so as to ensure the assignment of a register to variables which most need them. Great gains in run time can be realized by using register pointers into arrays rather than subscripted lookups.

In many cases a procedure is written to perform a simple action which is frequently required. A common example is that of a max function, which might be written as:

```
int max(a,b)
{
        return((a > b) ? a : b);
}
```

Macros, supported by the C pre-processor, can yield faster running programs. The max macro appears much the same as the procedure:

```
#define MAX(a,b) ((a > b) ? a : b)
```

but saves the stack overhead of parameter passing/returning and a context switch at the relatively insignificant expense of a few additional bytes of memory. The improvement in run-time performance is generally 10% or more.

As expensive as procedure calls are, a much more costly change of context is the use of fork followed by execl in the child process. This involves system overhead associated with initiating a new process and duplication of the current memory image of the process. In the child

this duplication is instantly made obsolete by the execl, involving swap and load overhead. Such control flow changes can make certain problems easy to solve, but are *very* expensive with respect to time and memory activity; the decision to use them should be made carefully.

Two common pitfalls with loops are (1) placing expressions which always yield the same result within the body of the loop (some optimizers, not including C's, are clever enough to correct this), and (2) use of an expression which does not change its value during loop execution for testing the stopping condition. For example:

```
while (j < 3 * relcore[thisrel].width)
/* 'thisrel' not changed in loop */
```

```
register int i;
i = j < 3 * relcore[thisrel].width;
while (j < i)}
```

The right loop uses one more word of memory, yet performs much more quickly than that on the left (at *least* twice as fast) where the array lookup (itself involving a multiply) and a multiply must be performed to yield the same result upon each iteration of the loop.

3.3.2 Memory Utilization

This implementation makes no attempt to be conservative in its use of memory except when it has reason to believe that the hosting environment cannot satisfy its greed. In such a case, allocation at run time is as conservative as is reasonable. No attempt was made to use bytes and 16 bit short integers to conserve space; in an implementation where memory is 'tight', some modest saving might be realized by doing this. An approximate breakdown of MRDSc memory utilization is:

System support structures	13.6 K
Character strings, messages	11.3 K
Statically allocated workspaces	1.0 K
Executable binary	41.5 K
Total	67.4 K

Not reflected here are (1) automatic storage allocation upon entry to procedures (but the most demanding of these uses only 11K, and is for a procedure called only once, dbck) and (2) dynamic allocation at run time. The only component of MRDSc which makes large dynamic storage demands is sortrel which at present asks for a sort/merge buffer of 200K. The most any other component asks for is on the order of 1 - 2K (in *z*-order support), with most components content with fewer than 100 bytes.

A comparison of these figures with the size of the mrds load module shows a tremendous discrepancy. This can be attributed to the way the 'includes' have been set up. Had there been separate, smaller '.h' files for separate components, then not every procedure would have had to include the one, huge, mrds.h header. It is this one, all-encompassing header file organization which inflates the load module by over 380K!

Some structures used in MRDSc have been carefully designed so as to use as little excess space as possible. Wastage occurs in structures when bytes must be 'skipped' in order to ensure that a subsequent element will be correctly aligned. Compare, for example, these possible declarations for the structure holding tuples in the rel system relation:

```
char relname[11];          char relname[11];          long Zmap;
char mode;                 unsigned short width;      char mode;
unsigned short width;      char mode;                 long cursize;
short fd;                  short fd;                  char relname[11];
long Zmap;                 long Zmap;                 long maxsize;
long cursize;              long cursize;              unsigned short width;
long maxsize;              long maxsize;              long rindx;
long rindx;                long rindx;                short fd;
long windx;                long windx;                long windx;

Size: 36 bytes             Size: 40 bytes             Size: 44 bytes
```

all contain the same fields of the same size, yet they vary in total size by as much as 22%.

There are two components in MRDSc into which memory management code was introduced. Both z-ordering and B^*-trees use complicated structures during the processing of a particular z-ordering or tree. Typically, some routine is operating on a relation and at the tuple-at-a-time level calls a procedure to deliver or process the next tuple. The latter procedure refers to these structures in order to understand the particular z-ordering or tree involved. For one operation, this routine may be needed several times *per tuple* (*e.g.*, a join on two z-ordered relations producing a third). The structures, particularly in the case of z-ordering, are costly to build and initialize. The solution was to build into these routines a fixed number of *slots* to hold the structures and to allocate to each execution of the routine on a new target one slot using least recently used replacement. The number of slots for each case is fixed at system generation time in the header file mrds.h; currently there are six slots (see sections 3.8 and 3.9).

3.3.3 Portability

No point is so contentious about C programs under UNIX as is their portability: code which is assured to port effortlessly always requires effort. MRDSc makes a valiant (but undoubtedly not totally successful) attempt at being portable to any UNIX system. This claim is not idly made: the current implementation is well-traveled, having originated on a PDP 11/45, it has spent various stages of its life on an Amdahl 5850 running UTS under VM, a Masscomp 5500DP running RTU, and lastly on a VAX 11/780 running 4.3BSD. Its gypsy upbringing has brought about the removal of the more common portability pitfalls, summarized below. It must be pointed out, though, that its wanderings have been confined chiefly to 32-bit architectures offering virtual memory.

There are some differences in the way different C compilers view the language. These are generally innocuous, with one annoying exception being the way an array of strings, received as a parameter, is to be declared. Some compilers absolutely required

```
char *parm[];
```

whereas others were content with

```
char parm[][];
```

Beyond dialectic differences, the next stumbling block concerns operand sizes, chiefly of integers. Some UNIX implementations like those for the PDP 11 (understandably) and Cadmus (not-understandably) still use 16 bit integers. In procedures where integers are 'casually' used (*e.g.*, loop counters) MRDSc declares variables to be integers without regard for what size operand it gets. For parameters, and especially in structure declarations, it is vital that *short* and *long* are used exclusively since these are universally understood as 16 and 32 bit operands respectively. Much MRDSc code depends on structures being of a particular size: were elements declared as int a ported version might produce structures of a different size.

MRDSc does have a rather cavalier attitude towards assigning ints to shorts and *vice versa*. A C compiler is assumed to generate automatically the appropriate conversion code. A more conscientious attempt has been made in more recently developed code to use explicit casts in such assignments, but the bulk of MRDSc is free of such casting clutter.

Other operand data types are not so susceptible to these problems. Characters are 'universally' represented as individual bytes, and while MRDSc believes in ASCII characters, it is not dependent upon character set, beyond requiring that a character which is lexicographically greater than another be represented by a bit pattern which is greater.

The most difficult portability issue concerns UNIX library and system calls which are highly variable from one system to another. In particular, MRDSc is largely dependent upon Berkeley UNIX implementations. System V compilers/loaders may well have nervous breakdowns attempting to generate a load module. The macro definition feature of the C preprocessor can be used to direct translation according to the version of UNIX. In some instances this produces nearly unreadable source code (see timer.c), but is worth its weight in gold when routines end up porting effortlessly and correctly. An implementer is also directed to use older, often lower level, routines, rather than exploit newer ones available in new releases. For example, one is reasonably assured that the read and write system calls will behave identically across UNIX versions; higher level, potentailly more sophisticated ones, may not.

A serendipitous consequence of extensive use of the pre-processor to ease portability is ease of extension. Concentration of 'global' variables and definitions of 'magic numbers' in header files makes it much easier to extend existing features or add new ones by eliminating witch-hunts through old code for occurrences of values which must be changed to accommodate new code.

3.4 Program Organization

MRDSc is approximately 8000 lines of C organized as roughly 40 separate routines and one "all inclusive" header file. Users can access it by plugging some interactive interface into it, or by calling its user level procedures from a program in some programming language, and linking with the mrds.a archive library.

Virtually all the source code is contained in a single directory with some supporting code in adjacent ones. The price of this is hugely inflated symbol tables in object files; something later versions may correct.

The header file includes system include files (like stdio.h and sys/types.h) required for any of the procedures, and declares *all* global variables and #defines. The large number of these

latter definitions (approximately 200) arises from the desire to facilitate portability and expansion.

Logically, MRDSc is built from the following major components:

- system relations and routines which maintain them,
- input/output subsystem which interfaces directly with the host UNIX system,
- the Tuple Access Method component which understands the different data organizations of relations (*e.g.*, *z*-ordered) and acts as a filter to provide data organization independence to components logically 'higher' in the system,
- a small family of relational operators and the support procedures needed to implement them,
- the error and exception handling component which deals with the unexpected,
- relational sort/merge required to support some relational operators, but is a self-contained, hence separate, component,
- a test user interface: an interactive command interpreter of very limited sophistication designed to facilitate testing of completed components (and *not* to serve as a general purpose user interface)

The components are connected as shown in Figure 3.2. Generating the system is controlled by a makefile which is straightforward to use. Standard flags which should be set when compiling are: BSD if on a Berkely system (otherwise define OLDUNIX and hope for the best), and one of VAX, M68000 or AMDAHL depending on the family of CPU being used. XTRACE can be defined if one wants to enable compilation of #ifdefed debugging and trace code. The statements print messages upon entry to, and usually upon normal exit from, a procedure. Be warned: these debugging messages are *all* on or off, and if enabled, produce voluminous output. Some procedures include further print statements which have to be explicitly un-commented to provide further details of the procedure's workings.

The following guidelines are observed by MRDSc procedures:

- *all* have an explicitly declared type
- procedures of type int return a negative value to indicate failure: FAIL (-1) indicates an irrecoverable failure, other negative values imply that the procedure was partly, but not totally, successful in performing its task; otherwise they return a non-negative value whose meaning is peculiar to the procedure and the context in which it was used
- procedures of type *some type return (*some type)(NULL) to indicate failure to their callers; otherwise they return a value whose meaning is peculiar to the procedure and the context in which it was used
- a procedure *always* has a return statement; none 'falls off' the closing brace bracket

3.5 System Startup And Execution

The first order of business when MRDSc begins running is a startup sequence in which the following occurs:

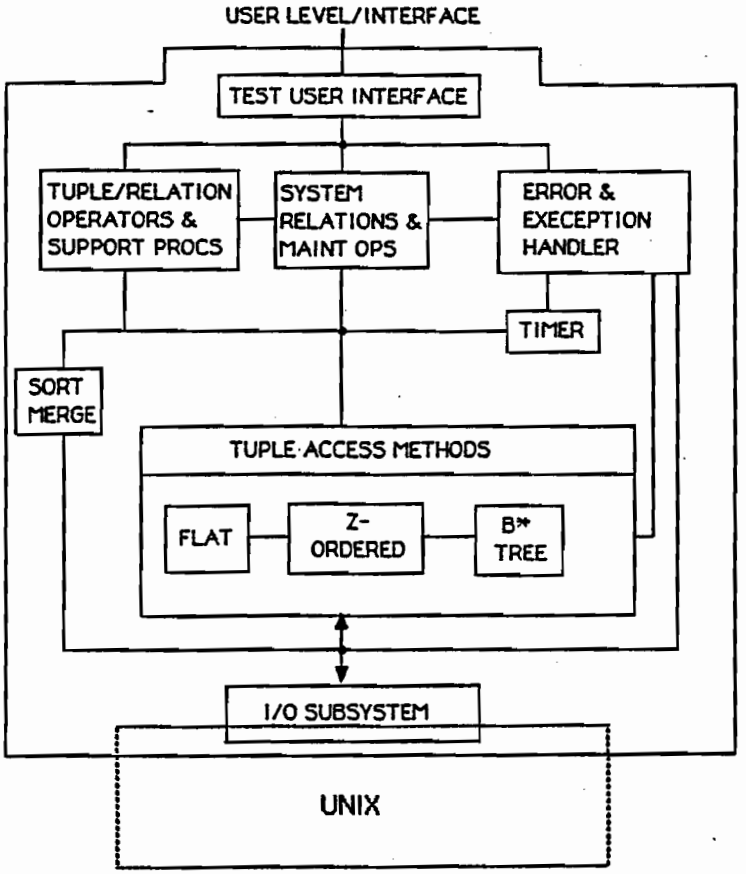


Figure 3.2: MRDSc Organization

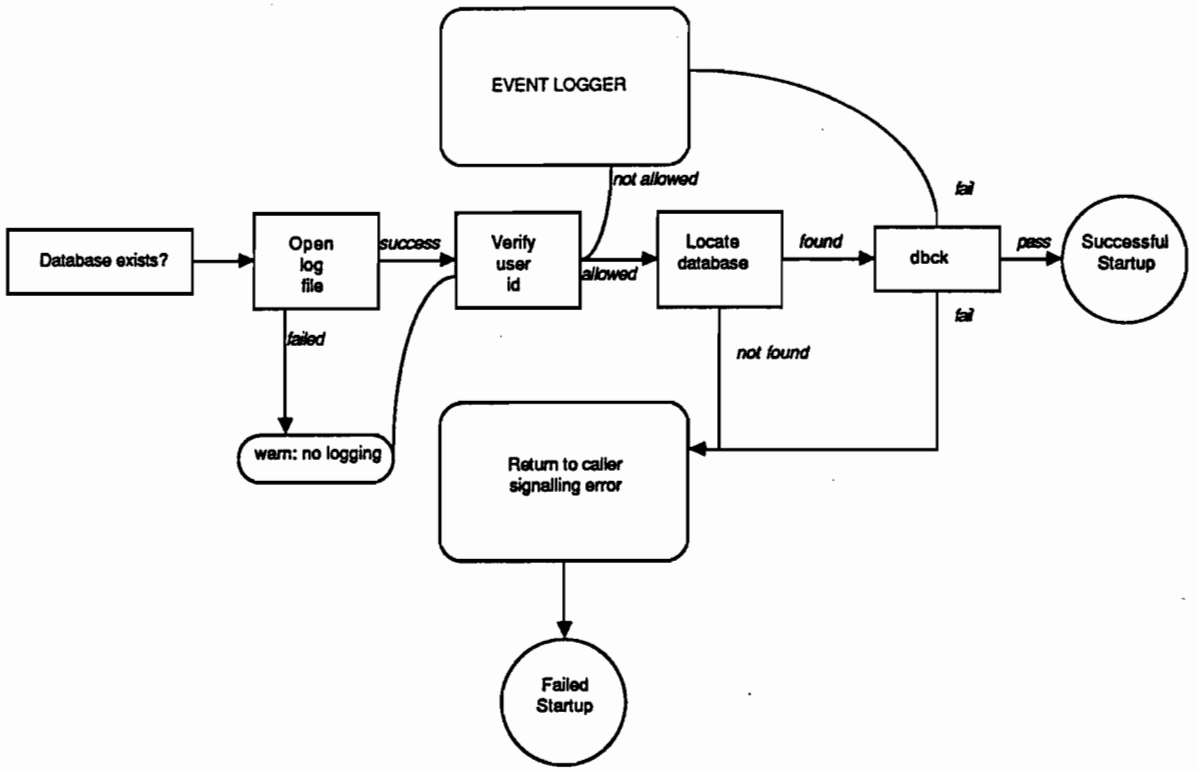


Figure 3.3: MRDSc System Startup Sequence

3.5.1 User Authorization Check

The checking performed to verify a user's access privileges is very simple minded. An MRDSc system file maintains a list of UNIX user idents (as found in '/etc/passwd') of authorized users. The format of the file is simple: one uid per line, terminated by a colon. MRDSc performs a sequential read of this file in gooduser(), attempting to match the effective uid with one of those in the file. Failure to match results in immediate termination of the program.

This authorization scheme serves only to deny access to unauthorized UNIX users. It does *not*, nor is it intended to, provide security amongst authorized users. The current implementation of MRDSc does not have any notion of selective access permissions; *i.e.*, any authorized user would be able to see any database. A partial solution to this, and one currently implemented, is that only a database's owner may access it (but see below).

3.5.2 Session Logging

MRDSc maintains a log file (known in the source code as LOGFILE) wherein are recorded startup and finishing times, along with the user ident and database name involved. The consistency checking procedure also records its findings here. The log file is provided so that an MRDSc administrator can monitor session and database activity, and in the event of crashes, receive early notice of potential problems with a database.

The log file is a plain ASCII file with entries time stamped to the nearest second, and consisting of two fields: an *event* and an explanatory *message*. For a normal session, there will appear four entries in the log file:

Event	Accompanying Message
INVOKED	the uid of the user
CHECK	database name and its owner's uid
ACTIVE	database name and owner's uid; the presence of this event and message implies that all consistency checks were successfully passed
CLOSED	database name

Should the consistency check fail, rather than the event ACTIVE, CONFUSED will be reported, followed by a TERM (for 'terminated') event.

More MRDSc activity may be logged in this file by adding calls to logent() as desired. In particular, added calls to logent would make it possible to obtain performance timings for relational operations to resolutions of one second. Such added logging could also facilitate consistency recovery following a crash.

3.5.3 Database Selection

Upon successful user authorization, MRDSc tries to establish the identity of the database to be worked on. If the database name was provided on the command line (via the '-n' option), MRDSc begins looking immediately. Otherwise it fails to setup a database and signals its invoking program (such as an interactive user interface) to this effect. The invoker can then respond appropriately, for example, by interactively soliciting the name of a database.

find_db is used to determine if the name received corresponds to that of any known database. The list of such databases is maintained in another MRDSc system file (known as DBLIST) with one line corresponding to one database (see section 3.2.3). The MRDSc administrator maintains this file by hand. MRDSc has a naive view of access permissions, allowing access only to the database's owner. A way in which a database may be shared is to repeat its entry in DBLIST, changing the uid ('owner') field on each line, leaving the name, path, and other information unchanged; access control is then governed by regular UNIX file permissions.

If the database is located, a pointer to a structure containing its entry from DBLIST is returned; otherwise (DATABASE *)(NULL) is returned.

3.5.4 Consistency Check

Whenever a UNIX system is booted-up a check is performed on the filesystems to assure that they are consistent. On a medium sized system, 25 to 40 thousand files (or approximately 800 Mbytes), these checks take about 15 minutes. The consequences of attempting to run with corrupt filesystems (partially or wholly lost files or directories, system panics and crashes) make the performing of such checks worthwhile.

The consequences of running a database program on an inconsistent database can, on a relative scale, be similarly disastrous. For this reason MRDSc first performs a check on the database upon which it has been asked to work. These checks were particularly valuable during development of the system, since undebugged code frequently crashed leaving the test database in a shambles. The fact that inconsistencies were reported prevented further damage and the actual inconsistency reports were helpful in isolating the source of the failure.

The procedure dbck performs the consistency check after first calling procedures to load the three system relations into their respective memory resident structures. The routines loadrd, loadrel and loaddom load the corresponding system relations and return the number of entries encountered during the load (*e.g.*, loadrel returns *n*, the number of relations in that database). These load procedures check to ensure that the minimal quantity of system relation information is present (currently, 3 relations, 13 domains and 15 rd entries, or, MINRELS, MINDOMS and MINRDS in the header mrds.h). Failure to locate all system relation entries will abort the run of MRDSc.

Assuming the system relations are intact, dbck itself begins examining them and any user relations. The checks are limited to assuring that:

- for each relation encountered in rel there in fact exists a file corresponding to that relation,
- the file is the same size as the entry in rel suggests,
- the number of relations known in rel is the same as the number of relations mentioned in rd and indicates where there are mismatches (*i.e.*, total number of relations could be correct, but 'distribution' might be wrong),
- every domain name appearing in rd also appears in dom

All checks are run to completion and a count of the number of errors detected is returned by dbck. Normally the work is done silently, but by passing a non-zero (hence 'true') value for the second parameter, dbck will issue an error message each time an error is detected, as well informatory messages about what it is doing. A sample of an unsuccessful dbck appears below (dbck was called with a 'true' second parameter value):

```
dbck:  checking db 'plas'
       Checking rel
****   Missing relation 'redparts  '
****   Expected 7 but found 6 relations
       Checking rd
End of check on database 'plas' : 2 errors detected
```

Upon reaching the point of having a consistent database with system relations resident, dbck signals success to its caller. MRDSc continues to execute until: (1) the user requests termination, (2) the value of RUN has exceeded a preset threshold as a result of an excess of errors, or

(3) an irrecoverable error occurs. A fourth condition, present in MRDS but not implemented in MRDSc is to abend after too many I/O operations have been performed. If execution terminates because of either of the first two reasons, the memory resident image of the system relations is flushed back to disk and MRDSc exits. If an irrecoverable error occurred but MRDSc still has control, it will attempt the same graceful termination; otherwise it exits and the state of the database usually becomes inconsistent.

Should the consistency check prove to be objectionable (presumably by taking too long), it can be excised from the setup phase. If it is removed, then some mechanism must be made available to setup with which it can determine whether the last time the target database was 'closed' it was done so in a controlled way, and hence may be assumed to be consistent upon subsequent opening. This approach has been applied to UNIX filesystems by Masscomp for their RTU operating system and has been found to be not altogether reliable, *i.e.*, RTU will occasionally believe it has 'clean' filesystems when it does not.

3.5.5 System Relations At Run Time

setup loads the system relations rel, dom and rd into the arrays relcore, domcore and rdcore. The arrays are statically allocated at compile time of size MAXRELS, MAXDOMS and MAXRDS respectively. Once loaded into memory no further reference is made to the files containing the relations until syncrel is called when MRDSc ends execution.

rel is updated by both rdtuple() and wrtuple() which adjust the read index rindx or write index windx following a read or write to point at the 'next' tuple to be read or written. In the case of wrtuple, if the write action is appending to a relation, then cursize is also updated to reflect the increased size of the relation. rel is updated by mkrel() when a user level procedure (*e.g.*, select) which creates as output a relation runs. mkrel fills in the next available element (as reported by the structure sysrelstat) in relcore with particulars on the new relation, then fills in the appropriate entries in new elements of rdcore.

When a procedure needs particulars on a relation but only has the relation's name (as would be the case for any user level procedure) rel is consulted by findrel() which returns a pointer to the appropriate array element, or (REL *)(NULL) if no such relation name exists.

Since there is no way to create new domains under the current implementation dom is a read-only system relation, consulted by finddom() to confirm the existence of a domain and obtain its datatype and length. finddom returns a pointer to an element in domcore, or (DOM *)(NULL) when a sought entry cannot be found.

rd is updated by mkrel as described above, and is consulted by findrd() to answer questions about "what is where" in a relation. findrd is designed to provide the rd entries for one or more relation/domain combinations, possibly all of them, for any relation. It returns a pointer to the first rd entry matching what it was asked to find, and sets a global integer rdents to reflect the total number of rd entries for the relation.

When MRDSc is shutting down, the memory resident version of each system relation is written back to its disk file (in syncrel). Until that moment, the only current version of the system relations is that in memory; future versions might wish periodically to flush these memory images to disk during run time. A side effect of being memory resident is that access to the system relations is restricted exclusively to the procedures described above. Thus, for example, one cannot perform project or select on them.

In summary, there are three system relations (rel, dom and rd) containing all particulars on relations, domains and tuple layouts (themselves included). They appear in the directory containing the database as the hidden files '.rel', '.dom' and '.rd'. The files are loaded (by loadrel, loaddom and loadrd called from dbck) at startup into arrays of structures, and only the arrays are used during run time. At the end of execution the arrays are flushed back to their files.

The system relations are accessible only to MRDSc internal procedures. rel is updated by mkrel, rdtuple, and wrtuple and is consulted by findrel. dom is never updated and is consulted by finddom. rd is updated by mkrel and consulted by findrd. All are written by syncrel. *Note* that by rel, dom and rd above is meant "the memory resident images of".

Relevant Entries in header file:

#define Rel ".rel" #define Dom ".dom" #define Rd ".rd"	file names of the system relations used by loadrel, loaddom, and loadrd. Changing these defines allows the names of the system relation files to be changed since the only reference to the files by name occurs in the 'load' procedures.
#define MINRDS 15 #define MINDOMS 13 #define MINRELS 3	These reflect the minimum number of relations, domains and rd entries which must be present in the system relations. Any change to the composition of the system relations which would affect the minimal number of entries must be reflected here; *e.g.*, adding another attribute to rel would require changes to MINRDS and possibly also to MINDOMS.
#define MAXRELS 85 #define MAXDOMS 128 #define MAXRDS 340	impose limits on the size of a database and composition of its relations. Changes to these values have a small effect on the overall size of the system: the size of the statically allocated arrays increases by sizeof(REL) (now 36 bytes) for each additional relation allowed, and similarly by 16 bytes for each added domain and 24 bytes for each new rd entry. There are some z-order procedures which allocate arrays of these dimensions, hence automatic storage requirements would change.
mode bits	The byte field mode in the rel struct conveys data organization and read/write access information as follows:
#define CONREL 0x01	0 if general relation, 1 if constant relation
#define ZORD 0x02	0 if not z-orderd, 1 if is
#define PZREL 0x04	0 if not, 1 if is partially z-ordered
#define FLAT 0x08	0 if data organization not flat, 1 if is
#define ORDER 0x10	0 if not ordered, 1 if is
#define BTREE 0x20	0 if not, 1 if is organized as B^*-tree
#define APONLY 0x40	0 if writable anywhere, 1 if append only
#define WRINH 0x80	0 if writable, 1 if read only

	MRDSc does not necessarily have code to support all modes, *e.g.*, the current implementation does not understand constant relations
#define MAXRNMLEN 11 #define MAXDNMLEN MAXRNMLEN	maximum length for a relation or domain name, *including* the null byte terminator. Changing these will have very widespread effects (such as changing sizeof(REL)). Procedures directly affected include: dbck, findrel, mkrel, and openrel (change to MAXRNMLEN) and z, dbck, finddom, findrd, mkrel, project, select, and z (change to MAXDNMLEN).
static int relfd; static int domfd; static int rdfd;	file descriptors for the system relations used (set) in the respective load procedures. These are essentially redundant; the member fd of the rel structure for each system relation should be used for this, and these variables eliminated

typedef struct relrcrd {
  char relname[MAXRNMLEN];
  char mode;
  unsigned short width;
  short fd;
  long Zmap,cursize,maxsize,rindx,windx;
} REL;
	structure used to hold one rel entry. For details on members, see section 3.2.2; see also section 3.3.2 regarding arrangement of members for alignment and minimal size of structure if planning to add, remove or change structure.

typedef struct domrcrd {
  char domname [MAXDNMLEN];
  char spare;
  short domtype;
  unsigned short len;
} DOM;
	structure used to hold one dom entry. For details on members, see section 3.2.1; see also section 3.3.2 regarding arrangement of members for alignment and minimal size of structure if planning to add, remove or change structure.

typedef struct rdrcrd {
  char relname[MAXRNMLEN],
  domname[MAXDNMLEN];
  short pos;
} RD;
	structure used to hold one rd entry. For details on members, see section 3.2.2; see also section 3.3.2 regarding arrangement of members for alignment and minimal size of structure if planning to add, remove or change structure.

struct rstat {
  int numrelents;
  int numdoments;
  int numrdents; } sysrelstat;
	structure holding current number of relations, domains, and rd entries in a database. These values are used as subscripts into the arrays of relcore, domcore, and rdcore.

int rdents;	globally used integer holding the number of rd entries of the relation for which findrd was most recently called (this variable is set as a side-effect; see section 3.5.5)

3.6 Input/Output

MRDSc performs input/output on several fronts. Firstly, it communicates with a user by reading standard input and writing to both standard output and standard error. This makes it possible to operate MRDSc "interactively" from script files, and to save output using script or simple redirection.

The bulk of MRDSc I/O operations involve files (relations, to the user). The user is never aware of an I/O entity except as a relation, while MRDSc internals must be concerned with relations, tuples, and bytes, and at the same time, the different data organizations supported. This section concentrates on the implementation of the byte, tuple and relation interfaces shown in Figure 3.1. The way in which procedures at the different layers connect is illustrated below for reading a tuple from a *z*-ordered relation:

3.6.1 Byte Interface

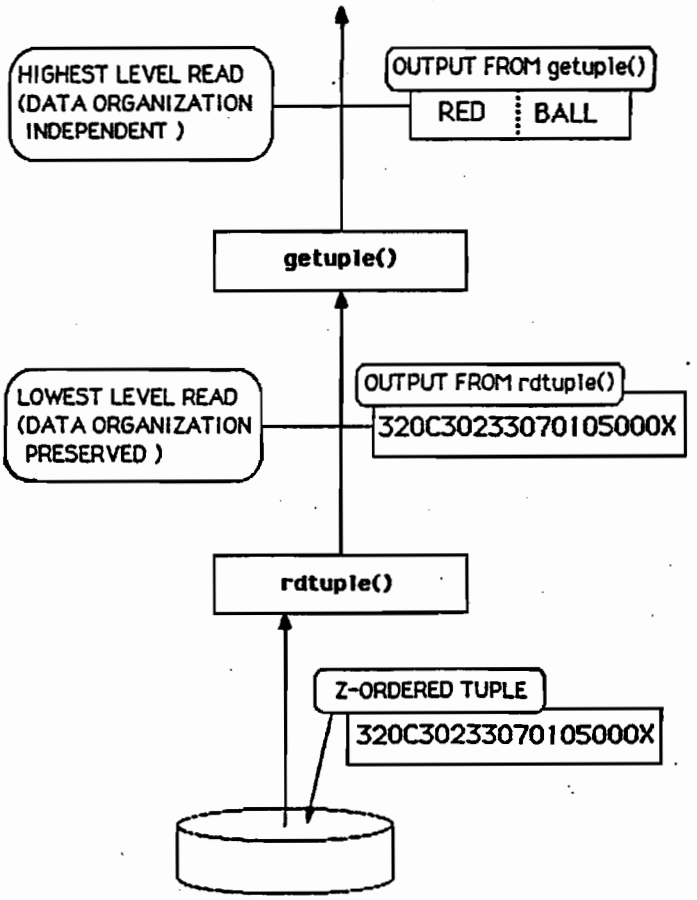


Figure 3.4: Read Access On a *z*-ordered Relation

All relations under MRDSc are UNIX files, hence this lowest I/O level is appropriately embedded within UNIX and is implemented by its open, read, write, lseek, and close system calls.

Invariably one first issues an open call, providing a UNIX file name and a flag wherein the intended mode of access is specified. In response, UNIX returns a file descriptor which is the element of the _iob array assigned to the now open file. Typically UNIX imposes a limit of 20 simultaneously open files; the new 4.3BSD imposes no restriction. The file now opened is a stream upon which direct access is provided by the lseek call. A single pointer to the current position in the file is kept; it can be moved by a read or write to the first byte following that which was just read or written, or by an lseek, to an arbitrary byte.

MRDSc routines above this level avoid using read and write directly (except loadrel, loaddom, loadrd, and syncrel for obvious reasons). All procedures wanting to read/write a relation must pass the request down until, at this level, the actual I/O operations are performed. The UNIX lseek call does appear in higher MRDSc routines, up to the level of adtuple(see below).

The UNIX read and write system calls are surreptitious in their operation. A call to read or write does not necessarily result in a disk I/O operation as UNIX performs read-ahead and write-behind caching. And while UNIX documentation implies that reads and writes can be performed on data of arbitrary size, the actual amount of data transferred to/from disk by UNIX is invariably a multiple of the device's blocksize, irrespective of the user specified size. Further, most implementations of UNIX perform 'double buffering' of read/written data. That is, a user requested read causes some number of disk blocks to be read into a buffer in UNIX system data space, then the actual data requested is copied into the user's specified buffer location (Masscomp's RTU does not do this, but works with user data space directly).

Owing to the way UNIX implements its file I/O and the design decision to implement a database as a collection of separate UNIX files, MRDSc does not perform any buffer management in the style of the original MRDS where a database was a single paged file. Rather, the buffer space allocated for each opened file/relation is minimal (see below).

3.6.2 Tuple Interface: Bottom Layer

This layer of MRDSc code translates relation/tuple I/O requests into 'file/record' I/O requests understandable to the byte interface and the UNIX system calls it uses. The procedures closerel, flushpage, insert, loadpage, openrel, rdtuple, split, and wrtuple implement this layer.

openrel receives a pointer to a relcore entry and an access mode, and attempts to 'open' the relation for input/output. It must first fiddle with the relation name obtained from the relcore entry since the latter may have trailing spaces to force it to MAXRNMLEN, but the filename does not have these spaces, and, it must prepend the path to the database's directory. Following this openrel uses a plain UNIX open on the file. If the open succeeds, the file descriptor is saved in the fd member of the relcore entry and a buffer is allocated for use with the relation. Since UNIX is performing disk caching, attempting to minimize asynchronous read faults caused by, *e.g.*, read requests for uncached data†, the buffer allocated by openrel is one tuple wide.

The address of this tuple buffer is kept in the global array buffer; the element of the array used is that indexed by the file descriptor. Since many of the MRDSc procedures which request a

† 4.3BSD UNIX may perform this even better since a user can specify the expected activity pattern for file accesses on the open, *e.g.*, sequential, random, *etc.*

tuple to be read/written already have their own buffer space allocated, particularly those supporting B^* trees, the buffer provided by openrel need not always be used. Thus, a procedure can be assured that there will always be a buffer available for I/O should it not wish to create its own.

closerel is used when work is finished on, or it is necessary to change the access modes (by a close and re-open) of, an opened relation. The file is closed using a standard UNIX close system call, the fd member of the relation's relcore entry set to -1, and the buffer space allocated by openrel is released.

rdtuple asks UNIX to read tuple-width bytes from the file containing the relation. The bytes read are placed in the default buffer for the relation, as provided by openrel, or in the caller's specificed location. If rdtuple is invoked on a closed relation (not considered an error) it calls openrel before continuing, opening the relation as read-only. rdtuple consults the relcore entry for the relation being read to find the value of the read index, rindx. It then uses the UNIX lseek system call to ensure the file pointer will be at position rindx within the file. Finally, it issues a read UNIX system call. Following a successful read, rindx is incremented by the width of one tuple for all data organizations except BTREE in which case the pointer to the next leaf entry is read from the relation and put into rindx. The intention is to leave rindx pointing to the logically next tuple in the relation. Any operation for which such sequential reading is inappropriate will have to change the value of rindx before issuing its next rdtuple call.

wrtuple is precisely analogous, except that it has more checking to do before issuing a UNIX write call. Specifically, it must determine if an illegal write operation is being attempted, and if so, return failure to its caller. A write is illegal if it occurs anywhere on a read-only relation (mode of which is WRINH), or if it fails to be an append to an APONLY mode relation, or, if by appending to the relation, it's size would exceed maxsize. Beyond this, wrtuple performs the same sequence of actions as rdtuple, leaving the write index incremented by the width of a tuple in *all* cases.

That rdtuple takes into account the possibility of a B^*tree organized relation and wrtuple does not is one unorthogonal feature of the MRDSc I/O system. A B^*tree may be read by a procedure as naive as rdtuple, but it cannot be updated so simply as wrtuple would have it. Adding a tuple to a B^*tree requires linked list manipulation, searches for the correct insertion point to maintain ordering, and may involve node splits. For this reason, writing to a relation organized as BTREE is handled by insert and split rather than by wrtuple.

insert relies on search to locate the point in the B^*tree at which the new tuple should be added (search is described in section 3.9.4). Unless search indicates that the tuple being added is already present, insert simply plugs the new tuple into the linked list of tuples in the appropriate leaf page. The updated memory copy of the leaf page is then written back to the file using flushpage.

flushpage (and the companion loadpage for input) are to B^*trees what wrtuple and rdtuple are to flat organized relations. Instead of reading or writing a single tuple from the relation, these read or write single nodes, either leaves or branches. They are highly analogous to the simple single tuple I/O routines they resemble: each performs a seek to the start position of the target node and then issues a read/write call of 'nodesize' on the file.

3.6.3 Tuple Interface: Top Layer

This layer of the I/O system, implemented by getuple and adtuple, is the first layer seen by user invoked procedures, *e.g.*, printrel reads a relation by issuing repeated getuples. These two routines do little else than issue calls to their counterparts in the next layer down, rdtuple and wrtuple. The distinction is that, at this level, tuples are viewed independently of their underlying data organization (see Figure 3.4). Thus, a printrel on a z-ordered or B^*tree organized relation can be performed without printrel being aware of that data organization. printrel calls getuple which, seeing a z-ordered relation, unshuffles the tuple it received from rdtuple. In the case of BTREE organization, getuple behaves as if it is accessing a flat relation, but adtuple calls insert rather than wrtuple.

3.6.4 Relation Interface

This is chiefly an output layer whose responsibility is to produce new output relations from user invoked operations like project or select. The procedures mkrel and replace implements this layer.

mkrel begins by verifying that it can work with the parameters it has received and that there is still space to accommodate more relations in the database. The verification process includes checking that one is not trying a create a duplicate relation and that all requested domains exist and are not duplicated within a tuple. The properties of the relation created by mkrel, *e.g.*, mode, Zmap, maxsize are determined from parameters sent by the caller. cursize, rindx, and windx are initialized to 0, fd to -1, and width is calculated by mkrel from the lengths of the individual domains which appear in tuples of the new relation. The next available positions in rel and rd are updated to contain the necessary data on this new relation and the file to hold the relation is created.

mkrel stoops to using a UNIX system call to create the file entry, rather than passing a request down to the byte interface level, for expediency's sake, and because calls to open at the byte interface level expect the file to exist already. The file so created is closed upon mkrel's return.

The ability to replace an existing relation is provided by replace. This need arises when the data organization of a relation is changed, thus replace is called by sortrel and mkindex to replace a current relation with a sorted or B^*tree organized form. Each of sortrel and mkindex perform their own file I/O on intermediate files used during their operation, and when finished replace the previous relation file with the last of their temporary files. replace truncates the 'old' relation file, then copies the replacement file into it. Upon successful completion of the copy both files are closed and the temporary file is removed. It might be faster to rename the file received from sortrel or mkrel rather than perform a copy, but since these temporary files are not generated in the database's directory, a file copy of some description is unavoidable.

In summary, the I/O system of MRDSc refers primarily to its file/relational I/O (as opposed to 'terminal' I/O with a user) and is implemented in layers, as shown below in Figure 3.5

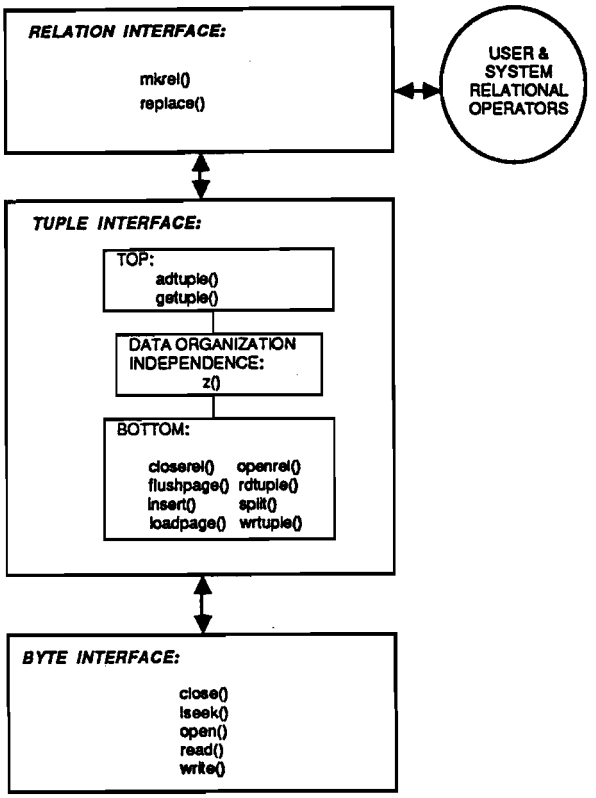


Figure 3.5: The MRDSc I/O System

Relevant Entries in header file:

#define MAXOPNFILES 16	maximum number of simultaneously active files; used to control number of elements in buffer array (see below)
#define APONLY 0x40	relation mode bit indicating that any writing done to the relation must be an append; overwriting existing tuples is not permitted. This bit is checked in wrtuple
#define WRINH 0x80	relation mode bit indicating that no writing is allowed on this relation; it is *write inh*ibited.
#define READMODE 0 #define WRITMODE 1 #define RDWRMODE 2	mode values to use with UNIX open system call. Newer MRDSc code uses mode names taken from <sys/file.h>; any new code should stick to these latter names rather than those in mrds.h (except that use of the names at left assure portability by reason of self containment)
#define FROMTOP 0 #define FROMCUR 1 #define FROMEND 2	origin values to use with UNIX lseek system call. The same observation applies regarding names in <sys/file.h>.
#define BLOCK 512	

	size of an I/O block used in replace when performing its copy, and in printrel as a liberal size to use for a work buffer for one print line. The value 512 was selected when MRDSc was still on a PDP-11; the value should be changed to something like 4K (or whatever is returned in st_blksize in the stat struct for the file) and printrel should use a separate value for a work buffer length
static int IOCOUNT = 0;	set upper bound on number of allowed I/O operations; not used in current implementation.
char *buffer[MAXOPNFILES];	an array of pointers to the buffers allocated by openrel; the opened relation's file descriptor is used as the subscript into the array at which its buffer pointer is found

3.7 Sort/Merge

The need to sort relations arises in MRDSc most often to facilitate the elimination of duplicate tuples from relations. It may also be used when converting a 'flat' unordered relation into a flat ordered one.

The sort is organized around the UNIX qsort routine which, given a pointer to a memory area containing the entries to be sorted, the quantity and length of each entry (*i.e.*, entries must be of fixed length) and a compare function, will quicksort the data. The compare function must expect pointers to two of the data to be sorted and return a value which is negative (first is less than second), zero (first is equal to second), or positive (first is greater than second). MRDSc uses as its compare function tplcmp which uses information from dom and rd to compare tuples, attribute by attribute until a determination can be made.

After convincing itself that its input parameters are acceptable, sortrel allocates a memory region of size SORTLIM Kbytes for use.† The present implementation gives up immediately if it cannot allocate the required buffer; future versions should iteratively try smaller sizes down to around 32K before bailing out. sortrel determines how many tuples fit in the sort buffer and decides whether a single sort or multiple sort/merge iterations will be needed. In either case it allocates a string to contain the name of the temporary merge file, and sets the name therein to a null string. sortrel enters a loop reading bufsize (if all is well, SORTLIM * 1024) bytes and calling qsort, and merge if needed, until all tuples have been sorted.

The merge subcomponent is slightly non-standard. It is designed to merge one file with an 'in core' buffer to save extraneous writes of temporary merge files. Upon qsort's return, an attempt is made to open the previously used merge temporary file. If there was no such file, one is opened now and the current in core buffer written to it, with duplicate tuple suppression done during the write and the file closed. If there was such a file, merge is called to perform a straightforward merge, eliminating duplicate tuples. The output is written to a new merge temporary file and the previous one is deleted. When all tuples have been sorted, the last merge temporary file replaces the file containing the unsorted relation and sortrel returns.

† currently, SORTLIM = 200 except on 16 bit machines, where it cannot ever exceed 64

Relevant Entries in header file:

```
#define MERTEMP "/tmp/smXXXXXX"
```

template string used to generate unique names for merge temporary files used with the UNIX mktemp routine

```
#define SORTLIM 200
```

size of memory region to be allocated for sort buffer; it is in units of *K*, and, in the even of machines with 16 bit addressing (maximum segment size of 64K), *must* be less than or equal to 64

```
#define DT_STRING  0
#define DT_INT     1
#define DT_CHAR    2
#define DT_FLOAT   3
#define DT_SHORT   4
#define DT_LONG    5
```

names of datatypes used in tplcmp to identify which comparison to used for an attribute compare; see section 3.2.1

```
union u_short {
    short sval;
    char c1,c2;
} short_coerce;

union u_int {
    int ival;
#ifdef INT16
    char c1,c2;
#endif
#ifndef INT16
    char c1,c2,c3,c4;
} int_coerce;
#endif

union u_float {
   float fval;
   char c1,c2,c3,c4;
} flt_coerce;

union u_long {
   long lval;
   char c1,c2,c3,c4;
} long_coerce;
```

structures used in tplcmp to 'convert' values pointed at by the char * pointers it receives to the necessary data type, without regard to alignment. This coercion is necessary since tplcmp receives only a pointer to the tuple, which may have any mix of different data types.

```
REL *reltosort;
```

global variable containing pointer to rel entry for the relation to be sorted

3.8 *Z*-Order

Two entries in the rel descriptor for a relation contain information pertinent to *z*-ordering. The byte mode may contain ZORD,PZREL, or 0 to indicate that all, some but not all, or none of the attributes participate in the *z*-ordering. The long integer Zmap is a bit map denoting

attributes appearing in the ordering with a one bit corresponding to the attribute's position within a tuple. For example, if the third attribute in a tuple is the only one participating in *z*-order, Zmap would equal 4.

The support for *z*-ordering is concentrated in z.c which performs both shuffling and unshuffling of tuples. More precisely, z only performs reshuffling: reshuffling a *z*-ordered tuple to a non *z*-ordered tuple achieves the effect of unshuffling. z receives tuples of one Zmap value and produces tuples represented by a new Zmap value. It does this by first completely unshuffling the tuple (new Zmap = 0) and then, if required, shuffling the now unshuffled tuple according to the new Zmap.

z relies on a somewhat involved collection of dynamically allocated structures to do its work efficiently. Allocating these structures each time a tuple is to be reshuffled is prohibitively expensive, so z maintains a pool of 'slots' (which should never be fewer than three) holding the necessary structures. Thus, the cost of reshuffling the first tuple is high, but subsequent tuples can be processed cheaply. The slots are recycled using a least recently used policy.

The shuffling/unshuffling process is inherently circular: to shuffle j attributes of a tuple, begin with bit 0 of attribute 1, then bit 0 of attribute 2, $\cdots$, to bit 0 of attribute j, then increment the bit number and return to attribute 1. This continues until all required bits have been interleaved. If, at some point, one attribute runs out of bits, it is skipped over for the remainder of the shuffling process.

This circularity prompts the allocation in z of a circular list (zclist) which contains, for each attribute in the *z*-ordering, the number of bits in the attribute and the *bit* position within the tuple at which this attribute begins in a completely non *z*-ordered tuple (*i.e.*, the position as reported in rd), plus pointers to the next and previous entries in the circular list.

The implemented *z*-ordering has the property that should not all of the attributes be involved in the ordering, the *z*-ordered bits are placed in the most significant bit positions of the tuple, and the non *z*-ordered bits are pushed towards the least significant positions. For example, in a tuple of 8 equal length attributes where only even numbered attributes are to be *z*-ordered, the interleaved bit string representing attributes 2, 4, 6, and 8 appears in the bit positions formerly held by attributes 1, 2, 3, and 4; attributes 1, 3, 5 and 7 are concatenated into attribute positions formerly occupied by attributes 5, 6, 7, and 8.

To accommodate relocated non-*z*-ordered attributes, z uses a linked list (zllist) which, for each such attribute, contains the byte positions where that attribute is supposed to begin and where it does now begin, plus the pointer to the next list entry.

Since z first unshuffles a tuple and then may need to shuffle it to produce a different *z*-ordering, one would expect two circular lists to be needed. In fact, four circular lists are used: two hold the abovementioned particulars (masters), and the other two serve as working copies. The working copies are initialized from the masters each time a tuple is reshuffled.

The sequence of events is as follows (it is recommended that one follow the source code when reading through these steps):

(1) locate the slot containing the structures corresponding to this relation and input/output Zmap values. It is necessary but not sufficient that the relation names match since a relation may have been used previously with different Zmaps.

If no slot is found, pick the least recently used one and initialize its time stamp, set up the particulars for this new reshuffling (pointer to rel entry, 'to' and 'from' Zmap values, number of attributes per tuple). Free any previously allocated list structures associated with this slot, set pointers to NULL and begin building new lists.

To build the lists proceed attribute by attribute, from most significant to least. If the bit position in the 'from' Zmap corresponding to the current attribute is set, attach another entry to the circular list, initializing its data members to the length and starting position (in bits) within the unshuffled tuple. Otherwise, attach another entry to the linked list, initializing its 'to' member; the 'from' member must be done later.

Having now generated the circular and linked lists needed to unshuffle the tuple, repeat the exercise to generate corresponding lists for use in producing the new *z*-ordered output tuple, *if necessary.*

The 'from' positions in the linked list(s) can now be initialized in a single pass through each list.

Finish setting up the slot by allocating a work buffer to hold one tuple, and the working copies of the circular lists.

(2) Given a slot for the current relation and its reshuffling, unshuffle it. Unshuffling is done in a loop which iterates once per bit now participating in the *z*-ordering. The unshuffling uses the tuple-width work buffer in the slot by first clearing every bit therein. Subsequently, for each set bit in the *z*-ordered tuple, set a bit in a byte-sized mask, and shift it by the current output bit position mod 8. For example, if the current bit is bit 3 in the third attribute which began at bit position 32 in the tuple, set bit 11 mod 8 in the mask then circular shift the bit by 35 mod 8 and OR the byte mask with the byte in the work buffer determined by dividing the output bit position by 8. Continue going around the circular list, each time decrementing the count for that attribute of the number of bits participating in the *z*-order. When the count becomes zero, this entry in the list is skipped, until all shuffled bits are unshuffled.

Now, any bits which were not involved in the *z*-order must be relocated to their correct attribute positions through a straightforward byte copy using the 'from' and 'to' values in the linked list.

(3) Now the tuple is completely unshuffled: if the 'to' Zmap is 0, return the tuple as it now is. Otherwise begin shuffling it following the same operation as above, by going through the circular and, if necessary, linked list(s).

(4) before returning, re-initialize the working versions of the circular lists.

There is room for some refinement to z in terms of its execution speed. For clarity of code the principal variable names should be changed from their current cryptic, but meaningful, values.

Relevant Entries in header file:

```
#define ZORD   0x02
#define PZREL  0x04
```

masks used to test a relation's mode to determine if all (ZORD) or some (PZREL) of its attributes participate in *z*-ordering

```
#define ZSLOTS 6
```

the number of slots for structures

```
typedef struct zcirclist {
    long numzbits;
    long outposn;
```

Beginning with the z-ordered tuple from Figure 3.4, 320C30233070105000X, and knowing that there are two attributes per tuple, write out the tuple in binary, distinguishing alternate bits:

```
  3   2   0   C   3   0   2   3   3   0   7  ...

0011 0010 0000 1100 0011 0000 0010 0011 0011 0000 0111 ...
```

By cyclically recombining 'like' bits, one arrives at the unshuffled attribute values:

```
0101 0010  0100 0101  0100 0100 ...
    R          E          D

0100 0010  0100 0001  0100 1100 ...
    B          A          L    ...
```

Figure 3.6: Unshuffling A Tuple

```
    struct zcirclist *next;
    struct zcirclist *prev;
} ZCLIST;
```

circular list member used in shuffling/unshuffling; members contain pointers to 'previous' and 'next' members, as well as the length of the attribute in bits, numzbits, and the starting position of the attribute as a bit offset from beginning of tuple, outposn.

```
typedef struct zlinklist {
    int tounz;
    int fromz;
    struct zlinklist *next;
} ZLLIST;
```

linked list member used in shuffling/unshuffling; members contain pointer to 'next' member, the positions (measured as bit offset from start of tuple) from which the tuple is taken, fromz, and to which it is shifted, toz.

```
typedef struct Zordslot {
    REL    *zrptr;
    long   zactime;
    long   tomap;
    long   frommap;
    ZCLIST *fromz;
    ZCLIST *toz;
    ZCLIST *wkfrom,*wkto;
    ZLLIST *froml,*tol;
    char   *tbuf;
    int    fromzbits;
    int    tozbits;
    int    zrdents;
} Zslot;
```

structure of a slot member; members are: pointer to the rel entry for the relation occupying the slot, *zrptr, the time of most recent access used by the LRU code, zactime, the frommap and tomap Zmap values to guide reshuffling, pointers to the circular lists used in unshuffling, fromz and shuffling, toz, and their corresponding working copies wkfrom and wkto, pointers to the linked lists froml, and tol, a pointer to the working tuple buffer, tbuf, how many bits are now in z-order, fromzbits, and how many are to be in the output z-order, tozbits, and the number of rd entries for this relation, zrdents

Zslot zslot[ZSLOTS]; the array of slots used by z

3.9 B*-Trees

B^*trees are supported by a collection of procedures as well as a totally different file organization from that used for 'flat' relations. Tuples appear only in leaf nodes and are linked into one linked list to allow rapid sequential processing. The sizes of leaf and branch nodes can be determined by the user for a particular tree and are independent of each other. For best performance, they should be chosen as multiples of the storage device's blocksize. Similarly, the load on leaf and branch pages is user selectable, and independent for leaves and branches. That is, the amount of used space within a node is selectable, as a percentage: when a node is split, tuple distribution between the overflowed node and the newly created one will leave the overflowed node as full as has been requested. The usual fill percentage is 50.

3.9.1 File Organization

A B^*tree has three types of entries: leaf and branch nodes, and a first-block which contains control information for that tree (see typedef struct Xfirstblock, below). It records the size and percentage fill for branch and leaf nodes, the name of the relation and several addresses. In the context of B^*trees, addresses may refer to the offset from the start of the file (disk address), or, a memory location at which a node page has been stored for manipulation (memory address). The addresses kept in the first block include the disk addresses of the root node, first logical tuple in the relation, and of the current branch and leaf nodes. The first block is a fixed 512 bytes, not all of which are used. The size is chosen as being a convenient block or fragment size for rapid handling under a 4.1BSD or higher filesystem.†

3.9.2 Branch Nodes

Branch nodes are structured as shown in Figure 3.7.

The node has an 8 byte header which identifies the page as being of type branch, possibly the root (STATUS byte), an unsigned short indicating how many bytes remain free for use (SPACE), thus no node can be larger than 64 Kbytes, and a pointer to the node's predecessor (which is NULL if this is the root). It is *absolutely essential* that this header on branch nodes have a size which is a multiple of 4 since the p_0 pointer, a long integer, is concatenated to it and must be on a 32 bit boundary.

Following the node header begins a linked list of entries which is entirely contained within the node (*i.e.*, the list links only those entries in this node). Each entry has a NEXT member which is an unsigned short holding the offset from the starting location of the node to that of the next entry, or 0 for end of list. 0 is assured to be a safe terminator since no entry can start at an offset smaller than the size of the node header.

† For best results, the first-block really should be the blocksize for the device on which the file resides. Unfortunately this can be quite wasteful as only about 60 bytes are used, and some devices, *e.g.*, DEC RM03, under UNIX, have 8K block sizes.

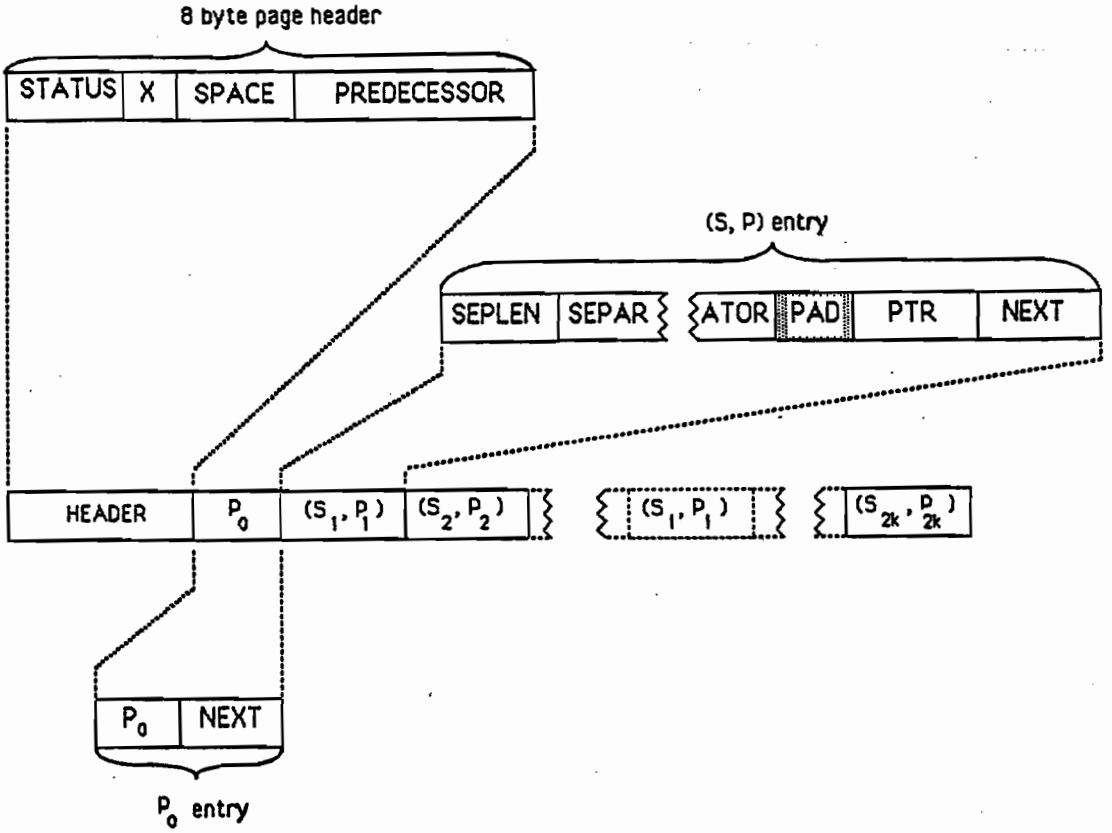


Figure 3.7: Layout of a Branch Node

The p_0 entry consists only of a long holding the disk address at which a child node begins, and the abovementioned NEXT pointer.

The (s_i, p_i) entries are somewhat more involved. Since the separator is of variable length, and z-ordered data must be accommodated, the first member (an unsigned short integer) of an (s_i, p_i) entry specifies the length *in bits* (thus maximum separator length is 8 Kbytes). The separator begins immediately after the length indicator, and is aligned to the nearest byte by padding, if necessary, with zero bits on the right. The next member of import is the pointer which, being a long, must start at a 32 bit boundary. The location at which the separator ends may or may not satisfy this requirement, thus a variable number $(0 \cdots 3)$ of alignment padding bytes is appended to the separator. By thus guaranteeing that the pointer is correctly aligned, one also guarantees that both the NEXT member of this (s_i, p_i) entry and the SEPLEN member of the physically next one will also be aligned. It has been observed that the cost in disk space of this alignment padding is acceptable when weighed against increased processing time necessitated by unaligned data (see Appendix 1).

A related alignment consideration is that the determination of how many alignment padding bytes will be necessary is made based on the disk address (within the node) mod 4. When these disk pages are loaded into memory buffers it will have to be the case that the memory

address within the buffer mod 4 has the same value. This condition is guaranteed by dynamically allocating buffers with the UNIX malloc call, which assures that the requested memory begins at a 32 bit boundary. This assumes that the disk pages always start on a 32 bit boundary, which seems assured.

Another vital property of branch nodes is that they must contain *no holes*. This is imperative since the way space on the node is allocated for new entries is by appending, starting at the node address + (nodesize - header_length - space).

A branch node grows as entries are added to its linked list by insert or split until an attempt to add another entry would require more space than is available. At that point split is called (possibly recursively) to resolve the overflow. The root of the B^*-tree will initially appear 512 bytes into the file, *i.e.*, immediately after the first-block. Following that node will be as many leaf nodes as the root can index. When the root must be split, two new branch nodes are appended to the file, one becoming the brother of the now former root, with the other becoming the new root. It is debatable whether there is a significant advantage to keeping the root at the same physical location within the file, particularly given the small number of accesses which are made to the file during a search (see Table VII, Appendix 1). Moreover, the current implementation does not attempt to enforce any particular control over where branch and leaf nodes appear physically within the file. This is justified by, as mentioned, an expected small number of access to 'pinpoint' a particular tuple, and, by the constraint of having to live within a general purpose, mult-user/multi-tasking filesystem. Were MRDSc to have direct control over its own physical I/O devices, more effort would be directed towards optimal disk layouts for B^*tree files.

3.9.3 Leaf Nodes

Leaf nodes are structured as shown in Figure 3.8.

In addition to the status, space and predecessor information found in a branch node's header, leaves have a pointer to their 'successor', *i.e.*, the node at which the linked list of all tuples in the relation continues from this node. This may be used in order to know what page to read next during sequential processing of tuples. Since the first physical entry in a leaf node need not be the first logical one (as is occurs with branch nodes), an offset to the first logical entry in the node appears in the header.

Data entries in a leaf consist of a tuple followed by a long pointer to the next logical tuple in the relation, which may or may not be on the same leaf page. As was the case with branch nodes, the pointer must be assured of being aligned on a 32 bit boundary; the way this is implemented is different for leaves. Since the length of each tuple is knowable (either they are of fixed length as required by this MRDSc implementation, or they are variable and the length can be determined), and the length of the pointer is fixed, the starting location of the tuple within the leaf is chosen so that the pointer, which begins immediately after the tuple, will begin on a 32 bit boundary. While this introduces 'holes' in the leaves, since the entries are linked by pointers, the holes are transparent. This alignment scheme is considerably simpler than that used on branch nodes and should perform slightly better. Future implementations should retrofit it to branch nodes.

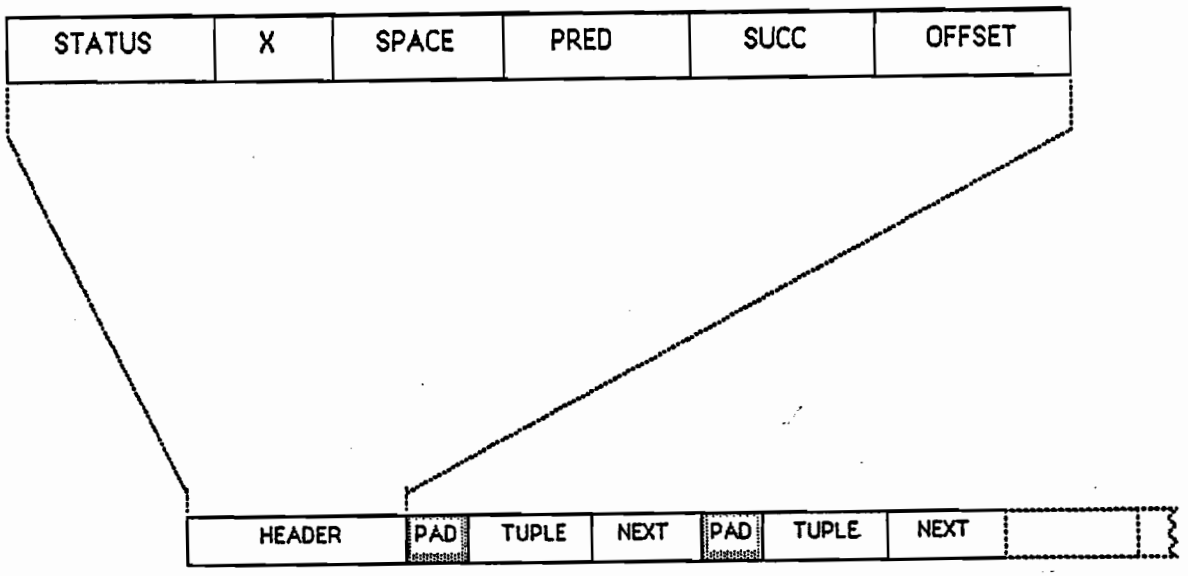


Figure 3.8: Layout of a Leaf Node

3.9.4 Procedures

Manipulation of relations in B^*trees is as different from that of flat relations as are their respective file organizations. The positioning of the software components within MRDSc attempts to make these differences transparent to components higher in the system.

For any component of MRDSc wanting a tuple from a B^*tree organized relation, the standard getuple will provide the tuple currently indicated by the rindx member of the relation's rel entry. Repeated calls to getuple provide logically sequential tuples.

To get a particular tuple, the procedure search is used. search uses the best strategy it believes can be applied to the relation in question, and for B^*trees will set-up several dynamically allocated structures needed for a search through the index. As with the structures required to perform z-order reshuffling, these structures are allocated to an available entry in a pool of slots, using LRU replacement. The slot holds the first-block for the B^*tree, and the memory regions to be used as buffers for branch and leaf nodes (1 each) are allocated and assigned to the slot. Following this set up work, references to the B^*tree occur via pointers to its rel and Xfb entries. This overhead for a first access on a B^*tree is slight compared to that involved in

setting up Zslot entries since there are no complex structures to generate and initialize. search returns the disk address of the sought tuple, if it exists, and this can be used to set rindx for a subsequent call to getuple.

Adding tuples to a B^*tree organized relation is performed by insert, which uses search to locate the insertion point; insert works only on leaves. Its operation is straightforward, involving the linking in to the list of tuples the new entry and updating the SPACE entry in the modified node's header. The node targeted for the insertion is loaded into the leaf buffer allocated in the Xfb slot for the relation using loadpage and is rewritten by flushpage after modification. UNIX disk write-behind caching ensures that insertions do not become bogged down needlessly waiting for updated disk writes.

If insert discovers there is insufficient space on the leaf, it calls split to resolve the space shortage *and* complete the insertion, *i.e.*, the entire insertion becomes split's responsibility. split, in the current implementation, is not as clever as it might be, in that it *always* performs a page split. split allocates three memory buffers the size of the node it has been asked to split. Two will become the buffers of the new sibling pages: one 'brand new', the other a new version of the node now being split. Apart from ease of implementation, this arrangement offers the security advantage that should something go wrong within split, the original page's contents remain undisturbed, so that split can return FAIL to its caller without corrupting the B^*tree.†

The entry which causes the overflow is linked into the list of the overflowed page by setting the 'pointer to next' of its logical predecessor in the list to a "magic" value which is understood by split to mean that the next entry in the list appears in a dynamically allocated structure, and not in the memory buffer containing the node. split then begins generating the buffers for the new siblings by copying the list from the overflowed node. Upon encountering the magic pointer, the new entry is written into the node buffer. When the replacement node buffer reaches its prescribed fill percentage, copying of the linked list continues in the node buffer corresponding to the newly generated node, until the list is exhausted.

At the point of switchover from one node buffer to another, split must ensure that an (s_i, p_i) entry is available for later insertion into the split node's parent. If a leaf page is being split, split must generate a new separator, for which it calls mksep. Otherwise, a branch node is being split, and the (s_i, p_i) entry is generated by taking the separator from what will become the first entry of the new branch node (the corresponding pointer becomes the p_0 pointer) and placing it, with the disk address of the new branch node (at this point unknown), in the parent. The disk address will be determined after the linked list has been completely copied and the node buffers are ready to be written.

The node buffers are written out, one replacing the formerly overflowed node, the other occupying hitherto unused disk space in the file. Now the parent node must be updated. If the root has just been split, then another branch-node-sized buffer is set up and the relevant header, p_0, and (s_i, p_i) entries are installed and this buffer written out, becoming the new root. Otherwise, upd parent is called to insert the new (s_i, p_i) entry into the parent node. This insertion will fail if the parent page cannot accommodate the new entry, and split will be called to resolve this insertion. This recursion can continue to the point of splitting the root of the tree.

Upon eventual return from a split, the caller can then itself return knowing the insertion has been completed.

---

† There is still one situation where a failure in split might corrupt the tree; see source listing for details. This case should be easy to correct.

There are some lower level routines used by search, insert, and split which are part of MRDSc B^*tree support.

cmpseptup compares a separator in a branch node with a given tuple and returns a value indicating the separator is greater than, less than or equal to the tuple. It uses a pool of slots holding information about the relation being worked on, *e.g.*, number of rd entries, domain type and length of each attribute. These vital statistics are easy to obtain at the expense of procedure calls; if a relation is to be worked on extensively it will be faster to make a 'local' copy of these particulars. Unlike the *z*-order and Xfb slots, these slots are not essential to its operation, but simply save time.

cmpseptup continues its comparison until a determination has been made which occurs at the first difference or when the separator is exhausted implying "equality". Equality of separator and tuple has to be understood in a restricted context: for as much tuple as appears in a separator, there was equality. Procedures which use cmpseptup must so understand what it reports. cmpseptup can accept both *z*-ordered (completely or partially), or plain tuples and separators.

When splitting a leaf, an (s_i, p_i) entry must be generated for the leafs' parent. mksep is responsible for making the separator. In the current implementation, only *z*-ordered tuples are guaranteed minimal length separators; otherwise the separator will include the fewest complete attributes (except strings) needed to differentiate entries. That is, if an attribute is not a string, it will either appear in its entirety, or not at all, in the separator. For example, in a tuple of two attributes (a floating point value followed by a string), mksep will produce a separator in which all of the floating point value appears, followed by (if necessary) as much of the string as is needed to differentiate the two entries. For a *z*-ordered tuple, only as many bits (padded on the right with zero bits to the next byte boundary) as are required for differentiation appear in the separator. If a tuple is partially *z*-ordered, and two entries cannot be differentiated by inclusion of all *z*-ordered bits, then the remaining non-*z*-ordered attributes are added one by one until an effective separator is generated.

findbro was written for split to use in order to locate a node's sibling(s). split could then examine them to determine if a split was avoidable. While the current implementation does not support split avoidance, findbro remains for use when this feature is implemented. Given pointers to the rel and Xfb entries for the relation, and a pointer to the node whose siblings are sought and that node's mode (leaf, branch, root), findbro will return in two longs the disk addresses of the nodes, or zero for non-existent siblings.

sepentlen determines the complete separator length of an *s* entry in an (s_i, p_i) pair. That is, it returns the length of the actual separator, plus any alignment pad bytes, plus the length of the unsigned short integer which precedes the separator. It is used when one knows the address at which an (s_i, p_i) entry begins, and wants the offset thence to the p_i pointer within that entry.

upd_parent is used by split following a successful split and redistribution of entries to insert a new (s_i, p_i) entry in the parent node. The parent node is updated by loading into a local buffer the parent of the page which has just been split. The entry for the split page is then sought by a linear search along the linked list in the node and when found, the (s_i, p_i) entry for the child node's new brother is linked into the list. If there is not enough space for the new (s_i, p_i) entry, upd_parent rearranges some buffered pages (notably the branch page buffer in the Xfb slot is copied then overwritten with the parent page which has now overflowed) thereby 'faking' parameters for a recursive call to split. Following a successful split, the page originally being split must be reloaded from disk to get its parent (which may have changed). That parent page, whether the same or different, is reloaded and upd_parent iterates. Should it still

find insufficient space for the needed insertion it gives up, rather than enter a loop; such a situation is indicative of a problem. After successful updating of the parent page, that page is rewritten and upd_parent returns.

Not directly used by the B^*tree supporting routines, but part of the overall B^*tree support is a procedure which converts a flat relation into a tree. mkindex receives a pointer to the target relation's rel entry and parameters specifying sizes and fill percentages for branch and leaf nodes. From these, in a temporary file, it creates the B^*tree to contain the relation. Firstly, mkindex builds the appropriate first block entries, then root and leaf pages. These pages are written out, and then a loop performing rdtuples (to keep the tuples in z-order if they already are) and inserts is entered. When the loop exits after all tuples have been read, the original file containing the flat relation is replaced using replace (see section 3.6.4), and the rel entry updated to reflect the new organization of the relation.

Relevant Entries in header file:

#define ZORD 0x02 #define PZREL 0x04 #define BTREE 0x08	masks used to test a relation's mode to determine if all (ZORD) or some (PZREL) of its attributes participate in z-ordering
#define XFBSLOTS 6 #define SEPSLOTS 6	the number of slots for structures
#define BTEMP "BXXXXXX"	template string for use in generating temporary file name using UNIX mktemp for use in mkindex
#define XLEAF 0x01 #define XFRSTLF 0x04 #define XROOT 0x02 #define XBRANCH 0x00	masks used to test a node's status to identify it as a leaf (XLEAF) or a branch (XBRANCH); if a leaf, is it the first logical leaf (XFRSTLF) or if a branch, is it the root (XROOT)
#define XHDRLEN 8 #define XLFHDRLEN 14	express the size of the node header structures for branch (XHDRLEN) and leaf (XLFHDRLEN) nodes. These should be done away with and where now referenced in code be replaced with appropriate sizeof()s. Note that XHDRLEN *must* be a multiple of 4.
#define XMASK 00600	file creation mask used when B^*tree files are created
#define XPGMINFIL 5 #define XPGMAXFIL 99 #define XPGDFLTFIL 60 #define LPGMINFIL 5 #define LPGMAXFIL 99 #define LPGDFLTFIL 85 #define LSPLITFILL 10 #define XPGMINSIZE 512	minimum, maximum allowed and default fill factors for branch (XPG...) and leaf (LPG...) nodes

```
#define XPGMAXSIZE 8192
#define XPGDFLTSIZE 4096
#define LPGMINSIZE 512
#define LPGMAXSIZE 32768
#define LPGDFLTSIZE XPGDFLTSIZE
```

minimum, maximum allowed and default sizes of branch (XPG...) and leaf (LPG...) nodes

```
#define XFBSIZE 512
```

size of the first block of the B^*tree which holds characteristic and status information for each tree (see Xfirstblk below)

```
#define X_SPCOFF 2
#define X_PREDOFF 4
```

offsets beyond start address of a branch node at which the unsigned short indicating space remaining on the page (X_SPCOFF) and the long pointing to this node's predecessor (X_PREDOFF) begin

```
#define L_SUCCOFF 8
#define L_FLEOFF 12
```

offsets beyond the start address of a leaf node at which the unsigned short indicating space remaining on the page (L_SPCOFF), the long pointers to this node's predecessor (L_PREDOFF) and successor (L_SUCCOFF) and the unsigned short with the offset into the page where the *f*irst *l*ogical *e*ntry appears (L_FLEOFF)

```
#define MAGICLINK 0xffff
```

used to signal split, when copying the linked list from the overflowed node to the new node buffers, that the next entry in the linked list comes from a dynamically allocated structure holding the entry to be inserted, and not from the node buffer currently being read

```
#define DT_STRING  0
#define DT_INT     1
#define DT_CHAR    2
#define DT_FLOAT   3
#define DT_SHORT   4
#define DT_LONG    5
```

datatype indicators used in cmpseptup and mksep to identify which comparison to be used for an attribute compare; see section 3.2.1

```
typedef struct Xfirstblk {
    long rootpos;
    long first_tpl;
    long xpgat;
    long lpgat;
    char *xbufat;
    char *lbufat;
    long xactime;
    int xpgfill;
    int lpgfill;
    int xpgsize;
    int lpgsize;
    char rname[MAXRNMLEN];
```

```
    char filler[XFBSIZE - (44+MAXRNMLEN)];
} Xfb;
```

first block structure of a B^*tree holding characteristics of the tree. These are: the disk addresses at which the following begin: root node, rootpos, first logical tuple in the relation, first tpl, most recently examined branch and leaf nodes, xpgat and lpgat; the memory addresses of buffers for leaf and branch nodes, lbufat and xbufat; time of last access, xactime, fill factors for branch and leaf nodes, xpgfill and lpgfill, sizes of branch and leaf nodes, xpgsize and lpgsize, and the name of the relation stored in this tree, relname. The final member of the structure is an unused array of characters used to 'round out' the size of the structure to force it to be of exactly XFBSIZE2.

```
Xfb xfb[XFBSLOTS];
```

statically allocated pool of slots used by B^*tree handling routines

```
typedef struct xbranchpghdr {
    char *status;
    char *spare;
    u_short *space;
    long *pred;
} Xpghdr;
```

structure of the header of a branch page, containing the status of the node, status, an unused byte, spare, needed to assure alignment of the subsequent short integer, space which reports the number of bytes remaining on the page, and the pointer to this node's immediate predecessor, pred.

WARNING: the size of Xpghdr typedef *must* guarantee that an instance of the type ends on a 32-bit boundary since the p_0 ptr appears immediately following the hdr on a branch page.

Note that the defined constant XHDRLEN must be equal to sizeof(Xpghdr).

```
typedef struct leafpghdr {
    char *status;
    char *spare;
    u_short *space;
    long *pred;
    long *succ;
    u_short *offset;
} Xlfhdr;
```

structure of the header of a leaf page, containing the status of the node, status, an unused byte, spare, needed to assure alignment of the subsequent short integer, space which reports the number of bytes remaining on the page, and the pointers to this node's immediate predecessor, pred and successor, succ, and the offset into the page at which the first logical entry starts, offset.

Note that the defined constant XHDRLEN must be equal to sizeof(Xlfhdr).

```
typedef struct SepSlot {
    REL *rptr;
    long actime;
```

```
    int rdentries;
    int Zdomlen;
    short domtype[MAXATTS];
    ushort domlen[MAXATTS];
} Sepslot;
```

slot structure used by search containing a pointer to the rel entry for the relation being searched, rptr, the time this slot was last referenced, actime, total length of domains now in *z*-order (in bytes), Zdomlen, the data type of each domain, domtype, and the length of each attribute, domlen.

```
Sepslot sepslot[SEPSLOTS];
```

statically allocated pool of slots used by cmpseptup.

```
typedef struct tree_ent {
    char *te_item;
    ushort te_len;
    ushort te_bnext;
    long te_ptr;
} Bentry;
```

type of dynamically allocated structure used in split to hold the item being inserted. It is from this structure that the new item is linked into one or the other of the 'replacement' node buffers when, during its copy of the linked list from the overflowed node split encounters a 'next' pointer value of MAGICLINK. The structure contains a pointer to the where the entry, tuple or separator, is stored, te_item, its length in *bits*, te_len, pointer to next item in linked list (used only when inserting onto a branch node), te_bnext, and the pointer to the next entry (leaves only) or the child page (branch only), te_ptr.

```
union u_short {
    short sval;
    char c1,c2;
} short_coerce;

union u_int {
    int ival;
#ifdef INT16
    char c1,c2;
#endif
#ifndef INT16
    char c1,c2,c3,c4;
} int_coerce;
#endif

union u_float {
    float fval;
    char c1,c2,c3,c4;
} flt_coerce;

union u_long {
    long lval;
    char c1,c2,c3,c4;
} long_coerce;
```

used for type coercion of unaligned tuple data in cmpseptup and mksep.

3.10 Error and Exception Handling

An *error* condition arises when, in performing some operation, an expected event fails to occur or an unexpected one occurs -- situations for which MRDSc was prepared. The error usually results in the termination of the operation, under MRDSc control. An *exception* is an unexpected event for which MRDSc was unprepared, and may be of such a nature that MRDSc loses control of the operation, usually followed by UNIX intervention resulting in MRDSc termination.

Error conditions are of different severities:

warning — a minor error condition, which MRDSc can correct. The warning serves to notify the user that some corrective action was taken on his/her behalf in order that a requested operation continue; the action may or may not have been what the user intended.

error — the error condition, often involving an underlying UNIX system call, has caused some component of MRDSc to fail to perform what was asked of it and is not MRDSc correctable. The operation encountering the error usually fails, signalling failure its caller, but MRDSc continues operating.

severe — the error condition causes an uncorrectable failure within MRDSc of such a nature that the current line of activity (possibly involving many operations) cannot continue (*e.g.*, work on a particular database cannot continue).

fatal — the error condition causes an uncorrectable failure in so critical an area of MRDSc that no further processing is possible, and MRDSc terminates

Exceptions arise from the hardware and UNIX system software upon which MRDSc is built. They range from user generated interrupts (*e.g.*, use of the intr key on the user's terminal), to hardware events such as floating point traps, or processes dying on swap errors. Most, but not all, of these exceptions ('signals' in UNIX vernacular) are delivered to the affected process should it wish to deal with it.

Procedures within MRDSc perform a large number of self consistency checks in their operation in order to catch error conditions. If an error is detected, a centralized error handler implemented by db_err is called. db_err prints an error message, and then determines whether execution should continue. The global variable RUN is a severity weighted counter incremented with each error or severe. If RUN's value exceeds a maximum limit, MRDSc terminates.

db_err receives four parameters from its callers:

Parameter	Type	Description
error number	int	identifies both the particular error and its severity. Error numbers are used as subscripts into an array of strings to locate the appropriate error number. The error number div 25 determines the severity of the error, which, from 0 to 3, are fatal, severe, error and warning, respectively.
ersect	int	an index into a table of strings containing procedure names, used so the appropriate procedure name appears with the error message
unix err	int	in many cases the error condition originates from a failed UNIX system call; in such cases this parameter contains the global errno which reports the operating system's idea of what caused the error. The convention of passing -1 as the value of this parameter was adopted when there is no associated UNIX error.
ermsg	char[]	character string containing some additional information to appear in the error message. Its content should be directly related to the particular instance of the error since anything more general will simply repeat what is already expressed by the message selected by the error number. The string may be null.

A typical error message appearing on stderr has the form:

>> Error << procname: [ernum,unix err] error_message ermsg

In the eventuality that MRDSc terminates, the last procedure entered, often called by db_err to mediate the termination, is abend. abend distinguishes values of its abend code parameter, abcode, always less than zero, to determine if the call represents an abnormal end or a controlled termination (as opposed to a normal MRDSc exit). Like db_err, the abend code acts as an index into a message table. The appropriate termination message, accompanied by an 'obituary' string parameter are printed, and abend performs an exit(abcode).

At present, db_err errors are not written to the LOGFILE whereas abend messages are.

MRDSc has no implemented exception handling code (except in cmd; see chapter 5). The proper installation of such code is not a trivial exercise as many signal handlers would need to be tailored to specific MRDSc procedures, and one signal type may require several different handlers. Originally it was intended that exceptions would be dealt with by a centralized facility as are errors. It remains true that some measure of control over exceptions could be installed centrally (*e.g.*, for SIGSEGV and SIGBUS), but no simple central handler could provide a flexible enough mechanism for recovery from all exception types. For example, a divide by zero

exception in printrel should be dealt with locally by a handler within printrel where it is a situation from which recovery is easy. The same exception arising in z or split involves a much more difficult recovery, nor would the technique used for recovery in printrel be appropriate here. Thus while a centralized handler could provide minimal insurance, it could not be made flexible enough to give MRDSc an acceptable level of recovery capability. For a production quality version of MRDSc this significant shortcoming would have to be remedied.

Relevant Entries in header file:

#define ABEND 1 #define ABORT 2 #define ADTUPLE 3 . . . #define WRTUPLE 43 #define YESNO 44 #define Z_ORD 45	names for error section numbers to be passed to db_err, which are used to index its table of procedure names; see source code for complete list
#define MAXERR 50	default maximum value of RUN; if RUN > MAXERR, db_err calls abend to terminate execution
static int RUN = 0;	global severity weighted error count, initially 0

3.11 Implemented Relational Operations

The current MRDSc implementation supports very few user callable relational operators: only the monadic select, project, and printrel exist.

printrel is a simple minded routine to display a relation on paper. It performs very rudimentary formatting, generating columns for attributes and printing on each page a caller supplied title, page number and attribute column headings. Attribute values are printed left justified within these columns. Tuples which are wider than the maximum line length are truncated. A companion procedure, outrel, is identical except that it is intended for output directed to a crt terminal, thus has different ideas about line and page lengths. outrel stops after every 'page' (twenty lines, a value hardcoded into it, not, as should be the case, a value obtained from the TERMCAP environment variable) and waits for a carriage return before continuing. The procedure was implemented for use with cmd. A more satisfying solution would be to dispose of printrel altogether and adopt outrel to direct its output to either the terminal via some paging mechanism, *e.g.*, /usr/ucb/more, a line printer via lpr, or to a user specified file. printrel will produce 'normal' appearing output without regard to the datatype of an attribute or the data organization of the relation since it acquires tuples from the operand relation through getuple.

Relevant Entries in header file:

`#define LINELEN 130`	length of line from line printer
`#define PGNUMLEN 5`	length of string holding page number string
`#define PGNUMHDR "Page "`	string appearing in page number message atop each page
`#define PAGELINES 55`	maximum number printed lines per page
`#define PGNUMFMT "%-4d"`	print format to use when printing page numbers
`static char *DATAFMT[] = { "%s", /* char strings */ "%d", /* integer, free format */ "%c", /* single char */ "%f", /* single precision float */ "%d", /* short integer, free fmt */ "%ld" /* long int, free fmt */ };`	formats to use in printing attributes of different data types

project may be used to make new relations which are an improper attribute subset of the input relation. Following checks that the operand relation exists, that all its requested domains can be found, and that all needed rd entries exist, project allocates a work buffer of the size required to hold the tuples it will be generating.

Upon successful return from mkrel to make the output relation, tuples are read by getuple from the input relation, requested attributes excised, positioned as directed by the user (by the sequence of domain names in the parameter domlist), and written out with adtuple. This loop continues, independent of possible failures in adtuple, until all tuples in the relation being projected have been read.

When all input tuples have been read, project calls sortrel to sort the projection and eliminate duplicates, and then returns to its caller.

The newly created relation inherits from the one being projected its Zmap and maxsize characteristics.

Relevant Entries in header file:

`int rdents;`	global integer holding number of entries in rd for current relation, set as side effect of calling findrd

Other related header entries may be found in the sections covering the procedures used by project.

select produces an output relation, created with mkrel, containing all tuples from the input relation which satisfy a user specified constraint or selector. The current implementation of select

accepts only one selector, thus a query involving a conjunction of multiple constraints will require multiple calls to select.

A selector is of the form:

domain cmp value

where *cmp* is an understood comparison operator (some meaningful combination of '<', '=', and '>'), *value* is a constant [as opposed to a an expression which yields a] value of an attribute in *domain*. select confirms that it understands the selector, although it does not posses the concept of type checking, and isolates the indicated domain.

select will attempt to use the fastest strategy that can be applied to the input relation. For example, if the domain in the selector is the primary key of an ordered relation, select will call search to locate the point in the input relation from which sequential processing should begin and/or end. For a *z*-ordered relation, a bit mask representing the attributes and value would be generated and used to restrict the amount of the relation which must be examined.

The current implementation does not support strategy selection, but allows for its introduction into the next version. If nothing better can be used, select performs a purely sequential read on the input relation using getuple. Each tuple is subjected to a comparison of the attribute in the selector domain to the value appearing in the selector. The comparison generates an intermediate result of less than, equal to or greater than zero; its meaning in the context of the selector's operator is then established.

For example, if the selector was "colour < red" and the colour attribute in the current tuple were green, then the intermediate result would be -1, which in the context of a 'less than' constraint, directs select to call adtuple since the current tuple satisfies the selection criterion.

Once all tuples from the input relation have been read, select calls sortrel to sort the newly created relation and remove any duplicates; it then returns. The new relation inherits its Zmap and maxsize characteristics from the relation upon which the select is performed.

Relevant Entries in header file:

```
#define CMP_OPS   3
```
number of recognized comparison operators used in select

```
#define CMPATT_EQ  1
#define CMPATT_GT  2
#define CMPATT_LT  4
```
names for recognized compare attribute operators used in switch statement in select

```
#define CMPVALID   7
```
mask used to test for valid compare operators.

```
static int CMPATT[] = {
   CMPATT_EQ,
   CMPATT_GT,
   CMPATT_LT };
```
array of known attribute operators

```
union u_short {
    short sval;
    char c1,c2;
} short_coerce;

union u_int {
    int ival;
#ifdef INT16
    char c1,c2;
#endif
#ifndef INT16
    char c1,c2,c3,c4;
} int_coerce;
#endif

union u_float {
   float fval;
   char c1,c2,c3,c4;
} flt_coerce;

union u_long {
   long lval;
   char c1,c2,c3,c4;
} long_coerce;
```

used for type coercion of unaligned tuple data in attribute comparisons with selector value

3.12 Useful Tools

A small set of useful procedures was developed during the course of MRDSc implementation which serve as 'helpful' auxiliary programs -- none is part of the MRDSc system.

cmd is a simple interactive command interpreter which is a *test tool only.* It is not, nor is it intended to be, the basis for a general purpose user interface. It allows a user to select a database and, following successful completion of setup, to choose from a small set of commands to perform, *e.g.*, projects or selects. Complete directions on its use are found in chapter 5.

cmd's structure is a simple loop of getting and executing user commands. The loop is broken only when the user chooses the 'exit' command, or, after generating a SIGINT or SIGQUIT from the terminal, replies negatively to cmd's query "resume?".

The command loop issues a prompt then reads a line from standard input. The string which begins that line, up to the first space or tab, is assumed to be a command name. If recognized, a routine which drives that specific command is invoked and passed the entire command line.

The driver routines perform very simple minded parsing of command line arguments, and use the result to generate a call to the MRDSc procedure which implements the user requested command.

cmd is completely open-ended, and while it could be turned into a 'real' user interface with modest effort, its *raison d'ètre* is to assist MRDSc system developers test new code, without being ambitious.

mkdb is an interactive program which facilitates the construction of relational databases. It assumes the existence of a plain text file for each relation in the intended database. This allows the data which is to become a relation to be typed in with an editor, or to be generated by some program. The files must have the property that, as with fields in /etc/passwd, attribute strings are separated by a colon (:). The interactive part of mkdb is required first to solicit from the user the name and home directory for the intended database. It then asks for relation names until the user provides a null name. For each relation, the maximum number of tuples must also be provided. Within each relation, mkdb builds a 'tuple template' by asking for domain name, type and length of attribute. A null domain name signals the end of a tuple, hence of input for the current relation.

When a null relation name, signalling 'no more relations in the database', has been received, mkdb inquires for each user relation the name of the text file to use to generate the relation. When all relations have thus been generated, mkdb updates the system relations and rewrites them. The database has now been created, and following its addition to the MRDSc master database list (which must be done separately by hand), is ready for use.

sizeof is useful when porting MRDSc to a new environment; it reports not only the sizes of standard primitive data types (int, float, *etc*), but also of all structures declared in the header mrds.h.

Chapter 4

MRDSc Programmer's Manual

General Notes

This chapter provides reference information on the use of routines provided in the library archive mrds.a for both user and systems level programmers.

Procedures in MRDSc are generally either of type int or "pointer to something". In cases where the routine acts as a procedure, *i.e.*, its return value is irrelevant to what it does, the return value will be SUCCESS (0) for complete success in performing its task.

Failure of int routines is signaled by negative return values (*i.e.*, < SUCCESS) with FAIL (-1) being the commonest. Some routines return specific negative integers to signal particular failures: these are described in the entry for the particular routine. Pointer valued routines return (type *)(NULL) to indicate failure.

In the following descriptions of MRDSc routines, the 'Action Taken' for each 'Error Condition' described distinguishes immediate from delayed returns caused by errors. Upon encountering an error condition which precludes further activity in the routine, the usual course of action is (possibly) to issue an error message via db_err, and then to return FAIL immediately. For less severe error conditions, an error message may still be issued, but execution continues in the routine. At some later point a return value different from FAIL but indicative of unsuccessful performance is returned.

The 'Call Table' entry for each routine lists alphabetically all external routines called by a particular routine, and identifies each as an MRDSc routine or a UNIX system/library call with a single letter, 'M' or 'U' respectively, appearing beside the name.

User callable procedures differ from system level ones in that relational entities are referenced by *name* only; the latter use pointers to structures related to the entities.

Relation and domain names *must* be of exactly the correct length and always terminated by a null byte. Under the current implementation, relation and domain names are the same fixed length, 11 bytes (10 characters plus 1 terminating null byte). In the programming examples shown, the appearance of the character '|' in a string indicates that a blank must appear in that position. Thus the string "relname|||" is a valid relation name since it is of length 10 bytes, a length reached by adding three trailing blanks.

Below appears a cross reference of the procedures described in this chapter with relevent section numbers from Chapter 3 and/or other entries in this chapter.

Procedure Name	Related Entries
abend	3.10
adtuple	3.2.2, 3.6.3, 3.9.1, insert, wrtuple
closerel	3.2.2,3.6.2
cmpseptup	3.9

A programmer would follow these steps in order to use the MRDSc routines from a C program:

(1) include in the C program the header file <mrds.h>,

(2) declare the MRDSc procedures in the invoking program according to the type of the procedure,

(3) compile the C program using a command line resembling:

cc myprog.c -D*CPU* -D*UNIX* -o myprog -l*mrdsc*

The CPU define identifies the type of processor for which object code is being generated: this is important in determining the size of some data structures at run time. The UNIX define identifies the version of the operating system under which the load module will be run: this affects some of the system calls and include files used in generating executable code.

While the example above is for a C program, any language which can co-exist with the C preprocessor should have little difficulty calling MRDSc procedures.

Everything shown above assumes that the library and header file have been properly installed -- see Appendix 2.

Find below a sample C program which performs some work on a previously established database named 'plas'.

```
#include "mrds.h"
main ()
{
        register int i;
        int find_db(),select();
        DBSTATUS mydb;

        printf("Regular C program finding db \"plas\"...");
        i = setup("plas",&mydb,1);
        printf("setup returned %d\n",i);
        if (i == FAIL) exit(-1);
        printf("Let us be really brave and make black parts...");
        select("partcol   ","colour    ","Black         ",CMPATT_EQ,"blackparts")
        printf("finished select\n");
        printf("Regular C program calls  syncrel on \"plas\"...\n");
        syncrel();
}
```

```
int abend(abcode,obit)

int abcode;
char *obit;
```

DESCRIPTION

Abend is called to exit MRDSc when, as the result of an irrecoverable error or exception, further processing is not possible.

Abend prints a message on stderr indicating an ABEND or TERMINATION, followed by the negative of abcode, and the obituary string, if one is provided. After this it writes an entry to the log file indicating an event of ABEND or TERM, a message chosen from an array of abend/termination messages indexed by -abcode, and then closes the logfile. Finally, it performs an exit(abcode).

abcode should always be negative, and if less than ABENDCODE (-50) implies a genuinely abnormal end of MRDSc execution. If greater, abend was called in response to a request to terminate execution prematurely (*e.g.*, in response to a user SIGINT).

EXAMPLE:

```
abend(-20,"example use");
```

results in the message:

```
MRDSc TERMINATED 20
```

on standard error, a timestamped TERM message appearing in the logfile, and an exit(-20) from MRDSc

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

does not return; exits with value of abcode.

CALL TABLE

```
exit    U
fclose  U
fflush  U
fprintf U
logent  M
```

SEE ALSO

Section 3.10

```
int adtuple(rel,tpl,x)
REL *rel;      /* rel being appended to */
char *tpl;     /* tuple to be added     */
Xfb *x;        /* Xfb ptr if BTREE      */
```

DESCRIPTION

Adtuple is used to write a tuple into a relation. The tuple is written at the position in the relation indicated by the relation's write index (rel -> windx) which must be set before calling adtuple. Adding tuples to read only relations or overwriting a tuple in an append only relation is an error. The tuple to be added is pointed at by tpl, and will be converted automatically to the data organization appropriate for the operand relation, *e.g.*, if adtuple is adding to a *z*-ordered relation, the tuple received by adtuple will be shuffled as necessary. Input tuples are assumed to be 'flat' *i.e.*, not in *z*-order, and not in the form of a leaf entry. Adtuple is intended for used directly with the code implementing the user callable procedures, *e.g.*, project uses it to build its output relation.

Adtuple does not support constant relations at present.

The parameter x points to the Xfb entry holding the tree's first-block and is required if the data organization of the relation is BTREE; it can be ignored otherwise. The Xfb entry for a relation is assigned only in search() which must therefore have been called at least once before attempting to add to the B^*-tree relation †.

EXAMPLE:

```
rptr = openrel("example|||",READ);
rptr -> windx = rptr -> cursize;
                .
                .
                .
adtuple(rptr,tuple);
```

appends the tuple pointed at by tuple to the relation example; since the relation is not organized as a B^*-tree the third parameter can be ignored.

ERROR CONDITION	ACTION TAKEN
relation pointer rel is NULL	error 58;return NULL
tuple pointer tpl is NULL	return NULL

† Future versions should assign Xfb slots to each B^*-tree relation upon opening them.

target relation is a constant relation	warn 83; return NULL
relation is z-ordered and attempt to shuffle tuple as required has failed	return NULL
relation is B^*-tree and attempt to insert tuple has failed	return NULL
actual writing of tuple has failed (wrtuple failed).	return NULL

RETURN VALUE(S)

NULL	failure to add tuple
$n > 0$	number of bytes successfully written

CALL TABLE

db_err	M
insert	M
wrtuple	M
z	M

SEE ALSO

Sections 3.2.2, 3.6.3, 3.9.1; wrtuple, insert

```
int closerel(rptr)
REL *rptr;
```

DESCRIPTION

Closerel is used to close the relation holding a now opened relation, mark the fd member of the relation's rel system relation entry invalid, and release dynamically allocated buffer space associated with the relation. It is intended for use by outer level system and user called routines, *i.e* printrel uses it close the relation when it has finished printing it.

EXAMPLE:

```
rptr = openrel("example|||",READ);
                .
                .
                .
closerel(rptr);
```

closes the relation example previously opened for processing.

ERROR CONDITION	ACTION TAKEN
could not close the file holding the relation	error 50; return FAIL

RETURN VALUE(S)

FAIL	if UNIX close call fails to close relation's file
ofd	for successful closure, return the UNIX file descriptor which was used to reference the file containing the relation.

CALL TABLE

close U
db_err M
free U

SEE ALSO

Sections 3.2.2, 3.6.2

```
int cmpseptup(rel,sep,tup)
REL *rel;
char *sep;
char *tup;
```

DESCRIPTION

Compare a tuple pointed at by tup with a separator pointed at by sep in the B^*-tree organized relation pointed at by rel. It returns < -1, 0, or > 0 reflecting a less than, equal to, or greater than relationship between the separator and the tuple. The procedure is use during searches of a tree to select which child pointer to follow from the current node, and is intended for used with search and similar lower level internals.

cmpseptup compares the tuple, attribute by attribute, with the separator until the separator is exhausted (implying 'equality') or until a distinction is made.

EXAMPLE:

For the tuple:

ball	red

pointed at by tup, in the relation whose entry in rel is pointed at by relptr, with sep pointing at 'b', the call

```
cmpseptup(relptr,sep,tup)
```

returns < -1 ('b' < "ball").

ERROR CONDITION	ACTION TAKEN
rel pointer is null	return FAIL
bad separator length: separator length was negative, or, for a non z-ordered relation was not a multiple of 8 (byte aligned)	error 66; return FAIL
beyond end of separator but still have not determined <, =, or >	warn 90; return 0 (separator presumed 'equal' to tuple)
have gone beyond end of comparison code, but failed to make determination. (*i.e* have reached a point which should be unreachable)	error 65; return FAIL

RETURN VALUE(S)

> 0	separator greater than tuple
= 0	separator 'equals' tuple
= -1	failure
< -1	separator less than tuple

CALL TABLE

db_err	M
finddom	M
findrd	M
sprintf	U
timer	M

SEE ALSO

Section 3.9

```
int db_err(ernum,ersect,unix_err,erstring)
int ernum,ersect,unix_err;
char *erstring;
```

DESCRIPTION

db_err is used to notify a user of a detected error; it writes a message on standard error and adjusts the value of RUN depending on the severity of the error. If RUN has exceeded the ceiling established for this run, or the error was fatal, db_err performs as controlled an abend as is possible, exiting through abend.

The error message printed is of the form:

>> severity << procname: [ernum,unix_err] ermsg erstring

where:

severity	reflects the class of error, one of: fatal, severe, error or warning
procname	is the name of the MRDSc procedure from which this error originates
ernum, unix_err	are the values of the corresponding parameters passed
ermsg	is an error message identifying the error; it is selected from an array of such messages using the ernum parameter as a subscript
erstring	is an optional string passed by the caller; if present, it should provide specific information pertinent to this instance of the error, since ermsg will contain a general purpose message about the type of error

The value of ernum identifies both the error *and* its severity, according to the quotient of an integer divide of ernum by 25:

ernum/25	severity	range of ernums
0	fatal	0..24
1	severe	25..49
2	error	50..74
3	warning	75..100

Should more than 25 messages of a class be needed, add 100, *e.g.*, the 26^{th} 'error' number would be 150.

Error handlers wishing to add entries to the log file must do so separately as db_err does

not report errors there.

Symbolic names for procedures for use with ersect are in the file db_err.c.

The convention of passing -1 as the value of unix_err in situations where the detected error does not arise from failure of a UNIX system call is obeyed by all MRDSc procedures.

EXAMPLE:

If an attempt to open a relation named *rname has failed in printrel one might issue:

```
db_err(51,PRINTREL,errno,rname)
```

which prints on standard error:

```
>> Error << printrel: [51,9] Cannot open rname
```

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

none

CALL TABLE

fprintf U

SEE ALSO

Section 3.10

```
int dbck(ptr,report)
DBSTATUS *ptr;
int report;
```

DESCRIPTION

dbck checks the database indicated by ptr for inconsistencies and reports anything amiss. It also updates the MRDSc system log file at the beginning and ending of its check, the latter entry reporting success or failure of the check.

The following checks are performed:

- verify existence of directory containing the database
- verify existence of and load into memory all three system relations
- for each system relation, confirm that it has the minimum required number of entries
- verify that there are no relation (domain) names in rd which are absent from rel (dom).
- verify that all system and user relations are of the correct size as recorded in rel

Any instance of a failure in the check signals an inconsistent database which should be repaired before any further work is done.

dbck is silent about its operation and findings if the parameter report is zero; otherwise it will report its progress and findings on standard output. Any non-zero return value is indicative of problems: FAIL implies a serious problem, a positive value is the number of inconsistencies found between/within system and user relations.

EXAMPLE:

To check on a database named fred: (the required pointer parameter is set by find_db)

```
.
.
.
find_db("fred",myuid,&dbstruct);
good_db = dbck(&dbstruct,1);
```

which, if successful, sets good_db to 0 and prints on standard output:

```
checking db 'fred'
        Checking rel
        Checking rd
End of check on database 'fred' : no errors detected
```

ERROR CONDITION	ACTION TAKEN
ptr is null, no database to check	error 74; return FAIL
directory alledged to contain database does not exist	error 25; return FAIL
directory alledged to contain database 'exists' but is a file, not a directory	error 26; return FAIL

RETURN VALUE(S)

FAIL	a serious error, usually that the database does not exist where it is supposed to (possibly not at all)
0	successful check: no errors of any kind detected
$n > 0$	the number of inconsistencies detected during the last three types of check performed (see list above)

CALL TABLE

db_err	M	loadrel	M	sprintf	U
getrelnam	M	logent	M	stat	U
loaddom	M	printf	U	strcmp	U
loadrd	M	qsort	U	strcpy	U

SEE ALSO

Sections 3.2, 3.5.4

```
long find(tpl)
char    *tpl;    /* tuple to look for */
```

DESCRIPTION

find, part of mkindex, looks through the B^*-tree representation of a relation for the tuple pointed at by tpl returning its offset from the beginning of the file if found, or the offset to the tree entry immediately after which tpl should be inserted. In doing this it is indistinguishable from search except that its implementation makes different assumptions about rel and Xfb information. These assumptions made it possible to shorten find and speed its operation. It was necessary to have find as a separate procedure since, during mkindex, there is no complete tree to which to apply search.

find is a specialized B^*-tree search procedure, not intended for use outside mkindex.

EXAMPLE:

From the code of mkindex, the loop which builds the B^*-tree from the flat organized input relation begins:

```
while(!(rdtuple(rel,0L) <= 0)) {
        i = find(buffer[rel->fd]);
```

ERROR CONDITION	ACTION TAKEN
loadpage cannot read a node from the file containing the B^*-tree	error 53; return FAIL
space reported to remain in a node is not what actually remains there	fatal 12; return FAIL

RETURN VALUE(S)

≥ 0	tuple was found; value is the offset beyond beginning of file containing the B^*-tree at which the tuple starts (suitable for use with lseek, for example [but use loadpage])
$= -1$	find fails because of some error condition

< -1	tuple was not found; if -2, then tuple should have appeared before first tuple now in relation; if less, then tuple should have appeared just after (*abs*(return value) + 3) B^*-tree file to start of tuple which is tpl's logically immediate predecessor. For example, if return value is -2, tpl's predecessor begins at offset 1.

CALL TABLE

cmpseptup	M
db_err	M
loadpage	M
tplcmp	M

SEE ALSO

Section 3.9; search, mkindex

```
int find_db(name,owner,ustruct) /* determine whether a database exists    */
char *name;          /* if so return pointer to struct holding */
int owner;              /* status of db, else return null ptr     */
DBSTATUS *ustruct; /* addrs of user structure to return entry in */
```

DESCRIPTION

Given the name of a database and the UNIX uid of its owner, find_db searches the MRDSc master list of known databases, DBLIST, to isolate the pertinent entry. DBLIST entries are matched on both database name and owner uid, thereby allowing different users to have databases of the same name and each user to have several, differently named, databases. If located, the DBSTATUS structure pointed at by ustruct is initialized to the entries from the list. If not found, all fields in ustruct are zeroed. Provided in ustruct is information for the user of the database; MRDSc works with its own separate copy of the DBLIST entry.

EXAMPLE:

To find my database named 'fred',

```
typedef struct dbsrcrd  {          /* database status structure   */
  short  dbs_owner;                 /* db owner's uid          */
  char dbs_name[MAXNAMLEN];            /* name of db             */
  char dbs_homedir[FILESTRING];       /* path to home dir of db   */
  short  dbs_ident;                /* db's ident number       */
  short dbs_dfltmode;               /* db modes to use         */
  char dbs_stat;                   /* db current status       */
}DBSTATUS fredstat;

got_db = find_db("fred",getuid(),&fredstat);
```

ERROR CONDITION	ACTION TAKEN
cannot open master list DBLIST	return FAIL
ustruct is a null pointer	return FAIL

RETURN VALUE(S)

FAIL	not found, either due to an error or because DBLIST has no matching entry (members of * ustruct are zeroed)
SUCCESS	an entry in DBLIST matches the name and owner; members of * ustruct contain information from DBLIST

CALL TABLE

```
fclose  U
fopen   U
fscanf  U
strcmp  U
strcpy  U
strlen  U
```

SEE ALSO

Sections 3.2.3, 3.5.3; getdbname

```
int findbro(rptr,pg,x,mode,lbro,rbro)
REL   *rptr;        /* ptr to REL entry for this relation */
long  pg;           /* addrs of page whose brothers are sought */
Xfb   *x;           /* Xfb ptr for this tree */
char  mode;         /* flag: leaf or branch type of page */
long  *lbro,*rbro;  /* addrs of left/right brother pages */
                    /* NULL ptr ===> no such brother    */
```

DESCRIPTION

findbro finds the left and right 'brother' nodes of a given node in a B^*-tree. It provides the disk address (offset from the start of disk file) at which the brother nodes begin, or 0 if there is no corresponding brother. The pg parameter is the disk address of the node whose siblings are sought; the node is identified as a branch or leaf by mode. The first-block of the tree is passed via pointer x; lbro, and rbro are the addresses of two long integers which will hold the sibling addresses.

findbro reads the parent of node pg and performs a linear search for pg therein. Upon locating it, it returns the starting addresses of the previous and subsequent entries in the parent (*i.e.*, pointer to left and right brothers, respectively).

EXAMPLE:

With thisrel pointing to the rel entry for the relation in question, thisnode, a leaf, holding the disk address at which the node whose siblings are sought begins, thisnode's left and right siblings are found by:

```
long thisleft,thisright;
findbro(thisrel,thisnode,thisx,XLEAF,&thisleft,&thisright);
```

ERROR CONDITION	ACTION TAKEN
cannot allocate buffer space to hold parent page	error 52; return FAIL; * lbro = * rbro = 0L
bad address for parent (currently tests for < 0, but should test for < filesize && aligned)	free buffer; return FAIL; * lbro = * rbro = 0L
cannot load parent node from disk	free buffer; return FAIL; * lbro = * rbro = 0L

RETURN VALUE(S)

FAIL	unable to determine existence of brothers because of some error condition; * lbro = * rbro = 0L
SUCCESS	succeeded in determining existence of brothers. If either of * lbro or * rbro is zero, then the node in question does not have a left or right brother; a non-zero value is the disk address of that brother node.

CALL TABLE

db_err	M
free	U
loadpage	M
malloc	U
sepentlen	M

SEE ALSO

Sections 3.6.2, 3.9

```
DOM *finddom(dname)
char *dname;
```

DESCRIPTION

finddom performs a linear search through the dom system relation for the domain name pointed at by dname and returns a pointer to the tuple in dom if the domain is found, (DOM *)(NULL) otherwise.

* dname must be a string MAXDNMLEN in length †.

EXAMPLE:

To find domain 'thisdomain':

```
DOM *hereitis;
hereitis = finddom("thisdomain");
```

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

(DOM *)(NULL)	no domain of * dname was found in dom
non - NULL	pointer to tuple in dom where domain of name * dname is found (containing name, datatype and length).

CALL TABLE

strcmp U

SEE ALSO

Section 3.2.1

† 10 bytes in current implementation

```
RD *findrd(rname,domlist,num,rdptr)
char *rname,domlist[][MAXDNMLEN]; /* was domlist[][] on masscomp */
short num;
RD *rdptr[];
```

DESCRIPTION

findrd performs a linear search through the rd system relation for occurrences of domain names in domlist within relation of name *rname, or all rd entries corresponding to the relation, depending upon whether num is non-zero (the number of domains in domlist), or zero (implying no names in domlist). For each match, a pointer to the rd tuple is added to the array rdptr. If a specific list of domains is provided, the sequence of entries in rd matches that in domlist; otherwise, rdptr entries correspond to the order of appearance of domains within a tuple of *rname. For any name in domlist which cannot be located in rd, the rdptr entry is set to (RD *)(NULL).

findrd performs its search by first establishing the first and last tuples in rd representing the relation in question, and confining searches to that range. A global integer, rdents, is set to the size of that range as a side effect of calling findrd.

EXAMPLE:

Find all rd entries for relation 'getallrds':

```
RD *rdtab[MAXATTS];
findrd("getallrds|",(char *)(0),0,rdtab);
```

To get rd entries for only attributes 'attr1' and 'attr2' in relation 'getsomerds':

```
char domnmlist[2][MAXDNMLEN];
RD *rdtab[2];
strcpy(domnmlist[0],"attr1|||||");
strcpy(domnmlist[1],"attr2|||||");
findrd("getsomerds",domnmlist,2,rdtab);
```

ERROR CONDITION	ACTION TAKEN
relation name pointer is null	return (RD *)(NULL)
relation name not found in rd	error 62; return (RD *)(NULL)
while establishing number of rd entries for current relation, quantity found exceeds MAXATTS (this generally implies corruption in rd as something will have been overwritten)	error 63; set number found to MAXATTS and continue

RETURN VALUE(S)

(RD *)(NULL)	unable to successfully match rd entries with requested domain names due to some error condition
non - NULL	always the pointer to the first tuple in rd for relation *rname, irrespective of num

CALL TABLE

db_err M
strcat U
strcmp U
strcpy U

SEE ALSO

Section 3.2.2

```
REL *findrel(rname)
char *rname;
```

DESCRIPTION

findrel performs a linear search through the rel system relation for the relation name pointed at by rname and returns a pointer to the tuple in rel if the relation is found, (REL *)(NULL) otherwise.

* rname must be a string MAXRNMLEN in length †.

EXAMPLE:

To find relation 'thisrel':

```
REL *hereitis;
hereitis = findrel("thisrel|||");
```

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

(REL *)(NULL)	no relation of * rname was found in rel
non - NULL	pointer to tuple in rel where relation of name * rname is found

CALL TABLE

strcmp U

SEE ALSO

Section 3.2.2

† 10 bytes in current implementation

```
int flushpage(from,fd,at,len)
char *from;    /* page starting addrs              */
int fd;        /* fd of file to flush to           */
long at;       /* position in file at which to write */
int len;/* length of page to write                 */
```

DESCRIPTION

flushpage performs a seek on file fd to position at, then writes len bytes from memory location from. It is used to write out a node (branch or leaf), from memory back to the file containing a B^*-tree organized relation.

EXAMPLE:

To write a leaf node now residing in memory, with treefd holding the appropriate UNIX file descriptor, and x pointing to the first-block of the B^*-tree,

```
flushpage(x -> lbufat,treefd,x -> lpgat, x -> lpgsize)
```

ERROR CONDITION	ACTION TAKEN
cannot seek on file fd to position at	error 54; return FAIL
failed to write len bytes	error 54; return FAIL

RETURN VALUE(S)

FAIL	wrote none or some of, but not all of, the node due to some error condition
$n > 0$	number of bytes written out by flushpage; can be compared with known page size for the node by caller to confirm correct writing of node

CALL TABLE

```
db_err M
lseek  U
write  U
```

SEE ALSO

Sections 3.6.2, 3.9

```
unsigned freeclist(cptr)
ZCLIST *cptr;
```

DESCRIPTION

freeclist releases all space dynamically allocated to accommodate circular list entries in a Zslot; it is used to release the old list before allocating and initializing a new one when such a slot is being re-used. The space is released entry by entry until the list is empty, and the number of free'd entries returned.

EXAMPLE:

To fully free up space now held by the list pointed at by thiscirclist:

```
freeclist(thiscirclist);
```

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

>0	number of list entries for which storage was released

CALL TABLE

free U

SEE ALSO

Section 3.8

```
unsigned freellist(lptr)
ZLLIST *lptr;
```

DESCRIPTION

freellist releases all space dynamically allocated to accommodate linked list entries in a Zslot; it is used to release the old list before allocating and initializing a new one when such a slot is being re-used. The space is released entry by entry until the list is empty, and the number of free'd entries returned.

EXAMPLE:

To fully free up space now held by the list pointed at by thislinkedlist:

```
freellist(thislinkedlist);
```

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

>0	number of list entries for which storage was released

CALL TABLE

free U

SEE ALSO

Section 3.8

```
char *getdbname()
```

DESCRIPTION

getdbname interactively solicits from a user the name of a database on which to work. It writes on standard output and reads from standard input, asking the user to type in the name of a database to use. Names which exceed MAXNAMLEN in length are truncated to that length, and the user warned (on standard error) of the truncation. The string is returned in a global character array (which other procedures also use), and thus should be copied to 'safe' storage.

EXAMPLE:

To get a database name from a user:

```
strcpy(safedbname,getdbname());
```

usually followed something akin to:

```
if (strlen(safedbname) == 0) { /* no name */
```

ERROR CONDITION	ACTION TAKEN
name too long	warn 84; truncate to acceptable length, continue

RETURN VALUE(S)

pointer to global character array into which user typed database name has been placed.

CALL TABLE

db_err M
fprintf U
getc U (used by macro READLN)
strlen U

SEE ALSO

Section 3.2.3, Chapter 5 (cmd); find_db

```
char *getrelnam(relp)
REL *relp;
```

DESCRIPTION

getrelnam looks into the tuple in rel pointed at by rptr to dig out the relation name. Any trailing blanks added for padding the name to length MAXRNMLEN-1 are removed, and the result is appended to a copy of the path to the database's home directory, thereby generating the full path to the file containing the relation. The string is returned in a global character array used exclusively by getrelnam and is overwritten with each call.

EXAMPLE:

To generate the full path to the file holding the relation whose tuple in rel is pointed to by thisrel,

```
strcpy(relfilename,getrelnam(thisrel));
```

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

pointer to global character array into which the filename string has been placed

CALL TABLE

strcat U
strcpy U

SEE ALSO

Section 3.2.2; openrel

```
char *getuple(rptr,mode)
REL *rptr;
char mode;
```

DESCRIPTION

getuple reads the 'next' tuple from the relation whose tuple in the system relation rel is pointed at by rptr and places the tuple in the pre-allocated buffer array (see *openrel*). getuple handles relations of any data organization, as identified by mode, automatically unshuffling and/or 'tree walking' as is needed. Tuples returned by getuple are of FLAT data organization and completely unshuffled, *e.g.*, suitable for display. getuple does no reading of the relation itself, but calls rdtuple for the actual read.

The 'next' tuple which will be read by getuple is that whose address (offset into the file containing the relation) is in the read index (rptr -> rindx) attribute of the tuple in the system relation rel. getuple leaves the read index set to the address of the physically next tuple in non B^*-tree relations, thus by setting (rptr -> rindx) to zero, before the first call to getuple, one can sequentially get each tuple simply by repeated calls to getuple. In B^*-tree relations, rptr -> rindx is left holding the address of the logically next tuple, thus one can similarly access tuples sequentially with repeated getuples.

EXAMPLE:

To sequentially get each tuple in a non B^*-tree relation from beginning to end:

```
rptr -> rindx = 0L;
while (getuple(nontreerelptr,FLAT)) {
        /* stay in loop until end of rel or error */
}
```

Similarly, for a B^*-tree relation, with x pointing at its first-block:

```
rptr -> rindx = x -> first_tpl;
while (getuple(nontreerelptr,FLAT)) {
        /* stay in loop until end of rel or error */
}
```

ERROR CONDITION	ACTION TAKEN
rptr pointer is NULL	error 58; return (char *)(NULL)
failure during attempt to unshuffle	return (char *)(NULL)
failure in reading the tuple (rdtuple fails)	return (char *)(NULL)

RETURN VALUE(S)

(char *)(NULL)	failed to get another tuple because of an error or end of relation has been reached
non - NULL	pointer to buffer containing tuple just 'got'

CALL TABLE

db_err	M
rdtuple	M
z	M

SEE ALSO

Sections 3.6, 3.8, 3.9; rdtuple, openrel

```
int gooduser ()
```

DESCRIPTION

gooduser determines if a current UNIX user is an authorized MRDSc user by first getting the user's effective uid then looking through the MRDSc list of known users. If found, the euid is returned; otherwise FAIL is returned.

EXAMPLE:

To check if a user is authorized to use MRDSc

```
if (gooduser < SUCCESS) { /* not allowed */
```

ERROR CONDITION	ACTION TAKEN
cannot open MRDSc user list file	error 61; return FAIL

RETURN VALUE(S)

FAIL	cannot read user list file or user is not authorized
≥ 0	user is authorized, return value is euid. Note that this may cause problems on 4.3BSD systems where negative valued user idents are valid.

CALL TABLE

```
atoi    U
db_err  M
fgets   U
fopen   U
geteuid U
index   U
```

SEE ALSO

Section 3.5.1; Appendix 2 (Installation)

```
int insert(rel,tpl,x,mode)
REL   *rel;        /* ptr to rel into which insertion occurs */
char  *tpl;        /* ptr to entry being inserted (tpl or (s,p)) */
Xfb   *x;          /* ptr to Xfb entry for this tree */
char  mode;        /* insertion of tuple or (s,p) */
```

DESCRIPTION

insert is used to add tuples to B^*-tree organized relations (thus is concerned only with leaf nodes). It searches the tree for the correct point at which to insert the tuple pointed at by tpl, and if necessary, will split the leaf node in order to accommodate the insertion.

insert writes out the updated leaf node upon completion of the insertion.

EXAMPLE:

To insert a tuple pointed at by thistuple into thisrel whose first-block structure is pointed at by xrel:

```
insert(thisrel,thistpl,xrel,BTREE);
```

ERROR CONDITION	ACTION TAKEN
not a B^*-tree	warning 97; return FAIL
search of relation fails	error 61; return FAIL
search succeeded in finding the tuple being inserted (*i.e.*, already there)	warn 91; do not insert tuple, return number of bytes added as 0
leaf node lost from memory buffer	error 60; return FAIL
split fails to split and add tuple	error 61; return FAIL
cannot write node back to file following insertion	error 60; return FAIL

RETURN VALUE(S)

FAIL	unable to insert tuple into tree because of some error condition
0	tuple to be inserted is already present in tree

> 0	number of bytes successfully written; should be same as size of leaf node

CALL TABLE

db_err	M
flushpage	M
loadpage	M
openrel	M
search	M
split	M

SEE ALSO

Sections 3.6.2, 3.9; adtuple, search, split

```
int loaddom(ptr)
DBSTATUS *ptr;
```

DESCRIPTION

loaddom is used by dbck at system startup to load tuples from the file containing the dom system relation into the array domcore to which all references to dom are directed when MRDSc is running. The file is opened while loaddom works and is closed once all tuples are loaded, remaining closed throughout MRDSc execution until shutdown (see *syncrel*). Returned is the number of tuples loaded, which is guaranteed to be not less than MINDOMS (the minimum number of domain entries needed to support an empty database containing only system relations).

EXAMPLE:

During system startup, with DBSTAUTS *thisdb initialized by find_db, the dom tuples are loaded by

```
found_doms = loaddom(thisdb);
```

where found_doms can be compared with the quantity expected.

ERROR CONDITION	ACTION TAKEN
cannot open file holding system relation dom	fatal 3; abend
number of domains found is less than the minimum needed to support the system relations	fatal 9; abend

RETURN VALUE(S)

$\geq$ MINDOMS	number of tuples loaded from dom

CALL TABLE

```
abend  M
close  U
db_err M
read   U
strcat U
strcpy U
```

SEE ALSO

Sections 3.2.1, 3.5.4,3.5.5; dbck, syncrel

```
int loadpage(file,x,from,to)
int file; /* descriptor for file                */
Xfb x;        /* hdr info for this index        */
long from;    /* position in file from which to read*/
char *to;     /* buffer pointer: put page 'to' there*/
```

DESCRIPTION

loadpage performs a seek on file to position from, then reads 'pagesize' bytes from the file into memory beginning at location to. The 'pagesize' is determined by reading the first byte at offset from into the file, the status byte for the node, which identifies it as a leaf or a branch whose sizes are indicated by x -> lpgsize and x -> xpgsize (in the first-block) respectively. It is used to read an entire node, branch or leaf, into memory from a file containing a B^*-tree organized relation. loadpage can read a node into the user specified buffer *to, or, if to is null, into whichever of x -> xbufat or x -> lbufat is appropriate for the node type.

EXAMPLE:

To read in a leaf node at position leafhere in file thisrelfd, whose first-block is pointed at by thisrelXfb into a buffer at location puthere:

```
loadpage(thisrelfd,thisrwlXfb,leafhere,puthere);
```

Note that the area set aside beginning at puthere must, in this case, be at least of size thisrelXfb -> lpgsize.

ERROR CONDITION	ACTION TAKEN
cannot seek on file to position from	error 54; return FAIL
failed to read status byte of node	error 54; return FAIL
cannot rewind to beginning of node	error 54; return FAIL
failed to read 'pagesize' bytes	error 54; return FAIL

RETURN VALUE(S)

FAIL	read none or some of, but not all of, the node due to some error condition

>0	number of bytes read from branch page
<-1	number of bytes read from leaf page

CALL TABLE

db_err M
lseek U
read U

SEE ALSO

Sections 3.6.2, 3.9; flushpage,

```
int loadrd(ptr)
DBSTATUS *ptr;
```

DESCRIPTION

loadrd is used by dbck at system startup to load tuples from the file containing the rd system relation into the array rdcore to which all references to rd are directed when MRDSc is running. The file is opened while loadrd works and is closed once all tuples are loaded, remaining closed throughout MRDSc execution until shutdown (see *syncrel*). Returned is the number of tuples loaded, which is guaranteed to be not less than MINRDS (the minimum number of rd entries needed to support an empty database containing only system relations).

EXAMPLE:

During system startup, with DBSTAUTS *thisdb initialized by find_db, the rd tuples are loaded by

```
found_rds = loadrd(thisdb);
```

where found_rds can be compared with the quantity expected.

ERROR CONDITION	ACTION TAKEN
cannot open file holding system relation rd	fatal 3; abend
number of rds found is less than the minimum needed to support the system relations	fatal 10; abend

RETURN VALUE(S)

$\geq$ MINRDS	number of tuples loaded from rd

CALL TABLE

```
abend  M
close  U
db_err M
read   U
strcat U
strcpy U
```

SEE ALSO

Sections 3.2.2, 3.5.4,3.5.5; dbck, syncrel

```
int loadrel(ptr)
DBSTATUS *ptr;
```

DESCRIPTION

loadrel is used by dbck at system startup to load tuples from the file containing the rel system relation into the array relcore to which all references to rel are directed when MRDSc is running. The file is opened while loadrel works and is closed once all tuples are loaded, remaining closed throughout MRDSc execution until shutdown (see *syncrel*). Returned is the number of tuples loaded, which is guaranteed to be not less than MINRELS (the minimum number of rel entries needed to support an empty database containing only system relations).

EXAMPLE:

During system startup, with DBSTAUTS *thisdb initialized by find_db, the rel tuples are loaded by

```
found_rels = loadrel(thisdb);
```

where found_rels can be compared with the quantity expected.

ERROR CONDITION	ACTION TAKEN
cannot open file holding system relation rel	fatal 3; abend
number of rels found is less than the minimum needed to support the system relations	fatal 10; abend

RETURN VALUE(S)

$\geq$ MINRELS	number of tuples loaded from rel

CALL TABLE

abend	M
close	U
db_err	M
read	U
strcat	U
strcpy	U

SEE ALSO

Sections 3.2.2, 3.5.4,3.5.5; dbck, syncrel

```
int logent(event,msg)
char *event, *msg;
```

DESCRIPTION

logent records the string *event with accompanying *msg in the MRDSc system log file, along with a time stamp. The logfile is selected at MRDSc generation time (see LOGFILE in mrds.h). logent is used by various MRDSc routines (including dbck and abend) to report on the progress of activity on a database; the entries are intended primarily for the use of database administrators.

The format of a logent entry is:

time: *event *msg

Note that the log file being written into *must* be open before calling logent (normally opened in setup).

EXAMPLE:

When dbck begins its consistency check on a database it calls logent to update the log to show the database entering the CHECK state. (fplog is a global FILE * pointer to the opened log file):

```
if (fplog) logent("CHECK",LOGBUF);
```

where LOGBUF is a string generated by dbck containing the database name and owner. An entry resulting from the above call appears as:

```
525389949:    CHECK:    testdb (owner 59): begin dbck
```

ERROR CONDITION	ACTION TAKEN
log file is not opened for writing	error 54; return FAIL

RETURN VALUE(S)

FAIL	log entry failed due to an error condition
SUCCESS	log entry successfully completed

CALL TABLE

```
db_err M
fprintf U
```

SEE ALSO

Section 3.5.2

```
int merge(file,core,out,rel,num)
int file;  /* fd of file containing previous merged output */
char *core; /* ptr to start of memory area */
char *out;  /* name of new output file */
REL *rel;   /* ptr to system rel entry for rel being merged */
int num;    /* number of tuples in mem to be merged */
```

DESCRIPTION

merge is part of the MRDSc sort/merge component and is not intended for standalone use. It expects to receive from sortrel a memory address containing num most recently sorted tuples and a file descriptor of a previously generated merge output file. The contents of the file and memory buffer are merged with duplicate tuple elimination into the new output file whose name is returned in string *out. Upon return to sortrel, the old merge file is removed. Buffer space in merge is dynamically allocated and released with each invocation and is at most 2K bytes.

For a complete description, see Section 3.7.

EXAMPLE:

The call to merge in sortrel merges the contents of file merfd with memory region sortbuf into the new merge file newmer for relation rel; there are tpb tuples in sortbuf:

```
merge(merfd,sortbuf,newmer,rel,tpb);
```

ERROR CONDITION	ACTION TAKEN
cannot allocate memory for input, output or working tuple buffers	in each case, error 52 followed by return FAIL
cannot open new merge output file	error 55; free allocated buffers and return FAIL
failed to write full buffer out to new merge file	error 54; free allocated buffers and return FAIL

RETURN VALUE(S)

FAIL	failed to merge file and memory buffer because of some error condition
SUCCESS	successfully merged file and memory buffer into new merge file

CALL TABLE

creat	U	read	U
db_err	M	sprintf	U
free	U	strcpy	U
malloc	U	tplcmp	M
mktemp	U	write	U

SEE ALSO

Sections 3.7; sortrel

```
int mkindex(rel,xsize,xfill,lsize,lfill)
REL   *rel;
int   xsize;  /* size of internal nodes */
int   xfill;  /* % fill internal nodes  */
int   lsize;  /* size of leaf nodes     */
int   lfill;  /* % fill leaf nodes      */
```

DESCRIPTION

mkindex is used to convert an existing non- B^*-tree organized relation into a B^*-tree organized one. It creates a new file in which it builds the tree, and when finished, removes the original file, giving the new one the name of the old one. The rel system relation tuple for the relation undergoing the conversion is pointed at by rel. The parameters xsize and lsize control the size of branch and leaf nodes, and xfill and lfill the extent to which branch/leaf nodes are to remain full following a split.

mkindex first generates the entries needed in the tree's first-block, then builds the root and first leaf nodes, placing one 'data' entry in each. These three B^*-tree entries are then written out, and the remainder of the conversion is performed by a rdtuple/ insert loop until all tuples have been read. Then, the tuple in rel for this relation is updated to reflect the new data organization, and the file containing the tree replaces the file formerly holding the relation. Note that the use of rdtuple instead of getuple ensures preservation of any z-ordering the non- B^*-tree form had.

mkindex returns SUCCESS immediately if the relation is already organized as a B^*-tree.

EXAMPLE:

To convert a relation thisflatrel to a B^*-tree organized one:

```
mkindex(thisflatrel,4096,60,8192,70);
```

The resulting B^*-tree will have branch nodes of size 4 Kbytes which remain 60% full after a split, and leaf nodes of size 8Kbytes which remain 70% full after a split.

ERROR CONDITION	ACTION TAKEN
rel pointer is (REL *)(NULL)	return FAIL
cannot open new file to contain tree	error 53; return FAIL
xfill and or lfill are not within the allowed range (typically between 5 and 95 %)	in each case: warning 83; set to default fill (XPGDFLTFILL/ LPGDFLTFILL); continue

xsize and/or lsize are not within the allowed range (typically 512 bytes to 8K bytes for branches, 512 bytes to 32 Kbytes for leaves)	in each case: warning 83; set to default size (XPGDFLTSIZE/ LPGDFLTSIZE); continue
cannot allocate memory for branch or leaf node work buffer	in each case: error 52; close and unlink output file; return FAIL
cannot read first tuple in relation to put onto leaf page	error 53; close and unlink output file; return FAIL
flushpage fails to write out first-block, root, or leaf node	in each case: error 54; close and unlink output file; return FAIL
failed to insert tuple	increment count of insert failures; at end of rdtuple/ insert loop, warning 81, reporting number of failed insertions; continue. Upon return, return -2
replace failed to replace old relation with new one	error 59; return FAIL

RETURN VALUE(S)

-2	none of, or some of, but not all of, the tuples in the input relation were inserted into the B^*-tree; the relation is not replaced, and the file containing the stunted tree is left as is
FAIL	conversion to B^*-tree organization failed due to some error condition
SUCCESS	conversion was successful

CALL TABLE

closerel	M	malloc	U	sprintf	U
db_err	M	mktemp	U	strcpy	U
find	M	open	U	timer	M
flushpage	M	rdtuple	M		
insert	M	replace	M		

SEE ALSO

Section 3.9

```
REL *mkrel(rname,mode,newZmap,numofdoms,doms,sizelim)
char *rname;                    /* name of new relation    */
char mode;                      /* boolean set ===> conrel */
long newZmap;                         /* bit map of Z ord atts   */
int numofdoms;                        /* num of doms in dom list */
char doms[][MAXDNMLEN];         /* list of domain names    */
long sizelim;                   /* max number of tuples    */
```

DESCRIPTION

mkrel is used to build the system relation framework needed when creating new relations by building the appropriate tuples in rel and rd.

mkrel performs a variety of checks on the parameters, and if all are in order, updates rel, creates a file in the directory holding the database of the appropriate name (which is closed before returning), and then updates rd. Typically a relational operation, *e.g.*, project, will call mkrel to create the output relation, then use adtuple to fill it.

The parameters provide the name of the relation, its mode and Zmap, the number of domains per tuple, an array containing the domain names for each attribute, ordered from most significant to least (left to right) and the maximum size (measured in *tuples*) that the relation is allowed to become.

EXAMPLE:

The call

```
mkrel(newrel,FLAT,0L,5,domnames,50L);
```

creates an empty new relation whose name is pointed at by *newrel, which is flat (*i.e.*, not a B^*-tree) and not z-ordered, as the Zmap reflects (0L). Tuples have five attributes; the name of the i^{th} domain is in domnames[i]. The relation is not allowed to have more than 50 tuples.

ERROR CONDITION	ACTION TAKEN
database is full: cannot add more relations	error 70; return (REL *)(NULL)
*rname points to a null string	error 58; return (REL *)(NULL)
*rname points to a name which is too long	warning 84; truncate to MAXRNMLEN (10 bytes in current implementation), continue
target relation already exists	warning 76; return (REL *)(NULL)

bad number of domains (less than 1 or more than 32)	error 71; return (REL *)(NULL)
a domain in domnames cannot be found in dom	warning 79; continue until end of list, then return (REL *)(NULL)
a domain appears more than once in domnames	warning 78; excise it from list, continue
rd relation is too full; not enough room to contain rd entries for new relation	error 71; return (REL *)(NULL)
cannot create file for new relation	error 55; return (REL *)(NULL)

RETURN VALUE(S)

FAIL	unable to make entries/file for new relation because of some error condition
>0	number of rd entries added for new relation; implies successful creation

CALL TABLE

creat	U	strcat	U
db_err	M	strcmp	U
findrel	M	strcpy	U
getrelnam	M	strlen	U
sprintf	U		

SEE ALSO

Sections 3.2.2, 3.5.5,3.6.4

```
long mksep(rptr,old,new,sep)  /* return separator length in BITS */
REL *rptr;     /* ptr to REL */
char *old;     /* ptr to tpl in old page */
char *new;     /* ptr to tpl in new page */
char *sep;     /* ptr to space for separator */
```

DESCRIPTION

mksep generates 'minimum' length separators to distinguish an entry pointed at by old from the one pointed at by new; the separator is returned in space pointed at by sep. Returned is the separator length in *bits*. sep should point to at least as many bytes as the tuples are wide.

mksep guarantees true minimal length separators only for *z*-ordered and pure string data. In building a separator, mksep proceeds attribute by attribute until a separator is produced or it is clear one cannot be made, *i.e.*, *old == *new. For *z*-ordered data, only as many bits as are needed will appear in the separator. If necessary, the separator will be padded with zero bits on the right (least significant positions) to align it to the next byte boundary. For string data, the separator contains the minimum length string which distinguishes *old from *new. For other data types, *e.g.*, integer, the entire attribute value will appear in the separator.

mksep is used by split when adding entries to B^*-tree (splitting leaves); it is not needed when splitting branch nodes (in a simple prefix B^*-tree).

EXAMPLE:

Where left is a pointer to the last tuple in an old leaf node, right is a pointer to the first tuple in a new leaf node. newsep points to an array whose length is that of thisrel -> width; following

```
i = mksep(thisrel,left,right,newsp);
```

it contains a separator of length *i* bits.

ERROR CONDITION	ACTION TAKEN
tuples are found to be identical	error 65; return FAIL

RETURN VALUE(S)

FAIL	could not produce separator: both tuples were identical, an error condition)
>0	length of generated separator, in *bits*

CALL TABLE

db_err	M
finddom	M
findrd	M

SEE ALSO

Sections 3.2.2, 3.6.2

```
ZCLIST *mkwrklist(size)
int size;
```

DESCRIPTION

mkwrklist builds an empty circular list used by *z*-order support code containing size elements. The elements are allocated and properly linked into a ZCLIST; returned is the address of the 'head' of the list. The *z*-order code uses this procedure to generate a working copy of the master circular list for use when reshuffling tuples; the actual contents of the list are filled in by copying from the master into the newly created working copy using restoreclist.

EXAMPLE:

To generate and initialize a working copy of the master circular list,

```
ZCLIST *worklist;
int lots;
if ((worklist = mkwrklist(lots)) != (ZCLIST *)(0)) {
        if ((restoreclist(masterclist,worklist,0)) ...
```

ERROR CONDITION	ACTION TAKEN
cannot allocate any elements	return (ZCLIST *)(0)

RETURN VALUE(S)

(ZCLIST *)(0)	allocated no elements for list due to some error condition
>0	pointer to 'head' of newly allocated list

CALL TABLE

malloc U

SEE ALSO

Sections 3.8; restoreclist

```
REL *rptr;
int mode;
```

DESCRIPTION

openrel will open the file containing the relation whose rel entry is pointed at by rptr. The name of the file is obtained through rptr -> relname and getrelnam. mode controls the access to the opened relation and may be RONLY (read only), RDWR (read/write), or APONLY (append only). The UNIX file descriptor for the opened file is saved in the rel tuple for the relation (rptr -> fd) and is returned by openrel.

Buffer space, sufficient to hold one tuple from the relation, is allocated by openrel so that procedures which access the relation are assured of buffer space (see rdtuple, wrtuple) should they not wish to provide their own. A pointer to the start of the buffer is stored in the global array buffer, in the element subscripted by the opened file's UNIX file descriptor. It is not an error to attempt to open an already opened relation, however one cannot change the access mode of a relation by an openrel call on an already opened relation; to do that, the relation must first be closed using closerel, then opened again with the new mode.

EXAMPLE:

If rptr -> relname points at 'thisrel|||', and

```
i = openrel(rptr,RONLY);
```

returns 6 in *i*, then the file in the database's directory called 'thisrel' has been successfully opened as read only, with UNIX file descriptor 6; a one tuple buffer is pointed at by buffer[6].

ERROR CONDITION	ACTION TAKEN
file containing relation could not be opened	return FAIL
could not allocate any buffer space for tuples	error 52; close the file and return FAIL
unable to close file during recovery from above error condition	error 50; return FAIL

RETURN VALUE(S)

FAIL	could not open relation due to some error condition; caller should check errno for further information regarding open failure
>0	file descriptor of successfully opened relation file; usually > 2 as descriptors 0, 1 and 2 are used. The descriptor value may be used as a subscript into array buffer to locate the pointer to buffer space allocated to the opened relation

CALL TABLE

close	U
db_err	M
getrelnam	M
malloc	U
open	U

SEE ALSO

Section 3.6; closerel, rdtuple, wrtuple

```
int outrel(rname,title)
char *rname;    /* name of relation to print */
char *title;    /* a page heading/title      */
```

DESCRIPTION

outrel will print the relation whose name is pointed at by rname on standard output. If *title is not a null string, it is printed atop each 'page' as well. Tuples are printed one per line, left justified in columns under attribute name headings. Tuples longer than outrel's idea of screen width are simply truncated.

Since its output is intended for display terminals, outrel contains hard coded ideas about line length and lines per 'page' (80 and 22, respectively, defined in header mrds.h).

outrel uses getuple to access tuples from the relation being printed, hence can be used to print z-ordered and/or B^*-tree organized relations without the need to 'convert' the relation to flat, printable form.

To print a relation on paper, the procedure printrel should be used as it is more attuned to the characteristics of a line printer.

EXAMPLE:

To print the relation 'thisrel ' with the message 'Test of outrel' appearing at the top of each screen:

```
outrel("thisrel|||","Test of outrel");
```

ERROR CONDITION	ACTION TAKEN
relation named *rname cannot be found	error 58; return FAIL
failed to locate all rd entries for relation	error 62; return FAIL
more rd entries found for *rname than are allowed for a relation	warning 96; set number of rd entries to MAXATTS (32 in the current implementation), continue
cannot find a domain referenced in rd	warning 79; continue
tuple length exceeds maximum line length	warning 88; truncate tuples for printing, continue

RETURN VALUE(S)

FAIL	unable to print relation because of some error condition
>0	number of tuples successfully printed

CALL TABLE

closerel	M	printf	U
db_err	M	sprintf	U
finddom	M	strcat	U
findrd	M	strcpy	U
findrel	M	strlen	U
getuple	M		

SEE ALSO

Section 3.11; printrel

```
int printrel(rname,title)
char *rname;     /* name of relation to print */
char *title;     /* a page heading/title      */
```

DESCRIPTION

printrel will print the relation whose name is pointed at by rname on standard output. If *title is not a null string, it is printed atop each page as well. Tuples are printed one per line, left justified in columns under attribute name headings. Tuples longer than printrel's idea of page width are simply truncated.

Since its output is intended for line printers, printrel contains hard coded ideas about line length and lines per page (132 and 55, respectively, defined in header mrds.h).

printrel uses getuple to access tuples from the relation being printed, hence can be used to print z-ordered and/or B^*-tree organized relations without the need to 'convert' the relation to flat, printable form.

For 'viewing' a relation on a terminal the procedure outrel should be used as it is more attuned to the characteristics of a display terminal.

EXAMPLE:

To print the relation 'thisrel ' with the message 'Test of outrel' appearing at the top of each page:

```
printrel("thisrel|||","Test of outrel");
```

ERROR CONDITION	ACTION TAKEN
relation named *rname cannot be found	error 58; return FAIL
failed to locate all rd entries for relation	error 62; return FAIL
more rd entries found for *rname than are allowed for a relation	warning 96; set number of rd entries to MAXATTS (32 in the current implementation), continue
cannot find a domain referenced in rd	warning 79; continue
tuple length exceeds maximum line length	warning 88; truncate tuples for printing, continue

RETURN VALUE(S)

FAIL	unable to print relation because of some error condition
>0	number of tuples successfully printed

CALL TABLE

closerel	M	printf	U
db_err	M	sprintf	U
finddom	M	strcat	U
findrd	M	strcpy	U
findrel	M	strlen	U
getuple	M		

SEE ALSO

Section 3.11; outrel

```
int project(rel,domlist,ndoms,newrel)
char *rel;                      /* rel to be projected          */
char domlist[][MAXDNMLEN];      /* list of domain names         */
short ndoms;                    /* number of domains in projection */
char *newrel;                   /* name of new (output) relation   */
```

DESCRIPTION

project produces a domain ('column') subset of relation *rel in the new output relation *newrel. domlist is a list of ndoms domains now in *rel which are to appear in tuples in the output relation. The sequence of appearance of attributes in the output relation is the sequence in which they appear in domlist. If ndoms is 0, the domlist parameter is ignored and all domains are projected in the order that they appear in *rel, *i.e.*, a copy of the relation is made.

domlist should have ndoms entries, each a domain name of length MAXDNMLEN (in the current implementation, 11 bytes [10 characters + terminating null byte]).

EXAMPLE:

The code below produces the new relation tcols containing the projection of the relation toycol, on its colour attribute:

```
strcpy(projdom[0],"colour||||");
project("toycol||||",projdom,1,"tcols|||||");
```

ERROR CONDITION	ACTION TAKEN
input relation rel cannot be found in system relations	error 58; return FAIL
findrd fails to locate all rd entries for rel	error 58; return FAIL
number of domains in ndoms is negative or exceeds the number found in rd	warning 77; assume ndoms is 0 and continue
cannot allocate temporary work buffer to use in construction of output tuples	error 52; return FAIL
mkrel cannot make new relation *outrel (possibly because *outrel already exists)	error 57; return FAIL

project fails to add tuples to the output relation	continue processing until all tuples read; then issue warning 81 and report number of failed additions; continue processing and ultimately return -2
sortrel fails to sort/eliminate duplicates from the projected entries in *outrel	warning 82; continue, ultimately returning -3

RETURN VALUE(S)

-3	projection occurred but intermediate output relation was not sorted successfully
-2	projection occurred unsuccessfully; some, possibly all, projected tuples were not added to output relation
FAIL	projection could not be performed because of some error condition
SUCCESS	projection successfully completed

CALL TABLE

adtuple	M	getuple	M
closerel	M	malloc	U
db_err	M	mkrel	M
finddom	M	sortrel	M
findrd	M	sprintf	U
findrel	M	strcpy	U

SEE ALSO

Section 3.11

```
int rdtuple(rptr,to)
REL  *rptr;
char *to;
```

DESCRIPTION

rdtuple is a low level relational read routine which will read the 'next' tuple from the relation whose rel entry is pointed at by rptr, placing the tuple either in a caller requested buffer (*to) or in the buffer allocated by openrel (if to === (char *)(NULL)).

rdtuple's idea of 'next' is based on the read index rptr -> rindx, which may be manipulated to cause the next tuple read to come from anywhere in the relation.

rdtuple simply scoops up rtpr -> width bytes from the current read index position, thus preserving *z*-ordering. It is intended for use by procedures which require tuples in 'raw' form. Procedures needing unshuffled, flat tuples should use getuple (which calls rdtuple and then automatically performs any unshuffling before returning a tuple to its caller).

If the file containing the relation is closed, not an error condition, rdtuple will first open it as read only (using openrel).

EXAMPLE:

To read the next tuple from the relation whose rel entry is pointed at by thisrelptr and put the tuple into mybuf:

```
rdtuple(thisrptr,mybuf);
```

ERROR CONDITION	ACTION TAKEN
rptr is a null pointer	return FAIL
cannot open file containing the relation (openrel failed)	error 51; return FAIL
attempt to get a non-existent tuple (read index does not point anywhere inside relation)	return FAIL
cannot seek to position in relation file indicated by read index	error 60; return FAIL
failed to read tuplewidth bytes from position indicated by read index	error 60; reset read index to start of tuple on which read attempt failed, return FAIL

in a B^*-tree organized relation, could not update read index because failed to get 'pointer to next' from leaf linked list	warning 92; reset read index to start of just read tuple, continue

RETURN VALUE(S)

FAIL	could not read all of tuple because of some error condition
>0	number of bytes successfully read; should be same as tuple width

CALL TABLE

db_err	M
lseek	M
openrel	M
read	U

SEE ALSO

Sections 3.2.2, 3.6.2, 3.6.3; openrel, getuple

```
int replace(oldrel,newfile)
REL *oldrel;   /* ptr to sysrel entry for rel being replaced */
char *newfile; /* name of file becoming the relation */
```

DESCRIPTION

replace is used to change the file containing a relation to another file. It first truncates the file holding the relation oldrel -> relname then copies the contents of *newfile to the truncated file. When copying is complete, *newfile is unlinked.

replace is used by processes which generate an output relation one tuple at a time (*e.g.*, project) and then sort the new output relation to produce the final result. Such procedures first open an output relation into which they write using adtuple. When finished, sortrel is used to order the output relation and eliminate duplicate tuples (except if result relation was of B^*-tree organization). The final output from sortrel is a temporary file which should now become the file holding the output relation; replace is used to cause that temporary file to replace the originally created (before sortrel) file and update the system relations (*e.g.*, concerning new size of output relation).

EXAMPLE:

A new output relation, newoutrel, has just been generated and is being sorted (primarily to facilitate elimination of duplicate tuples). When sortrel finishes sorting the file sortedrel contains the final form of the newly generated output relation. To cause sortedrel to become the file holding the relation,

```
replace(newoutrel,sortedrel);
```

ERROR CONDITION	ACTION TAKEN
file containing *oldrel is not closed	error 65; return FAIL
cannot open *newfile	error 51; return FAIL
cannot truncate file containing old relation	error 51; return FAIL
during copy, failed to write as many bytes as read	severe 25; close files, do not unlink *newfile and return -2
could not unlink *newfile	warning 83; return SUCCESS

RETURN VALUE(S)

-2	relation file partially replaced when an error occurred; the source file from which the replacement comes is left intact for consultation (since it now represents the only complete version of the relation)
FAIL	could not replace relation file because of some error condition
SUCCESS	successful replacement

CALL TABLE

close	U	read	U
creat	U	sprintf	U
db_err	M	strcpy	U
getrelnam	M	unlink	U
lseek	U	write	U
open	U		

SEE ALSO

Section 3.6.4

```
unsigned restoreclist(from,to,mode)
ZCLIST *from, *to;
short mode;
```

DESCRIPTION

restoreclist is used to initialize a Z circular list *to to the values currently found in the 'master list', *from. The mode determines whether the numzbits member of the list (number of bits in this attribute participating in z-order) is set to 0 (mode = 0) or to the value in the list *from.

EXAMPLE:

The working copy of the circular list used when reshuffling a tuple must be initialized for each tuple processed. The procedure mkwrklist is used to generate an empty Z circular list of the required size, then restoreclist is called each time a tuple needs to be processed:

```
restoreclist(masterzclist,wrkclist,1);
```

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

1	successful initialization of list

CALL TABLE

--

SEE ALSO

Section 3.8; mkwrklist

```
long search(rptr,tpl)
REL   *rptr;  /* rel to look through */
char  *tpl;   /* for this tuple      */
```

DESCRIPTION

search looks through the relation whose entry in rel is pointed at by rptr for the tuple pointed at by tpl, and returns the offset into the file where the tuple starts (suitable for assigning to rptr -> rindx prior to a call to getuple or rdtuple). If the tuple is not found in the relation, then search returns the offset to the tuple just after which the sought tuple would have appeared. If found, a positive valued offset is returned. If not found, the tuple's predecessor's offset is adjusted, negated and returned. The adjustment is necessary to ensure that FAIL (-1) is not returned as an offset; the largest negative offset returned is -2L, which would imply that the tuple being sought belongs before the first tuple now in the relation; -3L would indicate that it belongs after the first tuple, -4L after the second, *etc.*

search can deal with any data organization, and will [in future implementations] use the best strategy known to be applicable, *e.g.*, for a B^*-tree it uses the index in the branch nodes, for flat but ordered searches tuple by tuple until tuple is found or known to be absent † If no better technique is available, search looks through the relation tuple by tuple until a determination is made. Tuple comparisons are performed by tplcmp.

EXAMPLE:

To look through the relation indicated by thisrelptr for a tuple pointed at by lookfor:

```
position = search(thisrelptr,lookfor);
```

ERROR CONDITION	ACTION TAKEN
pointer rptr is NULL	return FAIL
cannot get mtime for file containing B^*-tree	warning 89; set mtime to -1, continue
failed to read first-block from B^*-tree	error 53; return FAIL
unale to allocate memory to hold one branch and/or one leaf node	in each case: error 52; in former case return FAIL; in latter, release branch node buffer, then return FAIL

† Although a binary search should be employed on a relation so ordered.

failed to read a node from file	error 53; free dynamically allocated buffer space and return FAIL
a branch node has been found to be corrupted (size mismatches)	severe 28; free dynamically allocated buffer space and return FAIL

RETURN VALUE(S)

$\leq$-3L	tuple not found but should have appeared just after the tuple which starts at offset *return value* + 3.
-2L	tuple not found but should have appeared before first tuple now in relation
FAIL	search could not be performed because of some error condition
$\geq$0	offset into file containing relation at which sought tuple was found

CALL TABLE

cmpseptup	M	rdtuple	U
db_err	M	read	M
free	U	stat	U
getrelnam	M	strcmp	U
malloc	U	timer	M
rdpage	M	tplcmp	M

SEE ALSO

Section 3.9; tplcmp

```
int select(rnam,domname,value,cmp,out)
char  *rnam;          /* name of  input relation to select from */
char domname[];             /* domain of selection                */
char *value;          /* value to compare against: TYPE UNKNOWN */
short cmp;            /* type of comparison to be done          */
char *out;            /* name of output relation                */
```

DESCRIPTION

select performs selection on a relation: it creates a new relation *out consisting of tuples from relation *rnam which satisfy the constraint

attribute in domname *cmp* *value

cmp may be any meaningful combination of $<$, $=$, or $>$, comparing the constant *value with an attribute of *rnam's tuples drawn from domain domname. *value can be of any datatype, hence is passed via a char * pointer which is interpreted during comparison as a pointer to the datatype of domain domname.

To specify a comparison, use the built-in names for the compare operators:

CMPATT_EQ	1
CMPATT_GT	2
CMPATT_LT	4

which may be combined to give, *e.g.*, a $\leq$ comparison.

EXAMPLE:

From the relation toycol produce a relation in which all toys are green in colour. The call to select would be:

```
select("toycol||||","colour||||","green",CMPATT_EQ,"greentoys|");
```

resulting in the creation of greentoys.

ERROR CONDITION	ACTION TAKEN
relation name, *rnam is a null string, or, the relation cannot be found	error 58; return FAIL
unrecognized compare operation	warning 83; return FAIL
cannot find domain domname	error 79; return FAIL
failed to get all rd entries for relation *rnam	error 58; return FAIL

domain domname not in relation *rnam	error 62; return FAIL
cannot make new relation *out	error 57; return FAIL
not all tuples processed were added to new relation	warning 81; indicate total number of failed additions, continue. Upon return, return -2
failed to sort/eliminate duplicates from newly created relation *out	warning 82; continue; upon return, return -3

RETURN VALUE(S)

-3	performed select but failed to sort/eliminate duplicates from output
-2	performed select but failed to add all selected tuples to output relation due to some error condition
FAIL	could not perform select due to some error condition
SUCCESS	successful select

CALL TABLE

closerel	M	findrel	M	sprintf	U
db_err	M	getuple	M	strcmp	U
finddom	M	mkrel	M	strcpy	U
findrd	M	sortrel	M		

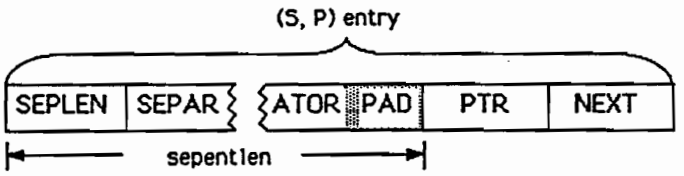
SEE ALSO

Section 3.11; sortrel

```
int sepentlen(s)
char    *s;             /* ptr to s in (s,p) entry */
```

DESCRIPTION

sepentlen returns the length, in bytes, of the *s* part of an (*s,p*) entry in a B^*-tree. This length, added to the location *s*, where the separator begins, yields a pointer to the *p* part:



EXAMPLE:

For an (*s,p*) entry with a separator 3 bytes long, the length of a separator entry:

```
i = sepentlen(thissepent);
```

where thissepent is on a longword boundary, would be returned as 8 (2 for seplen (unsigned short) + 3 for the separator + 3 for alignment pad).

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

>0	length of separator entry, in bytes, or, offset from start of (*s,p*) entry to *p* part

CALL TABLE

--

SEE ALSO

Section 3.9

```
int setup(dbname,db,verbose)
char  *dbname;                  /* name of database */
DBSTATUS *db;
int verbose;
```

DESCRIPTION

setup is used to make a database ready for use by MRDSc procedures. It will load the system relations into memory, perform a consistency check on the database using dbck, begin session logging and check a user's access privileges to a database.

The name of the database to set up is passed via dbname. db must point at a previously allocated occurrence of DBSTATUS which setup will fill in with the particulars of the database. The verbose parameter is passed on to dbck to control the extent to which it issues messages during its consistency check.

In normal MRDSc use, this should be the first MRDSc procedure called.

EXAMPLE:

Following determination of the name of a database name,

```
i = setup(thisdb,dbinfo,1);
```

will set up the database for use. Note that verbose is set to 1, which will cause dbck to issue several messages as it performs its consistency check.

ERROR CONDITION	ACTION TAKEN
the string *dbname or the pointer dbname is NULL	return FAIL
cannot open MRDSc system logfile for session logging	warning 95; continue
cannot locate database *dbname	return DB_NOFND (-2)
consistency check failed	return DB_FUBAR (-3)

RETURN VALUE(S)

DB_FUBAR (-3)	during setup, the database was found to be inconsistent, *i.e.*, unsafe for use

DB_NOFND (-2)	the database could not be located in the MRDSc master list (DBLIST) of known databases, hence could not be set up
FAIL (-1)	the database could not be set up because of some error condition
SUCCESS (0)	database successfully set up

CALL TABLE

db_err	M	fopen	U
dbck	M	gooduser	M
fclose	U	logent	M
find_db	M	sprintf	U

SEE ALSO

Section 3.5; dbck, find_db

```
int shuffle(rname,domlist,outname)
char *rname;                    /* name of relation to shuffle */
char domlist[][MAXDNMLEN];      /* list of domains to shuffle  */
char *outname;                  /* name of output relation     */
```

DESCRIPTION

shuffle allows the creation of a new *z*-ordered relation, *outname, from an existing relation, *rname, which may or may not already be *z*-ordered. Only the domains appearing in domlist will participate in the new *z*-ordering. If the domlist parameter is null, then all domains in *rname will be *z*-ordered to produce *outname. If domlist contains an actual list of i domain names, then the $i + 1^{st}$ entry *must* be a null string to signal 'end-of-list'. If relation *rname is already (partially or totally) *z*-ordered, its tuples are first completely unshuffled, then re-shuffled according to domlist. If the output relation would have the same *z*-ordering as the input relation, shuffle does no work (not even copying *rname to *outname), but signals this condition through its return value.

shuffle is essentially a front end to the MRDSc system procedure Z which performs the actual re-shuffling.

EXAMPLE:

To shuffle relation toycol into ztoycol:

```
i = shuffle("toycol|||",(char *)(0),"ztoycol|||");
```

Any non-zero value of i indicates a not completely successful shuffle.

ERROR CONDITION	ACTION TAKEN
relation *rname cannot be found	error 58; return FAIL
more than maximum allowed attributes in a relation said to be participating in *z*-ordering (*i.e.*, likely missing NULL domain name in domlist to signal end of list)	warning 77; set number of domains participating in *z*-ordering to all domains in the relation, continue.
no change in *z*-ordering: *outname would have same *z*-ordering as *rname	warning 98; return -2
a domain in domlist cannot be found	warning 79; continue checking domlist, then issue another warning 79 indicating number of missing domains encountered, and return FAIL

a domain name is duplicated in domlist	warning 78; ignore duplication, continue

RETURN VALUE(S)

-2	output relation would have had same z-ordering as input relation; shuffle not done, and no new relation made
FAIL	shuffle did not succeed because of some error condition
SUCCESS	successful shuffling of input relation
>0	number of errors during shuffle which has been only partially successful

CALL TABLE

db_err M
findrd M
findrel M
sprintf U
Z M

SEE ALSO

Section 3.8; Z

```
int sortrel(rel)
REL *rel;
```

DESCRIPTION

sortrel uses quicksort to sort the relation whose rel tuple is pointed at by the parameter rel, eliminates duplicate tuples, and replaces the file originally containing rel -> relname with the newly sorted version, updating rel -> cursize as necessary. It is used by procedures which produce relations as output, *e.g.*, project and select to produce the final form of their output relation.

sortrel has integrated sort and merge components, and will iteratively sort and merge if the relation is too large to sort in one pass. sortrel passes its output to merge via a pointer, *i.e.*, merge is constructed to merge a file (previous merge output) with a memory buffer (sort output) to produce a new output merge file. When all input tuples have been sorted, the last merge output file replaces the file holding the relation (using replace).

EXAMPLE:

In performing a select on relation inputrel creating selrel, whose rel tuple is pointed at by selrelptr, the last step is to sort and eliminate any duplicate tuples. This is done with

```
if (sortrel(selrelptr) != SUCCESS) ...
```

in select.

ERROR CONDITION	ACTION TAKEN
rel pointer is null	error 58; return FAIL
attempting to sort a B^*-tree	return -2
cannot allocate sort buffer memory	error 52; return FAIL
failed to get all needed rd entries	error 62; release buffer space; return FAIL
could not find a domain, *i.e.*, one or more is missing	warning 79; set domain's datatype to -1 (for 'unknown'); continue
could not allocate string buffer for old merge temp file name	warning 81; release buffer space; return FAIL
failed read on input relation	error 60; release buffer space; return FAIL
merge pass failed	error 55; release buffer space; return FAIL

unable to create a new merge temporary file	error 51; release buffer space; return FAIL
write failed while writing out merge temporary file	error 54; release buffer space; return FAIL
unable to open a previously written merge temporary file	error 51; release buffer space; return FAIL
unable to replace original relation file with new sorted file	error 59; free buffer space; return FAIL

RETURN VALUE(S)

-2	attempted to sort a B^*-tree
FAIL	sort failed, totally or partially, because of some error condition
SUCCESS	relation successfully sorted

CALL TABLE

close	U	findrd	M	open	U	sprintf	U
closerel	M	free	U	openrel	M	strcpy	U
creat	U	malloc	U	qsort	U	strlen	U
db_err	M	merge	M	read	U	unlink	U
finddom	M	mktemp	U	replace	M	write	U

SEE ALSO

Sections 3.7; merge, replace, tplcmp

```
int split(rptr,tpl,pos,x,mode,sptr)
char *tpl;      /* ptr to entry causing split: may be tpl or (s,p) */
REL *rptr;      /* ptr to rel whose page is being split */
long pos;       /* position at which to insert new entry */
Xfb *x;         /* ptr to Xfb for this relation */
char mode;      /* flag: split branch or leaf page */
long sptr;      /* 'p' in (s,p) pair */
```

DESCRIPTION

split is used to split a leaf or branch node during an insertion into a B^*-tree. When insert realizes that a leaf must be split to accommodate a new entry, it calls split, passing the pointer to the relation's tuple in rel (in rptr), the tuple which could not be inserted (in *tpl) and other particulars of the B^*-tree: both the split and completion of the insertion become the responsibility of split. This version of split will always perform a split, *i.e.*, it does not attempt to avoid splits by redistributing entries between adjacent nodes.

pos is the location returned by search to insert as the position after which the insertion should occur. x is a pointer to the memory area holding the first-block of the B^*-tree; the node being split *must* already be loaded into whichever of x -> lbufat or x -> xbufat is appropriate. mode identifies the node being split as a leaf or a branch (XLEAF,XBRANCH). If the node is a branch, then sptr points to the p part of the (s,p) pair at which the split is occurring. For leaf nodes, this parameter is ignored.

split is recursive so that if, following successful insertion of *tpl into one of the two new sibling leaves, the parent cannot be updated without being split, split calls itself. In this way, split may need to split branch nodes up the tree, possibly including the root.

A successful return from split implies that the tuple originally being inserted is now in the B^*-tree and all necessary updates to any affected branch nodes have been made.

EXAMPLE:

When insert realizes there is insufficient space on a leaf node to hold the tuple being inserted, it calls split:

```
if (split(rel,tpl,pos,x,XLEAF) == FAIL)
```

Note that since this is a leaf split, the parameter sptr is absent.

When updating a branch node requires it to be split, the call is

```
if (split(rptr,&(sp.te_item),posn,nodemode,sp.te_ptr) == FAIL)
```

where the sptr entry is present, as part of a structure called sp (which includes as its member te_item the separator part of the (s,p) pair).

ERROR CONDITION	ACTION TAKEN
pos is >0 implying that *tpl is already present in the relation	warning 91; return SUCCESS
cannot allocate buffers for nodes	in each case: error 52, free any previously allocated buffers and return FAIL
cannot allocate buffer for backup copy of node being split	warning 95; continue
failed to write out updated node passed to split	error 59; release buffer space; return FAIL
could not seek in file holding B^*-tree to location where new sibling node should be written	error 56; release buffer space; return FAIL
failed to write out new node	error 59; release buffer space; return FAIL
could not allocate buffer for branch node (for update of parent)	error 52; release buffer space; return FAIL
could not seek to position in file at which new root should be written	error 56; release buffer space; return FAIL
failed to write out new root node	error 59; release buffer space; return FAIL
failed to update parent node of now split node	error 61; release buffer space; return FAIL
could not seek to position in file at which to write updated child node	error 56; release buffer space; return FAIL
failed during write of updated child node	error 59; release buffer space; return FAIL
could not seek to position in file at which to write new leaf node	error 56; release buffer space; return FAIL
failed to write updated sibling nodes (replacement of original node, or new one)	for each case: error 59; release buffer space; return FAIL
failed to make separator between two new leaf nodes	error 68; release buffer space; return FAIL
failed to update parent	error 68; release buffer space; return FAIL

RETURN VALUE(S)

FAIL	split and subsequent insertion failed because of some error condition
SUCCESS	successful split and completed insertion

CALL TABLE

db_err	M
flushpage	M
lseek	U
malloc	U
mksep	M
upd_parent	M
write	U

SEE ALSO

Sections 3.9; insert, upd_parent

int syncrel()

DESCRIPTION

syncrel is used at the end of an MRDSc session to flush the current memory resident versions of the system relations out to their respective disk files and close the session log file prior to exiting. It updates only the system relations and returns. If it is undesirable to save updated system relations, MRDSc should be exited without calling syncrel thus leaving the system relations exactly as they were when the current session started.

EXAMPLE:

When winding up a session, to save the updated system relations:

```
syncrel();
```

ERROR CONDITION	ACTION TAKEN
cannot open the file holding a relation for writing	error 51; return value depends on which file could not be opened: -4 for rel, -2 for dom, and -1 for rd.
error during write to file holding relation	error 54; return value depends on which file could not be written: -4 for rel, -2 for dom, and -1 for rd.

RETURN VALUE(S)

<0	update of system relations failed due to an open or write failure. Absolute value of return value is a bit string in which bits conveying the values 4, 2, and 1 are OR'ed implying failure in rel, dom, or rd respectively.
SUCCESS	successful update of disk file holding system relations

CALL TABLE

close U
db_err M
logent M
open U
sprintf U
write U

SEE ALSO

Sections 3.5.5

long timer()

DESCRIPTION

timer returns the host UNIX system's idea of the current time of day as the number of seconds since "the epoch". It is used by the logging procedure logent to timestamp its entries and by the procedures using LRU replacement to establish time of last use.†

EXAMPLE:

To get the current time,

```
l = timer();
```

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

>0	long integer containing time, in seconds, since "the epoch"

CALL TABLE

gettimeofday	U (on BSD Unix systems)
time	U (on old Unix systems)

SEE ALSO

Sections 3.5.2, 3.8, 3.9

† Although these should get finer resolution time stamps

```
int tplcmp(a,b)  /* kludgey but quick */
char *a,*b;
```

DESCRIPTION

tplcmp compares two tuples by basically returning *a - *b. It returns ≤-1 if *a < *b, 0 if *a = *b, or ≥1 if *a > *b. It is used by sortrel and any other procedure which must determine the relationship between two tuples.

Since tuples can consist of a combination of one or more possibly different data types, they must be passed via char * pointers. tplcmp uses the datatype information in dom and the domain positioning information in rd to determine how to perform its comparison. If given an unknown datatype for a domain, tplcmp silently ignores that domain in its comparison. Only as much of the tuples as is needed to establish their relationship is examined (thus equality is worst case). tplcmp begins comparing the tuples' first attributes, using whatever comparison is appropriate to that attribute's datatype, and continues, attribute by attribute, until a determination is made. Coercion of individual attributes to their correct datatypes is handled automatically by tplcmp, by rather ugly code, since tuple attributes are not guaranteed to be correctly aligned.

EXAMPLE:

To compare tuple1 with tuple2, which are stored in memory areas pointed at by the char * pointers t1ptr and t2ptr respectively:

```
i = tplcmp(t1ptr,t2ptr);
```

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

≤-1	tuple *a is less than tuple *b
0	tuple *a is equal to tuple *b
≥1	tuple *a is greater than tuple *b

CALL TABLE

--

SEE ALSO

Section 3.7; sortrel, search

```
int unshuffle(rname,domlist,outname)
char *rname;                    /* name of relation to shuffle */
char domlist[][MAXDNMLEN];      /* list of domains to shuffle  */
char *outname;                  /* name of output relation     */
```

DESCRIPTION

unshuffle allows the creation of a new *z*-ordered relation, *outname, from an existing relation, *rname, which may or may not already be *z*-ordered. Only the domains appearing in domlist will participate in the new *z*-ordering. If the domlist parameter is null, then all domains in *rname will be unshuffled to produce *outname. If domlist contains an actual list of i domain names, then the $i + 1^{st}$ entry *must* be a null string to signal 'end-of-list'. If relation *rname is already (partially or totally) *z*-ordered, its tuples are first completely unshuffled, then re-shuffled according to domlist. If the output relation would have the same *z*-ordering as the input relation, unshuffle does no work (not even copying *rname to *outname), but signals this condition through its return value.

unshuffle is essentially a front end to the MRDSc system procedure Z which performs the actual re-shuffling.

EXAMPLE:

To unshuffle relation ztoycol into toycol:

```
i = unshuffle("toycol||||",(char *)(0),"ztoycol|||");
```

Any non-zero value of i indicates a not completely successful unshuffle.

ERROR CONDITION	ACTION TAKEN
relation *rname cannot be found	error 58; return FAIL
more than maximum allowed attributes in a relation said to be participating in *z*-ordering (*i.e.*, likely missing NULL domain name in domlist to signal end of list)	warning 77; set number of domains participating in *z*-ordering to all domains in the relation, continue.
no change in *z*-ordering: *outname would have same *z*-ordering as *rname	warning 98; return -2
a domain in domlist cannot be found	warning 79; continue checking domlist, then issue another warning 79 indicating number of missing domains encountered, and return FAIL

a domain name is duplicated in domlist	warning 78; ignore duplication, continue

RETURN VALUE(S)

-2	output relation would have had same *z*-ordering as input relation; unshuffle not done, and no new relation made
FAIL	unshuffle did not succeed because of some error condition
SUCCESS	successful unshuffling of input relation
>0	number of errors during unshuffle which has been only partially successful

CALL TABLE

db_err M
findrd M
findrel M
sprintf U
Z M

SEE ALSO

Section 3.8; Z

```
int upd_parent(rptr,sp,parent,x)
REL    *rptr;
Bentry sp;
long   parent;
Xfb    *x;
```

DESCRIPTION

upd_parent is used when splitting a leaf or a branch node and the parent of the node being split must be updated, *i.e.*, a new (s,p) pair must be inserted into it. rptr points at the relation's rel tuple, sp contains the actual s and p entries to be inserted into the parent, parent holds the disk address of the split node's parent and x points at the B^*-tree's first-block.

upd_parent searches the parent node for the split node's entry and attempts to insert the new (s,p) immediately after it in the linked list of (s,p) entries in the parent. Should there be insufficient space in the parent to accommodate such an insertion, upd_parent will set up the necessary parameters for a call to split to split the parent, following which it retries the insertion.

EXAMPLE:

When split has completed its split of a leaf node and has the (s,p) pair ready for insertion into the parent of the now sibling nodes, it calls:

```
if (upd_parent(rptr,upward,*(lf.pred),x) === FAIL)
```

where upward is a Bentry structure containing the (s,p) values to insert, and lf.pred is a pointer to the parent of the node received by split.

ERROR CONDITION	ACTION TAKEN
parent pointer is null	return FAIL
could not allocate space for branch node work buffer	error 52; return FAIL
failed to load parent page into memory (loadpage fails)	error 60; free buffer space; return FAIL
could not find pointer to current node in its parent	severe 29; free buffer space; return FAIL
failed to write out updated parent page (flushpage fails)	error 59; free buffer space; return FAIL
no room on parent page after split	severe 30; free buffer space; return FAIL

could not allocate space for branch node	error 52; free buffer space; return FAIL
split failed to split parent	error 68; free buffer space; return FAIL
unable to load split child following parent split (loadpage fails)	error 60; free buffer space; return FAIL

RETURN VALUE(S)

FAIL	unable to update parent node because of some error condition
SUCCESS	successful update of parent node

CALL TABLE

db_err	M
flushpage	M
free	U
loadpage	M
malloc	U
split	M

SEE ALSO

Section 3.9; split

```
int wrtuple(rptr,from)
REL  *rptr;
char *from;
```

DESCRIPTION

wrtuple is a low level relational write routine which will write the 'next' tuple from the relation whose rel entry is pointed at by rptr, getting the tuple from the caller designated buffer (*from). Note that this differs from rdtuple in that *from *must* point at a valid buffer location, either &buffer[rptr -> fd] or some user defined one; in rdtuple, the buffer pointer is allowed to be NULL to indicate use of the buffer space allocated by openrel.

wrtuple's idea of 'next' is based on the write index rptr -> windx, which may be manipulated to cause the next tuple written to come from anywhere in the relation.

wrtuple simply spits out rtpr -> width bytes at the current write index position, thus preserving z-ordering. It is intended for use by procedures which use tuples in 'raw' form. Procedures writing flat tuples into z-ordered or B^*-tree organized relations should use adtuple (which first performs any shuffling and then calls wrtuple to write the tuple).

If the file containing the relation is closed, not an error condition, wrtuple will first open it as read/write (using openrel).

EXAMPLE:

To write the next tuple into the relation whose rel entry is pointed at by thisrelptr from mybuf:

```
wrtuple(thisrptr,mybuf);
```

ERROR CONDITION	ACTION TAKEN
rptr is a null pointer	return FAIL
write attempt which is not an append on an append-only relation	error 59; return FAIL
cannot open file containing the relation (openrel failed)	error 51; return FAIL
attempt to add to 'too full' relation	warning 85; return FAIL
cannot seek to position in relation file indicated by write index	error 59; return FAIL

failed to write tuplewidth bytes to position indicated by write index	error 59; reset write index to start of tuple on which write attempt failed, return FAIL

RETURN VALUE(S)

FAIL	could not write all of tuple because of some error condition
>0	number of bytes successfully written; should be same as tuple width

CALL TABLE

db_err M
lseek U
openrel M
write U

SEE ALSO

Sections 3.2.2, 3.6.2, 3.6.3; openrel, adtuple

int yesno(string)

DESCRIPTION

yesno is used by interactive procedures to test whether *string is an affirmative or negative response from a user. If *string begins with upper or lower case 'y' yesno returns 2. 1 is returned for *string starting with upper or lower case 'n'; otherwise 0 is returned. Beware: in a simple-minded approach to leading blank suppression, characters which are neither 'y' nor 'n' are skipped over until a 'y' or 'n' is encountered or end-of-string is reached (thus, "maybe" would always be taken to mean yes).

EXAMPLE:

A user's response to some interactive question is contained in the string useransw. The following determines action based on what the user indicated:

```
switch (yesno(useransw)) {
case 2: /* said yes */
        .
        .
        .
case 1: /* said no */
        .
        .
        .
default: /* prompt again; don't grok user answer */
}
```

ERROR CONDITION	ACTION TAKEN
none	

RETURN VALUE(S)

2	user 'said' yes
1	user 'said' no
0	user did not 'say' either of yes or no

CALL TABLE

--

SEE ALSO

--

```
int z(rptr,fromZmap,toZmap,tplptr)
REL *rptr;
long fromZmap;
long toZmap;
char *tplptr;
```

DESCRIPTION

z reshuffles one tuple from the relation whose rel entry is pointed at by rptr. Tuples with *z*-order expressed by the bitmap fromZmap (where set bits positionally represent attributes now participating in *z*-order, bit 0 being most significant attribute [on a VAX, bits go leftward as domains go rightward across a tuple]) are shuffled into new tuples whose *z*-ordering is expressed by toZmap. tplptr points at the tuple which is to be reshuffled. When both fromZmap and toZmap are non-zero and different, z automatically unshuffles the tuple (to a Zmap of 0L) then shuffles it to toZmap. The reshuffled tuple is returned in *tplptr.

z implements a tuple-at-a-time reshuffling mechanism used by *e.g.*, getuple and adtuple, whereas Z provides a relation-at-a-time mechanism.

EXAMPLE:

When getuple has just received a raw tuple read by rdtuple, it calls z to unshuffle it:

```
if (z(rptr,rptr -> Zmap,0L,buffer[rptr -> fd])) == FAIL)
```

ERROR CONDITION	ACTION TAKEN
findrd fails to get all rd entries for relation	return FAIL
finddom cannot find a domain	warning 79; set the domain's length to 0, continue
cannot allocate memory for a needed list or working tuple buffer	in each case: error 52; increment error count, release any already allocated space and return FAIL
attempt to change Zmap in mid-relation (all tuples in a relation must have same Zmap value)	warning 87; use for toZmap already established map value for relation

RETURN VALUE(S)

FAIL	tuple was not reshuffled because of some error condition
SUCCESS	tuple successfully reshuffled

CALL TABLE

db_err	M	freellist	M
finddom	M	malloc	U
findrd	M	mkwrklist	M
free	U	restoreclist	M
freeclist	M	timer	M

SEE ALSO

Section 3.8; getuple, adtuple, Z

```
int Z(rptr,map,rds,out)
REL *rptr;
long map;
RD *rds[];
char *out;
```

DESCRIPTION

Z reshuffles entire relations from their current *z*-ordering to the new ordering specified in map. Each bit in map represents a domain of the relation whose rel tuple is pointed at by rptr; the leftmost attribute corresponds to bit 0, the next attribute (towards right) to bit 1, *etc.* Each bit which is set identifies a domain which is to participate in the new *z*-ordering of the output relation whose name is *out (which may be same as rptr -> relname *i.e.*, the intent is to replace a relation, which may already be partially or totally *z*-ordered, with a differently *z*-ordered version). If map is the same as rptr -> Zmap *i.e.*, no change in *z*-ordering, Z returns without doing anything.

This procedure implements a relation-at-a-time reshuffling mechanism, whereas z provides a tuple-at-a-time mechanism. Z is largely obsoleted by the newer z which achieves the same effect by iteratively reading/z/writing albeit more slowly.

EXAMPLE:

To change the data organization of the relation wasflatrel with 6 attributes per tuple to *z*-ordered on all attributes, one would call shuffle which ultimately calls Z to do the shuffling work with

```
Z(rptr,63L,rdtab,outname)
```

where rdtab is the result of a call to findrd to get all rd entries for the relation being reorganized.

ERROR CONDITION	ACTION TAKEN
performing an in-place reshuffling of a relation, but cannot open the relation as writable	error 51; return FAIL
output relation is new, but cannot make a new relation (mkrel fails)	error 57; return FAIL
a domain in rd cannot be found in dom (finddom fails)	warning 79; set length of missing domain to 0, continue
cannot allocate memory for a needed list or working tuple buffer	in each case: error 52; release already allocated space and return FAIL

failed to add a tuple to output relation (adtuple fails)	increment count of add failures; when finished getting tuples from input relation issue warning 81 and report quantity of failures; ultimately return FAIL

RETURN VALUE(S)

FAIL	shuffled relation was not completely successfully generated because of some error condition
SUCCESS	successful generation of new *z*-ordered relation

CALL TABLE

closerel	M	freellist	M
db_err	M	malloc	U
finddom	M	mkwrklist	M
findrd	M	restoreclist	M
free	U	strcpy	U
freeclist	M	timer	M

SEE ALSO

Sections 3.8; shuffle, unshuffle, z

Chapter 5

5: Two Useful Development Tools: cmd and mkdb

5.1 Overview of cmd

During the development of MRDSc it became necessary to have a quick and easy way to test various components. Originally it was sufficient to add specific code to main, recompile and reload, and run the new test code. This rapidly became unrealistic as the system grew.

A different approach was to provide a bare minimal interactive driver, analogous to the simple interactive monitor programs by which a console operator can control a VAX or an Amdahl. Such a driver is cmd; it allows a user to open, work on, and close a database. As new operators are implemented, code can be added to cmd to make it available to a user. cmd has no frills, few amenities, and is in no way intended to be a general purpose user interface.

cmd is implemented as a simple one-line command processor which expects the first blank or tab terminated string ('word') to be a command. Upon encountering a recognized request, it calls a routine which examines the line and therefrom generates a call to the MRDSc routine(s) which implement the requested action. This approach was adopted because of the extreme ease with which one can add code to test new routines.

The only 'amenity' offered by cmd is interrupt control from the terminal. In response to a SIGINT or SIGQUIT, cmd ceases its current activity, asks the user whether to continue the (now interrupted) action, to return to command mode, or to quit altogether. This allows an individual operation to be terminated without ending the cmd session.

cmd is invoked under the current implementation when a UNIX user types mrds as a command to the shell. Invoked in this way, the user is first prompted for a database name upon which to work. If invoked from the shell with

mrds -n *dbname*

database *dbname* will be opened and the user placed immediately into cmd's command loop, signalled by the prompt "mrds> ".

The following commands are understood by cmd:
bye, exit, help, quit: cmd control commands;
finddom, findrd, findrel, list: database commands;
convert, printrel, project, select: relational commands; each is described separately below, in alphabetical order.

5.2 BYE

request to terminate interactive session.

Following user confirmation of intent to exit, the system relations rel, rd, and dom are updated to reflect the new state of the database, and mrds exits to the invoking program, usually the UNIX shell.

The commands exit and quit are identical to bye.

5.3 CONVERT

convert current_rel *to* new_rel

convert the existing relation current_rel to the new relation new_rel where to determines the type of conversion and is one of:

b	convert to B^*-tree organization
z	convert to z-ordered organization
f	convert tree to flat organization

Example: To convert the relation toycol to a z-ordered relation ztoycol:

convert toycol z ztoycol

5.4 EXIT

see 'BYE'

5.5 HELP

prints a list of known commands, the syntax for their use, 'one line' descriptions of what they do, and a few basic rules for using cmd.

5.6 LIST

produces a summary of the current database, naming all relations and giving their current sizes (in bytes and tuples).

5.7 PRINTREL

printrel rel_to_print hdg

Displays the contents of rel_to_print on a terminal, with optional heading hdg atop each output 'page'. The named relation is roughly formatted to fit on a crt display and printed out, stopping after each screenful awaiting a carriage return to continue. † The data organization of the

† This should be changed so that the output produced by printrel actually goes through the UNIX utility pr and thence to a UNIX page-at-a-time display program, *e.g.*, more.)

relation is immaterial; any extra work to unshuffle or traverse a tree is automatically performed.

5.8 PROJECT

project cur_rel domname out_rel

Produce in relation out_rel the projection of relation cur_rel on domain domname. The cmd driver for project supports only a single domain name on which to project; the actual project procedure supports more than one.

Example: To produce the relation colours from toycol (having domains 'toyname' and 'colour') containing colours in which toys are available:

project toycol colour colours

5.9 QUIT

see 'BYE'

5.10 SELECT

select cur_rel domname *cmp* value out_rel

Produce in relation out_rel the selection from relation cur_rel of tuples having attributes in domain domname related to the constant value (*i.e.*, a value of an attribute in domname) according to *cmp*. cmp may be any one of '<', '=', '>', '<=', '>=', or '<>'. value is an actual attribute value and must contain no embedded blanks or tabs.

Example: To build the relation trucks containing all tuples from toycol which correspond to truck toys:

select toycol toyname = truck trucks

5.11 SHOWDOM, SHOWRD, SHOWREL

These commands produce a formatted display of the contents of the system relations, so they may be inspected during a session. The system relations exist only in main memory while MRDSc is running, thus a conventional command like printrel cannot be used to display their contents.

5.12 Adding to cmd

To add a command to cmd's repertoire, the following steps are necessary (refer to cmd.c in source listing):

(1) Add an entry to the table struct cmd tab containing [a] the exact string by which the command will be known, [b] a unique integer which identifies position in the switch at which the code for this command will appear.

The procedure command will examine user typed lines, comparing the first space or tab terminated string to those in struct cmd tab, returning -1 for an unrecognized command or the integer tagged in the command name in 1.b above. This number selects a case in a switch which will execute the command.

(2) Update the constant KNOWN CMDS to reflect any addition(s) just made to struct cmd tab.

(3) Add an entry in cmd's switch statement, at the position indicated in 1b above, which will invoke a driver for the command.

(4) Add a procedure to generate a call to the new code from the user typed command line. The procedure should return no value (hence be of type void), declarations in cmd.

5.13 mkdb

mkdb is a simple interactive utility designed to facilitate the construction of relational databases suitable for use with MRDSc. It interactively solicits information concerning each relation and the construction of its tuples, and from these queries builds the appropriate system relation entries. The files which make up individual user relations are created independently (*e.g.*, by vi) and are converted into relations by mkdb. There must be one file for each relation, and in each, attributes must be distinguished from each other by a delimiter (the colon, ':'). Thus files suitable for 'conversion' by mkdb resemble the UNIX /etc/passwd or /etc/termcap files.

In order to create a database for use with MRDSc

(1) Produce, through whatever means is convenient, text files of the needed format (see above).

(2) In file mkdb.c, define DBHOMEDIR to be the directory under which the new database is to be created. Recompile mkdb.c into mkdb.

(3) Run mkdb answering its questions (see below).

(4) Add an entry to the MRDSc master database list so that the existence of this new database will be recognized.

mkdb first requests the name of the new database. Given this, it tries to make a directory in DBHOMEDIR of that name, within which it creates the files .rel, .dom, and .rd. These system relation files are then initialized to the minimal content required to support a relational database.

The next phase of mkdb is a pair of (nested) for loops. The outer loop asks for relation particulars (name, maximum number of tuples). For each new relation, the inner loop solicits domain information (name, datatype, length). The order in which domains are supplied determines the order in which they appear in tuples. Entering a null domain name exits the inner loop. Similarly, a null relation name terminates the outer loop.

The final stage of mkdb is a loop wherein the user is asked for the name of a UNIX file which is to be used to generate a relation, for each of the relation names entered previously.

If all goes well, mkdb exits after successfully building the relations from the supplied files and updating the relevant rel entries to reflect the current sizes of the user relations.

Warning: As with cmd, mkdb is a quickly implemented utility created to fulfill a particular need during system implementation. It is not rigorously built, not particularly intelligent (it's downright dim), but does work as advertised. It is *not* intended for general use.

5.14 Source Listing

On the following pages are the listings for cmd.c and mkdb.c.

```
#include "mrds.h"

#define MAXCMDLEN 10
#define KNOWN_CMDS 12
struct cmd_tab {
        char    cmd_name[MAXCMDLEN];
        int     cmd_ident;
} CMD_NAMES[KNOWN_CMDS] = {
        "bye",0,
        "convert",9,
        "exit",0,
        "showdom",1,
        "showrd",2,
        "showrel",3,
        "help",6,
        "list",8,
        "printrel",4,
        "project",5,
        "quit",0,
        "select",7
};

char cmdstr[MAXCMDLEN + 1];
jmp_buf cmd_cont;

void cmd(db)
char *db;
{
        extern int errno;
        register int i;
        DBSTATUS mydb;
        char dbname[MAXNAMLEN],*getdbname();
        void do_exit(), do_showdom(),do_showrd(),do_showrel();
        void do_printrel(),do_project(),do_help(),do_select();
        void do_list(),do_convert(),usr_int();

        /* This is a quickly patched together, simple-minded
           command interpreter designed to facilitate testing
           of pieces built into MRDSc. It is NOT meant to be
           a general purpose user interface, although it could
           be grown into one with a modest amount of effort.

           See the project report (sections on "cmd" and section
           6.2) for more details on user interfaces.
        */


getname:
        if (strlen(db) == 0)
                strcpy(dbname,getdbname());
        else
                strcpy(dbname,db);

        i = setup(dbname,&mydb,1);
        if (strlen(dbname) == 0) abend(-2,"user quit");
```

```
        switch (i) {
        case FAIL:
                printf("XXX Can't set up database \"%s\"\n",dbname);
                break;
        case -2:
                printf("XXX Can't find database \"%s\" (new?)\n",dbname);
                break;
        case -3:
                printf("?XXX Database is inconsistent:");
                break;
        default:
                break;
        }
        if (i < SUCCESS) {
                *db = '\0';
                goto getname;
        }

        /* set up interrupt control */

        if (setjmp(cmd_cont) == 0) {
                signal(SIGQUIT,usr_int);
                signal(SIGINT,usr_int);
        }
        /* put prompt onto stdout and read from stdin */
cmd_loop:
        printf("%s ",PROMPT);
        READLN
        if (strlen(inbufr) == 0) goto cmd_loop;
        switch (command(inbufr)) {
        case CMD_BYE:
                do_exit();
                break;
        case CMD_FINDDOM:
                do_showdom();
                break;
        case CMD_FINDRD:
                do_showrd();
                break;
        case CMD_FINDREL:
                do_showrel();
                break;
        case CMD_PRINTREL:
                do_printrel();
                break;
        case CMD_PROJECT:
                do_project();
                break;
        case CMD_HELP:
                do_help();
                break;
        case CMD_SELECT:
                do_select();
                break;
        case CMD_LIST:
                do_list();
```

```
                break;
        case CMD_CONVERT:
                do_convert();
                break;
        default:
                printf("\tXXX unknown command: \"%s\"\n",cmdstr);
        }

        goto cmd_loop;
}


int command(buf)
char *buf;
{
        register int i;
        int strcmp();

        /* identify the command, if possible */

        i = 0;
        for (i = 0; i < MAXCMDLEN; i++) {
                cmdstr[i] = buf[i];
                if ((buf[i] == ' ') || (buf[i] == '\t'))
                        break;
        }

        cmdstr[i] = '\0'; /* terminate string, or truncate it if too long */

        for(i = 0; i < KNOWN_CMDS; i++)
                if (!strcmp(cmdstr,CMD_NAMES[i].cmd_name))
                        return(CMD_NAMES[i].cmd_ident);

        return(-1);
}
void do_exit()
{
        printf("Exit (y or n)? ");
        READLN
        if (yesno(inbufr) == 2) {
                /* do exit code...closeup shop */
                printf("Updating relations...(wait)");
                syncrel();
                printf(" :done.\nBye.\n");
                exit();
        }
        printf("\t ...continuing...\n");
}

void do_abort()
{
        syncrel();
        exit();
}
void do_showdom()
{
```

```
        extern DOM domcore[];
        extern struct rstat sysrelstat;
        register int i;
        static char *dtypes[] = {"string","int","char","float","short","long"};
        for(i = 0; i < sysrelstat.numdoments; i++){
                printf("%4d /%4d ",i + 1,sysrelstat.numdoments);
                printf("\"%s\"",domcore[i].domname);
                printf("\t%s\t",dtypes[domcore[i].domtype]);
                printf("\tlen:%7d\n",domcore[i].len);
        }
}

void do_showrd()
{
        extern RD rdcore[];
        extern struct rstat sysrelstat;
        register int i;
        printf("\f            relname       domname    \t   pos\n");
        for(i = 0; i < sysrelstat.numrdents; i++){
                printf("%4d /%4d ",i + 1,sysrelstat.numrdents);
                printf("\"%s\"",rdcore[i].relname);
                printf("\t\"%s\"",rdcore[i].domname);
                printf("\t%7d\n",rdcore[i].pos);
        }
}

void do_showrel()
{
        extern REL relcore[];
        extern struct rstat sysrelstat;
        register int i;

        for(i = 0; i < sysrelstat.numrelents; i++){
                printf("\fDump system relation REL:");
                printf("%4d of %4d entries\n\n",i + 1,sysrelstat.numrelents);
                printf("Relation \"%s\"\n",relcore[i].relname);
                printf("mode %xX\n",((long)(relcore[i].mode) & 255L));
                printf("tplwidth:%6d\n",(int)(relcore[i].width));
                printf("fd:%5d\n",(int)(relcore[i].fd));
                printf("Zmap: %lxX\n",relcore[i].Zmap);
                printf("cursize: %7d",relcore[i].cursize);
                printf("\t(%d tuples)\n",relcore[i].cursize / relcore[i].width);
                printf("maxsize: %7d",relcore[i].maxsize);
                printf("\t(%d tuples)\n",relcore[i].maxsize / relcore[i].width);
                printf("rindx: %7d",relcore[i].rindx);
                printf("\t(tpl number %d)\n",relcore[i].rindx / relcore[i].width);
                printf("windx: %7d",relcore[i].windx);
                printf("\t(tpl number %d)\n",relcore[i].windx / relcore[i].width);
                printf("\n\n\nMORE...(hit return)");
                getc(stdin);
        }
}

void do_printrel()
{
       /* set up call to printrel: take next two parms as */
```

```
        /* "rname" (name of relation to print) and */
        /* "title" (optional string to appear as title on each page of */
        /*         printed relation */

        char rname[MAXRNMLEN],ttl[MAXSUBSTR];
        register int i;
        register char *c,*d;
        int strlen();

        c = inbufr;
        d = rname;
        /* skip command name */
        while (*c != ' ' && *c != '\t')
                ++c;

        while (*c == ' ' || *c == '\t')
                ++c;

        /* now pointing at next arg: it becomes rname */

        for(i = 0; i < MAXRNMLEN; i++) {
                *d = *c;
                if (*c == ' ' || *c == '\t' || *c == '\0') {
                        *d = '\0';
                        break;
                }
                c++; d++;
        }

        *d = '\0';

        /* adjust name length */

        i = strlen(rname);
        while(i++ < (MAXRNMLEN - 1))
                *d++ = ' ';
        *d = '\0';

        d = ttl;
        while (*c == ' ' || *c == '\t')
                *c++;

        /* now pointing at next arg: it becomes ttl */

        for(i = 0; i < MAXRNMLEN; i++) {
                *d = *c;
                if (*c == ' ' || *c == '\t'){
                        *d = '\0';
                        break;
                }
                c++; d++;
        }

        *d++ = '\0';

        outrel(rname,ttl);
```

```
}

void do_help()
{
        printf("\f\tKnown commands and their syntax\n");
        printf("\t\targuments are separated by spaces or tabs ONLY");
        printf("\n-------------------------------------------------------------------  \n");
        printf("|    CMD NAME    |  ARGS                    |          DESCR            |\n");
        printf("+----------------+--------------------------+---------------------------+\n");
        printf("|  bye           |      (none)              | leave the command         |\n");
        printf("|                |                          | interpreter.              |\n");
        printf("+----------------+--------------------------+---------------------------+\n");
        printf("|  help          |      (none)              | display this help page.   |\n");
        printf("+----------------+--------------------------+---------------------------+\n");
        printf("|  printrel      | rel  [title]             | print relation \"rel\" with  |\n");
        printf("|                |                          | \"title\" atop each page of  |\n");
        printf("|                |                          | listing.                  |\n");
        printf("+----------------+--------------------------+---------------------------+\n");
        printf("|  project       | rel attr newrel          | make new relation \"newrel\" |\n");
        printf("|                |                          | from projection of \"rel\" on|\n");
        printf("|                |                          | attrs in \"attrlist\" (>= 1) |\n");
        printf("+----------------+--------------------------+---------------------------+\n");
        printf("|  select        | rel attr cmp const out   | from relation \"rel\" select |\n");
        printf("|                |                          | tuples with \"attr\" related |\n");
        printf("|                |                          | to constant valued \"const\" |\n");
        printf("|                |                          | as expressed by \"cmp\"      |\n");
        printf("|                |                          | and put into newrel \"out\". |\n");
        printf("+----------------+--------------------------+---------------------------+\n");
}

void usr_int()
{
        /* interrupt handler for user generated interrupt from */
        /* keyboard */

        printf("\n--- Interrupt: resume (y or n)? ");
        READLN
        if (yesno(inbufr) == 1) {
                printf("\n--- return to prompt (y or n)?");
                READLN
                if (yesno(inbufr) == 1)
                        do_abort();
                longjmp(cmd_cont,1);
        }
}

void do_select()
{
        extern int errno;
        register int i,j;
        register char *c,*d;
        short cmp,cerrs=0;
        char args[6][35];
        int strlen();
        DOM *cdom;
```

```
        c = inbufr;
        for (i = 0; i < 6; i++) {
                d = args[i];
                while (*c != ' ' && *c != '\t' && *c != '\0')
                        *d++ = *c++;
                *d = '\0';

                while((*c != '\0') && (*c == ' ' || *c == '\t'))
                        c++;
        }

        /* adjust size of each argument individually */

        /* 1 is relname */
        c = args[1];
        i = strlen(c);
        c += i;
        while (i++ < (MAXRNMLEN - 1))
                *c++ = ' ';
        *c = '\0';

        /* 2 is domain name */
        c = args[2];
        i = strlen(c);
        c += i;
        while (i++ < (MAXDNMLEN - 1))
                *c++ = ' ';
        *c = '\0';

        /* 3 is comare operator */
        c = args[3];
        i = strlen(c);
        switch (i) {
        case 1:
                switch(*c) {
                case '=':
                        cmp = CMPATT_EQ;
                        break;
                case '<':
                        cmp = CMPATT_LT;
                        break;
                case '>':
                        cmp = CMPATT_GT;
                        break;
                }
                break;
        case 2:
                switch (*c) {
                case '>':
                        if (*(c + 1) == '=')
                                cmp = CMPATT_GT | CMPATT_EQ;
                        else {
                                printf("XXX unknown comparison \"%c%c\"\n",c,(c + 1));
                                ++cerrs;
                        }
                        break;
```

```
                case '<':
                        if (*(c + 1) == '=')
                                cmp = CMPATT_LT | CMPATT_EQ;
                        else {
                                printf("XXX unknown comparison \"%c%c\"\n",c,(c + 1));
                                ++cerrs;
                        }
                        break;
                }
                default:
                        printf ("XXX unknown comparison \"%s\"\n",c);
                        ++cerrs;
        }

        /* 4 is attribute value */
        c = args[4];
        cdom = finddom(args[2]);
        if (cdom != (DOM *)(0)) {
                i = strlen(c);
                c += i;
                while (i++ < ((cdom -> len) -1 ))
                        *c++ = ' ';
                *c = '\0';
        }
        else {
                printf("XXX cannot find domain \"%s\"\n",args[2]);
                ++cerrs;
        }

        /* 5 is output relation name */
        c = args[5];
        i = strlen(c);
        c += i;
        if (!i) {
                printf("XXX missing output relation name\n");
                ++cerrs;
        }
        else {
                while (i++ < (MAXRNMLEN - 1))
                        *c++ = ' ';
                *c = '\0';
        }

        if (!cerrs)
                select(args[1],args[2],args[4],cmp,args[5]);
}

void do_project()
{
        extern int errno;
        register int i,j;
        register char *c,*d;
        short cerrs=0;
        char args[4][35],domlist[1][MAXDNMLEN];
        int strlen();
```

```
        c = inbufr;
        for (i = 0; i < 4; i++) {
                d = args[i];
                while (*c != ' ' && *c != '\t' && *c != '\0')
                        *d++ = *c++;
                *d = '\0';

                while((*c != '\0') && (*c == ' ' || *c == '\t'))
                        c++;
        }

        /* adjust size of each argument individually */

        /* 1 is relname */
        c = args[1];
        i = strlen(c);
        c += i;
        while (i++ < (MAXRNMLEN - 1))
                *c++ = ' ';
        *c = '\0';

        /* 2 is domain name */
        c = args[2];
        i = strlen(c);
        c += i;
        while (i++ < (MAXDNMLEN - 1))
                *c++ = ' ';
        *c = '\0';
        strcpy(domlist[0],args[2]);

        /* 3 is output relation name */
        c = args[3];
        i = strlen(c);
        c += i;
        if (!i) {
                printf("XXX missing output relation name\n");
                ++cerrs;
        }
        else {
                while (i++ < (MAXRNMLEN - 1))
                        *c++ = ' ';
                *c = '\0';
        }

        if (!cerrs)
                project(args[1],domlist,1,args[3]);
}

void do_list()
{
        register int i,j;
        extern DBSTATUS sys_db;
        extern struct rstat sysrelstat;
        extern REL relcore[];

        printf("\fDatabase \"%s\"[",sys_db.dbs_name);
```

```
        printf("%d relations, %d domains]\n",sysrelstat.numrelents,
                sysrelstat.numdoments);

        printf("\n\nRealtions:\n");
        for (i = 0; i < sysrelstat.numrelents; i++)
                printf("\t%10s\t%5d tuples\n",relcore[i].relname,
                        (relcore[i].cursize / relcore[i].width));

]

void do_convert()
{
        extern int errno;
        register int i,j;
        register char *c,*d;
        int strlen(),shuffle();
        char args[5][35];
        short cerrs=0;
        REL *thisrel,*findrel();

        c = inbufr;
        for (i = 0; i < 4; i++) {
                d = args[i];
                while (*c != ' ' && *c != '\t' && *c != '\0')
                        *d++ = *c++;
                *d = '\0';

                while((*c != '\0') && (*c == ' ' || *c == '\t'))
                        c++;
        ]

        /* adjust size of each argument individually */

        /* 1 is relname */
        c = args[1];
        i = strlen(c);
        c += i;
        while (i++ < (MAXRNMLEN - 1))
                *c++ = ' ';
        *c = '\0';

        /* 3 is output relation name */
        c = args[3];
        i = strlen(c);
        c += i;
        if (!i) {
                printf("XXX missing output relation name\n");
                ++cerrs;
        ]
        else {
                while (i++ < (MAXRNMLEN - 1))
                        *c++ = ' ';
                *c = '\0';
        ]

        if ((thisrel = findrel(args[1])) == FAIL){
```

```
                ++cerrs;
        ]

        if (!cerrs)
                switch (args[2][0]) {
                /*
                case 'f':
                        break;
                */
                case 'z':
                        shuffle(args[1],(char *)(0),args[3]);
                        break;
                case 'b':
                        mkindex(thisrel,4096,60,4096,70);
                        break;
                default:
                        printf("XXX unknown conversion mode \"%c\"\n",*c);
                ]
}
```

```
#include <errno.h>
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>
#include <math.h>
#include <sys/file.h>
#ifdef BSD
#include <sys/types.h>
#include <sys/time.h>
#include <sys/dir.h>
#include <sys/stat.h>
#endif
#ifdef OLDUNIX
#include <sys/types.h>
#include <sys/timeb.h>
#include <dir.h>
#include <stat.h>
#endif

#define DBHOMEDIR "/u4/charles/mrds"
#define MAXRNMLEN 11
#define MAXDNMLEN MAXRNMLEN
#define XMASK 00600
#define DMASK 00700
#define FAIL -1
#define MAXTYPES 9
#define MAXRELS 85
#define MAXDOMS 128
#define MAXRDS 340

char inbufr[512],outbufr[512],*inptr;

#define READLN  inptr = inbufr; \
        while ((( *inptr = getc(stdin)) != EOF) && \
                (*inptr != 0x0a)) inptr +=1; \
                *inptr = '\0';

typedef struct relrcrd {                /* rel record structure        */
   char relname[MAXRNMLEN];                /* relation name            */
   char mode;                              /* flag for const rels      */
   unsigned short width;                   /* parms of rel             */
   short fd;                               /* filedes if open          */
   long Zmap,cursize,maxsize,rindx,windx; /* more parms of rel         */
} REL;

typedef struct domrcrd {                /* dom record structure        */
   char domname [MAXDNMLEN];               /* name of domain           */
   char spare;                             /* spare byte : speed!      */
   short domtype;                          /* data type of domain      */
   unsigned short len;                     /* length of domain bytes   */
} DOM;

typedef struct rdrcrd {                 /* rd record structure         */
   char relname[MAXRNMLEN],
        domname[MAXDNMLEN];                /* name of rel and dom      */
   short pos;                              /* position within tuple    */
```

```
} RD;

REL relcore[50];
DOM domcore[50];
RD rdcore[150];
int rel,dom,rd,cur_rel,cur_dom,cur_rd;
main()
{
        extern int errno;
        register int i,j;
        register char *c,*d,*e;
        int line,numofdoms[50],rmxsize;
        int filemask,errs=0,doms,curpos,thisrel,k,out;
        int spintf(),open(),close(),strlen(),mkdir(),domlen();
        char dbname[MAXNAMLEN],rname[MAXRNMLEN],dname[MAXDNMLEN];
        char *krfgets(),*index(),*strcpy();
        FILE *fp,*fopen();

        /* get dbname */

        printf("Enter name of database to create: ");
        READLN
        if ((i = strlen(inbufr)) > MAXNAMLEN) {
                inbufr[MAXNAMLEN] = '\0';
                printf("XXX too long; truncated to \"%s\"\n",inbufr);
        }

        strcpy(dbname,inbufr);

        sprintf(inbufr,"%s/%s",DBHOMEDIR,dbname);
        if ((mkdir(inbufr,DMASK) )== FAIL) {
                printf("XXX [%d] cannot make home directory \"%s\"\n",errno,inbufr);
                exit();
        }

        /* create system relations in home directory */

        filemask = O_CREAT | O_RDWR;
        sprintf(inbufr,"%s/%s/.rel",DBHOMEDIR,dbname);
        if ((rel = open(inbufr,filemask,XMASK)) == FAIL) {
                printf("XXX [%d] open fail on \"%s\"\n",errno,inbufr);
                ++errs;
        }

        sprintf(inbufr,"%s/%s/.dom",DBHOMEDIR,dbname);
        if ((dom = open(inbufr,filemask,XMASK)) == FAIL) {
                printf("XXX [%d] open fail on \"%s\"\n",errno,inbufr);
                ++errs;
        }

        sprintf(inbufr,"%s/%s/.rd",DBHOMEDIR,dbname);
        if ((rd = open(inbufr,filemask,XMASK)) == FAIL) {
                printf("XXX [%d] open fail on \"%s\"\n",errno,inbufr);
                ++errs;
        }
```

```
        if (errs) exit();

        mksysrels();

        /* begin interactive section */

        cur_rel = 3; cur_dom = 13; cur_rd = 15;

        while(1) { /* get relations */
                printf("Enter relation name %2d: ",cur_rel - 2);
                READLN
                j = strlen(inbufr);
                inbufr[MAXRNMLEN - 1] = '\0';
                if (j == 0) break; /* no more rels */
                if (j > MAXRNMLEN)
                        printf("XXX too long; trunc to \"%s\"\n",inbufr);

                /* adjust to precise length */

                while (j < MAXRNMLEN - 1)
                        inbufr[j++] = ' ';
                inbufr[j] = '\0'; /* paranoia check */
                strcpy(relcore[cur_rel].relname,inbufr);
gettplcnt:
                printf("\tmax number of tpls? ");
                READLN
                if (((i = atoi(inbufr)) < 0) || (i > 65535)) {
                        printf("XXX bad num tpls %d\n",i);
                        goto gettplcnt;
                }
                rmxsize = i;
                doms = 0; curpos = 0;
                while (1) { /* get domains */
                        printf("\tEnter domain name %d: ",doms + 1);
                        READLN
                        j = strlen(inbufr);
                        if (j == 0) break; /* no more doms */
                        inbufr[MAXDNMLEN - 1] = '\0';
                        if (j > MAXDNMLEN)
                                printf("XXX too long; trunc to \"%s\"\n",inbufr);
                        /* adjust to precise length */
                        while (j < MAXDNMLEN - 1)
                                inbufr[j++] = ' ';
                        inbufr[j] = '\0';
                        strcpy(domcore[cur_dom].domname,inbufr);
gtyp:
                        printf("\t\ttype? ");
                        READLN
                        if (((i = atoi(inbufr)) < 0) || (i > MAXTYPES)) {
                                printf("XXX unknown type %d\n",i);
                                goto gtyp;
                        }

                        domcore[cur_dom].domtype = (short)(i);
gtlen:
                        printf("\t\tlen? ");
```

```
                        READLN
                        if (((i = atoi(inbufr)) < 0) || (i > 65535)) {
                                printf("XXX bad len %d\n",i);
                                goto gtlen;
                        }

                        domcore[cur_dom].len = (short)(i);
                        domcore[cur_dom].spare = 7; /* temp */

                        strcpy(rdcore[cur_rd].relname,relcore[cur_rel].relname);
                        strcpy(rdcore[cur_rd].domname,domcore[cur_dom].domname);
                        rdcore[cur_rd].pos = curpos;
                        curpos += domcore[cur_dom].len;
                        ++cur_dom;
                        ++cur_rd;
                        ++doms;
                        ++numofdoms[cur_rel];
                } /* end getting domains loop */

                relcore[cur_rel].mode = '\0';
                relcore[cur_rel].width = (short)(curpos);
                relcore[cur_rel].fd = -1;
                relcore[cur_rel].Zmap = 0L;
                relcore[cur_rel].maxsize = (long)(rmxsize * curpos);
                relcore[cur_rel].rindx = 0L;
                relcore[cur_rel].windx = 0L;

                ++cur_rel;

        } /* end getting relations loop */

        printf("\nMaking database \"%s\" with %d user relations\n",dbname,cur_rel - 3);

        /* now build it! */

        errs = 0;
        for (thisrel = 3; thisrel < cur_rel; thisrel++) {
                printf("Filename for relation \"%s\" ? ",relcore[thisrel].relname);
                READLN
                inbufr[MAXNAMLEN] = '\0';
                if ((fp = fopen(inbufr,"r")) == 0){
                        printf("XXX [%d] open fail \"%s\"\n",errno,inbufr);
                        ++errs;
                        continue;
                }

                /* open output file */

                strcpy(outbufr,relcore[thisrel].relname);
                if ((c = index(outbufr,' ')) != 0)
                        *c = '\0';
                sprintf(inbufr,"%s/%s/%s",DBHOMEDIR,dbname,outbufr);
                if ((out = open(inbufr,filemask,XMASK)) == FAIL) {
                        printf("XXX [%d] open fail for output rel file \"%s\"\n",errno,inbufr);
                        fclose(fp);
                        ++errs;
```

```
                        continue;
                }

                line = 1;
                while (*krfgets(inbufr,(2 * relcore[thisrel].width),fp)) {
                        i = 0;
                        d = inbufr;
                        e = outbufr;
                        *e = '\0';
                        while (i < numofdoms[thisrel]) {
                                if ((c = index(d,':')) == 0) {
                                        printf("XXX missing field mark, line %d\n",line);
                                        continue;
                                }
                                k = domlen(relcore[thisrel].relname,i);
                                strncat(e,d,(c -d));
                                j = strlen(e);
                                e += j; --k;
                                while (j < k) {
                                        *e++ = ' '; /* UGH! only good for char domtypes */
                                        ++j;
                                }
                                *e++ = '\0'; *e = '\0';
                                d = c + 1;
                                ++i;
                        } /* done this line */

                        if (1 /*!errs */) /* temp */
                                if (write(out,outbufr,relcore[thisrel].width) != relcore[thisrel].width){
                                        printf("XXX [%d] write fail: out tpl\n",errno);
                                        ++errs;
                                }
                ++line;
                } /* end this file: another relation is done */
                relcore[thisrel].cursize = relcore[thisrel].width * --line ;

        } /* end for each relation find a file */

        /* write out in core system relations */

        lseek(rel,0L,0);
        lseek(dom,0L,0);  /* rewind system rels */
        lseek(rd,0L,0);

        relcore[0].cursize += (cur_rel - 3) * relcore[0].width;
        relcore[1].cursize += (cur_dom - 13) * relcore[1].width;
        relcore[2].cursize += (cur_rd - 15) * relcore[2].width;

        for (i = 0; i < cur_rel; i++)
                if (write(rel,&relcore[i],sizeof(REL)) != sizeof(REL)){
                        printf("XXX write fail rel.%s\n",relcore[i].relname);
                        ++errs;
                }

        for (i = 0; i < cur_dom; i++)
                if (write(dom,&domcore[i],sizeof(DOM)) != sizeof(DOM)){
```

```
                        printf("XXX write fail dom.%s\n",domcore[i].domname);
                        ++errs;
                }

        for (i = 0; i < cur_rd; i++)
                if (write(rd,&rdcore[i],sizeof(RD)) != sizeof(RD)){
                        printf("XXX write fail rd.%s\n",rdcore[i].relname);
                        ++errs;
                }


        printf("Database \"%s\" created\nBye.\n",dbname);
} /* end main */
mksysrels()
{
        register int i,j,errs=0;
        int write();

        strcpy(relcore[0].relname,"rel       ");
        relcore[0].mode = '\0';
        relcore[0].width = sizeof(REL);
        relcore[0].fd = -1;
        relcore[0].Zmap = 0L;
        relcore[0].cursize = 108L;
        relcore[0].maxsize = (long)(MAXRELS * sizeof(REL));
        relcore[0].rindx = 0L;
        relcore[0].windx = 0L;

        strcpy(relcore[1].relname,"dom       ");
        relcore[1].mode = '\0';
        relcore[1].width = sizeof(DOM);
        relcore[1].fd = -1;
        relcore[1].Zmap = 0L;
        relcore[1].cursize = 208L;
        relcore[1].maxsize = (long)(MAXDOMS * sizeof(DOM));
        relcore[1].rindx = 0L;
        relcore[1].windx = 0L;

        strcpy(relcore[2].relname,"rd        ");
        relcore[2].mode = '\0';
        relcore[2].width = sizeof(RD);
        relcore[2].fd = -1;
        relcore[2].Zmap = 0L;
        relcore[2].cursize = 360L;
        relcore[2].maxsize = (long)(MAXRDS * sizeof(RD));
        relcore[2].rindx = 0L;
        relcore[2].windx = 0L;

        strcpy(domcore[0].domname,"relname   ");
        domcore[0].domtype = 0;
        domcore[0].len = MAXRNMLEN;

        strcpy(domcore[1].domname,"mode      ");
        domcore[1].domtype = 2;
        domcore[1].len = 1;
```

```
        strcpy(domcore[2].domname,"width     ");
        domcore[2].domtype = 4;
        domcore[2].len = 2;

        strcpy(domcore[3].domname,"fd        ");
        domcore[3].domtype = 4;
        domcore[3].len = 2;

        strcpy(domcore[4].domname,"Zmap      ");
        domcore[4].domtype = 5;
        domcore[4].len = 4;

        strcpy(domcore[5].domname,"cursize   ");
        domcore[5].domtype = 5;
        domcore[5].len = 4;

        strcpy(domcore[6].domname,"maxsize   ");
        domcore[6].domtype = 5;
        domcore[6].len = 4;

        strcpy(domcore[7].domname,"rindx     ");
        domcore[7].domtype = 5;
        domcore[7].len = 4;

        strcpy(domcore[8].domname,"windx     ");
        domcore[8].domtype = 5;
        domcore[8].len = 4;

        strcpy(domcore[9].domname,"domname   ");
        domcore[9].domtype = 0;
        domcore[9].len = MAXDNMLEN;

        strcpy(domcore[10].domname,"domtype   ");
        domcore[10].domtype = 4;
        domcore[10].len = 2;

        strcpy(domcore[11].domname,"len       ");
        domcore[11].domtype = 4;
        domcore[11].len = 2;

        strcpy(domcore[12].domname,"pos       ");
        domcore[12].domtype = 4;
        domcore[12].len = 2;

        strcpy(rdcore[0].relname,"rel       ");
        strcpy(rdcore[0].domname,"relname   ");
        rdcore[0].pos = 0;

        strcpy(rdcore[1].relname,"rel       ");
        strcpy(rdcore[1].domname,"mode      ");
        rdcore[1].pos = 11;

        strcpy(rdcore[2].relname,"rel       ");
        strcpy(rdcore[2].domname,"width     ");
        rdcore[2].pos = 12;
```

```
        strcpy(rdcore[3].relname,"rel       ");
        strcpy(rdcore[3].domname,"fd        ");
        rdcore[3].pos = 14;

        strcpy(rdcore[4].relname,"rel       ");
        strcpy(rdcore[4].domname,"Zmap      ");
        rdcore[4].pos = 16;

        strcpy(rdcore[5].relname,"rel       ");
        strcpy(rdcore[5].domname,"cursize   ");
        rdcore[5].pos = 20;

        strcpy(rdcore[6].relname,"rel       ");
        strcpy(rdcore[6].domname,"maxsize   ");
        rdcore[6].pos = 24;

        strcpy(rdcore[7].relname,"rel       ");
        strcpy(rdcore[7].domname,"rindx     ");
        rdcore[7].pos = 28;

        strcpy(rdcore[8].relname,"rel       ");
        strcpy(rdcore[8].domname,"windx     ");
        rdcore[8].pos = 32;

        strcpy(rdcore[9].relname,"dom       ");
        strcpy(rdcore[9].domname,"domname   ");
        rdcore[9].pos = 0;

        strcpy(rdcore[10].relname,"dom       ");
        strcpy(rdcore[10].domname,"domtype   ");
        rdcore[10].pos = 11;

        strcpy(rdcore[11].relname,"dom       ");
        strcpy(rdcore[11].domname,"len       ");
        rdcore[11].pos = 13;

        strcpy(rdcore[12].relname,"rd        ");
        strcpy(rdcore[12].domname,"relname   ");
        rdcore[12].pos = 0;

        strcpy(rdcore[13].relname,"rd        ");
        strcpy(rdcore[13].domname,"domname   ");
        rdcore[13].pos = 11;

        strcpy(rdcore[14].relname,"rd        ");
        strcpy(rdcore[14].domname,"pos       ");
        rdcore[14].pos = 22;

        /* write out the entries */

        for (i = 0; i < 3; i++)
                if (write(rel,&relcore[i],sizeof(REL)) != sizeof(REL)) {
                        printf("XXX [%d] write fail: rel\n",errno);
                        ++errs;
                }
```

```
        for (i = 0; i < 13; i++)
                if (write(dom,&domcore[i],sizeof(DOM)) != sizeof(DOM)) {
                        printf("XXX [%d] write fail: dom\n",errno);
                        ++errs;
                }

        for (i = 0; i < 15; i++)
                if (write(rd,&rdcore[i],sizeof(RD)) != sizeof(RD)) {
                        printf("XXX [%d] write fail: rd\n",errno);
                        ++errs;
                }

        if (errs) {
                printf("XXX giving up.\nBye.\n");
                exit();
        }
}

char *krfgets(s,n,iop)
char *s;
int n;
FILE *iop;
{
        register int c;
        register char *cs;

        cs = s;
        while (--n > 0 && (c = getc(iop)) != EOF)
                if ((*cs++ = c) == '\n')
                        break;

        *cs = '\0';

        return((c == EOF && cs == s) ? NULL : s);
}
int domlen(rel,domnum)
char *rel;
int domnum;
{
        register int i,j;
        register char *a,*b;
        int strcmp(),cumlen=0;

        j = 0;
        for (i = 15; i < cur_rd; i++) {
                j = strcmp(rel,rdcore[i].relname);
                if (!j) {
                        j = i;break;
                }
                else j = 0;
/*
                if (! strcmp(rel,rdcore[i].relname)){
                        j = i;
                        break;
                }
*/
```

```
        }

        if (!j) {
                printf("XXX missing entry in rd for rel \"%s\"\n",rel);
                return(0);
        }

        i = 0;
        while(i <= domnum) {
                j += i ;
                cumlen = rdcore[j].pos;
                ++i;
        }

        /* find len of domain rdcore[j].domname */

        for (i = 13; i < cur_dom; i++)
                if (! strcmp(rdcore[j].domname,domcore[i].domname))
                        return(domcore[i].len);

        printf("XXX missing entry in dom for domain \"%s\"\n",rdcore[j].domname);
        return(0);
}
```

Chapter 6

6: Concluding Notes

6.1 Restrictions and Shortcomings

Considerable effort was expended in the design and implementation of MRDSc to introduce few restrictions. The few inevitable restrictions that are present are not felt to be constraining, *e.g.*, no tuple may have more than 32 attributes, a tuple cannot be longer than 8 Kbytes and a relation no larger than 2^{32} tuples.

One implementation restriction which is potentially constrictive is the requirement of all tuples to be of the same length within a relation. Support for variable length tuples is not difficult to integrate into MRDSc ,but will be labour intensive. rd will no longer be able to provide offsets to attribute starting positions within a tuple, hence each tuple must be preceded by an array of 16 bit unsigned integers containing these offsets. This increases the size of the relation by *at least* $2mn$, (m = attributes per tuple, n = number of tuples). Byte offsets will be inadequate since a tuple may exceed 256 bytes in length before the beginning of its last attribute. Since each attribute is now pointed at, it can be correctly aligned for its datatype, thereby simplifying and expediting procedures like search and tplcmp. In addition, each tuple must now begin on a 16 bit boundary. Thus the relation will grow further because of space lost to alignment padding.

Some of the problems to overcome in adding support for variable length tuples include: (1) replacing all instances of tuple width determination, now taken from rptr -> width, with a macro which calculates the length from the array preceding the tuple, (2) buffer management, since a tuple's size is unknown before it is read, (3) new data structure(s) to hold, *e.g.*, the array preceding the tuple, (4) changes to the data type support to accommodate variable length attributes (*e.g*, introduction of a new data length of 0 to indicate variable length).

While the flexibility made possible with variable length tuples is attractive, it is expensive. Except in situations where it is genuinely valuable, fixed length tuples will provide better performance without necessarily using more disk space. MRDSc should not forget how to deal with fixed length tuples as it learns how to handle variable length ones.

If speed is truly a consideration, then there are some parts of MRDSc which should be implemented in assembler. Portions of tplcmp, cmpseptup, and support for z-ordering could be speeded up tremendously were they written at a lower level, since much of what they do is low level.

A major weakness of the current implementation is its concept of user access control. There is room for much improvement in the support for database/relation ownership, sharable/unsharable databases and relations and views and integrity checking at run time.

Finally, the dearth of dyadic operators renders the current implementation impotent -- it is in desperate need of code to implement joins.

6.2 A Different Approach

Stonebraker [Ston80] points out in his INGRES retrospective that he feels it would have been better to have implemented the system using special purpose hardware and file systems. While the OS underlying both INGRES and MRDSc provides an excellent development environment, it cannot simultaneously provide the full power of which the hardware is capable to the application.

The types of operations typically performed on relational databases can be adequately handled by general purpose hardware/instruction set combinations, but would be more effectively performed by a processor whose architecture is tailored specifically to relational operations. Similarly, file system layouts suited to general purpose computing manage relations adequately, but specialized file systems can do much better. Further, associative memory is clearly going to outperform conventional RAM on many relational operations.

While special purpose hardware and file systems are attractive from a performance standpoint, their cost and the unavailability of commonplace software (*e.g.*, operating systems, compilers) often preclude their use in a development environment. A compromise is to design a modularized *relational virtual machine*, RVM (Figure 6.1), wherein each module can be developed as a virtual component on a conventional timesharing system. A virtual component can be replaced at any time with real hardware performing the same function, without affecting other parts of the RVM which remain virtual.

The major components of the RVM are:

- Relational Processing Unit (RPU) which is a special purpose processor architecturally tailored to relational operations, having its own relational instruction set (RIS),
- Relational Memory Unit (RMU) containing ordinary RAM and a multi-way associative memory, with access control logic,
- Relational File System (Relfs) on the Relational I/O Bus to which may be attached various physical storage volumes supporting different access strategies (logarithmic, direct, *etc.*).

User access, interactive or batched, takes place through user interfaces which connect to the RPU and issue to it a stream of RIS instructions. The results of action so initiated are communicated back through the invoking interface. Embodied within the typical user interface would be a terminal driver, *e.g.*, support for bit mapped screens, a query language processor and error/exception handling facilities.

This high degree of modularization is the primary theme of this alternative implementation scheme. It makes possible a versatile, elegant, and open-ended implementation of a relational database system. User interfaces can be anything from simple personal computers to large mainframes attached through any kind of communications method, and located anywhere. The components of the RPU and RMU can be distributed across several machines without affecting the appearance of the RVM to outside users.

Further study of the idea of implementing a relational database system on a modularized, distributed virtual machine lies beyond the scope of the current project, but should be pursued.

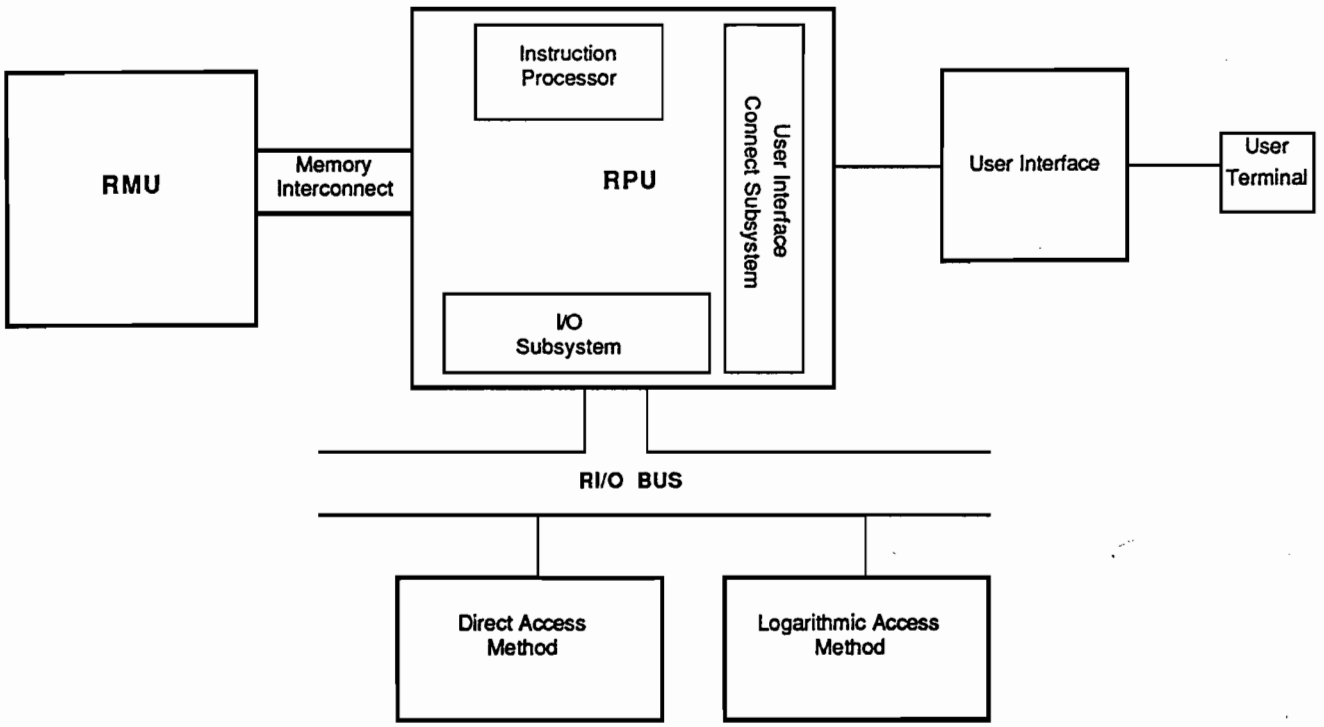


Figure 6.1: Organization of the Relational Virtual Machine

Bibliography

Bayer, R., and E. McCreight, "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica*, **1**, 173-189, 1972

Bayer, R., and K. Unteraur, "Prefix *B*-Trees", *ACM Transactions on Database Systems*, Vol 2 No 1, 11-26, 1977

Blasgen, M. W., and K. P. Eswaran, "Storage and Access in Relational Data Bases", *IBM Systems Journal*, Vol 16 No 4, 363-377, 1977

Blasgen, M. W., R. G. Casey and K. P. Eswaran, "An Encoding Method for Multifield Sorting and Indexing", *Communications of the ACM*, Vol **20** No 11, 874-878, 1977

Blasgen, M. W. *et. al.*, "System R: An Architectural Overview", *IBM Systems Journal*, Vol **20** No 1, 42-62, 1981

Chiu, G., "MRDSA User's Manual", *School of Computer Science Technical Report* **SOCS82.9**, McGill University, 1982

Codd, E. F., "A Relational Model Of Data For Large Shared Data Banks", *Communications of the ACM*, Vol **13**, No 6, 377-87, 1970

Codd, E. F., [1971a], "Relational Completeness of Data Base Sublanguages", in *Data Base Systems*, R. Rustin, editor, Prentice-Hall Publishing Co., 34-64, Englewood Cliffs, N.J., 1972

Codd, E. F., [1971b], "A Data Base Sublanguage Founded On The Relational Calculus", *SIGFIDET* **71**, 35-68

Date, C. J., *An Introduction to Database Systems*, second edition, Addison Wesley Publishing Co., 1977

Horspool, R. N. S., *C Programming in the Berkely UNIX Environment*, Prentice Hall Canada Inc., 1986

Joy, W., *et. al.*, "4.2BSD System Manual (Revised July 1983)", documentation accompanying the 4.2BSD UNIX Release, 1983

Kerninghan, B. W. and R. Pike, *The UNIX Programming Environment* , Prentice Hall Inc., 1984

McKusick, M. K., *et. al.*, "A Fast File System for UNIX (Revised July 27, 1983", documentation accompanying the 4.2BSD UNIX Release, 1983

McKusick, M. K., "Fsck - The UNIX File System Check Program (Revised July 28, 1983)", documentation accompanying the 4.2BSD UNIX Release, 1983

Merrett, T. H., *Relational Information Systems*, Reston Publishing Co. Inc., 1984

Orenstein, J. A., "Algorithms and Data Structures for the Implementation of a Relational Database", *School of Computer Science Technical Report* **SOCS8217**, McGill University, 1982

Scheuermann, P., and M Ouksel, "Multidimensional *B*-Trees For Associative Searching In Database Systems", *Information Systems* Vol **7** No 2, 123-137, 1982

Stonebraker, M., *et. al.*, "The Design and Implementation of INGRES", *ACM Transactions on Database Systems* Vol **1** No 3, 189-222, 1976

Stonebraker, M., "Retrospection on a Database System", *ACM Transactions on Database Systems* Vol **5** No 2, 225-240, 1980

Wedekind, H., "On The Selection Of Access Paths In A Data Base System", in *Proceedings of the IFIP Working Conference On Database Management,* J. W. Klimbie and K. L. Koffeman, editors, North-Holland Publishing Co., 385-397, Amsterdam, 1974

Appendix 1

Effect On Storage Utilization & Performance

Of Data Alignment

An inherent characteristic of current digital computers is a data bus of a particular bit width (currently popular sizes are 16, 32 and more). Data transfers on these buses, usually synchronous, occur between two bus addresses, or a bus address and a CPU register, and range in size from a single byte up to full bit-width per bus cycle. The bus control and arbitration hardware, along with the bus itself, provide optimal transfer speeds only when the data unit being transferred has as both origin and destination an address which is an integral multiple of the data unit's width. Indeed, most processors, including VAX and Motorola, are incapable of data transfers where this condition is not met.

The index portion of a simple prefix B^* tree contains pages of fixed size containing:

$$p_0,(s_1,p_1),\ \cdots\ ,(s_{2k},p_{2k})$$

In supporting various data types, the separators (s_i) may have varying lengths. Further, in a dynamic index accommodating insertions and deletions, it is reasonable for the (s_i,p_i) to be maintained in a linked list within each page. Thus, a single (s_i,p_i) entry takes on the form:

```
struct   spentry {
             unsigned short   sep_len;        /* num bits in separator */
             char             sep[varying];   /* actual separator */
             long             ptr_to_child;   /* ptr to node in lower level */
             unsigned short   nxt_ent;        /* next mem on linked list */
}
```

Separators are aligned to the nearest byte boundary by right padding with 0 bits.

In reading such an index to locate a datum (*e.g.* a tuple), it may be necessary to examine many (s_i,p_i) entries. To examine a single entry requires being able to extract the fields correctly, *i.e.* successfully to extract two short integers and a long integer (2 bytes and 4 bytes on most machines). If a branch page contains (s_i,p_i) entries "streamed" into the linked list, that is appearing exactly as shown above, the address alignment of the short and long integers is not predictable. Hence, processing an entry entails consecutive byte-by-byte accesses and subsequent type coercion manipulated as, for example, to get the value for ptr_to_child:

```
char    *a;
long    ptr_to_child;

union   tolong {
            long    l_val;
            char    c1,c2,c3,c4;
} long_coerce;

long_coerce.c1 = *a;
*(&(long_coerce.c1) + 1) = *(a + 1);
```

```
*(&(long_coerce.c1) + 2) = *(a + 2);
*(&(long_coerce.c1) + 3) = *(a + 3);
a += sizeof(long);
ptr_to_child = long_coerce.lval;
```

The alternative to this type coercion overhead is to guarantee that the three integer fields of (s_i, p_i) are *always* aligned; this can only be accomplished by introducing pad bytes into the entry. Thus, from 0 to 3 additional bytes *per entry* are introduced, appended to the separator, so that the long integer ptr_to_child will always appear on a 4-byte boundary. Aligning this one field assures that both short integer fields are also correctly aligned.

Thus, to the problem associated with the data type alignment there are two solutions: one costing in processing overhead, the other in disk space. In the course of investigating theses costs and their associated benefits, other interesting points came to light concerning procedure call overhead and compiler code generation.

To test the two access strategies (referred to hereafter as "unaligned" and "aligned") two test files were constructed, each containing 10,000 (s_i, p_i) entries. For the purpose of these tests only branch pages were considered, and each such page contained (s_i, p_i) entries exclusively (no p_0, no header information). A page size of 4K bytes was used throughout, chosen as being both a typical branch page size and the optimal block size for Unibus disk data transfers (from an Ampex 9300 disk drive) on a VAX 11/780 running 4.2BSD UNIX.

The (s_i, p_i) entries were generated using *random()* to generate separators of varying length. Attention was focused on test files with average separator lengths of 3, 5, 6, 10, 15, and 20 bytes. In each test, two data files were generated using the same separator length sequences: one file containing pad bytes to assure data alignment, the other having entries streamed across the page without regard to alignment. No entry was permitted to cross a page boundary, thus in each file pages contained unused space 'at the ends'.

On the two test files a series of read tests was performed, consisting of block by block reads and processing of each (s_i, p_i) entry on the page. Observed for each read test were the user and system CPU times and the I/O count (blocks read). The data files were compared for overall difference in size. Based on these observations, the processing and disk performance of the two access strategies are assessed.

Table I shows, for the case of each different separator length, the relative file sizes and amount of "wasted space". For table columns containing two columns of numbers, the left column is for the aligned file, the right for the unaligned.

Avg. Sep. length	Filesize (pages)		End-Of-Block Padding (% totl)		Alignment Padding bytes	%	Page Capacity		Ratio
3	33	29	1.0	1.5	16,735	12.4	306	349	0.877
5	38	34	2.4	3.0	16,790	10.8	269	303	0.888
6	40	36	1.0	0.2	15,057	9.2	252	278	0.906

Effect On Storage Utilization & Performance Of Data Alignment

$$(\alpha x)^{h+1} = 1 - N(1 - \alpha x)$$

assuming N is large,

$$h = \frac{log(\alpha x - 1) + log(N)}{log(\alpha x)} - 1$$

for $\alpha = 1$, and x reasonably large, this reduces to

$$h = \frac{log(N)}{log(x)}$$

From this it is apparent that $log(N)$ will overshadow the effect of $log(\alpha x - 1)$ since αx is almost never as large as 1000, whereas N is seldom as small as 1000.

One way to compare tree flatness between aligned and unaligned data is to look for cases where the former case requires a higher tree than does the latter to contain the same number of indices. For the average separator lengths examined, in very few cases did aligned data necessitate a tree of one level more than unaligned data:

Avg Sep Len	$\log_{10}N$	$\lceil h_{align} \rceil$	$\lceil h_{unalign} \rceil$
3	10	5	4
10	7	4	3
10	14	7	6
20	15	8	7

Table VI: Cases Where Aligned Data Required Higher Tree

Table VII shows the effect on tree capacity for various values of h for an average separator length of 6:

h	Nodes In Tree Aligned	Unaligned	Ratio
1	253	279	0.907
2	63,757	77,563	0.822
3	16,066,765	21,562,515	0.745
4	4,048,824,781	5,994,379,170	0.675
5	1.020×10^{12}	1.666×10^{12}	0.612
6	2.571×10^{14}	4.633×10^{14}	0.555

Table VII: Aligned / Unaligned Tree Capacity

Figure 1 shows the effect on tree capacity for varying h for each of the different separator

From Table II average system and user processing times, in seconds, per page can be established as:

Avg Sep Len	Aligned utime	Unaligned utime	Aligned stime	Unaligned stime
3	0.008939	0.013517	0.007000	0.008966
5	0.008079	0.011676	0.006105	0.008147
6	0.007750	0.011333	0.005950	0.007000
10	0.006163	0.009733	0.008143	0.006400
15	0.004113	0.007814	0.007419	0.006153
20	0.002689	0.006657	0.008689	0.006529

Table VIII: User and System Processing Times Per Page

As one would expect, for increased separator size, user CPU time decreases as each page contains fewer entries to process and system CPU time increases as more pages are required to accommodate the entries.

For trees of varying heights, the times to process every entry in the tree are (average separator length = 6):

h	Aligned utime	Unaliged utime	Aligned stime	Unaligned stime
1	1.96	3.16	1.51	1.95
2	494.12	879.05	379.35	542.94
3	124,517.43	244,375.17	95,597.25	150,937.61
4	31,378,392.05	67,936,297.27	24,090,507.45	41,960,654.20
5	7,907,354,797.30	18,886,290,641.44	6,070,807,876.64	11,665,061,866.77
6	1,992,653,408,919.80	5,250,388,798,320.89	1,529,843,584,912.62	3,242,887,198,962.90

Table IX: User and System Tree Processing Times For Aligned and Unaligned Data

Comparing the times shown above indicates that aligned data was processed more quickly than unaligned data. But conflated in such a comparison is the fact that aligned data contained fewer entries per page to process. To produce times which are comparable, *i.e.* reflecting equivalent processing work, the times for aligned data have been 'corrected' as shown below for the case of average separator length 6, $h = 1$:

$$253 \text{ aligned nodes} \times 7.75 \times 10^{-3} \frac{\text{seconds}}{\text{block}} = 1.96 \text{ seconds}$$

$$253 \text{ aligned nodes} \times 252 \frac{\text{entries}}{\text{node}} = 63{,}756 \text{ entries}$$

A tree of unaligned nodes of same height contains:

$$279 \text{ aligned nodes} \times 278 \frac{\text{entries}}{\text{node}} = 77{,}562 \text{ entries}$$

Thus, the time that would have been required for an equivalent number of entries to be processed had they been aligned is:

$$1.96 \times \frac{77{,}562}{62{,}756} = 2.39 \text{ corrected seconds}$$

Profiled execution of the read programs showed that over 98% of the reported system time, in either the aligned or unaligned cases, was spent in the *read()* system call. This being the case, the procedure for correcting system times deals only with read times: multiply the system time per block for aligned data by the quotient of number of unaligned data entries divided by block capacity of aligned data blocks:

$$5.95 \times 10^{-3} \frac{\text{seconds}}{\text{aligned block}} \times \frac{77{,}562 \text{ data entries}}{252 \frac{\text{entries}}{\text{aligned block}}} = 1.83 \text{ corrected seconds}$$

Combining the user and system times to give total processing times, the corrected aligned *vs.* unaligned comparison becomes (for an average separator length of 6):

h	Corrected Aligned Total	Unaligned Total	Ratio
1	4.22	5.12	0.82
2	1,172.25	1,421.99	0.82
3	325,884.90	395,312.78	0.82
4	90,596,002.03	109,896,951.47	0.82

Effect On Storage Utilization & Performance Of Data Alignment

```
#define FILE1   "/u1/charles/mrds/test/file1"
#define MEMSIZE 4096
#define REPS 100
#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>

main()
{
        extern int errno;
        register int i,fd,j,k,m;
        char mem[MEMSIZE],*sep,out[50],strncpy();
        unsigned short len,next;
        long ptr,lseek();
        int open(),close(),read();

        /* open file for reading */

        if ((fd = open(FILE1,O_RDONLY)) < 0) {
                printf("Open fail [%d]\n",errno);
                exit(-1);
        }

        /* do some reading */

        for (i = 0; i < REPS; i++) {
                fprintf(stdout,"%3d...",i);fflush(stdout);
                while(read(fd,mem,MEMSIZE)) {
                        j = 0;
                        while((j < MEMSIZE) && mem[j+1] != '&') {
                                len = *(u_short *)(&mem[j]); j += 2;
                                sep = &mem[j]; j += 1;
```

```
Jul 14 19:56 1986  read4.c Page 1

#define FILE2   "/ul/charles/mrds/test/file2"
#define MEMSIZE 4096
#define REPS 100
#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>

union U_long {
        long lval;
        char c1,c2,c3,c4;
} cvtl;

union U_short {
        unsigned short sval;
        char c1,c2;
} cvts;

main()
{
        extern int errno;
        register int i,fd,j,k,m;
        char mem[MEMSIZE],*sep,out[50],strncpy();
        unsigned short len,next,getushort();
        long ptr,lseek(),getlong();
        int open(),close(),read();

        /* open file for reading */

        if ((fd = open(FILE2,O_RDONLY)) < 0) {
                printf("Open fail [%d]\n",errno);
                exit(-1);
        }

        /* do some reading */

        for (i = 0; i < REPS; i++) {
                fprintf(stdout,"%3d...",i);fflush(stdout);
                while(read(fd,mem,MEMSIZE)) {
                        j = 0;
                        while((j < MEMSIZE) && mem[j+1] != '&') {
        cvts.c1 = mem[j];
        *(&(cvts.c2) + 1) = mem[j + 1]; j += 2;
        len = cvts.sval;
/*                              len = getushort(&mem[j]); j +=2; */
                                sep = &mem[j]; j += len >> 3;
#ifdef TRACE
strncpy(out,sep,((len>>3) /* +((j % 4)?(4-(j%4)):0) */ ));
#endif
/*                              if (j % 4)
                                        j += (4 - (j % 4));
*/
/*                              ptr = getlong(&mem[j]); j += 4; */
        cvtl.c1 = mem[j];
        *(&(cvtl.c2) + 1) = mem[j + 1];
        *(&(cvtl.c3) + 2) = mem[j + 2];
        *(&(cvtl.c4) + 3) = mem[j + 3];
```

```
Jul 14 19:56 1986  read4.c Page 2

        ptr = cvtl.lval; j+= 4;
/*                              next = getushort(&mem[j]); j +=2; */
        cvts.c1 = mem[j];
        *(&(cvts.c2) + 1) = mem[j + 1]; j += 2;
        next = cvts.sval;
                                k++;
#ifdef TRACE
fprintf(stdout,"|%2d|%20s|%41d|%6d|\n",len>>3,out,ptr,next);
fflush(stdout);for(m = 0; m < 50;m++)out[m] = '\0';
#endif
                        }
                }
                lseek(fd,0L,0);
        }

        printf("done.");
        close(fd);
        printf("\nBye.\n");
}

unsigned short getushort(a)
char *a;
{

        cvts.c1 = *a;
        *(&(cvts.c2) + 1) = *(a + 1);
        return(cvts.sval);
}


long getlong(a)
char *a;
{
        cvtl.c1 = *a;
        *(&(cvtl.c2) + 1) = *(a + 1);
        *(&(cvtl.c3) + 2) = *(a + 2);
        *(&(cvtl.c4) + 3) = *(a + 3);
        return(cvtl.lval);
}
```

Appendix 2

Installation of MRDSc

Proper installation of MRDSc from tape requires following these steps:

(1) Make a directory *mrdsc* in some filesystem containing 5 Mbytes of available space. Not all of this space will be required following installation.

(2) cd to this directory and extract all files: tar x.

(3) The current directory should contain the subdirectories doc, src, and supp. cd to the src subdirectory and edit the Makefile, making any site specific changes to the CFLAGS:

-DVAX	define your CPU type as one of: AMDAHL, CADMUS, M68000, or VAX. If yours is none of these, use VAX and hope for the best.
-DBSD	identify your hosting OS: either a real BSD or else OLDUNIX

You may also remove the -g flag from both CLFAGS and loader commands if you do not foresee needing to use a symbolic debugger with the system. *Do not* run make yet!

(4) edit the file mrds.h adjusting the paths to these files to suit your site:

USRFILE	file containing list of authorized users by their UNIX uids
DBLIST	file containing list of databases known to exist (see section 3.5.3)
LOGFILE	file containing logged event reports of MRDSc activity
MERTEMP, BTEMP	file name templates used in mktemp() calls to generate names for potentially large temporary files

(5) run the Makefile now. When finished, do make lib to generate the library archive of MRDSc routines.

```
/*****************************************************************************
*                                                                            *
*            H E A D E R   F I L E   F O R   M R D S c                       *
*                                                                            *
*   This file contains all #defines used for MRDSc procedures                *
*   as well as many variable declarations/initializations, and               *
*   all relevant #includes.                                                  *
*                                                                            *
*   It must be included in every module of MRDSc in order for                *
*   compilation & linking to succeed.                                        *
*                                                                            *
*                                                                            *
*                                                                            *
*****************************************************************************/

#define VERSION "0.9  Oct 86"

/*   Files and filenames needed by MRDSc kernel                              */

#define USRFILE "/u4/charles/mrds/sysfiles/.sysusrs"
#define DBLIST  "/u4/charles/mrds/sysfiles/dblist"
#define LOGFILE "/u4/charles/mrds/sysfiles/syslog"
#define MERTEMP "/tmp/smXXXXXX"
#define BTEMP   "BXXXXXX"

/*    #include  needed by modules (kernel and library)                        */

#include <sys/errno.h>
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>
#include <math.h>
#include <sys/file.h>
#ifdef BSD
#include <sys/types.h>
#include <sys/time.h>
#include <sys/dir.h>
#include <sys/stat.h>
#endif
#ifdef OLDUNIX
#include <sys/types.h>
#include <sys/timeb.h>
#include <dir.h>
#include <stat.h>
#endif

#define SORTLIM 200
#ifdef CADMUS
#define INT16
#define MAXINT 0x7fff
#endif
#ifdef VAX
#define INT32
#define MAXINT 0x7fffffff
#endif
#ifdef M68000
```

```
#define INT32
#define MAXINT 0x7fffffff
#endif
#ifdef AMDAHL
#define INT32
#define MAXINT 0x7fffffff
#endif
#define MAXERR 25                       /* dflt max # of errs permitted*/
#define MAXSUBSTR 256                   /* max length of a substinrg   */
#define MXLENUID 4                      /* max  length of a uid        */
#define MXLINUID 30                     /* length of line in usrs file */
#define RDMODE "r"                      /* read mode for fopen         */
#define APMODE "a"                      /* append mode for fopen       */
#define CRMASK 00644                    /* file creation mask          */
#define BLANK " "                       /* field delim on input lines  */
#define BLANKC ' '                      /* char constant blank         */
#define NULLC '\0'                      /* char constant null          */
#define DOT "."                         /* character period            */
#define SLASH "/"                       /* character slash (dir paths) */
#define MAXARGS 6                       /* max num args on cmnd line   */
#define MAXIO 1000                      /* maxnum of page i/o reqs     */
#define DELIM " | "                     /* print field delimiter       */

#define READLN  inptr = inbufr; \
        while ((( *inptr = getc(stdin)) != EOF) && \
                (*inptr != EOL)) inptr +=1; \
                *inptr = NULLC;

static int IOCOUNT = 0;                 /* count i/o actions           */
static int RUN = 0;                     /* run mode setting & dflt     */
static char DIAGNOSE = 0x00;            /* dflt exec trace setting     */
static char DBNAME[MAXNAMLEN];          /* name of db; dflt is null$   */
static int relfd;                       /* file des for sysrel .rel    */
static int domfd;                               /* file des for sysrel .dom */
static int rdfd;                        /* file des for sysrel .rd     */
static char OLD = 0x00;                 /* flag for setup: old db      */
static char NEW = 0x01;                 /* flag for setup: new db      */
static char STRBUF[MAXSUBSTR];          /* array for substrs           */
static char EOL = 0x0a;                 /* end of line char            */
static char NOSTRING = 0x00;            /* empty text string           */
#define FAIL -1                         /* return code for failure     */
#define SUCCESS 0                       /* return code for success     */
#define FILESTRING 64                   /* max length of pathnames     */
#define Rel ".rel"                      /* string form of filename rel */
#define Dom ".dom"                      /* string form of filename dom */
#define Rd ".rd"                        /* string form of filename rd  */
#define MAXRNMLEN 11                    /* max+1 len of relation name  */
#define MAXDNMLEN MAXRNMLEN             /* max+1 len of domain name    */
#define PAGESIZE 2048                   /* for 2K pages                */
#define MAXOPNFILES 16                  /* maxnum simult open files    */
#define MINRDS 15                       /* min num of rd entries       */
#define MINDOMS 13                      /* min num of dom entries      */
#define MINRELS 3                       /* min num of rel entries      */
#define CONPGSIZE 256                   /* size of pages for cons rels */
#define MAXCONPAGE 15                   /* maxnum frames for cons rels */
#define MAXATTS 32                      /* max number of attributes    */
```

```
#define RONLY 0x1                       /* relation is read only       */

/*              bits in "rel" mode tag                                  */

#define CONREL 0x01                     /* relation is constant rel    */
#define ZORD   0x02                     /* some domains are in z order */
#define PZREL  0x04                     /* some, not all, atts are z   */
#define FLAT   0x08                     /* rel is flat, no tree        */
#define ORDER  0x10                     /* rel is ordered              */
#define BTREE  0x20                     /* rel is prefix B tree        */
#define APONLY 0x40                     /* rel is append only          */
#define WRINH  0x80                     /* rel is write inhibited      */

#define XLEAF  0x01                     /* xpage is a leaf             */
#define XFRSTLF 0x04                    /* xpage is first leaf pg      */
#define XROOT  0x02                     /* xpage is root page          */
#define XBRANCH 0x00                    /* xpage is a branch page      */

#define ZSLOTS 6                        /* num of slots for z codec    */
#define XFBSLOTS 6                      /* num of slots for xfb hdrs   */
#define SEPSLOTS 6                      /* num of slots for sep frames */
#define XHDRLEN  8                      /* hdr len on internal nodes   */
#define XLFHDRLEN 14                    /* hdr len on leaf nodes       */

#define XMASK   00600                   /* file creat mask for BTREE   */
#define XPGMINFIL  5                    /* min % fill internal node    */
#define XPGMAXFIL 99                    /* max % fill internal node    */
#define XPGDFLTFIL 60                   /* dflt % fill intternal node  */
#define LPGMINFIL  5                    /* min % fill leaf node        */
#define LPGMAXFIL 99                    /* max % fill leaf node        */
#define LPGDFLTFIL 85                   /* dflt % fill leaf node       */
#define LSPLITFILL 10                   /* % avail below which no split*/
#define XPGMINSIZE 512                  /* min size internal node      */
#define XPGMAXSIZE 8192                 /* max size internal node      */
#define XPGDFLTSIZE 4096                /* dflt size internal node     */
#define XSPLITFILL 10                   /* % avail below which no split*/
#define LPGMINSIZE 512                  /* min size leaf node          */
#define LPGMAXSIZE 32768                /* max size leaf node          */
#define LPGDFLTSIZE XPGDFLTSIZE         /* dflt size leaf node         */
#define XFBSIZE 512                     /* size of first block page    */
#define X_SPCOFF 2                      /* offset to space on branch   */
#define X_PREDOFF 4                     /* offset to pred ptr on branch*/
#define L_SPCOFF 2                      /* offset to space on leaf     */
#define L_PREDOFF 4                     /* offset to pred ptr on leaf  */
#define L_SUCCOFF 8                     /* offset to ptr to successor  */
#define L_FLEOFF 12                     /* offset to First Log Entry pt*/
#define MAGICLINK 0xffff                /* tag ==> link off pg to mem  */
#define RIGHT_SPLIT 0x8000              /* direction of split is --->  */

static char *abmsg[] = {
   "main:       quit- bad arg count",
   "main:       quit- unauthorized user",
   "main:       quit- user entered null dbname",
   "setup:      abort- inconsistent database"
};
```

```
static char *DATAFMT[] = {
        "%s",           /* char strings */
        "%d",           /* integer, free format */
        "%c",           /* single char */
        "%f",           /* single precision float */
        "%d",           /* short integer, free fmt */
        "%ld",          /* long int, free fmt */
};

static char CONFRAME[MAXCONPAGE][CONPGSIZE]; /* con rel page array    */

typedef struct dbsrcrd  {               /* database status structure   */
   short  dbs_owner;                       /* db owner's uid           */
   char dbs_name[MAXNAMLEN];               /* name of db               */
   char dbs_homedir[FILESTRING];           /* path to home dir of db   */
   short  dbs_ident;                       /* db's ident number        */
   short dbs_dfltmode;                     /* db modes to use          */
   char dbs_stat;                          /* db current status        */
}DBSTATUS;

DBSTATUS sys_db;

typedef struct relrcrd {                /* rel record structure        */
   char relname[MAXRNMLEN];                /* relation name            */
   char mode;                              /* flag for const rels      */
   unsigned short width;                   /* parms of rel             */
   short fd;                               /* filedes if open          */
   long Zmap,cursize,maxsize,rindx,windx; /* more parms of rel         */
} REL;

typedef struct domrcrd {                /* dom record structure        */
   char domname [MAXDNMLEN];               /* name of domain           */
   char spare;                             /* spare byte : speed!      */
   short domtype;                          /* data type of domain      */
   unsigned short len;                     /* length of domain bytes   */
} DOM;

typedef struct rdrcrd {                 /* rd record structure         */
   char relname[MAXRNMLEN],
        domname[MAXDNMLEN];                /* name of rel and dom      */
   short pos;                              /* position within tuple    */
} RD;

struct rstat {
        int numrelents;
        int numdoments;
        int numrdents;
} sysrelstat;

typedef struct zcirclist {              /* circ list in shuffles       */
        long numzbits;                     /* num of bits this attrib  */
        long outposn;                      /* starting output bit posn */
        struct zcirclist *next;            /* ptr to next in list      */
        struct zcirclist *prev;            /* ptr to previous in list  */
} ZCLIST;
```

```
typedef struct zlinklist {              /* linked list in shuffles     */
        int tounz;                         /* where to put bits        */
        int fromz;                         /* where bits come from     */
        struct zlinklist *next;            /* ptr to next in list      */
} ZLLIST;

typedef struct Zordslot {               /* slots used by zorder stuff  */
        REL     *zrptr;                    /* rel in this slot         */
        long    zactime;                   /* time of last access      */
        long    tomap;                     /* map converting TO        */
        long    frommap;                   /* map converting FROM      */
        ZCLIST  *fromz;                    /* circ list for current z  */
        ZCLIST  *toz;                      /* circ list for result z   */
        ZCLIST  *wkfrom,*wkto;             /* working copies of above  */
        ZLLIST  *froml,*tol;               /* non-z attr lists         */
        char    *tbuf;                     /* working tuple buffer     */
        int     fromzbits;                 /* num of bits now in z ord */
        int     tozbits;                   /* num of bits to be in z   */
        int     zrdents;                   /* num of rd ents this rel  */
} Zslot;

Zslot zslot[ZSLOTS];

typedef struct Xfisrstblk {
        long    rootpos;                   /* addrs of root page       */
        long    first_tpl;                 /* addrs of first tuple     */
        long    xpgat;                     /* disk addrs of *xbufat    */
        long    lpgat;                     /* disk addrs of *lbufat    */
        char    *xbufat;                   /* addrs of intrnl node buf */
        char    *lbufat;                   /* addrs of leaf buffer     */
        long    xactime;                   /* time of access for LRU   */
        int     xpgfill;                   /* % fill internal nodes    */
        int     lpgfill;                   /* % fill leaf nodes        */
        int     xpgsize;                   /* size of internal nodes   */
        int     lpgsize;                   /* size of leaf nodes       */
        char    rname[MAXRNMLEN];          /* copy of this rel name    */
        char    filler[XFBSIZE - (44+MAXRNMLEN)];  /* fill out block size      */
} Xfb;

Xfb xfb[XFBSLOTS];

/* the size of Xpghdr typedef ***MUST*** guarantee that an instance   */
/* of the type ends on a 32-bit boundary since the P0 ptr appears     */
/* IMMEDIATELY following the hdr on a branch page                     */
typedef struct xbranchpghdr {
        char    *status;                   /* status byte of page      */
        char    *spare;                    /* unused: skip for align   */
        u_short *space;                    /* space left on this page  */
        long    *pred;                     /* back ptr to pred page    */
} Xpghdr;

typedef struct leafpghdr {
        char    *status;                   /* status byte of page      */
        char    *spare;                    /* unused: skip for align   */
        u_short *space;                    /* space left on this page  */
        long    *pred;                     /* back ptr to pred page    */
```

```
        long    *succ;                  /* fwd ptr to successor pg  */
        u_short *offset;                /* offset to lst logical tpl*/
} Xlfhdr;

typedef struct SepSlot {
        REL     *rptr;                  /* rel being searched       */
        long    actime;                 /* time of last reference   */
        int     rdentries;              /* num of rd entries in rel */
        int     Zdomlen;                /* len of doms in Z order   */
        short   domtype[MAXATTS];       /* domtype of each att      */
        ushort  domlen[MAXATTS];        /* domlen of each att       */
} Sepslot;

Sepslot sepslot[SEPSLOTS];

typedef struct tree_ent { /* apologies to Tolkien */
        char    *te_item;               /* ptr to data item         */
        ushort  te_len;                 /* length in BITS of item   */
        ushort  te_bnext;               /* offset to next ent list  */
        long    te_ptr;                 /* ptr to nxt pg or tpl     */
} Bentry;

union u_short {
        short sval;
        char c1,c2;
} short_coerce;

union u_int {
        int ival;
#ifdef INT16
        char c1,c2;
#endif
#ifndef INT16
        char c1,c2,c3,c4;
} int_coerce;
#endif

union u_float {
        float fval;
        char c1,c2,c3,c4;
} flt_coerce;

union u_long {
        long lval;
        char c1,c2,c3,c4;
} long_coerce;

typedef struct vmemrcrd {               /* what's in mem recrd struct  */
   long page;                           /* what page num               */
   char wflg;                           /* read/write flag for page */
} VMEM;

/* defines for error handler "db_err" to identify procedures */
#define ABEND           1
#define ABORT           2
#define ADTUPLE         3
```

```
#define CLOSEREL         4
#define CMPSEPTUP        5
#define DB_ERR           6
#define DBCK             7
#define FIND             8
#define FIND_DB          9
#define FINDBRO         10
#define FINDDOM         11
#define FINDRD          12
#define FINDREL         13
#define FLUSHPAGE       14
#define GETUPLE         15
#define GOODUSER        16
#define INSERT          17
#define LOADDOM         18
#define LOADPAGE        19
#define LOADRD          20
#define LOADREL         21
#define LOGENT          22
#define MERGE           23
#define MKINDEX         24
#define MKREL           25
#define MKSEP           26
#define OPENREL         27
#define PRINTREL        28
#define PROJECT         29
#define RDPAGE          30
#define RDTUPLE         31
#define REPLACE         32
#define SEARCH          33
#define SELECT          34
#define SETUP           35
#define SORTREL         36
#define SPLIT           37
#define SPLITAT         38
#define SYNCREL         39
#define TIMER           40
#define UPD_PAREN       41
#define VACANCY         42
#define WRTUPLE         43
#define YESNO           44
#define Z_ORD           45
/****************** ENDS HERE ******************************/
/* static VMEM frame[MAXFRAMES];          /* one record per frame       */
static int freeframe;                   /* do I need this ??????      */
#define LINELEN 130                     /* length of printed line     */
#define PGNUMLEN 5                      /* length of $ holding pgnum  */
#define PGNUMHDR "Page "                 /* msg appearing in header    */
#define PAGELINES  55                   /* maxnum printed lines / page */
#define PGNUMFMT "%-4d"                 /* printe format for pg nums  */

#define READMODE 0                      /* read mode for open         */
#define WRITMODE 1                      /* write mode for open        */
#define RDWRMODE 2                      /* read/write mode for open   */
#define FROMTOP 0                       /* lseek indx:from top of file */
#define FROMCUR 1                       /* lseek indx:from cur in file */
```

```
#define FROMEND 2                       /* lseek indx:from end of file */
#define ABENDCODE -50                   /* distinguish abend/terminate */
#define ONE 1                           /* for read/write: char count  */
#define BLOCK 512                       /* for read/write: char count  */
#define RELWIDTH 36                     /* record length for REL       */
#define DOMWIDTH (MAXDNMLEN+1+4)        /* record length for DOM       */
#define RDWIDTH (2*MAXRNMLEN+2)         /* record length for RD        */
#define MAXRELS 85                      /* maximum number of relations */
#define MAXDOMS 128                     /* maximum number of domains   */
#define MAXRDS 340                      /* maximum number of rds       */
#define BIGNUM 32767                    /* a big number                */

/* macros */

#define ACTIVE 'a'
#define NEW 'n'
#define CHECK  'c'
#define CONFUSED '?'
#define CLOSED '-'
#define YES    2
#define NO     1
#define FALSE  0
#define TRUE   1
#define MAXCONRELS 50
int rdents; /* number of entries matched in findrd */
/* define numeric constants for know mrds commands */
#define CMD_BYE 0
#define CMD_EXIT 0
#define CMD_SHOWDOM 1
#define CMD_SHOWRD  2
#define CMD_SHOWREL 3
#define CMD_PRINTREL 4
#define CMD_PROJECT  5
#define CMD_HELP 6
#define CMD_SELECT 7
#define CMD_LIST 8
#define CMD_CONVERT 9
#define PROMPT "mrds> "

#define DT_STRING  0
#define DT_INT     1
#define DT_CHAR    2
#define DT_FLOAT   3
#define DT_SHORT   4
#define DT_LONG    5

#define CMP_OPS    3
#define CMPATT_EQ  1
#define CMPATT_GT  2
#define CMPATT_LT  4
#define CMPVALID   7
static int CMPATT[] = {CMPATT_EQ,CMPATT_GT,CMPATT_LT};

#define DB_NOFND -2
#define DB_FUBAR -3
```

```
/* access modes */

#define ALLMODES 0xff
#define IS_OWNER 0x04

/* UNIX hooks: calls to UNIX specific user-level routines */
#define MORE /usr/bin/more

char inbufr[MAXSUBSTR],*inptr,LOGBUF[MAXSUBSTR];
char *buffer[MAXOPNFILES],*conrels[MAXCONRELS];
DBSTATUS *data_base;
REL relcore[MAXRELS];
DOM domcore[MAXDOMS];
RD rdcore[MAXRDS];
FILE *fplog;
char names[4][MAXRNMLEN];
RD *entries[4];
REL *reltosort;
static char filenm[FILESTRING + MAXNAMLEN];
static int maxerr = MAXERR;

#define STARS   "*************************************************************************"
```

```
#include "mrds.h"

int abend(abcode,obit)

   int abcode;
   char *obit;

   /* perform graceful exit from program when things        */
   /* have become hopeless                                  */

   {
           extern char *abmsg[];
           int fclose();

#ifdef XTRACE
          fprintf(stdout,"--> abend(%d,%s)\n",-abcode,obit);
          fflush(stdout);
#endif
      if (abcode < ABENDCODE) {
         fprintf(stderr,"\nMRDSc ABEND %d\n",abmsg[-abcode]);
         logent("ABEND",abmsg[-abcode]);
         if (fplog) {fflush(fplog); fclose(fplog);}
         exit(abcode);
      }
      else {
         fprintf(stderr,"\nMRDSc TERMINATED %s\n",abmsg[-abcode]);
         logent("TERM",abmsg[-abcode]);
         if (fplog) {fflush(fplog); fclose(fplog);}
         exit(abcode);
      }
   }
```

```
#include "mrds.h"

/*
* int adtuple(rel,tpl,x)
*
*       add the tuple pointed at by the char *tpl to the end of
*       the relation whose .rel entry is pointed at by REL *rel.
*
*       return the number of bytes actually added (written) for
*       comparison with rel->width: any value different from this
*       is indicative of an error.
*
*       will issue its own error messages if:
*               (1) given a null pointer for rel
*               (2) told to append to a constant relation
*                   [currently unsupported]
*               (3) cannot open relation file for writing
*               (4) write index is negative or exceeds size limit
*               (5) cannot seek to write index offset in file
*
*/

int adtuple(rel,tpl,x)
REL *rel;       /* rel being appended to */
char *tpl;      /* tuple to be added     */
Xfb *x;         /* Xfb ptr if BTREE      */
{
        extern int errno;
        register int i;
        register char *from,*to;
        char *here;
        int openrel(),closerel(),db_err(),wrtuple(),z(),loadpage();
        int flushpage();
        long pos,lseek(),pgnum,*nxtptr,next;

#ifdef XTRACE
fprintf(stdout,"--> adtuple(%s,X%lx,%d)\n",
        rel->relname,tpl,rel->width); fflush(stdout);
#endif

        /* good rel pointer? */
        if (!rel) {
                db_err(58,ADTUPLE,-1,"null");
                return(NULL);
        }

        if (!tpl)
                return(NULL);

        if (rel -> mode & CONREL) {
                db_err(83,ADTUPLE,-1,"add to const rel");
                return(NULL);
        }

        /* ------------ G E N E R A L  R E L -------------- */
```

```
        if (rel -> mode & ZORD) /* must shuffle */
                if ((z(rel,0L,rel -> Zmap,tpl)) == FAIL)
                        return(NULL);

        if (rel -> mode & BTREE)  /* insert into tree */
                if (insert(rel,tpl,x,XLEAF) == FAIL)
                        return(NULL);
                else
                        return(rel -> width);

       /* otherwise, is just an ordinary tuple to write out */

        if ((i = wrtuple(rel,tpl)) < SUCCESS)
                return(NULL);


#ifdef XTRACE
                fprintf(stdout,"<-- adtuple(%d)\n",i);
                fflush(stdout);
#endif

                return(i);
}
```

```
int flushpage(from,fd,at,len)
char *from;     /* page starting addrs                */
int fd;         /* fd of file to flush to             */
long at;        /* position in file at which to write */
int len;        /* length of page to write            */

{
        extern int errno;
        register int i;
        int write();
        long lseek();

        if (fd < 0) return(FAIL);

        if (lseek(fd,at,FROMTOP) != at) {
                db_err(56,FLUSHPAGE,errno,"seek");
                return(FAIL);
        }

        /* should check that len is correct pagesize */

        if ((i = write(fd,from,len)) != len) {
                db_err(54,FLUSHPAGE,"write");
                return(FAIL);
        }

        return(i);
}
```

```
#include "mrds.h"

/* This file contains a "library" of utilities used by
   several MRDSc procedures to maintain and manipulate
   B*trees. [appearing in alphabetical order]
*/
/*****************************************************************
int findbro(rptr,pg,x,mode,lbro,rbro)

        Find left and right brother pages of "pg" page in a B*tree.
        Page whose brothers are sought MUST be in x -> xbufat.
        Return FAIL if failed during search, otherwise return SUCCESS.
        "lbro" and "rbro" will contain disk addresses of the respective
                brother pages;  NULL implies no such brother
                exits. A failed "findbro" will return with both
                lbro and rbro NULL.
*****************************************************************/

int findbro(rptr,pg,x,mode,lbro,rbro)
REL     *rptr;          /* ptr to REL entry for this relation */
long    pg;             /* addrs of page whose brothers are sought */
Xfb     *x;             /* Xfb ptr for this tree */
char    mode;           /* flag: leaf or branch type of page */
long    *lbro,*rbro;    /* addrs of left/right brother pages */
                        /* NULL ptr ==> no such brother     */

{
        extern int errno;
        register long i;
        register char *left,*mid,*right;
        long parent;
        ushort onpage;
        char *pred,*malloc();

        /* set initial value in return parms */

        *lbro = 0L;
        *rbro = *lbro;

        /* check if work to do */

        if (*(x -> xbufat) & XROOT) /* no siblings */
                return(SUCCESS);

        /* get space for parent page */

        if ((pred = malloc(x -> xpgsize)) == (char *)(0)) {
                db_err(52,FINDBRO,errno,"findbro: parent pg");
                return(FAIL);
        }

        /* get parent's address */
        /* this check should be cleverer; addresses <= 0 could */
        /* happen but are awfully unlikely                     */

        if ((parent = *(long *)(pg + X_PREDOFF)) <= 0L) {
```

```
                free(pred);
                return(FAIL);
        }

        /* get parent page */

        if (loadpage(rptr -> fd,x,parent,pred) == FAIL) {
                free(pred);
                return(FAIL);
        }

        /* begin search */

        left = (char *)(0);
        mid = pred + XHDRLEN; /* ptr to P0 */
        onpage = *(ushort *)(pred + XHDRLEN + sizeof(long));
        right = pred + onpage;
        do {
                right += sepentlen(right);
                if (*(long *)(mid) == pg) { /* found */
                        *lbro = (left ? (*(long *)(left)) : 0L);
                        *rbro = *(long *)(right);
                        free(pred);
                        return(SUCCESS);
                }
                left = mid;
                mid = right;
                onpage = *(ushort *)(right + sizeof(long));
                right = pred + onpage;
        } while(onpage);
        free(pred);
        return(SUCCESS);
}
```

```
/*****************************************************************
int sepentlen(s)

        Return length of 's' part of an (s,p) entry in a branch
                page.
        The "s part" consists of:
                +--------+-----< >-----+-----+
                | ushort |   varying   | PAD |
                +--------+-----< >-----+-----+
        where:
                ushort  is unsiged short containing length of
                        separator in bits,
                varying is the actual separator itself, aligned
                        to nearest byte address,
                PAD     is a string of 0 to 3 bytes needed to align
                        the pointer part of the (s,p) entry: it
                        assures that next physical address is on
                        a 32 bit boundary.
*****************************************************************/
int sepentlen(s)
char    *s;             /* ptr to s in (s,p) entry */

{
        register int i,j;

        i = (int)((* (ushort *)(s)) >> 3);
        i += sizeof(ushort);
        if (j = (long)(s) & 0x3)
                i += 4 - j;

        return(i);
}
```

```
/*****************************************************************
int splitat(pg,x,mode,penult,ult,first)

        Find position on page at which a split should be performed.
        Splits may be to the left or the right, with the returned
              ptr values indicating:
            <----                                   ---->
        last of entries from  penult    entry before last to stay
        current page to be              on current page.
        inserted onto left
        brother page.

        first of entries from   ult     last entry to stay on cur-
        current page to stay            page
        on current page

        entry from current    first     entry from current page
        page which becomes              which becomes first added
        first inserted onto             entry on right brother page.
        left brother page.

*****************************************************************/
int splitat(rptr,pg,x,mode,penult,ult,first)
REL     *rptr;          /* ptr to this rel */
char    *pg;            /* mem addrs of page to be spilt */
Xfb     *x;             /* Xfb ptr for this index */
short   mode;           /* lo byte = mode (leaf/branch),
                           hi byte = direction (to left/to right) */
char    **penult,       /* ptrs to entries in index (see above) */
        **ult, **first;

{
        register int i,j;
        register char *a,*b,*c;
        register int pgsize,k;
        short direction,oncurpg=0;
        extern Bentry newent;

        /* find split point on page */

        if (mode & (short)(XLEAF)) {
                pgsize = x -> lpgsize;
                i = (int)(pgsize * x -> lpgfill);
        }
        else {
                pgsize = x -> xpgsize;
                i = (int)(pgsize * x -> xpgfill);
        }

        /* i = num of bytes which should stay on current page */

        direction = (mode & RIGHT_SPLIT); /* is 0 for left */
        b = c = (char *)(0);
        if (mode & (short)(XLEAF)) {
                /* figure padding allowance for tuple entries */
```

```
		/* this is constant for fixed width tuples, so */
		/* do it here, outside loop */
		k = ((k = (rptr -> width & 0x3)) ? (4 - k) : 0);
		a = pg + *(ushort *)(pg + L_FLEOFF);
		j = XLFHDRLEN;
		while (1) {
			c = b;
			b = a;
			if (a == (char *)(MAGICLINK))
				a = (char *)(newent.te_ptr);
			else
				a = (char *)(*(long *)(a + rptr -> width));
			if (a != (char *)(MAGICLINK))
				a = ((long)(a) - x -> lpgat) + x -> xbufat;
			if ( b == (char *)(MAGICLINK)) {
			/* determine length of tuple here if variable */
				++oncurpg;
			}
			j += rptr -> width + sizeof(long) + k;
			if (direction && (j > i)) { /* --> */
				*first = a;
				*ult = b;
				*penult = c;
				break;
			}
			else if ((!direction) && ((pgsize - j) < i)) {
				*ult = a;
				*penult = b;
				*first = pg + *(ushort *)(pg + L_FLEOFF);
				break;
			}
		}
	}
	else { /* is a branch page */
		a = pg + XHDRLEN;
		while (1) {
			c = b;
			b = a;
			if (a == (char *)(MAGICLINK))
				a = (char *)(newent.te_bnext);
			else
				a = (char *)(*(ushort *)(a + sizeof(long)));
			if (a != (char *)(MAGICLINK))
				a = (char *)((unsigned long)(a) + (unsigned long)(pg));
			if ( b == (char *)(MAGICLINK)) {
				j += newent.te_len >> 3;
				++oncurpg;
			}
			else
				j += sepentlen(b);
			j +=  sizeof(long) + sizeof(ushort);
			if (direction && (j > i)) {  /* --> */
				*first = a;
				*ult = b;
				*penult = c;
				break;
```

```
                        }
                        else if ((!direction) && ((pgsize -j) < i)) { /* <-- */
                                *ult = a;
                                *penult = b;
                                *first = pg + XHDRLEN;
                                break;
                        }
                }
        }
        /* want to return number of bytes which remain on current page irrespective
           of direction of split. If value is poisitive then "linked in" new entry
           has NOT been encountered during traverse; otherwise is negative implying
           that the entry has been passed.
        */
        if (direction)
                return(oncurpg ? -j : j);
        else {
                pgsize -= j;
                pgsize = ( (mode & (short)(XLEAF)) ? pgsize - XLFHDRLEN : pgsize - XHDRLEN);
                return(oncurpg ? -pgsize : pgsize);
        }
}
```

```
/* Close the relation now opened, ie. close the now open file      */
/* for the relation, and make invalid the field in the rel         */
/* holding the relation's file descriptor. Also, reclaim the       */
/* buffer that was allocated by openrel and set appropriate        */
/* buffer pointer to invalid value.                                */
/* Return fd of successfully closed rel or FAIL for failed close   */

#include "mrds.h"

int closerel(rptr)
REL *rptr;

{
        extern int errno;
        extern char *buffer[];
        int oldfd,free();
#ifdef XTRACE
fprintf(stdout,"--> closrel(%s [%d])\n", rptr -> relname, rptr -> fd);
fflush(stdout);
#endif

        /* close file */

        if (close ((int)(rptr -> fd))) {
                db_err(50,CLOSEREL,errno,rptr -> relname); /* cannot close file */
                return (FAIL);
        ]

        /* reclaim buffer space */
        free(buffer[rptr -> fd]);
        oldfd = (int)(rptr->fd) ;
        buffer[oldfd] = 0L;   /* macdp: ng for 16 bit addrs */

        /* invalidate rel's fd field */
        oldfd = rptr -> fd;
        rptr -> fd = FAIL;

#ifdef XTRACE
fprintf(stdout,"<-- closerel(%d)\n",oldfd); fflush(stdout);
#endif

        return(oldfd);
]
```

```
#include "mrds.h"

int db_err(ernum,ersect,unix_err,erstring)
   int ernum,ersect,unix_err;
   char *erstring;

   {
        extern int maxerr;
        int abend();
        static char *ermesg[] = {
                "Permission denied ",                   /* err  0 */
                " ",                                    /* err  1 */
                " ",                                    /* err  2 */
                "Missing system relation",              /* err  3 */
                " ",                                    /* err  4 */
                " ",                                    /* err  5 */
                " ",                                    /* err  6 */
                " ",                                    /* err  7 */
                "Too few system relations",             /* err  8 */
                "Too few system domains ",              /* err  9 */
                "Too few rd entries ",                  /* err 10 */
                "Too many rd entries",                  /* err 11 */
                " ",                                    /* err 12 */
                " ",                                    /* err 13 */
                " ",                                    /* err 14 */
                " ",                                    /* err 15 */
                " ",                                    /* err 16 */
                " ",                                    /* err 17 */
                " ",                                    /* err 18 */
                " ",                                    /* err 19 */
                " ",                                    /* err 20 */
                " ",                                    /* err 21 */
                " ",                                    /* err 22 */
                " ",                                    /* err 23 */
                " ",                                    /* err 24 */
                "-Non-existant path",                   /* err 25 */
                "-Not a directory",                     /* err 26 */
                "-Cannot setup database",               /* err 27 */
                "Corrupted internal node (xsize)",      /* err 28 */
                "Parent missing ptr to its child",      /* err 29 */
                "No space after split",                 /* err 30 */
                " ",                                    /* err 31 */
                " ",                                    /* err 32 */
                " ",                                    /* err 33 */
                " ",                                    /* err 34 */
                " ",                                    /* err 35 */
                " ",                                    /* err 36 */
                " ",                                    /* err 37 */
                " ",                                    /* err 38 */
                " ",                                    /* err 39 */
                " ",                                    /* err 40 */
                " ",                                    /* err 41 */
                " ",                                    /* err 42 */
                " ",                                    /* err 43 */
                " ",                                    /* err 44 */
                " ",                                    /* err 45 */
```

```
            " ",                                        /* err 46 */
            " ",                                        /* err 47 */
            " ",                                        /* err 48 */
            " ",                                        /* err 49 */
            "Cannot close",                             /* err 50 */
            "Cannot open",                              /* err 51 */
            "Cannot allocate ",                         /* err 52 */
            "Cannot read",                              /* err 53 */
            "Cannot write",                             /* err 54 */
            "Cannot create",                            /* err 55 */
            "Cannot seek",                              /* err 56 */
            "Cannot make relation",                     /* err 57 */
            "Cannot find relation",                     /* err 58 */
            "Cannot write relation",                    /* err 59 */
            "Cannot read relation",                     /* err 60 */
            "Cannot add to relation",                   /* err 61 */
            "No match in rd",                           /* err 62 */
            "Bad number of entries in rd",              /* err 63 */
            "Bad substring count",                      /* err 64 */
            "Unexpected tuple match",                   /* err 65 */
            "Bad separator length",                     /* err 66 */
            "Missing index entry",                      /* err 67 */
            "Cannot update index",                      /* err 68 */
            " ",                                        /* err 69 */
            "No more rels",                             /* err 70 */
            "No more rds",                              /* err 71 */
            "Bad number of domains",                    /* err 72 */
            " ",                                        /* err 73 */
            "No database given",                        /* err 74 */
            " ",                                        /* err 75 */
            "Attempt to overwrite rel: ignored",        /* err 76 */
            " ",                                        /* err 77 */
            "Duplicate domain name",                    /* err 78 */
            "Missing domain",                           /* err 79 */
            "Non-existant database",                    /* err 80 */
            "Failed to add tuple(s)",                   /* err 81 */
            "Failed to sort",                           /* err 82 */
            "Unsupported operation",                    /* err 83 */
            "Name too long, truncated",                 /* err 84 */
            "Tuple overflow",                           /* err 85 */
            "Unknown option ignored; using default",    /* err 86 */
            "Z map change mid relation - ignored",      /* err 87 */
            "Line too long, truncated",                 /* err 88 */
            "Cannot stat relation",                     /* err 89 */
            "Bad == determination in cmpseptup",        /* err 90 */
            "Duplicate tpl insertion; not done",        /* err 91 */
            "Failed to update rindx",                   /* err 92 */
            "Unable to make backup copy - continue",    /* err 93 */
            "Duplicate relation name",                  /* err 94 */
            "Open failed; ignored",                     /* err 95 */
            "Excessive rd entries",                     /* err 96 */
            "Insert on non B*tree: ignored",            /* err 97 */
            "No change to Zorder: ignored",             /* err 98 */
            " ",                                        /* err 99 */
    };
    static char *er_pnames[] = {
```

```
                                        "?unkown",
                /*      ABEND   */      "abend",
                /*      ABORT   */      "abort",
                /*      ADTUPLE */      "adtuple",
                /*      CLOSEREL*/      "closerel",
                /*      CMPSEPTUP*/     "cmpseptup",
                /*      DB_ERR  */      "db_err",
                /*      DBCK    */      "dbck",
                /*      FIND    */      "find",
                /*      FIND_DB */      "find_db",
                /*      FINDBRO */      "findbro",
                /*      FINDDOM */      "finddom",
                /*      FINDRD  */      "findrd",
                /*      FINDREL */      "findrel",
                /*      FLUSHPAGE*/     "flushpage",
                /*      GETUPLE */      "getuple",
                /*      GOODUSER*/      "gooduser",
                /*      INSERT  */      "insert",
                /*      LOADDOM */      "loaddom",
                /*      LOADPAGE*/      "loadpage",
                /*      LOADRD  */      "loadrd",
                /*      LOADREL */      "loadrel",
                /*      LOGENT  */      "logent",
                /*      MERGE   */      "merge",
                /*      MKINDEX */      "mkindex",
                /*      MKREL   */      "mkrel",
                /*      MKSEP   */      "mksep",
                /*      OPENREL */      "openrel",
                /*      PRINTREL*/      "printrel",
                /*      PROJECT */      "project",
                /*      RDPAGE  */      "rdpage",
                /*      RDTUPLE */      "rdtuple",
                /*      REPLACE */      "replace",
                /*      SEARCH  */      "search",
                /*      SELECT  */      "select",
                /*      SETUP   */      "setup",
                /*      SORTREL */      "sortel",
                /*      SPLIT   */      "split",
                /*      SPLITAT */      "splitat",
                /*      SYNCREL */      "syncrel",
                /*      TIMER   */      "timer",
                /*      UPD_PAREN*/     "upd_parent",
                /*      VACANCY */      "vacancy",
                /*      WRTUPLE */      "wrtuple",
                /*      YESNO   */      "yesno",
                /*      Z_ORD   */      "z"

    };

    static char *severity[] =
       {  "Fatal  ","Severe ","Error  ","Warning"  };
    static short RUN;

    fprintf(stderr,"\n>> %s << %15s: [%3d,%3d] %s %s \n",
            severity[ernum/25],er_pnames[ersect],ernum,unix_err,
            ermesg[ernum],erstring);
```

```
/* if ERRLOGON {  %* write ertrap to error log file *%
   }
*/

   if (!RUN)
      switch (ernum / 25) {
         case 3 :
                     break;
         case 2 : RUN++;
                     break;
         case 1 : abend(-3);
         case 0 : abend(-3);
      }
      else
      switch (ernum / 25) {
         case 3 :
                     break;
         case 2 : RUN++;
                     break;
         case 1 : RUN++;
                     break;
         case 0 : abend(-3);
      }

      if (RUN > maxerr)
            abend(-3,"RUN over max err");

return;
}
```

```
#include "mrds.h"

int dbck(ptr,report)
DBSTATUS *ptr;
int report;

{


        extern int errno;
        char rd_doms[MAXRDS][MAXDNMLEN], dom_doms[MAXRDS][MAXDNMLEN];
        char reltmp[MAXRNMLEN],relnames[MAXRDS][MAXRNMLEN],*index();
        char *getrelnam();
        int expected[3],found[3],errors = 0;
        register int i,j,k;
        int fdr,rels,doms,strcmp();
        struct stat stbuf;

#ifdef XTRACE
        fprintf(stdout,"--> dbck(%s,%d)\n",ptr->dbs_name,report);
        fflush(stdout);
#endif

        if (!ptr) {
                db_err(74,DBCK,-1,BLANK); /* no db to check! */
                return(FAIL);
        }


        if (report) printf("dbck:\tchecking db \'%s\'\n",ptr->dbs_name);
        sprintf(LOGBUF,"%s (owner %d): begin dbck",ptr->dbs_name,ptr->dbs_owner);
        if (fplog) logent("CHECK",LOGBUF);

        /* check if home directory exits */

        if (stat(ptr -> dbs_homedir,&stbuf) == FAIL) { /* no path */
                db_err(25,2,errno,ptr ->dbs_homedir);
                return(FAIL);
        }

        if ((stbuf.st_mode & S_IFMT) != S_IFDIR) { /* not a dir */
                db_err(26,2,-1,ptr ->dbs_homedir);
                return(FAIL);
        }

        /* open system rels and load into memory */

        expected[0] = loadrel(ptr);
        expected[1] = loaddom(ptr);
        expected[2] = loadrd(ptr);

        /* check consistency of REL */

        if (report) printf("\tChecking rel\n");
        found[0] = MINRELS;
        i = MINRELS; j = sysrelstat.numrelents - i;
```

```
        while (j--) { /* check each rel */
                strcpy(LOGBUF,getrelnam(&relcore[i]));
                if ((stat(LOGBUF,&stbuf)) == FAIL) {
                        if (report) printf("****\tMissing relation \'%s\'\n",
                        relcore[i].relname);
                        ++errors;
                }
                else {
                        if (relcore[i].cursize != (long)(stbuf.st_size)) {
                                ++errors;
                                if (report) printf("****\tWrong size: %s should be %ld but is %d\n",
                                relcore[i].relname,relcore[i].cursize,stbuf.st_size);
                        }
                        ++found[0];
                }
                ++i;
        }

        if (expected[0] != found [0]) { /* trouble in Gotham city */
                ++errors;
                if (report) printf("****\tExpected %d but found %d relations\n",
                expected[0],found[0]);
        }

        /* check RD consistency */

        if (report) printf("\tChecking rd\n");
        rels = 0;
        doms = 0;
        j = 0;

        k = sysrelstat.numrdents;
        for (i = 0; i < k; i++) {
                ++doms;
                strcpy(rd_doms[i],rdcore[i].domname);
                if(strcmp(reltmp,rdcore[i].relname)) {
                        strcpy(reltmp,rdcore[i].relname);
                        ++rels;
                        strcpy(relnames[j],rdcore[i].relname);
                        ++j;
                }
        }

        k = sysrelstat.numdoments;
        for (i = 0; i < k; i++)
                strcpy(dom_doms[i],domcore[i].domname);

        qsort(dom_doms,i,MAXDNMLEN,strcmp);
        qsort(rd_doms,doms,MAXDNMLEN,strcmp);

        /* CHANGED 18 AUG 86: .windx  BECOMES .cursize */
        if ((relcore[0].cursize / ((long)(relcore[0].width))) != rels) {
                ++errors;
                if (report) {
                        printf("****\tMismatch: num of rels in rel and rd ");
                        printf("%ld != %ld\n",relcore[0].cursize / ((long)(relcore[0].width)),
```

```
                        rels);
                }
        }

        i = 0;
        k = 0;
        while ( *(relcore[i].relname) && *(relnames[k])) {
                if (j = strcmp(relcore[i].relname,relnames[k])) { /* not same */
                        ++errors;
                        if(j > 0) {
                                if (report) printf("****\tMissing relname \'%s\' in rel\n",relnames[k]);
                                ++k;
                        }
                        else {
                                if (report) printf("****\tMissing relname \'%s\' in rd\n",relcore[i].relname);
                                ++i;
                        }
                }
                else {
                        ++i;
                        ++k;
                }
        }

        i = 0;
        k = 0;
        reltmp[0] = NULLC;
        while ( (dom_doms[i][0] != NULL) ||
            (rd_doms[k][0] != NULL) ) {
                if(strcmp(reltmp,rd_doms[k])) { /* new dom name */
                        strcpy(reltmp,rd_doms[k]);

                        if (j = strcmp(dom_doms[i],rd_doms[k])) {
                                ++errors;
                                if (j > 0) {
                                        if (report) printf("****\tMissing domname \'%s\' in dom\n",rd_doms[k]);
                                        ++k;
                                }
                                else {
                                        if (report) printf("****\tMissing domname \'%s\' in rd\n",dom_doms[i]);
                                        ++i;
                                }
                        }
                        else { /* are same */
                                ++i;
                                ++k;
                        }
                }
                else { /* same domain as last time */
                        ++k;
                }
        }

        /* end of check */

        if (report) {
```

```
                printf("End of check on database \'%s\' ",ptr->dbs_name);
                if (!errors) printf(": no errors detected\n");
                else if (errors > 1)
                        printf(": %d errors detected\n",errors);
                else
                        printf(": %d error detected\n",errors);
        }

#ifdef XTRACE
        fprintf(stdout,"<-- dbck(%d)\n",errors);
        fflush(stdout);
#endif

        return(errors);
}
```

```
#include "mrds.h"


int find_db(name,owner,ustruct) /* determine whether a database exists    */
char *name;            /* if so return pointer to struct holding */
int owner;                  /* status of db, else return null ptr     */
DBSTATUS *ustruct;      /* addrs of user structure to return entry in */

{
        FILE *fptr, *fopen();
        extern DBSTATUS sys_db;
        register int i,j;
        register DBSTATUS *dbt,*sysdbt;
        short search = 1;
        int kludge; /* needed because Masscomps can't put big integers */
                    /* like '8' into a short in fscanf */

#ifdef XTRACE
        fprintf(stdout,"--> find_db(%s,%d,%lx)\n",name,owner,ustruct);
        fflush(stdout);
#endif

        if ((fptr = fopen(DBLIST,RDMODE)) == NULL)
                return(FAIL);
        else { /* look for the db */
                if ((dbt = ustruct) == (DBSTATUS *)(0))
                        return(FAIL); /* nowhere to put it */
                sysdbt = &sys_db;
                while ((search) &&
                    fscanf(fptr,"%d %s %s %d %c",&(kludge),
                dbt->dbs_name,dbt->dbs_homedir,&(dbt->dbs_ident),&(dbt->dbs_dfltmode))
                    != EOF)
                        if ((!strcmp(name,dbt->dbs_name)) &&  /* names are equal */
                        (owner == (short)(kludge)) )     /* owners are same */
                                search = 0;
        }
        fclose(fptr);
        dbt->dbs_owner = (short)(kludge);
#ifdef XTRACE
        fprintf(stdout,"find_db: owner %d\n",dbt->dbs_owner);
        fprintf(stdout,"         name  %s\n",dbt->dbs_name);
        fprintf(stdout,"         home  %s\n",dbt->dbs_homedir);
        fprintf(stdout,"         ident %d\n",dbt->dbs_ident);
        fprintf(stdout,"         dflt  %c\n",dbt->dbs_dfltmode);
        fprintf(stdout,"<-- find_db(%d)\n",search);

        fflush(stdout);
#endif

        /* for better security should:
                zero the user's struct,
                use local struct as image of user's structure
                if found, copy local image to user's true struct
        */
        if (search) { /* unfound: 0 the user's struct before return */
                j = strlen(dbt -> dbs_name);
```

```
                for (i = 0; i < j; i++) {
                        dbt -> dbs_name[i] = '\0';
                        sysdbt -> dbs_name[i] = '\0';
                }
                j = strlen(dbt -> dbs_homedir);
                for (i = 0; i < j; i++){
                        dbt -> dbs_homedir[i] = '\0';
                        sysdbt -> dbs_homedir[i] = '\0';
                }
                dbt -> dbs_owner = -1; sysdbt -> dbs_owner = -1;
                dbt -> dbs_ident = -1; sysdbt -> dbs_ident = -1;
                dbt -> dbs_dfltmode = '\0'; sysdbt -> dbs_dfltmode = '\0';
                return(FAIL);
        }
        else { /* found: copy into system db entry */
                strcpy(sysdbt -> dbs_name,dbt -> dbs_name);
                strcpy(sysdbt -> dbs_homedir,dbt -> dbs_homedir);
                sysdbt -> dbs_owner = dbt -> dbs_owner;
                sysdbt -> dbs_ident = dbt -> dbs_ident;
                sysdbt -> dbs_dfltmode = dbt ->dbs_dfltmode;
                return(SUCCESS);
        }

}
```

```
#include "mrds.h"

DOM *finddom(dname)
char *dname;

{

        extern DOM domcore[];
        int strcmp();
        register int i,found=TRUE;

#ifdef XTRACE
        fprintf(stdout,"--> finddom(%s)\n",dname);
        fflush(stdout);
#endif

        if (dname) /* there is a name */
                for (i = 0; i < MAXDOMS; i++) {
                        if (! *(domcore[i].domname))
                                break; /* no more doms */
                        if (! (found = strcmp(dname,domcore[i].domname)))
                                break;
                }

#ifdef XTRACE
        fprintf(stdout,"<-- finddom(%d,%d)\n",found,i);
        fflush(stdout);
#endif

        return(found ? (DOM *)(NULL) : &domcore[i]);
}
```

```
#include "mrds.h"


RD *findrd(rname,domlist,num,rdptr)
char *rname,domlist[][MAXDNMLEN]; /* was domlist[][] on masscomp */
short num;
RD *rdptr[];

{

        register int i,j,rstart=FALSE,rend,domfound;
        extern int rdents;
        extern RD rdcore[];
        int strcmp();

#ifdef XTRACE
        fprintf(stdout,"-->findrd(%s,X%lx,%d,X%lx\n",rname,domlist,num,rdptr);
        fflush(stdout);
#endif

        rdents = 0;
        /* see if there is a name */
        if (! *rname)
                return((RD *)(NULL)); /* no name */

        /* find first entry in rd for 'rname' */
        for (i = 0; i < MAXRDS; i++)
                if (!strcmp(rname,rdcore[i].relname)) {
                        rstart = i;
                        break;
                }

        if (rstart < 0) { /* rname not in rd */
                db_err(62,FINDRD.-1,rname);
                return((RD *)(NULL));
        }

        /* find last entry in rd for rname */
        rend = rstart;
        while (!strcmp(rname,rdcore[rend].relname))
                ++rend;
        rdents = rend - rstart ;

        if (rdents > MAXATTS) { /* VERY dangerous: something got clobbered */
                strcpy(inbufr,rname); /* this should never happen (ha!) */
                strcat(inbufr,"(");
                strcat(inbufr,itoa(rdents));
                strcat(inbufr,")");
                db_err(96,FINDRD,-1,inbufr);
                rdents = MAXATTS;
        }

        if (num) /* lookup supplied domain names */

{
#ifdef XTRACE
```

```
fprintf(stdout,"    findrd: num = %d, rstart = %d , rend = %d\n",
num,rstart,rend);fflush(stdout);
#endif
                for (i = 0; i < num; i++) {
                        domfound = 0;
                        j = rstart;
                        while ((!domfound) && (j < rend))
                                if(!strcmp(domlist[i],rdcore[j].domname))
                                        ++domfound;
                                else
                                        ++j;
                        if (domfound)
                                rdptr[i] = &rdcore[j];
                        else
                                rdptr[i] = (RD *)(NULL);
                }

        }else /* get all the domains */
                for (i = rstart; i <= rend; i++)
                        rdptr[i - rstart] = &rdcore[i];

#ifdef XTRACE
        fprintf(stdout,"<-- findrd(%d)\n",rdents);
        fflush(stdout);
#endif

        return(&rdcore[rstart]);
}
```

```
#include "mrds.h"

REL *findrel(rname)
char *rname;

{
        extern REL relcore[];
        register int i,found=TRUE;
        int strcmp();

#ifdef XTRACE
        fprintf(stdout,"--> findrel(%s)\n",rname);
        fflush(stdout);
#endif

        if ( *rname)    /* if there is a name */
                for (i = 0; i < MAXRELS; i++) {
                        if (!(*(relcore[i].relname)))
                                break;  /* no more rels */
                        if(!(found = strcmp(rname,relcore[i].relname)))
                                break;
                }

#ifdef XTRACE
        fprintf(stdout,"<-- findrel(%d,%d)\n",found,i);
        fflush(stdout);
#endif

        return(found ? (REL *)(NULL) : &relcore[i]);
}
```

```
#include "mrds.h"

char *getdbname()
{
        register int i;
        int strlen();

        while(1) {
                fprintf(stdout,"\nEnter name of db to use: ");
                READLN
                if ((i = strlen(inbufr)) > MAXNAMLEN) {
                        db_err(84,1,-1,inbufr);
                        inbufr[MAXNAMLEN - 1] = NULLC;
                        break;
                }
                else if (i == 0) break;
                return(inbufr);
        }
        return(inbufr);
}
```

```
#include "mrds.h"

char *getuple(rptr,mode)
REL *rptr;
char mode;

{

        extern char *conrels[],*buffer[];
        extern int errno;
        int rdtuple(),db_err(),z();
        register int i;
        long pos;

#ifdef XTRACE
        fprintf(stdout,"--> getuple(%s,X%lx)\n",rptr->relname,mode);
        fflush(stdout);
#endif

        if (!rptr) {
                db_err(58,GETUPLE,-1,NOSTRING);
                return((char *)(NULL));
        }

        if (rptr->mode & CONREL) {  /* -------  C O N S T A N T  R E L  ------- */

                if (rptr->fd == FAIL) {
                        db_err(70,GETUPLE,-1,rptr->relname);
                        return((char *)(NULL));
                }

                if ((rptr->rindx + mode >= rptr->cursize + (long)(conrels[rptr->fd])) ||
                    (rptr->rindx + mode < (long)(conrels[rptr->fd]))) {
                        db_err(83,GETUPLE,-1,rptr->relname);
                        return((char *)(NULL));
                }

                rptr->rindx += mode + rptr->width;
#ifdef XTRACE
                fprintf(stdout,"<-- getuple(con @ X%lx)\n",rptr->rindx + mode +
                    conrels[rptr->fd]);
                fflush(stdout);
#endif
        }

        /* --------  G E N E R A L   R E L   -------- */


        if ((rdtuple(rptr,(char *)(NULL))) < SUCCESS)
                return((char *)(NULL));


        if (rptr -> mode & ZORD) /* must unshuffle */
                if ((z(rptr,rptr -> Zmap,0L,buffer[rptr -> fd])) == FAIL)
                        return((char *)(NULL));
```

```
#ifdef XTRACE
                fprintf(stdout,"<-- getuple(gen,X%lx)\n",buffer[rptr->fd]);
                fprintf(stdout,"   TUPLE= |%s|\n",buffer[rptr->fd]);
                fflush(stdout);
#endif

                return(buffer[rptr->fd]);
}
```

```
int loadpage(file,x,from,to)
int file;       /* descriptor for file                  */
Xfb x;          /* hdr info for this index              */
long from;      /* position in file from which to read*/
char *to;       /* buffer pointer: put page 'to' there*/


{
        extern int errno;
        register int i,j;
        register long pos;
        int read();
        long lseek();
        char pgtype,*To;

#ifdef XTRACE
fprintf(stdout,"--> loadpage(%d,%lxX,%ld.,%lxX)\n",file,x,from,to); fflush(stdout);
#endif

        if (lseek(file,from,FROMTOP) != from) {
                db_err(56,LOADPAGE,errno,"seek");
                return(FAIL);
        }

        if ((i = read(file,&pgtype,1)) != 1) {
                db_err(53,LOADPAGE,errno,"lread");
                return(FAIL);
        }

        if (!to)
                if (pgtype = (pgtype & XLEAF))  /* is a leaf page */
                        To = x.lbufat;
                else    To = x.xbufat;
        else To = to;

        if ((pos = lseek(file,-1L,FROMCUR)) != from) {
                db_err(56,LOADPAGE,errno,"backseek");
                return(FAIL);
        }

        i = (pgtype ? x.lpgsize : x.xpgsize);

        if ((j = read(file,To,i)) != i) {
                db_err(53,LOADPAGE,errno,"read");
                return(FAIL);
        }

        /* update xfb particulars */

        if (pgtype) /* is leaf */
                x.lpgat = pos;
        else    x.xpgat = pos;

#ifdef XTRACE
fprintf(stdout,"<-- loadpage(%d)\n",(pgtype ? -j : j)); fflush(stdout);
```

```
#endif
        return(pgtype ? -j : j); /* negative ==> leaf was read */
}
```

```
#include "mrds.h"

int gooduser ()

{

        char inbuf[MXLINUID],*index(),uidstr[10],*substr();
        int  i = FAIL,uid,atoi(),db_err(),geteuid();
        FILE *fptr, *fopen();

#ifdef XTRACE
        fprintf(stdout,"--> gooduser()\n");
        fflush(stdout);
#endif

        if ((fptr = fopen(USRFILE,RDMODE)) == (FILE *)(NULL)) {
                db_err(61,GOODUSER,-1,"user list");
                return(FAIL);
        }

        uid = geteuid();
        while (fgets(inbuf,MXLINUID,fptr) != NULL)
                if (uid ==  atoi(substr(inbuf,uidstr,0,
                    ((int)(index(inbuf,':') - inbuf))))) {
                        i = uid;
                        break;
                }

#ifdef XTRACE
        fprintf(stdout,"<-- gooduser(%d)\n",i);
        fflush(stdout);
#endif
        return(i);
}
```

```
#include "mrds.h"

int insert(rel,tpl,x,mode)
REL     *rel;           /* ptr to rel into which insertion occurs */
char    *tpl;           /* ptr to entry being inserted (tpl or (s,p)) */
Xfb     *x;             /* ptr to Xfb entry for this tree */
char    mode;           /* insertion of tuple or (s,p) */

{
        extern int errno;
        register int i,j;
        register char *to,*from;
        char *here;
        long pos,next,*nxtptr;
        int pgnum;

                /* check that relation is a B* tree */
                if  (!(rel -> mode & BTREE)) { /* not a tree */
                        db_err(97,INSERT,-1,rel -> relname);
                        return(FAIL);
                }

                /* locate insertion point */
                if ((pos = search(rel,tpl)) == FAIL) {
                        db_err(61,INSERT,-1,rel -> relname);
                        return(FAIL);
                }

                if (pos > 0L) {
                        db_err(91,INSERT,-1,rel -> relname);
                        return(0);
                }

                pgnum = (-pos / x -> lpgsize);
                if (x -> lpgat != pgnum) /* lost leaf: should never happen */
                        if (loadpage(openrel(rel,RDMODE),x,pos,(char *)(0)) == FAIL){
                                db_err(60,INSERT,errno,rel -> relname);
                                return(FAIL);
                        }

                /* check space remaining on page: need to split? */
                /* 3 is the maximum amount of skip bytes which could */
                /* preceed a tuple */
                if ((rel -> width + sizeof(long) + 3) > *(ushort *)(x -> lbufat + L_SPCOFF)) {
                        if (split(rel,tpl,pos,x,XLEAF) == FAIL){
                                db_err(61,INSERT,-1,rel -> relname);
                                return(FAIL);
                        }
                        else
                                return((int)(rel -> width)); /* done ! */
                }
                else { /* add to linked list on this leaf page */
                        /* find physical space on page */
                        here = x -> lbufat + ((x -> lpgsize - XLFHDRLEN) - (*(ushort *)(x -> lbufat + L_SPCOFF)));
                        from = tpl; to = here;
                        for (i = 0; i < (int)(rel -> width); i++)
```

```
                        *to++ = *from++;

                /* link into list */
                pgnum = pgnum + *(ushort *)(x -> lbufat + L_SUCCOFF);
                if (((-pos - 1) - pgnum) == 1) { /* add before first */
                        *(long *)(to) = (x -> lpgat) + (*(ushort *)(x -> lbufat + L_SUCCOFF));
                        *(ushort *)(x -> lbufat + L_SUCCOFF) = (ushort)(here - x -> lbufat);
                        if (*(x -> lbufat) & XFRSTLF)
                                x -> first_tpl = (long)(here - x -> lbufat + x -> lpgat);
                }
                else {
                        nxtptr = (long *)(x -> lbufat + (-pos % x -> lpgsize) + (rel -> width));
                        next = *nxtptr;
                        *(long *)(to) = next;
                        *nxtptr = (long)(here - x -> lbufat) + x -> lpgat;
                }
        }

/* write out modified page */

        if ((i = flushpage(x -> lpgat,rel ->fd,x -> lbufat,x -> lpgsize)) != x -> lpgsize) {
                db_err(60,INSERT,errno,rel -> relname);
                return(FAIL);
        }
        return(i);
}
```

```
long mksep(rptr,old,new,sep)  /* return separator length in BITS */
REL *rptr;      /* ptr to REL */
char *old;      /* ptr to tpl in old page */
char *new;      /* ptr to tpl in new page */
char *sep;      /* ptr to space for separator */

{
        extern int errno;
        extern union u_short short_coerce;
        extern union u_int int_coerce;
        extern union u_long long_coerce;
        extern union u_float flt_coerce;
        extern short domtype[];
        register int i,j,k;
        register char *a,*b,*c;
        RD *rd,*rdtab[MAXATTS],*findrd();
        DOM *domain,*finddom();
        ushort domlen[MAXATTS];
        int seplen;
        long longl;
        float floatl;
        short shortl;
        char zmask;

#ifdef XTRACE
fprintf(stdout,"--> mksep(%s,%lx,%lx,%lx)",rptr->relname,old,new,sep);
fflush(stdout);
#endif
        a = old;
        b = new;
        c = sep;

        for (i = 0; i < (int)(rptr -> width); i++)
                *c++ = '\0';
        c = sep;

        /* get rd and dom particulars: expensive, but justifiable since
           this code is performed relatively infrequently                */

        rd = findrd(rptr -> relname,0L,0,rdtab); /* macdep: 0L null ptr */
        for (i = 0; i < rdents; i++) {
                domain = finddom(rdtab[i] -> domname);
                if (domain){
                        domtype[i] = domain -> domtype;
                        domlen[i] = domain -> len;
                }
        }

        /* build separator: only z order gets minimum length separator */
        /* check for data organization: all except z order handled same way */

        if (! rptr -> Zmap) { /* no z ordering */
                seplen = 0;
                for (i = 0; i < rdents; i++) {
#ifdef XTRACE
```

```
fprintf(stdout,"III: attr %d in %s of type %d\n",i,rptr->relname,domtype[i]);
fprintf(stdout,"separ len now %d\n",seplen);fflush(stdout);
#endif
                        switch (domtype[i]) {
                        case DT_CHAR:
                                *c++ = *b;
                                if (*a - *b)
                                        return(seplen + 8);
                                break;

                        case DT_STRING:
                                while (*a) {
                                        *c++ = *b;
                                        seplen += 8;
                                        if (*a - *b)
                                                return(seplen);
                                        else {
                                                ++a;
                                                ++b;
                                        }
                                }
                                *c++ = '\0'; ++a; ++b; seplen += 8;
                                break;
                        case DT_LONG:
#ifndef INT16
                        case DT_INT:
#endif
                                long_coerce.c1 = *a;
                                *(&(long_coerce.c2) + 1) = *(a + 1);
                                *(&(long_coerce.c3) + 2) = *(a + 2);
                                *(&(long_coerce.c4) + 3) = *(a + 3);
                                long1 = long_coerce.lval;
                                long_coerce.c1 = *b;
                                *(&(long_coerce.c2) + 1) = *(b + 1);
                                *(&(long_coerce.c3) + 2) = *(b + 2);
                                *(&(long_coerce.c4) + 3) = *(b + 3);
                                *c++ = long_coerce.c1;
                                *c++ = *(&(long_coerce.c2) + 1);
                                *c++ = *(&(long_coerce.c3) + 2);
                                *c++ = *(&(long_coerce.c4) + 3);
                                seplen += sizeof(long);
                                long_coerce.lval = long_coerce.lval - long1;
                                if (long_coerce.lval > 0L)
                                        return(seplen);
                                break;

                        case DT_SHORT:
#ifdef INT16
                        case DT_INT:
#endif
                                short_coerce.c1 = *a;
                                *(&(short_coerce.c2) + 1) = *(a + 1);
                                short1 = short_coerce.sval;
                                short_coerce.c1 = *b;
                                *(&(short_coerce.c2) + 1) = *(b + 1);
                                *c++ = short_coerce.c1;
```

```
                        *c++ = *(&(short_coerce.c2) + 1);
                        seplen += sizeof(short);
                        short_coerce.sval = short_coerce.sval - short1;
                        if (short_coerce.sval > 0)
                                return(seplen);
                        break;

                case DT_FLOAT:
                        flt_coerce.c1 = *a;
                        *(&(flt_coerce.c2) + 1) = *(a + 1);
                        *(&(flt_coerce.c3) + 2) = *(a + 2);
                        *(&(flt_coerce.c4) + 3) = *(a + 3);
                        float1 = flt_coerce.fval;
                        flt_coerce.c1 = *b;
                        *(&(flt_coerce.c2) + 1) = *(b + 1);
                        *(&(flt_coerce.c3) + 2) = *(b + 2);
                        *(&(flt_coerce.c4) + 3) = *(b + 3);
                        *c++ = flt_coerce.c1;
                        *c++ = *(&(flt_coerce.c2) + 1);
                        *c++ = *(&(flt_coerce.c3) + 2);
                        *c++ = *(&(flt_coerce.c4) + 3);
                        seplen += sizeof(float);
                        flt_coerce.fval = flt_coerce.fval - float1;
                        if (flt_coerce.fval > 0.0)
                                return(seplen);
                        break;
                }
        }
        /* entire tuple same: not good */
        db_err(65,MKSEP,-1,rptr->relname);
        return(FAIL);

} /* if not zordered */
else { /* is some z-ordering */
        if (!(rptr -> mode & PZREL)) { /* is all z order */
                /* make separator: go through as many bytes as
                neccessary, bit by bit, from most significant
                bit to least significant bit
                */
                for (i = 0; i < (int)(rptr -> width); i++) {
                        zmask = 0x80; /* set bit 7 */ /* ?????????????????????? */
                        for (j = 7; j >= 0; j--) {
                                *c |= (*b & zmask);
                                seplen++;
                                if ((*b & zmask) & !(*a & zmask))
                                        return(seplen);
                                zmask >>= 1;
                        }
                        ++a;
                        ++b;
                        ++c;
                }

                /* if here, tuples were equal ! */
                db_err(65,MKSEP,-1,rptr->relname);
                return(FAIL);
```

```
		}
		else { /* partial z ordering */
			j = 0;
			for (i = 0; i < rdents; i++)
				if (rptr -> Zmap & (1L << i))
					k += (int)(domlen[i]);

			for (i = 0; i < k; i++)
				zmask = 0x80; /* set bit 7 */
			for (j = 7; j >= 0; j--) {
				*c |= *b & zmask;
				++seplen;
				if ((*b & zmask) & !(*a & zmask))
					return(seplen);
				zmask >>= 1;
			}
			++a;
			++b;
			++c;
		}

		/* if here, continue building separator using
		repositioned un-zordered attributes */
		for (i = 0; i < rdents; i++) {
			if (rptr -> Zmap & (1L << i))
				continue;
			switch (domtype[i]) {
			case DT_CHAR:
				*c++ = *b;
				if (*a - *b)
					return(seplen + 8);
				break;

			case DT_STRING:
				while (*a) {
					*c++ = *b;
					seplen += 8;
					if (*a - *b)
						return(seplen);
					else {
						++a;
						++b;
					}
				}
				break;
			case DT_LONG:
#ifndef INT16
			case DT_INT:
#endif
				long_coerce.c1 = *a;
				*(&(long_coerce.c2) + 1) = *(a + 1);
				*(&(long_coerce.c3) + 2) = *(a + 2);
				*(&(long_coerce.c4) + 3) = *(a + 3);
				longl = long_coerce.lval;
				long_coerce.c1 = *b;
```

```
                        *(&(long_coerce.c2) + 1) = *(b + 1);
                        *(&(long_coerce.c3) + 2) = *(b + 2);
                        *(&(long_coerce.c4) + 3) = *(b + 3);
                        *c++ = long_coerce.c1;
                        *c++ = *(&(long_coerce.c2) + 1);
                        *c++ = *(&(long_coerce.c3) + 2);
                        *c++ = *(&(long_coerce.c4) + 3);
                        seplen += sizeof(long);
                        long_coerce.lval = long_coerce.lval - longl;
                        if (long_coerce.lval > 0L)
                                return(seplen);
                        break;

                case DT_SHORT:
#ifdef INT16
                case DT_INT:
#endif
                        short_coerce.c1 = *a;
                        *(&(short_coerce.c2) + 1) = *(a + 1);
                        shortl = short_coerce.sval;
                        short_coerce.c1 = *b;
                        *(&(short_coerce.c2) + 1) = *(b + 1);
                        *c++ = short_coerce.c1;
                        *c++ = *(&(short_coerce.c2) + 1);
                        seplen += sizeof(short);
                        short_coerce.sval = short_coerce.sval - shortl;
                        if (short_coerce.sval > 0)
                                return(seplen);
                        break;

                case DT_FLOAT:
                        flt_coerce.c1 = *a;
                        *(&(flt_coerce.c2) + 1) = *(a + 1);
                        *(&(flt_coerce.c3) + 2) = *(a + 2);
                        *(&(flt_coerce.c4) + 3) = *(a + 3);
                        floatl = flt_coerce.fval;
                        flt_coerce.c1 = *b;
                        *(&(flt_coerce.c2) + 1) = *(b + 1);
                        *(&(flt_coerce.c3) + 2) = *(b + 2);
                        *(&(flt_coerce.c4) + 3) = *(b + 3);
                        *c++ = flt_coerce.c1;
                        *c++ = *(&(flt_coerce.c2) + 1);
                        *c++ = *(&(flt_coerce.c3) + 2);
                        *c++ = *(&(flt_coerce.c4) + 3);
                        seplen += sizeof(float);
                        flt_coerce.fval = flt_coerce.fval - floatl;
                        if (flt_coerce.fval > 0.0)
                                return(seplen);
                        break;
                }
        }
    }
                db_err(65,MKSEP,-1,rptr->relname);
                return(FAIL);
}
```

```
/* itoa function from Kerninghan and Ritchie, page 60 */

int itoa(n, s)  /* convert n to characters in s */
char s[];
int n;
{

        int i, sign;

        if ((sign = n) < 0) /* record sign */
                n = -n; /* make n positive */
        i = 0;
        do { /* generate digits in reverse order */
                s[i++] = n % 10 + '0'; /* get next digit */
        }
        while ((n /= 10) > 0); /* delete it */
        if (sign < 0)
                s[i++] = '-';
        s[i] = '\0';
        reverse(s);
}

int reverse(s)          /* reverse string s in place */
char s[];

{

        register int i,j;
        register char c;
        int strlen();

        for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
                c = s[i];
                s[i] = s[j];
                s[j] = c;
        }
}
```

```
#include "mrds.h"

/* NEW FASTER VERSION USING PACKED STRUCT : AUGUST 1985 */
int loaddom(ptr)
DBSTATUS *ptr;

{

        extern struct rstat sysrelstat;
        extern union u_short short_coerce;
        extern int domfd;
        extern DOM domcore[];
        register int i,j,k;
        char tplbufr[DOMWIDTH],string[FILESTRING + 5];
        int numofdoms = 0,abend(),open(),close();
        long p;

#ifdef XTRACE
        fprintf(stdout,"--> loaddom(%s)\n",ptr->dbs_name);
        fflush(stdout);
#endif

        strcpy(string,ptr->dbs_homedir);
        strcat(string,Dom);
        if((domfd = open(string,RDWRMODE)) == FAIL) {
                db_err(3,LOADDOM,-1,"no dom");
                /* abend(-99,"no .dom"); BAD subscript 99 */
        }

        for (i = 0; i < MAXDOMS; i++)
                if (!(j = read(domfd,&domcore[i],DOMWIDTH)))
                        break; /* end of file */
                else
                        ++numofdoms;
         /*
                                                fprintf(stdout,"dom[%d].name = |%s|\n",i,domcore[i].domname);
                                                fprintf(stdout,"dom[%d].domtype = %d\n",i,domcore[i].domtype);
                                                fprintf(stdout,"dom[%d].len = %d\n",i,domcore[i].len);
                                                fprintf(stdout,"numofdoms now = %d\n",numofdoms);

        */
        sysrelstat.numdoments = numofdoms;

        if (numofdoms < MINDOMS) {
                db_err(9,LOADDOM,-1,BLANK);
                /* abend(-97,BLANK); BAD subscript 97 */
        }

        close(domfd); domfd = FAIL;

#ifdef XTRACE
        fprintf(stdout,"<-- loaddom(%d)\n",numofdoms);
        fflush(stdout);
#endif

        return(numofdoms);
```

```
]
```

```
#include "mrds.h"

int loadrd(ptr)
DBSTATUS *ptr;

{

        extern struct rstat sysrelstat;
        extern union u_short short_coerce;
        extern int rdfd;
        extern RD rdcore[];
        register int i,j,k;
        char string[FILESTRING + 5], tplbufr[RDWIDTH];
        int numofrds,abend(),open(),close();

#ifdef XTRACE
        fprintf(stdout,"--> loadrd(%s)\n",ptr->dbs_name);
        fflush(stdout);
#endif

        strcpy(string,ptr -> dbs_homedir);
        strcat(string,Rd);
        if ((rdfd = open(string,RDWRMODE)) == FAIL) {
                db_err(3,LOADRD,-1,string);
                /* abend(-99,"no .rd"); BAD subscript 99 */
        }

        for (i = 0; i < MAXRDS; i++)
                if (!(j = read(rdfd,&rdcore[i],RDWIDTH)))
                        break; /* end of file */

        sysrelstat.numrdents = i;
        if ((numofrds = i) < MINRDS) {
                db_err(10,LOADRD,-1,BLANK);
                /* abend(-96,BLANK); BAD subscript 96 */
        }

        close(rdfd); rdfd = FAIL;

#ifdef XTRACE
        fprintf(stdout,"<-- loadrd(%d)\n",numofrds);
        fflush(stdout);
#endif

        return(numofrds);
}
```

```
#include "mrds.h"

int loadrel(ptr)
DBSTATUS *ptr;

{
        extern struct rstat sysrelstat;
        extern int relfd,errno;
        extern REL relcore[];
        extern union u_short short_coerce;
        extern union u_int int_coerce;
        extern union u_long long_coerce;
        register int i,j,k;
        char string[FILESTRING + 5],tplbufr[RELWIDTH];
        int numofrels=0,abend(),close(),open();

#ifdef XTRACE
        fprintf(stdout,"--> loadrel(%s)\n",ptr->dbs_name);
        fflush(stdout);
#endif

        /* load rel structs for as many rels as there are */
        /* MAKE SURE STRINGS END UP BEING NULL TERMINATED!!!!! */

        strcpy(string,ptr ->dbs_homedir);
        strcat(string,Rel);
        if ((relfd = open(string,RDWRMODE)) == FAIL) {
                db_err(3,LOADREL,-1,errno,string);
                /* abend(-99,"no .rel"); BAD susbscript 99 */
        }
        for (i = 0; i < MAXRELS; i++) {
                if (!(j = read(relfd,&relcore[i],RELWIDTH)))
                        break; /* eof on .rel */
                else
                        ++numofrels;

/*
                                fprintf(stdout,"rel[%d].name = |%s|\n",i,relcore[i].relname);fflush(stdout);
                                fprintf(stdout,"rel[%d].mode = X%x\n",i,relcore[i].mode);fflush(stdout);
                                fprintf(stdout,"rel[%d].width = %d\n",i,relcore[i].width);fflush(stdout);
                                fprintf(stdout,"rel[%d].Zmap = %ld\n",i,relcore[i].Zmap);fflush(stdout);
                                fprintf(stdout,"rel[%d].cursize = %ld (X%lx)\n",i,relcore[i].cursize,relcore[i].cursize);fflush(stdout);
                                fprintf(stdout,"rel[%d].maxsize = %ld (X%lx)\n",i,relcore[i].maxsize,relcore[i].maxsize);fflush(stdout);
                                fprintf(stdout,"rel[%d].rindx = %ld (X%lx)\n",i,relcore[i].rindx,relcore[i].rindx);fflush(stdout);
                                fprintf(stdout,"rel[%d].windx = %ld (X%lx)\n",i,relcore[i].windx,relcore[i].windx);fflush(stdout);
                                fprintf(stdout,"numofrels now = %d\n",numofrels);
                                fflush(stdout);
*/
                        }
        /* to get here (1) eof was reached or
                       (2) null relname encountered, end of rels
        */
        sysrelstat.numrelents = numofrels;
        if ((numofrels) < MINRELS) {
                db_err(8,LOADREL,-1,BLANK);
                /* abend(-98,BLANK);D subscript 98 */
```

```
        }

        close(relfd); relfd = FAIL;

#ifdef XTRACE
        fprintf(stdout,"<-- loadrel(%d)\n",numofrels);
        fflush(stdout);
#endif
        return(numofrels);
}
```

```
#include "mrds.h"

int logent(event,msg)
char *event, *msg;

{
        extern FILE *fplog;

#ifdef XTRACE
        fprintf(stdout,"--> logent(%s,%s)\n",event,msg);
        fflush(stdout);
#endif
        /* file must already be fopen with FILE pointer fplog */

        if (fplog == (FILE *)(0)) {
                db_err(54,LOGENT,-1,"log file not open");
                return(FAIL);
        }
        else fprintf(fplog,"%ld:\t%s:\t%s\n",timer(),event,msg);

#ifdef XTRACE
        fprintf(stdout,"<-- logent()\n");
        fflush(stdout);
#endif

        return(SUCCESS);
}
```

```
/*****************************************************************
*                                                                *
*             MRDSc  --> main <--    version of Jan 1986         *
*                                                                *
*****************************************************************/

#include "mrds.h"


main(argc,argv)

int argc;
char *argv[];
{
        extern int errno;
        register int i;
        char *s;
        int abend(),strcpy();
        void cmd();
        int find_db();
        DBSTATUS data_base;
        REL *rptr;
        DOM *dptr;
        void cmd();

        printf("MRDSc\tVersion %s\n",VERSION);
        /* check correct invocation from shell  */

        if (argc > MAXARGS) { /* too many args */
                fprintf(stderr,"Usage: mrds [-r[0|1]] [-t] [-n xxx]\n");
                abend(0,"");
        }

        /* process command line arguments */

        i = 1;
        while (--argc > 0) { /* get next arg */
                s = argv[i++];
                if (*s == '-')
                        switch(*++s) {
                        case 'r' :
                                RUN = atoi(*++s);
                                if ((RUN != 1) || (RUN != 0)) {
                                        db_err(80,0,-1,"RUN");
                                        RUN = 1;
                                }
                                break;
                        case 't' :
                                DIAGNOSE = 1;
                                break;
                        case 'n' :
                                ++s;
                                ++s;
                                ++i;
                                for (i = 0; i < MAXNAMLEN; i++) {
                                        DBNAME[i] = *s++;
```

```
                                        if (!*s) break;
                                }
                                DBNAME[MAXNAMLEN] = NULLC;
                                if (*s) db_err(84,0,-1,DBNAME);
                                break;
                        default :
                                fprintf(stderr,"\nunknown parm: %s\n",s);
                        }
        }

        /* ############################  GO INTERACTIVE HERE ################## */
        cmd(DBNAME);
}
/* E N D   O F   M A I N */
```

```
#include "mrds.h"
#define MKXBOMB close(xfd);unlink(newname);if(!(x.lbufat))free(x.lbufat);if(!(x.xbufat))free(x.xbufat);return(FAIL);

/* TO DO:
        +  add code to set bit for "first leaf" in status byte on first leaf
*/

Xfb x;          /* global x struct pointer */
int xfd;        /* global file descriptor for Btree file */
REL *rptr;      /* global rel pointer */

int mkindex(rel,xsize,xfill,lsize,lfill)
REL     *rel;
int     xsize;  /* size of internal nodes */
int     xfill;  /* % fill internal nodes  */
int     lsize;  /* size of leaf nodes     */
int     lfill;  /* % fill leaf nodes      */

{
        extern int errno;
        extern char inbufr[],*buffer[];
        register int i,j;
        int filemask,read(),close(),mksep();
        char newname[MAXNAMLEN],*mktemp(),*malloc(),*pgptr,*c;
        char test[3][12];
        long timer(),newmax,newcur;
        Xpghdr xhd;
        Xlfhdr xlfhd;

#ifdef XTRACE
fprintf(stdout,"--> mkindex(%s,%d,%d,%d,%d)\n",rel->relname,xsize,xfill,lsize,lfill); fflush(stdout);
#endif
        if (!rel) return(FAIL);

        rptr = rel;

        if (rel -> mode & BTREE) return(SUCCESS);

        x.lbufat = (char *)(0);
        x.xbufat = (char *)(0);
        newmax = rel -> maxsize / rel -> width; /* max as num of tpls */
        newcur = rel -> cursize / rel -> width; /* cur as num of tpls */
        c = &x;
        for (i = 0; i < XFBSIZE; i++) c[i] = '\0';
        strcpy(newname,BTEMP);
        mktemp(newname);
        filemask = O_CREAT | O_RDWR;
        if ((xfd = open(newname,filemask,XMASK)) < NULL) {
                db_err(53,MKINDEX,errno,newname);
                return(FAIL);
        }

        /* set up "first block" for this relation */

        x.xactime = timer();
        x.rootpos = XFBSIZE;
```

```
        if (xfill < XPGMINFIL || xfill > XPGMAXFIL) {
                sprintf(inbufr,"xfill in %s: %d",rel -> relname,xfill);
                db_err(83,MKINDEX,-1,inbufr);
                xfill = XPGDFLTFIL;
        }
        x.xpgfill = xfill;
        if (lfill < LPGMINFIL || lfill > LPGMAXFIL) {
                sprintf(inbufr,"lfill in %s: %d",rel -> relname,lfill);
                db_err(83,MKINDEX,-1,inbufr);
                lfill = LPGDFLTFIL;
        }
        x.lpgfill = lfill;
        if (xsize < XPGMINSIZE || xsize > XPGMAXSIZE) {
                sprintf(inbufr,"xsize in %s: %d",rel -> relname,xsize);
                db_err(83,MKINDEX,-1,inbufr);
                xsize = XPGDFLTSIZE;
        }
        x.xpgsize = xsize;
        if (lsize < LPGMINSIZE || lsize > LPGMAXSIZE) {
                sprintf(inbufr,"lsize in %s: %d",rel -> relname,lsize);
                db_err(83,MKINDEX,-1,inbufr);
                lsize = LPGDFLTSIZE;
        }
        x.lpgsize = lsize;
        x.first_tpl = XFBSIZE  + x.xpgsize + XLFHDRLEN; /* disk addrs of */
        if ((x.xbufat = malloc(x.xpgsize)) == NULL) {
                db_err(52,MKINDEX,errno,"xpg buffer");
                MKXBOMB
        }
        for (i = 0; i < x.xpgsize; i++) *((x.xbufat) + i) = '\0';
        xhd.status = x.xbufat;
        xhd.spare = (x.xbufat) + 1;
        xhd.space = (ushort *)((x.xbufat) + 2);
        xhd.pred = (long *)((x.xbufat) + 4);

        if ((x.lbufat = malloc(x.lpgsize)) == NULL) {
                db_err(52,MKINDEX,errno,"xlf buffer");
                MKXBOMB
        }
        for (i = 0; i < x.lpgsize; i++) *((x.lbufat) + i) = '\0';
        xlfhd.status = x.lbufat;
        xlfhd.spare = (x.lbufat) + 1;
        xlfhd.space = (ushort *)((x.lbufat) + 2);
        xlfhd.pred = (long *)((x.lbufat) + 4);
        xlfhd.offset = (ushort *)((x.lbufat) + 8);

        x.xpgat = -1L;
        x.lpgat = -1L;

        /* prepare first leaf page */

        *(xlfhd.status) = XLEAF | XFRSTLF ;
        *(xlfhd.space) = (ushort)(x.lpgsize - (XLFHDRLEN + rel -> width + sizeof(long)));
        *(xlfhd.pred) = (long)(XFBSIZE);
        *(xlfhd.offset) = (ushort)(XLFHDRLEN);
        pgptr = x.lbufat + XLFHDRLEN;
```

```
        rptr -> rindx = 0L; /* ensure at start of relation */
        if (rdtuple(rel,pgptr) <= 0) {
                db_err(53,MKINDEX,errno,rel -> relname);
                MKXBOMB
        }

        /* prepare root page */

        *(xhd.status) = XROOT;
        *(xhd.space) = (ushort)(x.xpgsize - (XHDRLEN + sizeof(long)));
        *(xhd.pred) = 0L;
        pgptr = x.xbufat + XHDRLEN;
        *(long *)(pgptr) = XFBSIZE + x.xpgsize;

        /* pages are ready, write them out */

        if (flushpage(x.lbufat,xfd,(long)(XFBSIZE + x.xpgsize),x.lpgsize) == FAIL) {
                sprintf(inbufr,"%s: first leaf",rel -> relname);
                db_err(54,MKINDEX,errno,inbufr);
                MKXBOMB
        }

        if (flushpage(x.xbufat,xfd,(long)(XFBSIZE),x.xpgsize) == FAIL) {
                sprintf(inbufr,"%s: root",rel -> relname);
                db_err(54,MKINDEX,errno,inbufr);
                MKXBOMB
        }

        pgptr = (char *)(&x);
        if (flushpage(pgptr,xfd,0L,XFBSIZE) == FAIL) {
                sprintf(inbufr,"%s: xfb",rel -> relname);
                db_err(54,MKINDEX,errno,inbufr);
                MKXBOMB
        }

        /* now begin real work! */

        printf(":::testing \"find\": find first tuple returns ");
        i = 0;
        while (!(rdtuple(rel,0L) <= 0)) {
/*              i = (int)(find(buffer[rel->fd]));
                printf("%d\n",i);
*/
#ifdef XTRACE
                rdtuple(rel,test[0]);fprintf(stdout,"|");
for(i=0;i<12;i++)fprintf(stdout,"%c",test[0][i]);fprintf(stdout,"|\n");
                rdtuple(rel,test[1]);
for(i=0;i<12;i++)fprintf(stdout,"%c",test[1][i]);fprintf(stdout,"|\n");
                i = mksep(rel,test[0],test[1],test[2]);
                fprintf(stdout,"Separator len = %d |",i);
for(i=0;i<12;i++)fprintf(stdout,"%c",test[2][i]);fprintf(stdout,"|\n");
                fprintf(stdout,"|\n"); fflush(stdout);
#endif
                /*TEMP*/ return(SUCCESS);
/*              if ((insert(xfd,buffer[rel->fd])) == FAIL) {
                        ++i;
```

```
                }
*/
        }

        if (i) {
                sprintf(inbufr,"%s (%d) left in %s",rel -> relname,i,newname);
                db_err(81,MKINDEX,-1,inbufr);
        }

        else { /* replace original with temporary copy */
                closerel(rel);
                if (replace(rel,newname) < SUCCESS) {
                        /* very bad news! */
                        sprintf(inbufr,"%s replacing %s",newname,rel->relname);
                        db_err(59,MKINDEX,errno,inbufr);
                        return(-2); /* special ret val: replace failed */
                }
                rel -> mode |= BTREE | ORDER;
                rel -> mode &= ~FLAT;
                rel -> maxsize = newmax;
                rel -> cursize = newcur;
        }
/* remember to trash page pointers */
}
```

```
#define XNXTLL(a) ( (char *) ((long)((*(ushort *)(a+(*(ushort *)(a)>>3)+sizeof(long)))) + (long)(x.xbufat)) )
#define LNXTLL(a) (*(long *)((long)(a)+(long)(rptr->width)))
#define XNXTPG(a) (*(long *)(a+(*(ushort *)(a)>>3)))

/*
long find(tpl)  (basically same code as "seach" uses)
        Look through index for the tuple pointed at by "tpl". Return:
                < -1  tuple not found. Absolute value plus 1 is relation
                      address of tuple just after which sought tuple
                      would have appeared.

                = -1  something caused find to fail

                >= 0  tuple was found. Return value is address of tuple
                      within relation.

        Local macros:
                XNXTLL(a)       find start of next entry in linked list
                                on internal page; "a" must point at (s,p)
                                pair (and NOT at (p0)).

                XNXTPG(a)       return the "p" value from an (s,p) pair
                                pointed at by "a".

                LNXTLL(a)       find start of next tuple after current one
                                pointed at by "a" in linked list of tuples.
                                NOTE that tuple addrs returned may be off
                                current leaf page.


*/

long find(tpl)
char    *tpl;    /* tuple to look for */

{
        extern int errno;
        char ordered,*c,*malloc(),found,scanpg,*pgptr,*prevpgptr,*getrelnam();
        register int i,j;
        int tplcmp(),cmpseptpl(),free(),stat();
        struct stat fstatbuf;
        long pos,xp0,pgtoread;

                found = '\0';
                pgtoread = x.rootpos;

                while (!found) {
                        if((i = loadpage(pgtoread)) == FAIL) { /* read error */
                                db_err(53,FIND,errno,rptr -> relname);
                                return(FAIL);
                        }

                        if (i > 0) { /* internal node page */
                                xp0 = *(long *)((x.xbufat) + XHDRLEN); /* get p0 pointer */
```

```
                        pgptr = (x.xbufat) + XHDRLEN + sizeof(long); /* get sl pointer */

                        /* are there separators on this page? */
                        i = *(ushort *)((x.xbufat) + 2) - (x.xpgsize - XHDRLEN - sizeof(long));
                        if (i > 0) { /* corruption */
                                db_err(12,FIND,-1,rptr -> relname);
                                return(FAIL);
                        }
                        if (i < 0)  { /* yes, are separators */
                                scanpg = 't';
                                prevpgptr = xp0;
                                while (scanpg) {
                                        j = cmpseptup(pgptr,tpl);
                                        if (j > 0) { /* sep > tpl */
                                                pgtoread = prevpgptr;
                                                scanpg = '\0';
                                        }
                                        else { /* sep <= tpl; keep looking */
                                                c = XNXTLL(pgptr);
                                                if (c) {
                                                        prevpgptr = XNXTPG(pgptr);
                                                        pgptr = c;
                                                }
                                                else {
                                                        pgtoread = XNXTPG(pgptr);
                                                        scanpg = '\0';
                                                }
                                        }
                                }
                        }
                        else { /* no separators this page */
                                pgtoread = xp0;
                        }
                } /* end if was internal node page */
                else { /* is a leaf page */
                        pgptr = (char *)((long)(x.lbufat) +(long) (*(ushort *)((x.lbufat)+8))); /* get first tpl */
                        scanpg = 't';
                        prevpgptr = (char *)(NULL);
                        while (scanpg) {
                                j = tplcmp(pgptr,tpl);
                                if (j == 0) /* hit */
                                        return((pgptr - x.lbufat) + x.lpgat);
                                if (j < 0) { /* don't know yet */
                                        pos = LNXTLL(pgptr);
                                        if (pos && pos < (x.lpgat + x.lpgsize)) {
                                                prevpgptr = pgptr;
                                                pgptr = pos - x.lpgat + x.lbufat;
                                        }
                                        else /* miss */
                                                return(-(pgptr - (x.lbufat) + x.lpgat));
                                }
                                else if (j > 0) { /* miss */
#ifdef XTRACE
fprintf(stdout,"pgptr,x.lbufat,x.lpgat:%lxX,%lxX,%lxX\n",pgptr,x.lbufat,x.lpgat);fflush(stdout);
#endif
                                        if (prevpgptr == (char *)(0))  /* goes before first */
```

```
                                                return (-2L);
                                        else
                                                return(-((prevpgptr - x.lbufat) + x.lpgat) -3L);

                                }
                        } /* end while scanpg */
                } /* end else is leaf pg */
        } /* end while not found */
} /* end find() */
```

```
#include "mrds.h"

REL *mkrel(rname,mode,newZmap,numofdoms,doms,sizelim)
char *rname;                    /* name of new relation    */
char mode;                      /* boolean set ==> conrel  */
long newZmap;                   /* bit map of Z ord atts   */
int numofdoms;                  /* num of doms in dom list */
char doms[][MAXDNMLEN];         /* list of domain names    */
long sizelim;                   /* max number of tuples    */

{
        extern int errno;
        extern REL relcore[];
        extern RD rdcore[];
        extern struct rstat sysrelstat;
        extern char inbufr[];
        extern DOM *finddom();
        extern DBSTATUS *data_base;
        register int i,j,k,m;
        DOM *domains[MAXATTS];
        int width=0,creat(),strlen();
        char *relfile,*getrelnam();
        REL *newrel;

#ifdef XTRACE
fprintf(stdout,"--> mkrel(%s,%x,%d,domlist,%ld)\n",
        rname,mode,numofdoms,sizelim); fflush(stdout);
#endif

        /* any room left? */
        if (sysrelstat.numrelents > MAXRELS) {
                db_err(70,MKREL,-1,"rel full");
                return((REL *)(NULL));
        }

        /* verify rname */
        if (! *(rname)) {
                db_err(58,MKREL,-1,"null name");
                return((REL *)(NULL));
        }

        if (strlen(rname) >= MAXRNMLEN) {
                rname[MAXRNMLEN - 1] = NULLC;
                db_err(84,MKREL,-1,rname);
        }

        if (findrel(rname)) {
                db_err(76,MKREL,-1,rname); /* target rel already exists */
                return((REL *)(NULL));
        }

        /* verify domains and list */
        if ((numofdoms < 1) || (numofdoms > MAXATTS)) {
                sprintf(inbufr," %d",&numofdoms);
                db_err(71,MKREL,-1,inbufr); /* bad number of domains */
                return((REL *)(NULL));
```

```
        }

        /* check doms for duplicates and fix if any */
        j = 0;
        m = 1;
        while (numofdoms > m) {
                for (i = m; i < numofdoms; i++) {
                        if (!(strcmp(doms[i],doms[j]))) {
                                --numofdoms;
                                db_err(78,MKREL,-1,doms[i]); /* this should be warn not err */
                                for (k = i; k < numofdoms; k++)
                                        /* doms[k] = doms[k+1]; */
                                        strcpy(doms[k],doms[k+1]);
                                --i;
                        } /* if duplicated */
                } /* for */
                ++m;
        } /* end while */

        /* check that all these domains really exist */
        j = 0;
        for (i = 0; i < numofdoms; i++)
                if ((domains[i] = finddom(doms[i])) == NULL) {
                        ++j;
                        db_err(79,MKREL,-1,doms[i]);
                }
                else
                        width += domains[i]->len;
        if (j)
                return((REL *)(NULL)); /* try fix domlist? */

        if ((numofdoms + sysrelstat.numrdents) > MAXRDS) {
                db_err(71,-1,MKREL,"would oflow rd");
                return((REL *)(NULL));
        }

#ifdef XTRACE
fprintf(stdout,"*** tuple width in mkrel is %d\n",width);fflush(stdout);
#endif

        /* make entry for rel */
        newrel = &(relcore[sysrelstat.numrelents]);
        strcpy(newrel -> relname,rname);
        newrel -> width = width;
        newrel -> mode = mode;
        newrel -> Zmap = newZmap;
        newrel -> fd = FAIL;
        newrel -> cursize = 0L;
        newrel -> maxsize = (long)(width) * sizelim;
        newrel -> rindx = 0L;
        newrel -> windx = 0L;
        ++(sysrelstat.numrelents);

        /* make file for relation */
        relfile = getrelnam(newrel);
/*      strcpy(inbufr,data_base -> dbs_homedir);
```

```
        strcat(inbufr,rname); */
        if ((i = creat(relfile,CRMASK)) == FAIL) { /* trunc trail blanks!! */
                sprintf(inbufr," %d",&errno);
                strcat(inbufr," ");
                strcat(inbufr,rname);
                db_err(55,MKREL,-1,inbufr); /* cannot create file for */
                --(sysrelstat.numrelents);
                return((REL *)(NULL));
        }
        close(i);

        /* make entry for rd */
        width = 0;
        for (i = 0; i < numofdoms; i++) {
                strcpy(rdcore[sysrelstat.numrdents].relname,rname);
                strcpy(rdcore[sysrelstat.numrdents].domname,doms[i]);
                rdcore[sysrelstat.numrdents].pos = width;
                width += (int)(domains[i]->len);
                ++(sysrelstat.numrdents);
        }
#ifdef XTRACE
fprintf(stdout,"<-- mkrel(X%lx)\n",newrel); fflush(stdout);
#endif

        return(newrel); /* number of entries put in rd for this rel */

}
```

```
#include "mrds.h"

int openrel(rptr,mode)

REL *rptr;
int mode;

{
        extern int errno;
        extern char *buffer[];
        register int fd;
/*      char fname[MAXNAMLEN],name[FILESTRING + MAXNAMLEN]; */
        char *getrelnam(),fname[MAXNAMLEN],*name;

#ifdef XTRACE
        fprintf(stdout,"--> openrel(%s,%d)\n",rptr->relname,mode);
        fflush(stdout);
#endif

        if (rptr -> fd != FAIL) {
#ifdef XTRACE
fprintf(stdout,"<-- openrel(%d) AO\n",rptr -> fd);fflush(stdout);
#endif
                return(rptr -> fd);
        }

        /* was not already open, so open it */

        fd = 0;
        name = getrelnam(rptr);
        fd = open(name,mode);
        if (fd == FAIL)
                return(FAIL);

        rptr -> rindx = 0L; /* WRONG FOR BTREE FILES */
        rptr -> fd = (short)(fd);
        if ((buffer[fd] = malloc(rptr -> width)) == NULL) {
                db_err(52,OPENREL,-1,"openrel");        /* no buffer space; bad news! */
                if (!close(fd))
                        db_err(50,OPENREL,errno,NOSTRING);
                return(FAIL);
        }

#ifdef XTRACE
        fprintf(stdout,"<-- openrel(%d)\n",fd);
        fflush(stdout);
#endif
        return(fd);
}
```

```
char *getrelnam(relp)
REL *relp;

{
        extern char filenm[];
        extern DBSTATUS sys_db;
        register int i = 0;
        char nm[MAXNAMLEN];

#ifdef XTRACE
        fprintf(stdout,"--> getrelnam(%s)\n",relp -> relname);
        fflush(stdout);
#endif
        while ((relp->relname[i] != BLANKC) && (i < (MAXRNMLEN - 1)))
                nm[i] = relp->relname[i++];
        nm[i] = NULLC;

        strcpy(filenm,sys_db.dbs_homedir);
        strcat(filenm,nm);

#ifdef XTRACE
        fprintf(stdout,"<-- getrelnam(%s)\n",filenm);
        fflush(stdout);
#endif

        return(filenm);
}
```

```
#include "mrds.h"
#define SCRNLEN 80
#define SCRNLINES 22

/* TO DO
        +       be sure relation is closed after printing
*/

int outrel(rname,title)
char *rname;     /* name of relation to print */
char *title;     /* a page heading/title      */

{
        extern char *DATAFMT[],inbufr[];
        extern int rdents;
        extern union u_short short_coerce;
        extern union u_int int_coerce;
        extern union u_float flt_coerce;
        extern union u_long long_coerce;
        int pgnum=1,lines=BIGNUM;
        char ttl[LINELEN],hdg[LINELEN],uscore[LINELEN],*tuple,line[BLOCK];
        register int i,j,cursor,k;
        int len, cols, tab[MAXATTS],delimlen,collen[MAXATTS],closerel();
        int dtab[MAXATTS];
        REL *reltoprint;
        RD *rds[MAXATTS];
        DOM *doments[MAXATTS];

#ifdef XTRACE
        fprintf(stdout,"--> printrel(|%s|,|%s|)\n",rname,title);
        fflush(stdout);
#endif

        /* does this rel exist ? */
        if ((reltoprint = findrel(rname)) == NULL) {
                db_err(58,PRINTREL,-1,rname);
                return(NULL);
        }

        /* get the scoop on this relation */
        if ((findrd(rname,0L,0,rds)) == NULL) { /* macdep: 0L ng for 16 bit addrs */
                db_err(62,PRINTREL,-1,rname);
                return(NULL);
        }
        cols = rdents;
        if (rdents > MAXATTS) { /* VERY dangerous: something got clobbered */
                strcpy(inbufr,rname); /* this should never happen (ha!) */
                strcat(inbufr,"(");
                strcat(inbufr,itoa(rdents));
                strcat(inbufr,")");
                db_err(11,PRINTREL,-1,inbufr);
                cols = MAXATTS;
        }

        for (i = 0; i < cols; i++)
                if ((doments[i] = finddom(rds[i]->domname)) == NULL)
```

```
                        db_err(79,PRINTREL,-1,rds[i]->domname);

        /* check line length */
        len = 0;
        for (i = 0; i < cols; i++)
                if (doments[i]) /* don't lookup bad ones */
                        switch (doments[i]->domtype) {
                        case 1: /* integer */
                                len += 11; /* worst case + sign (macdep: 32 bit ints) */
                                collen[i] = 11;
                                break;

                        case 2: /* single char */
                                ++len;
                                collen[i] = 1;
                                break;

                        case 3: /* float */
                                len += 16; /* %g compromise */
                                collen[i] = 16;
                                break;

                        default: /* string */
                                len += doments[i]->len;
                                collen[i] = doments[i]->len;
                                --len; /* don't count null terminator */
                        } /* end switch */
#ifdef XTRACE
        for (i = 0;i < cols; i++) fprintf(stdout,"collen[%d] is %d\n",i,collen[i]);
#endif
        if (len > SCRNLEN) {
                sprintf(inbufr," %d",&len);
                db_err(88,PRINTREL,-1,inbufr);
        }

        /* build centered title line with page number */
        i = strlen(title);
        j = SCRNLEN;
        j = (j >> 1);
        i = (i >> 1);
        j -= i; /* j is offset in SCRNLEN to start title at */
        for (i = 0; i < SCRNLEN;i++) ttl[i] = BLANKC;
        strcpy(&(ttl[j]),title);
        for (j = 0; ttl[j] != NULLC; j++);
        ttl[j] = BLANKC;
        j = SCRNLEN - PGNUMLEN;
        strcpy(&(ttl[j]),PGNUMHDR);

        /* build domain header line */
        delimlen = strlen(DELIM);
        for (i = 0; i < SCRNLEN; i++) hdg[i] = BLANKC;
        cursor = 0;
        for (i = 0; i < cols; i++) {
                j = strlen(doments[i]->domname);
                if (j >= collen[i]) {
                        strcpy(&(hdg[cursor]),doments[i]->domname);
```

```
                        tab[i] = cursor + (((j >> 1) + 1) - ((collen[i] >> 1) + 1));
                        cursor += j;
                        strcpy(&(hdg[cursor]),DELIM);
                        dtab[i] = cursor;
                        cursor += delimlen;
                }
                else {
                        tab[i] = cursor;
                        cursor += (collen[i] >> 1) - (j >> 1);
                        strcpy(&(hdg[cursor]),doments[i]->domname);
                        cursor += j;
                        dtab[i] = cursor;
                        strcpy (&(hdg[cursor]),DELIM);
                        cursor += delimlen;
                }
#ifdef XTRACE
fprintf(stdout,"dtab[%d] is %d\n",i,dtab[i]);
fprintf(stdout,"domain is [%s], j/collen is %d/%d; wrote at %d\n",doments[i]->
domname,j,collen[i],(cursor-delimlen-j));
fprintf(stdout,"HDG is >%s<\n",hdg);
#endif
        }

        for (i = 0; i < cursor; i++)
                if (hdg[i] == NULLC)
                        hdg[i] = BLANKC;
        /* build underscore line */
        for (i = 0; i < SCRNLEN; i++)
                if ((hdg[i] == BLANKC) || (hdg[i] == NULLC))
                        uscore[i] = BLANKC;
                else
                        uscore[i] = '_';

                uscore[SCRNLEN -1 ] = NULLC;
        /* begin printing: have to format each line separately */
        while (tuple = getuple(reltoprint,0L)) {
                for (i = 0; i < SCRNLEN; i++) line[i] = BLANKC;
                k = 0;
                cursor = 0;
                for (i = 0; i < cols; i++) {
                        cursor  = tab[i];
                        sprintf(&(line[dtab[i]]),"%s",DELIM);
                        switch (doments[i]->domtype) {
                        case DT_STRING:
                                sprintf(&(line[cursor]),DATAFMT[DT_STRING],(tuple + k));
                                break;

                        case DT_CHAR:
                                sprintf(&(line[cursor]),DATAFMT[DT_CHAR],(tuple + k));
                                break;

                        case DT_FLOAT:
                                flt_coerce.c1 = *(tuple + k);
                                *(&(flt_coerce.c2) + 1) = *(tuple + k + 1);
                                *(&(flt_coerce.c3) + 2) = *(tuple + k + 2);
                                *(&(flt_coerce.c4) + 3) = *(tuple + k + 3);
```

```
                                sprintf(&(line[cursor]),DATAFMT[DT_FLOAT],flt_coerce.fval);
                                k += 4;
                                break;
                        case DT_SHORT:
#ifdef INT16
                        case DT_INT:
#endif
                                short_coerce.c1 = *(tuple + k);
                                *(&(short_coerce.c2) + 1) = *(tuple + k + 1);
                                sprintf(&(line[cursor]),DATAFMT[DT_SHORT],short_coerce.sval);
                                k += 2;
                                break;

                        case DT_LONG:
#ifndef INT16
                        case DT_INT:
#endif
                                long_coerce.c1 = *(tuple + k);
                                *(&(long_coerce.c2) + 1) = *(tuple + k + 1);
                                *(&(long_coerce.c3) + 2) = *(tuple + k + 2);
                                *(&(long_coerce.c4) + 3) = *(tuple + k + 3);
                                sprintf(&(line[cursor]),DATAFMT[DT_LONG],long_coerce.lval);
                                k += 4;
                                break;

                        default:
                                break;

                        } /* end switch */
j = strlen(&line[cursor]);
line[cursor + (j)] = BLANKC;
                        /* cursor += collen[i]; */
                        k += doments[i]->len; /* former kludge */
                } /* end for */

                if (lines > SCRNLINES) { /* need new page */
                        printf("\nMORE...");
                        getc(stdin);
                        printf("\f\n%s%d\n\n%s\n%s\n",ttl,pgnum,hdg,uscore);
                        lines = 1;
                        ++pgnum;
                }
                for (i = 0; i < SCRNLEN; i++) /* assure no stray nulls */
                        if (! line[i])
                                line[i] = BLANKC;
                line[SCRNLEN - 1] = NULLC; /* assure end of line */
                printf("%s\n",line);
                /* for(j=0; j<SCRNLEN;j++)printf("%c",line[j]);printf("\n") */
                ++lines;
        } /* end while */

#ifdef XTRACE
        fprintf(stdout,"<-- printrel (%d)\n",((--pgnum * SCRNLINES) + lines));
        fflush(stdout);
#endif
```

```
        closerel(reltoprint);
        return ((--pgnum * SCRNLINES) + lines); /* num tuples printed */

}
```

```
#include "mrds.h"

/* TO DO
        +       be sure relation is closed after printing
*/

int printrel(rname,title)
char *rname;     /* name of relation to print */
char *title;     /* a page heading/title      */

{
        extern char *DATAFMT[],inbufr[];
        extern int rdents;
        extern union u_short short_coerce;
        extern union u_int int_coerce;
        extern union u_float flt_coerce;
        extern union u_long long_coerce;
        int pgnum=1,lines=BIGNUM;
        char ttl[LINELEN],hdg[LINELEN],uscore[LINELEN],*tuple,line[BLOCK];
        register int i,j,cursor,k;
        int len, cols, tab[MAXATTS],delimlen,collen[MAXATTS],closerel();
        int dtab[MAXATTS];
        REL *reltoprint;
        RD *rds[MAXATTS];
        DOM *doments[MAXATTS];

#ifdef XTRACE
        fprintf(stdout,"--> printrel(|%s|,|%s|)\n",rname,title);
        fflush(stdout);
#endif

        /* does this rel exist ? */
        if ((reltoprint = findrel(rname)) == (REL *)(NULL)) {
                db_err(58,PRINTREL,-1,rname);
                return(FAIL);
        }

        /* get the scoop on this relation */
        if ((findrd(rname,(char *)(0),0,rds)) == (RD *)(NULL)) {
                db_err(62,PRINTREL,-1,rname);
                return(FAIL);
        }
        cols = rdents;
        if (rdents > MAXATTS) { /* VERY dangerous: something got clobbered */
                strcpy(inbufr,rname); /* this should never happen (ha!) */
                strcat(inbufr,"(");
                strcat(inbufr,itoa(rdents));
                strcat(inbufr,")");
                db_err(96,PRINTREL,-1,inbufr);
                cols = MAXATTS;
        }

        for (i = 0; i < cols; i++)
                if ((doments[i] = finddom(rds[i]->domname)) == (DOM *)(NULL))
                        db_err(79,PRINTREL,-1,rds[i]->domname);
```

```
        /* check line length */
        len = 0;
        for (i = 0; i < cols; i++)
                if (doments[i]) /* don't lookup bad ones */
                        switch (doments[i]->domtype) {
                        case 1: /* integer */
                                len += 11; /* worst case + sign (macdep: 32 bit ints) */
                                collen[i] = 11;
                                break;

                        case 2: /* single char */
                                ++len;
                                collen[i] = 1;
                                break;

                        case 3: /* float */
                                len += 16; /* %g compromise */
                                collen[i] = 16;
                                break;

                        default: /* string */
                                len += doments[i]->len;
                                collen[i] = doments[i]->len;
                                --len; /* don't count null terminator */
                        } /* end switch */
#ifdef XTRACE
        for (i = 0;i < cols; i++) fprintf(stdout,"collen[%d] is %d\n",i,collen[i]);
#endif
        if (len > LINELEN) {
                sprintf(inbufr," %d",&len);
                db_err(88,PRINTREL,-1,inbufr);
        }

        /* build centered title line with page number */
        i = strlen(title);
        j = LINELEN;
        j = (j >> 1);
        i = (i >> 1);
        j -= i; /* j is offset in LINELEN to start title at */
        for (i = 0; i < LINELEN;i++) ttl[i] = BLANKC;
        strcpy(&(ttl[j]),title);
        for (j = 0; ttl[j] != NULLC; j++);
        ttl[j] = BLANKC;
        j = LINELEN - PGNUMLEN;
        strcpy(&(ttl[j]),PGNUMHDR);

        /* build domain header line */
        delimlen = strlen(DELIM);
        for (i = 0; i < LINELEN; i++) hdg[i] = BLANKC;
        cursor = 0;
        for (i = 0; i < cols; i++) {
                j = strlen(doments[i]->domname);
                if (j >= collen[i]) {
                        strcpy(&(hdg[cursor]),doments[i]->domname);
                        tab[i] = cursor + (((j >> 1) + 1) - ((collen[i] >> 1) + 1));
                        cursor += j;
```

```
                        strcpy(&(hdg[cursor]),DELIM);
                        dtab[i] = cursor;
                        cursor += delimlen;
                }
                else {
                        tab[i] = cursor;
                        cursor += (collen[i] >> 1) - (j >> 1);
                        strcpy(&(hdg[cursor]),doments[i]->domname);
                        cursor += j;
                        dtab[i] = cursor;
                        strcpy (&(hdg[cursor]),DELIM);
                        cursor += delimlen;
                }
#ifdef XTRACE
fprintf(stdout,"dtab[%d] is %d\n",i,dtab[i]);
fprintf(stdout,"domain is [%s], j/collen is %d/%d; wrote at %d\n",doments[i]->
domname,j,collen[i],(cursor-delimlen-j));
fprintf(stdout,"HDG is >%s<\n",hdg);
#endif
        }

        for (i = 0; i < cursor; i++)
                if (hdg[i] == NULLC)
                        hdg[i] = BLANKC;
        /* build underscore line */
        for (i = 0; i < LINELEN; i++)
                if ((hdg[i] == BLANKC) || (hdg[i] == NULLC))
                        uscore[i] = BLANKC;
                else
                        uscore[i] = '_';

                uscore[LINELEN -1 ] = NULLC;
        /* begin printing: have to format each line separately */
        while (tuple = getuple(reltoprint,0L)) {
                for (i = 0; i < LINELEN; i++) line[i] = BLANKC;
                k = 0;
                cursor = 0;
                for (i = 0; i < cols; i++) {
                        cursor  = tab[i];
                        sprintf(&(line[dtab[i]]),"%s",DELIM);
                        switch (doments[i]->domtype) {
                        case DT_STRING:
                                sprintf(&(line[cursor]),DATAFMT[DT_STRING],(tuple + k));
                                break;

                        case DT_CHAR:
                                sprintf(&(line[cursor]),DATAFMT[DT_CHAR],(tuple + k));
                                break;

                        case DT_FLOAT:
                                flt_coerce.c1 = *(tuple + k);
                                *(&(flt_coerce.c2) + 1) = *(tuple + k + 1);
                                *(&(flt_coerce.c3) + 2) = *(tuple + k + 2);
                                *(&(flt_coerce.c4) + 3) = *(tuple + k + 3);
                                sprintf(&(line[cursor]),DATAFMT[DT_FLOAT],flt_coerce.fval);
                                k += 4;
```

```
                                break;
                        case DT_SHORT:
#ifdef INT16
                        case DT_INT:
#endif
                                short_coerce.c1 = *(tuple + k);
                                *(&(short_coerce.c2) + 1) = *(tuple + k + 1);
                                sprintf(&(line[cursor]),DATAFMT[DT_SHORT],short_coerce.sval);
                                k += 2;
                                break;

                        case DT_LONG:
#ifndef INT16
                        case DT_INT:
#endif
                                long_coerce.c1 = *(tuple + k);
                                *(&(long_coerce.c2) + 1) = *(tuple + k + 1);
                                *(&(long_coerce.c3) + 2) = *(tuple + k + 2);
                                *(&(long_coerce.c4) + 3) = *(tuple + k + 3);
                                sprintf(&(line[cursor]),DATAFMT[DT_LONG],long_coerce.lval);
                                k += 4;
                                break;

                        default:
                                break;

                        } /* end switch */
j = strlen(&line[cursor]);
line[cursor + (j)] = BLANKC;
                        /* cursor += collen[i]; */
                        k += doments[i]->len; /* former kludge */
                } /* end for */

                if (lines > PAGELINES) { /* need new page */
                        printf("\f\n%s%d\n\n%s\n%s\n",ttl,pgnum,hdg,uscore);
                        lines = 1;
                        ++pgnum;
                }
                for (i = 0; i < LINELEN; i++) /* assure no stray nulls */
                        if (! line[i])
                                line[i] = BLANKC;
                line[LINELEN - 1] = NULLC; /* assure end of line */
                printf("%s\n",line);
                /* for(j=0; j<LINELEN;j++)printf("%c",line[j]);printf("\n") */
                ++lines;
        } /* end while */
        printf("\f"); /* eject last page */

#ifdef XTRACE
        fprintf(stdout,"<-- printrel (%d)\n",((--pgnum * PAGELINES) + lines));
        fflush(stdout);
#endif

        closerel(reltoprint);
        return ((--pgnum * PAGELINES) + lines); /* num tuples printed */
```

```
}
```

```
#include "mrds.h"

/* TO DO:
        +       be sure relation is closed after projection
*/

int project(rel,domlist,ndoms,newrel)
char *rel;                          /* rel to be projected             */
char domlist[][MAXDNMLEN];          /* list of domain names            */
short ndoms;                        /* number of domains in projection */
char *newrel;                       /* name of new (output) relation   */

{
        register int i,j,k;
        extern int rdents,errno;
        extern char inbufr[];
        int domlength[MAXATTS],n,m,outrelwidth;
        int db_err(),closerel(),sortrel(),adtuple();
        long numoftpls,bitmap=0L;
        RD *rdtab[MAXATTS],*findrd();
        REL *inrel,*outrel,*findrel(),*mkrel();
        DOM *domptr,*finddom();
        char *tplin,*tplout,domnames[MAXATTS][MAXDNMLEN];
        char *getuple(),*malloc();

#ifdef XTRACE
fprintf(stdout,"--> project(%s,domlist,%d,%s)\n",rel,
        ndoms,newrel); fflush(stdout);
#endif
        /* find input relation */
        if ((inrel = findrel(rel)) == (REL *)(NULL)) {
                db_err(58,PROJECT,-1,rel);
                return(FAIL);
        }

        /* check number of domains supplied */
        if ((findrd(rel,(char *)(0),0,rdtab)) == (RD *)(NULL)) {
                db_err(58,PROJECT,-1,rel); /* diff ersect from above */
                return(FAIL);
        }
        j = rdents;
        numoftpls = inrel -> cursize  / (long)(inrel -> width);

        if (ndoms > rdents  ||  ndoms < 0) {
                sprintf(inbufr,"%s (%d)",rel,ndoms);
                db_err(77,PROJECT,-1,inbufr);
                ndoms = 0;
        }

        if (ndoms)
            findrd(rel,domlist,ndoms,rdtab);
        else
                for (i = 0; i < rdents; i++)
                        strcpy(domnames[i],rdtab[i] -> domname);

       /* for a subset of full tuple need individual domain lengths */
```

```
        outrelwidth = 0;
        if (ndoms) {
                for (i = 0; i < ndoms; i++) {
                        if ((domptr = finddom(domlist[i])) == (DOM *)(NULL)){
                                db_err(79,PROJECT,-1,domlist[i]);
                                domlength[i] = 0;
                        }
                        else {
                                domlength[i] = (int)(domptr -> len);
                                bitmap |= 1L << i;
                        }
                        outrelwidth += domlength[i];
                }
        }
        else {
                outrelwidth = inrel -> width;
                bitmap = (1L << rdents) - 1L;
        }
#ifdef XTRACE
fprintf(stdout,"*** tuple width in project is %d\n",outrelwidth);
fflush(stdout);
#endif

        if (ndoms)
                if (!(tplout = malloc(outrelwidth))) {
                        db_err(52,PROJECT,errno,"tplout");
                        return(FAIL);
                }

#ifdef XTRACE
fprintf(stdout,"\ttplout alloc @ %lX\n",tplout);fflush(stdout);
#endif
        /* set up new output relation */

        bitmap &= inrel -> Zmap;
        if (ndoms)
            outrel = mkrel(newrel,inrel->mode,bitmap,ndoms,domlist,numoftpls);
        else
                outrel = mkrel(newrel,inrel->mode,bitmap,j,domnames,numoftpls);
        if (outrel == (REL *)(0)){
                db_err(57,PROJECT,-1,newrel);
                return(FAIL);
        }

        /* project */
        m = 0;
        while (tplin = getuple(inrel,0L)) {
                if (ndoms) {
                        k = 0;
                        for (i = 0; i < ndoms; i++) {
                                for (j = 0; j < domlength[i]; j++)
                                        tplout[k + j] = tplin[j + rdtab[i] -> pos];
/* fprintf(stdout,"tplout[%d] became |%c|\n",k+j,tplout[k+j]);fflush(stdout);*/
                                k += j;
                        }
                }
```

```
                else
                        tplout = tplin;
                if ((adtuple(outrel,tplout)) != (int)(outrel -> width)) {
                    ++m;
                    n = errno;
                }
        }

        /* did project work? */
        if (m) {
                sprintf(inbufr,"%d",m);
                db_err(81,PROJECT,n,inbufr);
        }

        if ((int)(outrel -> fd) != FAIL)
            closerel(outrel);
        if ((int)(inrel -> fd) != FAIL)
            closerel(inrel);

        if ((j = sortrel(outrel)) < SUCCESS)
            db_err(82,PROJECT,-1,outrel -> relname);

#ifdef XTRACE
fprintf(stdout,"<-- project(1)\n");fflush(stdout);
#endif
        if (m == 0)
                if (j == SUCCESS) return(SUCCESS);
                else return (-3);
        else return(-2);

}
```

```
#include "mrds.h"

/* TO DO:
        + make sure that the relevant xpgat or lpgat fields
          are correctly updated upon read.
*/

int rdpage(rptr,ptr,x)
REL  *rptr;
long *ptr;
Xfb  *x;

{
        extern int errno;
        register int i,j;
        register long pos;
        int read(),openrel(),db_err();
        long lseek();
        char leaf, *buf;

#ifdef XTRACE
fprintf(stdout,"--> rdpage(%s,X%lx)\n",rptr -> relname,ptr);
fflush(stdout);
#endif

        if (!rptr) return (FAIL);

        if (rptr->fd == FAIL) /* file holding rel not open yet */
                if ((rptr->fd = openrel(rptr,READMODE)) == FAIL) {
                        db_err(70,RDPAGE,-1,rptr->relname);
                        return(FAIL);
                }

        /* conservatism since lseek check should catch this error */

        if ((ptr < 0L) || (ptr > rptr -> cursize)) { /* ptr not within rel */
                db_err(85,RDPAGE,-1,rptr -> relname);
                return(FAIL);
        }

        /* could add check to assure 'pos' is on a page boundary here */

        /* go to that page */

        if ((pos = lseek(rptr -> fd,ptr,FROMTOP)) != ptr) {
                db_err(60,RDPAGE,errno,rptr -> relname);
                return(FAIL);
        }

        if ((i = read(rptr -> fd, leaf, 1) != 1)) {
                db_err(60,RDPAGE,errno,rptr -> relname);
                return(FAIL);
        }

        if ((pos = lseek(rptr -> fd,-1L,FROMCUR)) != ptr) { /* cannot happen */
                db_err(60,RDPAGE,errno,rptr -> relname);
```

```
                return(FAIL);
        }

        leaf &= XLEAF;
        if (leaf) /* is a leaf page */
                buf = x -> lbufat;
        else
                buf = x -> xbufat;

        j = (leaf ? x -> lpgsize : x -> xpgsize);

        if ((i = read(rptr -> fd, buf, j)) != j) {
                db_err(60,RDPAGE,errno,rptr -> relname);
                return(FAIL);
        }

        /* no pointers to update: rindx should only point at tuples */

#ifdef XTRACE
fprintf(stdout,"<-- rdpage(size = %d, leaf = %d)\n",i,(leaf ? 1 : 0));
fflush(stdout);
#endif

        return((int)(leaf));
}
```

```
#include "mrds.h"

int rdtuple(rptr,to)
REL  *rptr;
char *to;

{
        extern int errno;
        extern char *buffer[];
        register int i;
        register long pos;
        int read(),openrel(),db_err();
        long lseek();

#ifdef XTRACE
fprintf(stdout,"--> rdtuple(%s,X%lx)\n",rptr -> relname,to);
fflush(stdout);
#endif

        if (!rptr) return (FAIL);

        if (rptr->fd == FAIL) /* file holding rel not open yet */
                if ((rptr->fd = openrel(rptr,READMODE)) == FAIL) {
                        db_err(51,RDTUPLE,-1,rptr->relname);
                        return(FAIL);
                }

        /* conservatism since lseek check should catch this error */

        if (rptr -> rindx > (rptr -> cursize - (long)(rptr -> width))) {
/*              db_err(85,RDTUPLE,-1,rptr -> relname);  */
                return(FAIL);
        }

        if ((pos = lseek(rptr -> fd, rptr -> rindx,FROMTOP)) != rptr -> rindx){
                db_err(60,RDTUPLE,errno,rptr -> relname);
                return(FAIL);
        }

        to = (to ? to : buffer[rptr -> fd]);
        if ((i = read(rptr -> fd, to,(int)(rptr -> width))) != (int)(rptr -> width)){
                db_err(60,RDTUPLE,errno,rptr -> relname);
                rptr -> rindx = pos; /*reset to tuple start */
                return(FAIL);
        }

        /* read succeeded */
        if (rptr -> mode & BTREE) {
                pos = rptr -> rindx;
                if (read(rptr -> fd,&(rptr -> rindx),sizeof(long)) != sizeof (long)) {
                        db_err(92,RDTUPLE,errno,rptr -> rindx);
                        rptr -> rindx = pos;
                }
        }
        else rptr -> rindx += (long)(rptr -> width);
#ifdef XTRACE
```

```
fprintf(stdout,"<-- rdtuple(width = %d)\n",rptr -> width);
fflush(stdout);
#endif

        return(i); /* number of bytes read */
]
```

```
#include "mrds.h"

int replace(oldrel,newfile)
REL *oldrel;   /* ptr to sysrel entry for rel being replaced */
char *newfile; /* name of file becoming the relation */

{
        extern int errno;
        extern char inbufr[];
        struct stat tmpstat;
        char oldrelfile[FILESTRING + MAXNAMLEN], block[BLOCK];
        char *getrelnam();
        int i,j,k,m,n,badnews,open(),unlink(),stat();
        long newsize,lseek();

#ifdef XTRACE
fprintf(stdout,"--> replace(%s,%s)\n",oldrel->relname,newfile);
fflush(stdout);
#endif
        /* get filename of existing relation */
/*      oldrelfile = getrelnam(oldrel); */
        strcpy(oldrelfile,getrelnam(oldrel));

        /* check rel file not in use */
        if (oldrel -> fd != FAIL) {
                db_err(65,REPLACE,-1,oldrel -> relname);
                return(FAIL);
        }

        /* get new relation size */
        if ((i = open(newfile,READMODE)) == FAIL) {
                db_err(51,REPLACE,errno,newfile);
                return(FAIL);
        }
        newsize = lseek(i,0L,FROMEND);
        lseek(i,0L,FROMTOP);

        /* move the newfile to oldrelfile */
        if ((j = creat(oldrelfile,CRMASK)) == FAIL) {
                db_err(51,REPLACE,errno,oldrel -> relname);
                return(FAIL);
        }
        badnews = 0;

        while ((k = read(i,block,BLOCK)) > 0) {
                m = (k < BLOCK ? k : BLOCK);
                if ((n = write(j,block,m)) != k) {
                        sprintf(inbufr,"%d:%d (%s)",k,n,newfile);
                        db_err(25,REPLACE,errno,inbufr);
                        ++badnews;
                        newsize = lseek(j,0L,2);
                        break;
                }
        }

        /* tidy up */
```

```
        oldrel -> cursize = newsize;
        close(i);
        close(j);
        oldrel -> rindx = 0L;
        oldrel -> windx = newsize;
        if (!badnews)
                if (unlink(newfile) == FAIL)
                        db_err(83,REPLACE,errno,newfile);

#ifdef XTRACE
fprintf(stdout,"<-- replace()\n");fflush(stdout);
#endif
        if (badnews) return(-2);
        else return(SUCCESS);
}
```

```
#include "mrds.h"

#define SBOMB free(x->xbufat);free(x->lbufat);x->xbufat=(char *)(0);x->lbufat=(char *)(0)
#define XNXTLL(a) ((char *)(*(ushort *)(a+sepentlen(a)+sizeof(long))))
#define LNXTLL(a) (*(long *)(a+rptr->width))
#define XNXTPG(a) (*(long *)(a+sepentlen(a)))

/*
long search(rptr,tpl)
        Look through relation pointed at by "rptr" for the tuple
        pointed at by "tpl". Return:
                < -1  tuple not found. Absolute value plus 2 is relation
                      address of tuple just after which sought tuple
                      would have appeared.

                = -1  something caused search to fail

                >= 0  tuple was found. Return value is address of tuple
                      within relation.

        Local macros:
                XNXTLL(a)       find offset to next entry in linked list
                                on internal page; "a" must point at (s,p)
                                pair (and NOT at (p0)).

                XNXTPG(a)       return the "p" value from an (s,p) pair
                                pointed at by "a".

                LNXTLL(a)       find start of next tuple after current one
                                pointed at by "a" in linked list of tuples.
                                NOTE that tuple addrs returned may be off
                                current leaf page.


*/

long search(rptr,tpl)
REL     *rptr;   /* rel to look through */
char    *tpl;    /* for this tuple      */

{
        extern int errno;
        extern char *buffer[];
        char ordered,*c,*malloc(),found,scanpg,*pgptr,*getrelnam();
        char *prevtpl;
        register int i,j;
        int tplcmp(),cmpseptpl(),free(),stat();
        Xfb *x,xfb[];
        struct stat fstatbuf;
        long timer(),deltatm,now,etime,xp0,pgtoread,prevpgptr;
        ushort *space;

        if (!rptr) return(FAIL);

        if (rptr -> mode & FLAT) {
                ordered = (rptr -> mode & ORDER);
```

```
                rptr -> rindx = 0L;  /* ensure relation is rewound */
                while (rdtuple(rptr,0L)) {
                        i = tplcmp(tpl,buffer[rptr -> fd]);
                        if (!i) /* got it */
                                return(rptr -> rindx - (long)(rptr -> width));
                        else if (i > 0 && ordered) { /* have overshot */
                                pgtoread = rptr -> rindx - (long)(rptr -> width);
                                if (pgtoread == 0L) /* goes before first */
                                        return(-2L);
                                return (-pgtoread - 3L);
                        }
                }
        }
        else if (rptr -> mode & BTREE) {
                /* find slot for this relation */

                x = (Xfb *)(0);
                if (stat(getrelnam(rptr),&fstatbuf) == FAIL) {
                        db_err(89,SEARCH,errno,rptr->relname);
                        fstatbuf.st_mtime = 0x7fffffff; /* assumes times are always in longs */
                }
                for (i = 0; i < XFBSLOTS; x++)
                        if (!strcmp(xfb[i].rname,rptr->relname) && fstatbuf.st_mtime < xfb[i].xactime) {
                                x = &xfb[i];
                                xfb[i].xactime = timer();
                                break;
                        }

                if (x == (Xfb *)(0)){ /* set up new slot */
                        /*find one to use */
                        deltatm = etime = 0L;
                        now = timer();
                        for (i = 0; i < XFBSLOTS; i++)
                                if ((deltatm = now - xfb[i].xactime) > etime) {
                                        etime = deltatm;
                                        x = &xfb[i];
                                }

                        /* initialize slot */
                        if (( i = read(rptr -> fd, x,XFBSIZE)) != XFBSIZE) {
                                db_err(53,SEARCH,errno,rptr -> relname);
                                return(FAIL);
                        }

                        x -> xactime = now;
                        if (x -> xbufat == (char *)(NULL)) /* alloc internal node buf */
                                if ((x -> xbufat = malloc(x -> xpgsize)) == (char *)(NULL)) {
                                        db_err(52,SEARCH,errno,rptr -> relname);
                                        return(FAIL);
                                }

                        if (x -> lbufat == (char *)(NULL))
                                if ((x -> lbufat = malloc(x -> lpgsize)) == (char *)(NULL)) {
                                        db_err(52,SEARCH,errno,rptr -> relname);
                                        free(x -> xbufat);
                                        x -> xbufat = (char *)(NULL);
```

```
                                return(FAIL);
                        ]

        }
        /* now begin search */

        found = '\0';
        pgtoread = x -> rootpos;
        prevtpl = x -> lbufat;
        space = (ushort *)(x -> xbufat + X_SPCOFF);

        while (!found) {
                if((i = rdpage(rptr,pgtoread)) == FAIL) { /* read error */
                        db_err(53,SEARCH,errno,rptr -> relname);
                        SBOMB;
                ]

                if (!i) { /* internal node page */
                        xp0 = *(long *)((x -> xbufat) + XHDRLEN); /* get p0 pointer */
                        pgptr = (x -> xbufat) + XHDRLEN + sizeof(long); /* get sl pointer */

                        /* are there separators on this page? */
                        i = *space - (x ->xpgsize - XHDRLEN - sizeof(long) - sizeof(ushort));
                        if (i > 0) { /* corruption */
                                db_err(28,SEARCH,-1,rptr -> relname);
                                SBOMB;
                        }
                        if (i < 0)  { /* yes, are separators */
                                scanpg = 't';
                                prevpgptr = xp0;
                                while (scanpg) {
                                        j = cmpseptup(rptr,pgptr,tpl);
                                        if (j > 0) { /* sep > tpl */
                                                pgtoread = prevpgptr;
                                                scanpg = '\0';
                                        }
                                        else { /* sep <= tpl; keep looking */
                                                c = XNXTLL(pgptr);
                                                if (c) {
                                                        prevpgptr = XNXTPG(pgptr);
                                                        pgptr =(char *)((unsigned long)(c) + (unsigned long)(x -> xbufat));
                                                }
                                                else {
                                                        pgtoread = XNXTPG(pgptr);
                                                        scanpg = '\0';
                                                }
                                        }
                                }
                        }
                        else { /* no separators this page */
                                pgtoread = xp0;
                        }
                } /* end if was internal node page */
                else { /* is a leaf page */
                        prevtpl = (char *)(0);
                        pgptr = x -> lbufat + *(ushort *)(x -> lbufat + L_FLEOFF);/* get first tpl */
```

```
                                scanpg = 't';
                                while (scanpg) {
                                        j = tplcmp(pgptr,tpl);
                                        if (j == 0) /* hit */
                                                return((pgptr - x -> lbufat) + x -> lpgat);
                                        if (j < 0) { /* don't know yet */
                                                c = LNXTLL(pgptr);
                                                /* assumes logically "higher" pages
                                                are always physically higher
                                                as well */
                                                if ( c && c < (x -> lpgat + x -> lpgsize)) {
                                                        prevtpl = pgptr;
                                                        pgptr = c;
                                                }
                                                else {
                                                        pgtoread = *(long *)(x -> lbufat + L_SUCCOFF);
                                                        scanpg = '\0';
                                                }
                                        }
                                        else if (j > 0) { /* miss */
                                                if (prevtpl == (char *)(0))  /* goes before first */
                                                        return (-2L);
                                                else
                                                        return(-((prevtpl - x -> lbufat) + x -> lpgat) -3L);
                                        }
                                } /* end while scanpg */
                        } /* end else is leaf pg */
                } /* end while not found */
        } /* end BTREE search */
} /* end search() */
```

```
/*
int cmpseptup(rel,sep,tup)
REL *rel;
char *sep;
char *tup;

        Compare tuple pointed at by 'tup' with separator pointed at by
        'sep' in relation pointed at by 'rel'. Possible return values
        are:
                >  0    separator > tuple
                =  0    separator = tuple
                = -1    failure
                < -1    separator < tuple
*/

int cmpseptup(rel,sep,tup)
REL *rel;       /* ptr to rel       */
char *sep;      /* ptr to separator */
char *tup;      /* ptr to tuple     */
{

        extern int rdents;
        extern char inbufr[];
        register int i,j,seplen;
        Sepslot *sinfo;
        register REL *r;
        register char *s,*t;
        extern union u_short short_coerce;
        extern union u_long long_coerce;
        extern union u_float flt_coerece;
        long timer(),longcmp,deltatm,now,etime;
        int intcmp;
        short shortcmp;
        float floatcmp;
        char zmask;
        RD *rd,*findrd(),rdtab[MAXATTS];
        DOM *domain,*finddom();

        if (!rel) return (FAIL);

        /* find slot for this relation */

        sinfo = (Sepslot *)(NULL);
        for (i = 0; i < SEPSLOTS; i++)
                if (rel == sepslot[i].rptr) {
                        sinfo = &sepslot[i];
                        sepslot[i].actime = timer();
                        break;
                ]

        if (sinfo == (Sepslot *)(NULL)) { /* set up new slot */
                deltatm = 0L; now = timer(); etime = 0L;
                for (i = 0; i < SEPSLOTS; i++)
                        if ((deltatm = now - sepslot[i].actime) > etime) {
                                etime = deltatm;
```

```
                                sinfo = &sepslot[i];
                        }

                /* initialize slot */

                rd = findrd(rel -> relname,0L,0,rdtab); /* macdep: 0L null ptr */
                sinfo -> rdentries = rdents;
                sinfo -> Zdomlen = 0;
                for (i = 0; i < rdents; i++) {
                        domain = finddom(rdtab[i].domname);
                        if (domain){
                                sinfo -> domtype[i] = domain -> domtype;
                                sinfo -> domlen[i] = domain -> len;
                        }
                        if (rel -> Zmap & (1L << i))
                                sinfo -> Zdomlen += (int)(domain -> len);
                }
        }

                /*slot ready; go to work on comparison */

        r = rel;        s = sep;        t = tup;        /* speed things up */
        /* get length of separator and fix 's' */

        seplen = (int)(*(ushort *)(s)); s += 2; /* skip over short */

        if (! r -> Zmap) {/* no z ordered attributes: seplen % 8 must = 0 */
                if ((seplen % 8) || (seplen < 0)) {
                        sprintf(inbufr,"%d % 8 != 0 in %s (non Z)",seplen,rel->relname);
                        db_err(66,CMPSEPTUP,-1,inbufr);
                        return(FAIL);
                }
                seplen >>= 3; /* convert length to bytes from bits */

                for (i = 0; i < sinfo -> rdentries; i++) {
#ifdef XTRACE
fprintf(stdout,"CMP: attr %d in %s of type %d\n",i,r->relname,sinfo->domtype[i]);
fprintf(stdout,"separ len now %d\n",seplen);fflush(stdout);
#endif
                        switch (sinfo -> domtype[i]) {
                        case DT_CHAR:
                                if (intcmp = (int)(*s - *t))
                                        return((intcmp < 0) ? -2 : intcmp);
                                ++s;
                                ++t;
                                --seplen;
                                break;

                        case DT_STRING:
                                while (*s && seplen) {
                                        if (intcmp = (int)(*s - *t))
                                                return((intcmp < 0) ? -2 : intcmp);
                                        else {
                                                ++s;
                                                ++t;
                                                --seplen;
```

```
                        }
                }
                ++s; ++t;
                if (!seplen)
                        /* --seplen; */
                        return(-2); /* string but no more sep */
                break;

        case DT_LONG:
#ifndef INT16
        case DT_INT:
#endif
                long_coerce.c1 = *s;
                *(&(long_coerce.c2) + 1) = *(s + 1);
                *(&(long_coerce.c3) + 2) = *(s + 2);
                *(&(long_coerce.c4) + 3) = *(s + 3);
                longcmp = long_coerce.lval;
                long_coerce.c1 = *t;
                *(&(long_coerce.c2) + 1) = *(t + 1);
                *(&(long_coerce.c3) + 2) = *(t + 2);
                *(&(long_coerce.c4) + 3) = *(t + 3);
                if (longcmp -= long_coerce.lval)
                        return(longcmp > 0L ? 1 : -2);
                seplen -= sizeof(long);
                break;

        case DT_SHORT:
#ifdef INT16
        case DT_INT:
#endif
                short_coerce.c1 = *s;
                *(&(short_coerce.c2) + 1) = *(s + 1);
                shortcmp = short_coerce.sval;
                short_coerce.c1 = *t;
                *(&(short_coerce.c2) + 1) = *(t + 1);
                if (shortcmp -= short_coerce.sval)
                        return((shortcmp < 0) ? -2 : (int)(shortcmp));
                seplen -= sizeof(short);
                break;

        case DT_FLOAT:
                flt_coerce.c1 = *s;
                *(&(flt_coerce.c2) + 1) = *(s + 1);
                *(&(flt_coerce.c3) + 2) = *(s + 2);
                *(&(flt_coerce.c4) + 3) = *(s + 3);
                floatcmp = flt_coerce.fval;
                flt_coerce.c1 = *t;
                *(&(flt_coerce.c2) + 1) = *(t + 1);
                *(&(flt_coerce.c3) + 2) = *(t + 2);
                *(&(flt_coerce.c4) + 3) = *(t + 3);
                if (floatcmp -= flt_coerce.fval)
                        return(floatcmp > 0.0 ? 1 : -2);
                seplen -= sizeof(float);
                break;
        }
        if (seplen == 0) /* no more separator */
```

```
                        return(0); /* sep = tup within seplen */
                }
                /* separator must have been as long as tuple and was same */
                /* this should be unreachable! */
                db_err(90,CMPSEPTUP,-1,r->relname);
                return(0);

        } /* if not zordered */
        else { /* is some z-ordering */
                if (seplen < 0) {
                        sprintf(inbufr,"%d < 0 in %s (Z)",seplen,rel -> relname);
                        db_err(66,CMPSEPTUP,-1,inbufr);
                        return(FAIL);
                }
                if (!(r -> mode & PZREL)) { /* is all z order */
                        for (i = 0; i < (int)(r -> width), seplen; i++) {
                                zmask = 0x80; /* set bit 7 */
                                for (j = 7; j >= 0; j--) {
                                        if ((*s & zmask)  (*t & zmask))
                                                return((*s & zmask) ? 1 : -2);
                                        seplen--;
                                        zmask >>= 1;
                                }
                                ++s;
                                ++t;
                        }

                        /* if here, sep = tpl within seplen */
                        return(0);

                }
                else { /* partial z ordering */
                        for (i = 0; i < sinfo -> Zdomlen; i++)
                                zmask = 0x80; /* set bit 7 */
                                for (j = 7; j >= 0; j--) {
                                        if ((*s & zmask)  (*t & zmask))
                                                return((*s & zmask) ? 1 : -2);
                                        --seplen;
                                        zmask >>= 1;
                                }
                        ++s;
                        ++t;
                }

                /* if here, continue comparing separator and tuple
                   using repositioned un-zordered attributes */
                for (i = 0; i < rdents; i++) {
                        if (r -> Zmap & (1L << i))
                                continue;
                        switch (sinfo -> domtype[i]) {
                        case DT_CHAR:
                                if (intcmp = (int)(*s - *t))
                                        return((intcmp < 0) ? -2 : intcmp);
                                ++s;
                                ++t;
                                --seplen;
```

```
                        break;

                case DT_STRING:
                        while (*s && seplen) {
                                if (intcmp = (int)(*s - *t))
                                        return((intcmp < 0) ? -2 : intcmp);
                                else {
                                        ++s;
                                        ++t;
                                        --seplen;
                                }
                        }
                        ++s; ++t;
                        if (!seplen)
                                /* --seplen; */
                                return(-2);
                        break;

                case DT_LONG:
#ifndef INT16
                case DT_INT:
#endif
                        long_coerce.c1 = *s;
                        *(&(long_coerce.c2) + 1) = *(s + 1);
                        *(&(long_coerce.c3) + 2) = *(s + 2);
                        *(&(long_coerce.c4) + 3) = *(s + 3);
                        longcmp = long_coerce.lval;
                        long_coerce.c1 = *t;
                        *(&(long_coerce.c2) + 1) = *(t + 1);
                        *(&(long_coerce.c3) + 2) = *(t + 2);
                        *(&(long_coerce.c4) + 3) = *(t + 3);
                        if (longcmp -= long_coerce.lval)
                                return(longcmp > 0L ? 1 : -2);
                        seplen -= sizeof(long);
                        break;

                case DT_SHORT:
#ifdef INT16
                case DT_INT:
#endif
                        short_coerce.c1 = *s;
                        *(&(short_coerce.c2) + 1) = *(s + 1);
                        shortcmp = short_coerce.sval;
                        short_coerce.c1 = *t;
                        *(&(short_coerce.c2) + 1) = *(t + 1);
                        if (shortcmp -= short_coerce.sval)
                                return((shortcmp < 0) ? -2 : (int)(shortcmp));
                        seplen -= sizeof(short);
                        break;

                case DT_FLOAT:
                        flt_coerce.c1 = *s;
                        *(&(flt_coerce.c2) + 1) = *(s + 1);
                        *(&(flt_coerce.c3) + 2) = *(s + 2);
                        *(&(flt_coerce.c4) + 3) = *(s + 3);
                        floatcmp = flt_coerce.fval;
```

```
                                flt_coerce.c1 = *t;
                                *(&(flt_coerce.c2) + 1) = *(t + 1);
                                *(&(flt_coerce.c3) + 2) = *(t + 2);
                                *(&(flt_coerce.c4) + 3) = *(t + 3);
                                if (floatcmp -= flt_coerce.fval)
                                        return(floatcmp > 0.0 ? 1 : -2);
                                seplen -= sizeof(float);
                                break;
                        }
                        if (seplen == 0) /* no more separator */
                                return(0); /* sep = tup within seplen */
                }
                /* separator must have been as long as tuple and was same */
                /* this should be unreachable! */
                db_err(90,CMPSEPTUP,-2,r->relname);
                return(0);
        }
        /* this better be unreachable */

        db_err(65,CMPSEPTUP,-1,r->relname);
        return(FAIL);
}
```

```
#include "mrds.h"

/* TO DO
        +       be sure relation is closed following selection
*/

/*
* int select(rnam,domname,value,cmp,out)
*
*       select from relation named `rptr' tuples with
*       attributes in `domname' domain match `value' according
*       to the type of comparison specified by `cmp' (currently
*       must be equals, greater than or less than in any
*       combination) and add those tuples to a new relation
*       to be named `out'.
*
*       will issue its own error messages if:
*               (1) given a null pointer for rel
*               (2) type of comparison is unknown
*               (3) cannot locate domain `domname' in .dom
*               (4) fails to get rd entries (rname not in rd)
*               (5) domain `domname' not in relation rptr
*               (6) cannot create output relation
*               (7) fails to sort output relation
*
*/

int select(rnam,domname,value,cmp,out)
char  *rnam;            /* name of  input relation to select from */
char domname[];         /* domain of selection                    */
char *value;            /* value to compare against: TYPE UNKNOWN */
short cmp;              /* type of comparison to be done          */
char *out;              /* name of output relation                */

{
        extern int rdents,errno;
        extern char inbufr[];
        extern union u_short short_coerce;
        extern union u_long long_coerce;
        extern union u_float flt_coerce;
        register char *a,*b;
        register int i,j,k;
        int m,n,dompos,closerel(),adtuple();
        float floatcmp;
        long longcmp;
        char *nxtpl,Doms[MAXATTS][MAXDNMLEN],*strcpy(),*getuple();
        DOM *dptr,*finddom();
        REL *rptr,*newrel,*findrel(),*mkrel();
        RD *rdtab[MAXATTS],*findrd();

#ifdef XTRACE
        fprintf(stdout,"--> select(%s,%s,%lX,%d,%s\n",
        rnam,domname,value,cmp,out);
        fflush(stdout);
#endif
```

```
        /* good input relation? */
        if (!rnam) {
                db_err(58,SELECT,-1,"null");
                return(FAIL);
        }

        if (!(rptr = findrel(rnam))) {
                db_err(58,SELECT,-1,rnam);
                return(FAIL);
        }

        /* good compare operator? */
        if ( (cmp & ~CMPVALID) || (!(cmp & CMPVALID)) ) {
                db_err(83,SELECT,-1,"unknown compare");
                return(FAIL); /* or assume a default? */
        }

        /* good domain name? */
        if (!(dptr = finddom(domname))) {
                db_err(79,SELECT,-1,domname);
                return(FAIL);
        }

        /* get domain list for new relation */
        if ((findrd(rptr -> relname,0L,0,rdtab)) == (RD *)(NULL)) {
                db_err(58,SELECT,-1,rptr -> relname);
                return(FAIL);
        }

        for (i = 0; i < rdents; i++)
                strcpy(Doms[i], rdtab[i] -> domname);

        /* isolate domain involved in selection */
        dompos = FAIL;
        for (i = 0; i < rdents; i++)
                if (! strcmp(domname,rdtab[i] -> domname)) {
                        dompos = i;
                        break;
                }
        if (dompos == FAIL) {
                db_err(62,SELECT,-1,domname);
                return(FAIL);
        }

        /* make new relation */
        long_coerce.lval = rptr -> maxsize / (long)(rptr -> width);
        if ((newrel = mkrel(out,rptr->mode,rptr->Zmap,rdents,Doms,long_coerce.lval)) == (REL *)(NULL)) {
                db_err(57,SELECT,-1,out);
                return(FAIL);
        }

        /* begin select */
        i = (1 << dompos);
        if (0) {  /* used to check for Bindx, was changed to Zmap */
                  /* FIX THIS LATER */
        }
```

```
        else { /* have to do it sequentially */
                dompos = rdtab[dompos] -> pos;
#ifdef XTRACE
fprintf(stdout,"+++dompos offset is %d\n",dompos);fflush(stdout);
#endif
                m = 0;

                while (nxtpl = getuple(rptr,0L)) {
                        a = nxtpl + dompos;
                        b = value;
                        switch (dptr -> domtype) {
                        case DT_STRING:
#ifdef XTRACE
fprintf(stdout,"+++ compare |%s| to |%s|\n",a,b);fflush(stdout);
#endif
                                while (*a)
                                        if (i = *a - *b)
                                                break;
                                        else {
                                                ++a;
                                                ++b;
                                        }

                        case DT_CHAR:
                                i = *a - *b;
                                break;

                        case DT_FLOAT:
                                flt_coerce.c1 = *a;
                                *(&(flt_coerce.c2) + 1) = *(a + 1);
                                *(&(flt_coerce.c3) + 2) = *(a + 2);
                                *(&(flt_coerce.c4) + 3) = *(a + 3);
                                floatcmp = flt_coerce.fval;
                                flt_coerce.c1 = *b;
                                *(&(flt_coerce.c2) + 1) = *(b + 1);
                                *(&(flt_coerce.c3) + 2) = *(b + 2);
                                *(&(flt_coerce.c4) + 3) = *(b + 3);
                                floatcmp -= flt_coerce.fval;
                                if (floatcmp == 0.0)
                                        i = 0;
                                else
                                        i = (floatcmp > 0.0 ? 1 : -1);
                                break;

                        case DT_SHORT:
#ifdef INT16
                        case DT_INT:
#endif
                                short_coerce.c1 = *a;
                                *(&(short_coerce.c2) + 1) = *(a + 1);
                                i = (int)(short_coerce.sval);
                                short_coerce.c1 = *b;
                                *(&(short_coerce.c2) + 1) = *(b + 1);
                                i -= (int)(short_coerce.sval);
                                break;
```

```
                        case DT_LONG:
#ifndef INT16
                        case DT_INT:
#endif
                                long_coerce.c1 = *a;
                                *(&(long_coerce.c2) + 1) = *(a + 1);
                                *(&(long_coerce.c3) + 2) = *(a + 2);
                                *(&(long_coerce.c4) + 3) = *(a + 3);
                                longcmp = long_coerce.lval;
                                long_coerce.c1 = *b;
                                *(&(long_coerce.c2) + 1) = *(b + 1);
                                *(&(long_coerce.c3) + 2) = *(b + 2);
                                *(&(long_coerce.c4) + 3) = *(b + 3);
                                longcmp -= long_coerce.lval;
                                if (! longcmp)
                                        i = 0;
                                else
                                        i = (longcmp > 0L ? 1 : -1);
                                break;
                        } /* end switch */

                        /* take action depending on `cmp' and `i' */
                        k = 0;
                        for (j = 0; j < CMP_OPS; j++) {
                                switch (cmp & CMPATT[j]) {
                                case CMPATT_EQ:
                                        if (!i) {
                                                ++k;
                                                if (adtuple(newrel,nxtpl) != rptr -> width) {
                                                        ++m;
                                                        n = errno;
                                                }
                                        }
                                        break;

                                case CMPATT_GT:
                                        if (i > 0) {
                                                ++k;
                                                if (adtuple(newrel,nxtpl) != rptr -> width) {
                                                        ++m;
                                                        n = errno;
                                                }
                                        }
                                        break;

                                case CMPATT_LT:
                                        if (i < 0) {
                                                ++k;
                                                if (adtuple(newrel,nxtpl) != rptr -> width) {
                                                        ++m;
                                                        n = errno;
                                                }
                                        }
                                        break;

                                } /* end switch */
```

```
                                if (k)
                                        break;
                        }
                } /* end while getuple */
        } /* end else do it sequentially */

        /* did all tuples get successfully added? */
        if (m) {
                sprintf(inbufr,"%d",m);
                db_err(81,33,n,inbufr);
        }

        /* close relations */
        if (rptr -> fd != FAIL)
                closerel(rptr);
        if (newrel -> fd != FAIL)
                closerel(newrel);

        if ((j = sortrel(newrel)) < SUCCESS)
                db_err(82,SELECT,-1,newrel -> relname);

#ifdef XTRACE
        fprintf(stdout,"<-- select(1)\n");
        fflush(stdout);
#endif

        if (m == 0)
                if (j == SUCCESS) return(SUCCESS);
                else return (-3);
        else return(-2);
}
```

```
#include "mrds.h"

int setup(dbname,db,verbose)
char  *dbname;          /* name of database */
DBSTATUS *db;
int verbose;

{
        extern int errno;
        register int result=TRUE;
        int uid= -1,getuid(),geteuid(),fclose();
        int yesno(),abend(),db_err(),logent(),gooduser(),find_db();
        FILE *fopen();

#ifdef XTRACE
        fprintf(stdout,"--> setup(%s)\n",dbname);
        fflush(stdout);
#endif

        if ((dbname == (char *)(0)) || (*dbname == (char *)(0)))
                return(FAIL);

        /* start session logging */

        if ((fplog = fopen(LOGFILE,APMODE)) == (FILE *)(NULL) )
                db_err(95,SETUP,errno,"log file");

        if ((uid = gooduser()) == FAIL) {
                sprintf(inbufr,"%d %d \"%s\"",getuid(),geteuid(),dbname);
                if (fplog) {
                        logent("ACFAIL",inbufr);
                        fclose(fplog);
                }
                db_err(0,SETUP,-1,inbufr);
                return(FAIL);
        }

        /* used to put command line arguments into log entry when this was */
        /* done in "main"; should find some way to keep that feature */
/*
        sprintf(LOGBUF,"user %d ",uid);
        for (i = 0; i < argc; i++) {
                strcat(LOGBUF,argv[i]);
                strcat(LOGBUF,BLANK);
        }
*/
        logent("INVOKED",LOGBUF);

        if ((find_db(dbname,uid,db)) == FAIL)
                return(DB_NOFND);

        /* db found: do consistency check */

        if (dbck(db,verbose) != 0) { /* oh oh */
                sprintf(LOGBUF,"%s (owner %d)",db -> dbs_name,db -> dbs_owner);
                logent("CONFUSED",LOGBUF);
```

```
                return(DB_FUBAR);
        }

        sprintf(LOGBUF,"%s (owner %d)",db -> dbs_name,db -> dbs_owner);
        logent("ACTIVE",LOGBUF);

#ifdef XTRACE
        fprintf(stdout,"<-- setup(X%x)\n",db -> dbs_dfltmode);
        fflush(stdout);
#endif

        return(SUCCESS);
}
```

```
#include "mrds.h"

short domtype[MAXATTS];
char newmer[FILESTRING + MAXNAMLEN];

int sortrel(rel)
REL *rel;

{
        int i,j,npass,tpb,merfd,state,bufsize;
        int qsort(),creat(),open(),read(),write(),replace();
        int strlen(),tplcmp(),openrel(),closerel();
        char *strcpy(),*mktemp(),*sortbuf,*oldmer,*malloc();
        DOM *domain,*finddom();
        RD *rds[MAXATTS];
        extern short domtype[];
        extern int rdents,errno;
        extern REL *reltosort;
        extern char inbufr[];

#ifdef XTRACE
        fprintf(stdout,"--> sortrel(%s)\n",rel->relname);
        fflush(stdout);
#endif
        /* verify rel exists */
        if (! rel) {
                db_err(58,SORTREL,-1,"null");
                return(FAIL);
        }
        reltosort = rel;

        /* don't sort a B* tree */
        if (rel -> mode & BTREE)
                return(-2);

        /*determine number of iterations & buffersize */
        /* WARNING! IF SIZEOF(INT) == 2, SORTLIM MUST BE <= 64 */
        bufsize = (int)(SORTLIM * 1024L);
        if (rel -> windx <= bufsize) {
                npass = 1;
                bufsize = (int)(rel -> windx);
        }
        else { /* need multiple sort/merge passes */
                i = bufsize % (int)(rel -> width);
                bufsize -= i; /* align to nearest tuple */
                npass = (int)(rel -> cursize / (long)(bufsize));
                if (rel -> cursize % (long)(bufsize))
                        ++npass;
        }
        tpb = bufsize / (int)(rel -> width); /* tuples per buffer */
#ifdef XTRACE
fprintf(stdout,"@@@ tpb now %d in %d passes\n",tpb,npass);fflush(stdout);
#endif

        /* get buffer space */
        if ((sortbuf = malloc(bufsize)) == NULL) { /* not enough mem */
```

```
                db_err(52,SORTREL,errno,"sortbuf");
                /* should have clever retry with smaller buffer */
                return(FAIL);
        }

        /* get particulars on rel for tuplcmp()     */
        /* determine number of attributes per tuple */
        if (! findrd(rel -> relname, 0L, 0, rds)) {
                db_err(62,SORTREL,-1,rel -> relname);
                state = FAIL;
                goto xsort;
        }

        /* get domain datatypes */
        for (i = 0; i < rdents; i++) {
                domain = finddom(rds[i] -> domname);
                if (domain)
                        domtype[i] = domain -> domtype;
                else {
                        db_err(79,SORTREL,-1,rds[i] -> domname);
                        domtype[i] = -1;
                }
        }

        i = strlen(MERTEMP);
        if ((oldmer = malloc(i)) == NULL) {
                db_err(52,SORTREL,-1,"merfnms");
                state = FAIL;
                goto xsort;
        }
        *oldmer = NULLC;

        openrel(rel,READMODE);

        /* off to the races */
        for (; npass > 0; npass--) {
                /*get next chunk to sort */
                if ((i = read((int)(rel -> fd),sortbuf,bufsize)) <= 0) {
                        sprintf(inbufr,"(%d)",i);
                        db_err(60,SORTREL,errno,inbufr);
                        state = FAIL;
                        goto xsort;
                }

                /* sort new chunk */
                if (i < bufsize) /* buf not full -- don't sort all */
                        tpb = i / (int)(rel -> width);
#ifdef XTRACE
fprintf(stdout,"@@@ tpb at qsort = %d\n",tpb);fflush(stdout);
#endif
                qsort(sortbuf,tpb,(int)(rel -> width),tplcmp);

                /* merge */
                if ((strlen(oldmer) != 0) &&(merfd = open(oldmer,RDMODE)) > 0) {
                        if (merge(merfd,sortbuf,newmer,rel,tpb) == FAIL) {
                                db_err(55,SORTREL,-1,"");
```

```
                                state = -1;
                                goto xsort;
                        }
                        unlink(oldmer);
                        strcpy(oldmer,newmer);
                }
                else { /* couldn't open a former merge file */
                        if (errno = ENOENT) {
                                strcpy(oldmer,MERTEMP);
                                mktemp(oldmer);
                                if ((merfd = creat(oldmer,CRMASK)) == FAIL) {
                                        db_err(51,SORTREL,errno,"mertemp");
                                        state = -1;
                                        goto xsort;
                                }
                                if ((i = write(merfd,sortbuf,rel -> width)) != rel -> width) {
                                        sprintf(inbufr,"%d:%d",rel -> width,i);
                                        db_err(54,SORTREL,errno,inbufr);
                                        state = -1;
                                        goto xsort;
                                }
                                for (i = 1; i < tpb; i++) {
                                        j = i * rel -> width;
                                        if (tplcmp(&sortbuf[j - rel -> width],
                                        &sortbuf[j]))
                                                write(merfd,&sortbuf[j],rel -> width);
                                }
                                close(merfd);
                        }
                        else { /* something is wrong */
                                db_err(51,SORTREL,errno,"mertmp");
                                state = -1;
                                goto xsort;
                        }
                }
        }

        closerel(rel);

        /* replace unsorted relation with new relation */
        if (replace(rel,oldmer) < SUCCESS) {
                db_err(59,SORTREL,-1,rel -> relname);
                state = -1;
                goto xsort;
        }

        state = 0;
xsort:
#ifdef XTRACE
        fprintf(stdout,"<-- sortrel(%d)\n",state);
        fflush(stdout);
#endif
        free(sortbuf);
        free(newmer);
        free(oldmer);
        return(state);
```

```
]
```

```
int tplcmp(a,b)  /* kludgey but quick */ /* DOESN'T GROK Z-ORDER (YET) */
char *a,*b;

{
        extern REL *reltosort;
        extern int rdents;
        extern short domtype[];
        extern union u_short short_coerce;
        extern union u_int int_coerce;
        extern union u_long long_coerce;
        extern union u_float flt_coerce;
        register int i;
        register char *aa,*bb;
        int skipped = 0;

        /* add here for more datatypes */
        int intcmp = 0;
        long longcmp = 0;
        float floatcmp = 0.0;

#ifdef XTRACE
        fprintf(stdout,"$$$ tuplcmp(X%lx,X%lx)\n",a,b);
        fflush(stdout);
#endif

        aa = a;
        bb = b;
        for (i = 0; i < rdents; i++) {
                switch (domtype[i]) {
                case DT_CHAR:
                        if (intcmp = (int)(*aa - *bb))
                                return(intcmp);
                        ++aa;
                        ++bb;
                        break;

                case DT_FLOAT:
                        flt_coerce.c1 = *aa;
                        *(&(flt_coerce.c2) + 1) = *(aa + 1);
                        *(&(flt_coerce.c3) + 2) = *(aa + 2);
                        *(&(flt_coerce.c4) + 3) = *(aa + 3);
                        floatcmp = flt_coerce.fval;
                        flt_coerce.c1 = *bb;
                        *(&(flt_coerce.c2) + 1) = *(bb + 1);
                        *(&(flt_coerce.c3) + 2) = *(bb + 2);
                        *(&(flt_coerce.c4) + 3) = *(bb + 3);
                        if (floatcmp -= flt_coerce.fval)
                                return (floatcmp > 0.0 ? 1 : -1);
                        aa += sizeof(float);
                        bb += sizeof(float);
                        break;

                case DT_SHORT:
#ifdef INT16
                case DT_INT:
```

```
#endif
                        short_coerce.c1 = *aa;
                        *(&(short_coerce.c2) + 1) = *(aa + 1);
                        intcmp = (int)(short_coerce.sval);
                        short_coerce.c1 = *bb;
                        *(&(short_coerce.c2) + 1) = *(bb + 1);
                        if (intcmp -= (int)(short_coerce.sval))
                                return(intcmp);
                        aa += sizeof(short);
                        bb += sizeof(short);
                        break;

                case DT_LONG:
#ifndef INT16
                case DT_INT:
#endif
                        long_coerce.c1 = *aa;
                        *(&(long_coerce.c2) + 1) = *(aa + 1);
                        *(&(long_coerce.c3) + 2) = *(aa + 2);
                        *(&(long_coerce.c4) + 3) = *(aa + 3);
                        longcmp = long_coerce.lval;
                        long_coerce.c1 = *bb;
                        *(&(long_coerce.c2) + 1) = *(bb + 1);
                        *(&(long_coerce.c3) + 2) = *(bb + 2);
                        *(&(long_coerce.c4) + 3) = *(bb + 3);
                        if (longcmp -= long_coerce.lval)
                                return(longcmp > 0L ? 1 : -1);
                        aa += sizeof(long);
                        bb += sizeof(long);
                        break;

                case DT_STRING:
                        while(*aa)
                                if (intcmp = (int)(*aa - *bb))
                                        return(intcmp);
                                else {
                                        ++aa;
                                        ++bb;
                                }
                                ++aa; ++bb;
                                break;

                default: /* unknown domtype */
                        ++skipped;
                }
        }

        return(NULL); /* tuples are equal */
}
```

```
int merge(file,core,out,rel,num)
int file;   /* fd of file containing previous merged output */
char *core; /* ptr to start of memory area */
char *out;  /* name of new output file */
REL *rel;   /* ptr to system rel entry for rel being merged */
int num;    /* number of tuples in mem to be merged */

{
        extern int errno;
        extern char inbufr[];
        char *inbuf,*outbuf,*out,*malloc(),*strcpy(),*mktemp();
        register int i,j,k;
        register char *lastpl,*in,*mem;
        int state,newfd,open(),close(),creat(),read(),write(),tplcmp();
        unsigned int bufsize;


#ifdef XTRACE
        fprintf(stdout,"--> merge(%d,X%lx,%s,%s,%d)\n",
        file,core,out,rel,num);
        fflush(stdout);
#endif
        /* set up block buffers for merge */
        if ((i = (num * rel -> width)) > 1024)
                if (rel -> width < 1024)
                        i = 1024 - (1024 % rel -> width);
                else
                        i = rel -> width;

#ifdef XTRACE
        fprintf(stdout,"@@@ bufsize becomes %d\n",i);
        fflush(stdout);
#endif

        if ((inbuf = malloc(i)) == NULL) {
                db_err(52,MERGE,errno,"merbufi");
                return(FAIL);
        }

        if ((outbuf = malloc(i)) == NULL) {
                free(inbuf);
                db_err(52,MERGE,errno,"merbufo");
                return(FAIL);
        }
        bufsize = i;

        if ((lastpl = malloc(rel -> width)) == NULL) {
                free(inbuf);
                free(outbuf);
                db_err(52,MERGE,errno,"merbufl");
                return(FAIL);
        }

        /* initialize */
        in = inbuf;
```

```
        out = outbuf;
        mem = core;
        *lastpl = NULLC;

        /* make new file */
        strcpy(out,MERTEMP);
        mktemp(out);
        if ((newfd = creat(out,CRMASK)) == FAIL) {
                db_err(55,MERGE,errno,out);
                state = FAIL;
                goto emerge;
        }

        /* merge */
        while ((i = read(file,inbuf,bufsize)) > 0) {
                if (i < bufsize)
                        bufsize = i;

                while (in < (inbuf + bufsize)) {
                        if (!(j = tplcmp(in,mem)))  /* tuples are equal */
                                if ((k = tplcmp(in,lastpl)))  /* but not same as previous */
                                        for (j = 0; j < rel -> width; j++) {
                                                *out++ = *in++;
                                                *(lastpl + j) = *mem++;
                                        }
                                else { /* same; discard from both */
                                        in += rel -> width;
                                        mem += rel -> width;
                                }
                                else /* tuples not equal, write 'smaller' */
                                        if (j < 0)
                                                for (j = 0; j < rel -> width; j++) {
                                                        *out++ = *in;
                                                        *(lastpl + j) = *in++;
                                                }
                                        else
                                                for (j = 0; j < rel -> width; j++) {
                                                        *out++ = *mem;
                                                        *(lastpl + j) = *mem++;
                                                }

                                /* time to empty output buffer? */
                                if ((outbuf - out) >= bufsize) { /* flush it */
                                        if ((j = write(newfd,outbuf,bufsize)) != bufsize) {
                                                sprintf(inbufr,"%d:%d (%s)",bufsize,j,out);
                                                db_err(54,MERGE,errno,inbufr);
                                                state = FAIL;
                                                goto emerge;
                                        }
                                        out = outbuf;
                                }
                        }
                        in = inbuf;
                }
                state = SUCCESS;
```

```
emerge:
                free(inbuf);
                free(outbuf);
                free(lastpl);
                close(newfd);
#ifdef XTRACE
        fprintf(stdout,"<-- merge(%d)\n",state);
        fflush(stdout);
#endif
                return(state);
}
```

```
/* Early version of "split" written so that MRDSc can get done
   on time: ALWAYS does a split, even if a redistribution of entries
   would have been sufficient. Later versions can correct this shortcoming
*/

/*=====================================================================
PROBLEMS:
FIXED   size of padding alignment checks done wrong: value to & with 0x3
        is start addrs of separator PLUS LENGTH OF SEP

FIXED   all child nodes whose P are moved to a new brother page at parent
        level must be "notified" that their PRED value needs changing

        when copying entries off 'old' node to its new image copy, should
FIXED   not be copying offsets from 'old', but rather installing new
        offsets, thereby straightening list into which insertions may
        have been made.

FIXED   in splitting branch pages, when initializing the status and
        space parms in the hdr, forgot to set the PRED of the new
        branch page. NOTE that this pred may be affected by splitting
        the grandparent page!

        ?write nulls throughout newly allocated memory image of pages?

FIXED   make sure root branch node can be correctly split

        change new page flushing so that if a recursive call to split
        fails, the new versions of the originally split pages will not
        have been written out...then at least we can recover from the
        recursive split failure (eg. as when size > maxsize)
=====================================================================*/
#include "mrds.h"

#define SPLBOMB free(dpg);free(epg);if(Epg)free(Epg);

Bentry newent,upward;

int split(rptr,tpl,pos,x,mode,sptr)
char *tpl;      /* ptr to entry causing split: may be tpl or (s,p) */
REL *rptr;      /* ptr to rel whose page is being split */
long pos;       /* position at which to insert new entry */
Xfb *x;         /* ptr to Xfb for this relation */
char mode;      /* flag: split branch or leaf page */
long sptr;      /* 'p' in (s,p) pair */

/* page being split ***MUST*** currently be in xbufat or lbufat for */
/* branch or leaf page resp. BEFORE calling split */

{
        extern int errno;
        register int i,j;
        register char *a,*b,*c;
        char *dpg,*epg,*rpg,*npg,*last,*first,*Epg,*b4last,*sepat;
        Xpghdr pg;
        Xlfhdr lf;
```

```
        int entsize,used,pgsize,oldpad,k;
        long pred,posn,lbro,hbro,*left,*right,thispg;
        ushort *oldspace,*uslast,*usnext,dirmode;
        char isnewent='\0',onleft='\0',b4fle='\0',*malloc();

        if (pos > 0) {
                db_err(91,SPLIT,-1,"already present");
                return(SUCCESS);
        }

        if (mode)
                entsize = (int)(rptr -> width) + sizeof(long);
        else
                /* sizeof complete entry with worst case padding */
                entsize = *(ushort *)(tpl) >> 3 + sizeof(long) + sizeof(ushort) + 3;

        /* allocate space for split work */

        pgsize = (mode ? x -> lpgsize : x -> xpgsize);

        /* should check here to see if by adding "pgsize" to the */
        /* relation it will exceed "maxsize"; can't do this now  */
        /* see bottom note in PROBLEMS above for reason why */

        if ((dpg = malloc(pgsize)) == (char *)(NULL)) { /* if it doesn't want to play */
                db_err(52,SPLIT,errno,"dpg");
                return(FAIL);
        }

        if ((epg = malloc(pgsize)) == (char *)(NULL)) {
                free(dpg);
                db_err(52,SPLIT,errno,"epg");
                return(FAIL);
        }

        if ((Epg = malloc(pgsize)) == (char *)(NULL)) {
                db_err(93,SPLIT,errno,"Epg");
        }

        /* set up values in space just allocated */

        if (mode == XLEAF) {
                lf.status = dpg;
                lf.space = (ushort *)(dpg + 2);
                lf.pred = (long *)(dpg + 4);
                lf.offset = (ushort *)(dpg + 8);
        }
        else if (mode == XBRANCH) {
                pg.status = dpg;
                pg.space = (ushort *)(dpg + 2);
                pg.pred = (long *)(dpg + 4);
        }

        if (a = Epg) { /* make backup copy */
                b = (mode ? x -> lbufat : x -> xbufat);
                for (i = 0; i < pgsize; i++)
```

```
                *a++ = *b++;
        }

        /* install overflow entry and 'link' it into list */

        if (mode == XLEAF) { /* on a leaf */
                if ((-pos + 2) != x -> lpgat) { /* in midst of list */
                        left = (long *)(-pos + 2 + (long)(rptr -> width));
                        newent.te_ptr = *left;
                        *left = -1L; /* magic ptr */
                }
                else { /* before first tuple */
                        oldspace = (ushort *)(x -> lbufat + L_FLEOFF);
                        newent.te_ptr = x -> lpgat + (long)(*oldspace);
                        *oldspace = MAGICLINK; /* magic FLE value */
                }

                newent.te_item = tpl;
        } /* end if was a leaf */
        else {  /* is on a branch */
                /* only special case here is if (-pos + 2) */
                /* points at P0 */
                if (-pos + 2 == x -> xpgat + (long)(XHDRLEN)) {
                        oldspace = (ushort *)(x -> xbufat + XHDRLEN + sizeof(long));
                        newent.te_bnext = *oldspace;
                }
                else { /* pos pts at ordinary (s,p) entry */
                        c = -pos + 2 - x -> xpgat + x -> xbufat;
                        oldspace = (ushort *)(sepentlen(c) + sizeof(long));
                        newent.te_bnext = *oldspace;
                }
                *oldspace = MAGICLINK; /* magic offset */
                newent.te_len = *(ushort *)(tpl);
                newent.te_ptr = sptr;
                newent.te_item = tpl;

        } /* end if branch on new item insertion */

        if (mode == XBRANCH) { /* split branch, may even be root */
                a = x -> xbufat + XHDRLEN;
                b = dpg + XHDRLEN;
                npg = dpg;

                /* recycle some variables...*/

                lf.status = epg;
                lf.space = epg + X_SPCOFF;
                lf.pred = epg + X_PREDOFF;

                *(pg.status) = XBRANCH;
                *(lf.status) = XBRANCH;
                *(pg.space) = x -> xpgsize - XHDRLEN;
                *(lf.space) = *(pg.space);
                *(pg.pred) = *(long *)(dpg + X_PREDOFF);
                *(lf.pred) = *(pg.pred);
```

```
        k = x -> xpgfill / 100.0 * x -> xpgsize;

        /* copy P0 ptr into new copy of 'left' page */

        *(long *)(b) = *(long *)(a);
        a += sizeof(long);
        b += sizeof(long);
        *(pg.space) -= sizeof(long);
        a = (char *)(*(ushort *)(a));
        *(ushort *)(b) = XHDRLEN + sizeof(long) + sizeof(ushort);
        b += sizeof(ushort);
        *(pg.space) -= sizeof(ushort);
        while (a != (char *)(0)) {
                /* is it time to switch frames? */
                j = sizeof(long) + sizeof(ushort);
                if (a == (char *)(MAGICLINK))
                        /* worst case approx */
                        j += (newent.te_len >> 3) + 3 + sizeof(ushort);
                else
                        j += sepentlen(a);

                if ((x -> xpgsize - *(pg.space)) > k) {

                        /* time to switch */

                        npg = epg;
                        b = epg + XHDRLEN;
                        pg.status = lf.status;
                        pg.space = lf.space;

                        /* install new P0, hold S for parent */

                        if (a == (char *)(MAGICLINK)) {
                                i = newent.te_len;
                                c = newent.te_item;
                        }
                        else {
                                i = (int)(*(ushort *)(a));
                                c = a + sizeof(ushort);
                        }
                        upward.te_len = i;
                        for (j = 0; j < i; j++)
                                *(upward.te_item + i) = *c++;

                        if (a == (char *)(MAGICLINK))
                                *(long *)(b) = newent.te_ptr;
                        else {
                                a += sepentlen(a);
                                *(long *)(b) = *(long *)(a);
                                a += sizeof(long);
                        }
                        b += sizeof(long);
                        *(pg.space) -= sizeof(long);
                        k = MAXINT;
                        i = (int)(b + sizeof(ushort)) - (int)(npg);
                        *(ushort *)(b) = (ushort)(i);
```

```
                b += sizeof(ushort);
                *(pg.space) -= sizeof(ushort);

                /* P0 now established on new brother */

                if (a == (char *)(MAGICLINK))
                        a = (char *)(newent.te_bnext);
                else
                        a = (char *)(*(ushort *)(a));
                continue;
        }


        if (a == (char *)(MAGICLINK)) {
                /* copy in new item */
                *(ushort *)(b) = newent.te_len;
                b += sizeof(ushort);
                *(pg.space) -= sizeof(ushort);
                c = b;
                for (i = 0; i < newent.te_len >> 3; i++)
                        *b++ = *(newent.te_item + i);
                j = ((j = (long)(c + i) & 0x3) ? (4 - j) : 0);
                for (i = 0; i < j; i++)
                        *b++ = 0xff;
                *(pg.space) -= newent.te_len >> 3 + j;
                *(long *)(b) = sptr;
                *(pg.space) -= sizeof(long);
                b += sizeof(long);
                i = (int)(b + sizeof(ushort)) - (int)(npg);
                *(ushort *)(b) = (ushort)(i);
                *(pg.space) -= sizeof(ushort);
                b += sizeof(ushort);
        }
        else {
                a = (char *)((unsigned long)(a) + (unsigned long)(x -> xbufat));
                /* !!! don't copy offset pointers!!! */
                /* rather than correct "pred" ptrs in child nodes
                   during this pass, which would require this
                   now incomplete page to have been written out
                   to disk (to claim its space), finish the work
                   in memory, then write out the complete page
                   then do a second pass to update child pred pointers.
                */
                k = *(ushort *)(a);
                last = a;
                *(ushort *)(b) = (ushort)(k);
                a += sizeof(ushort);
                b += sizeof(ushort);
                *pg.space -= sizeof(ushort);
                sepat = b;
                for (i = 0; i < (k >> 3); i++)
                        *b++ = *a++;
                j = ((j = (long)(sepat + k) & 0x3) ? (4 - j) : 0);
                for (i = 0; i < j; i++)
                        *b++ = 0xff;
                b += k + j;
```

```
                                *pg.space -= k + j;
                                a += sepentlen(last);
                                *(long *)(b) = *(long *)(a);
                                a += sizeof(long);
                                b += sizeof(long);
                                *pg.space -= sizeof(long);
                                i = (int)(b + sizeof(ushort)) - (int)(npg);
                                *(ushort *)(b) = (ushort)(i);
                                b += sizeof(ushort);
                                *pg.space -= sizeof(ushort);

                                a = (char *)(*(ushort *)(a));
                        } /* end else copy from *a */
                } /* end while not done */


/* After finished with new brother page, write it out to new address */
/* in file, then go back to its beginning and load each referenced */
/* child page to update the PRED ptr. Can cheat and try just writing */
/* the new long value at know offset into file, but this will read */
/* one disk block anyway */

                if (flushpage(dpg,rptr -> fd,x -> xpgat,x -> xpgsize) == FAIL) {
                        db_err(59,SPLIT,errno,"rewr dpg");
                        SPLBOMB
                }

                if ((posn = lseek(rptr -> fd,0L,FROMEND)) == FAIL){
                        db_err(56,SPLIT,errno,"add epg");
                        SPLBOMB
                }

                if (flushpage(epg,rptr -> fd,posn,x -> xpgsize) == FAIL){
                        db_err(59,SPLIT,errno,"wr epg");
                        SPLBOMB
                }
                /* put "upward" S into parent unless this was the root */

                if (*(x -> xbufat) == XROOT) {
                        if ((rpg = malloc(x -> xpgsize)) == (char *)(NULL)) {
                                db_err(52,SPLIT,errno,"rpg");
                                SPLBOMB
                                return(FAIL);
                        }

                        pg.status = rpg;
                        pg.space = (ushort *)(rpg + X_SPCOFF);
                        pg.pred = (long *)(rpg + X_PREDOFF);
                        *(pg.status) = XROOT;
                        *(pg.space) = x -> xpgsize - XHDRLEN;
                        *(pg.pred) = 0L;

                        a = rpg + XHDRLEN;
                        *(long *)(a) = x -> xpgat; /* P0 */
                        a += sizeof(long);
                        *(ushort *)(a) = a + sizeof(ushort);
```

```
                a += sizeof(ushort);
                *(pg.space) -= sizeof(ushort);
                *(ushort *)(a) = upward.te_len;
                a += sizeof(ushort);
                *(pg.space) -= sizeof(ushort);
                b = a;
                j = upward.te_len >> 3;
                for (i = 0; i < j; i++)
                        *a++ = *(upward.te_item + i);
                i =((i = (long)(b + j) & 0x3) ? (4 - i) : 0);
                for (k = 0; k < i; k++)
                        *a++ = 0xff;
                *(pg.space) -= j + i;
                *(long *)(a) = posn;
                *(pg.space) -= sizeof(long);

                /* new root generated; find place on disk */

                if ((posn = lseek(rptr -> fd,0L,FROMEND)) == FAIL){
                        db_err(56,SPLIT,errno,"new root");
                        free(rpg);
                        SPLBOMB
                        return(FAIL);
                }

                if (flushpage(rpg,rptr -> fd,posn,x -> xpgsize) == FAIL) {
                        db_err(59,SPLIT,errno,"new root");
                        free(rpg);
                        SPLBOMB
                        return(FAIL);
                }

                /* now patch ptr to root in Xfb */

                x -> rootpos = posn;
                free(rpg);
        }
        else { /* regular branch update */
                upward.te_ptr = posn;
                pred = *(long *)(x -> xbufat + X_PREDOFF);
                if (upd_parent(upward,pred,x) == FAIL) {
                        db_err(61,SPLIT,errno,"upd parent");
                        SPLBOMB
                        return(FAIL);
                }
        }

        /* now update the "pred" ptrs in children whose predecessor */
        /* has now moved to brother node */

        b = epg + XHDRLEN;
        if (lseek(rptr -> fd,*(long *)(b),FROMTOP) == FAIL){
                db_err(56,SPLIT,errno,"lost child");
        }
        else if(write(rptr -> fd,&posn,sizeof(long)) != sizeof(long)){
                db_err(59,SPLIT,errno,"child pred upd");
```

```
                }
                b += sizeof(long);
                b = (char *)(*(ushort *)(b));
                while (b != (char *)(0)) {
                        if (lseek(rptr -> fd,*(long *)(b),FROMTOP) == FAIL){
                                db_err(56,SPLIT,errno,"lost child");
                        }
                        else if(write(rptr -> fd,&posn,sizeof(long)) != sizeof(long)){
                                db_err(59,SPLIT,errno,"child pred upd");
                        }
                        b = (char *)(*(ushort *)(b));
                }
        } /* end if splitting branch page */
        else { /* splitting leaf page */
                pred = 0L;
                npg = dpg;
                thispg = x -> lpgat;
                k = x -> lpgfill / 100.0 * x -> lpgsize;
                *(lf.status) = XLEAF;
                *(lf.space) = x -> lpgsize - XLFHDRLEN;
                a = x -> lbufat + *(ushort *)(x -> lbufat + L_FLEOFF);
                c = dpg + L_FLEOFF;
                *(ushort *)(c) = XLFHDRLEN;
                j = rptr -> width;
                i = ((i = (long)(dpg + XLFHDRLEN + j) & 0x3) ? (4 - i) : 0);
                b = dpg + XLFHDRLEN + i;
                *(ushort *)(c) += i;
                *(lf.space) -= i;
                do {
                        /* time to switch frames? */

                        if ((pgsize - *lf.space) > k) { /* sw */
                                if ((pred = lseek(rptr -> fd,0L,FROMEND)) == FAIL){
                                        db_err(56,SPLIT,errno,"new lf");
                                        SPLBOMB;
                                        return(FAIL);
                                }
                                /* "pred" is disk addrs of new leaf page */
                                /* first, fix last ptr already copied to point at */
                                /* this new page, then copy the rest */

                                c = b - sizeof(long);
                                last = b - sizeof(long) - j; /* addrs of last tpl on old pg */
                                lf.status = epg;
                                lf.space = (ushort *)(epg + L_SPCOFF);
                                lf.pred = (long *)(epg + L_PREDOFF);
                                *(lf.status) = XLEAF;
                                *(lf.space) = x -> lpgsize - XLFHDRLEN;
                                *(lf.pred) = *(long *)(x -> lbufat + L_PREDOFF);
                                b = epg + L_FLEOFF;
                                *(ushort *)(b) = *(ushort *)(dpg + L_SPCOFF);
                                /* this [↑]ONLY works with fixed width tuples */
                                b = epg + *(ushort *)(epg + L_FLEOFF);
                                *(long *)(c) = (long)(b - epg) + pred;
                                k = MAXINT;
                                thispg = pred;
```

```
                        npg = epg;
                        first = b; /* addrs of first tpl on new page */
                }

                for (i = 0; i < j; i++)
                        *b++ = *a++;
                *(lf.space) -= j;
                posn = *(long *)(a);
                i = ((i = (long)(b + sizeof(long) + j) & 0x3) ? (4 - i) : 0);
                *(long *)(b) = (long)(b + sizeof(long) + i -npg) + thispg;
                a += sizeof(long);
                b += sizeof(long);
                *(lf.space) -= sizeof(long);
        } while ((posn < x -> lpgat + x -> lpgsize) && (posn > x -> lpgat));
        /* write out the new leaves */

        if (flushpage(dpg,rptr -> fd,x -> lpgat,x -> lpgsize) == FAIL) {
                db_err(59,SPLIT,errno,"upd old");
                SPLBOMB
                return(FAIL);
        }

        if (flushpage(epg,rptr -> fd,pred,x -> lpgsize) == FAIL) {
                db_err(59,SPLIT,errno,"upd new");
                SPLBOMB
                return(FAIL);
        }

        /* now generate (s,p) entry to put into parent page */
        /* "pred" is disk address of the new entry; use mksep() */
        /* to determine new separator */

        upward.te_ptr = pred; /* disk ptr to this new page */

        /* need space for separator: scribble on dpg */


        if ((i = mksep(rptr,last,first,dpg)) == FAIL) {
                db_err(68,SPLIT,-1,"mksep fail");
                SPLBOMB
                /* at this point, some recovery action should be */
                /* undertaken, eg. to restore the now split pg */
                /* to its original form, and report that the split */
                /* failed, signalling to insert that the requested */
                /* insertion could not be done, but the integrity */
                /* of the relation is not lost */

                return(FAIL);
        }

        /* install (s,p) into parent */

        if (upd_parent(rptr,upward,*(lf.pred),x) == FAIL) {
                db_err(68,SPLIT,-1,"upd parent");
                SPLBOMB
                /* same remarks as above re. recovery*/
```

```
                        return(FAIL);
                }
        }
        SPLBOMB
        return(SUCCESS);
}
/* ================= GOT TO HERE: NEW VERSION ======================== */
```

```
int upd_parent(rptr,sp,parent,x)
REL     *rptr;
Bentry  sp;
long    parent;
Xfb     *x;
{
        extern int errno;
        register int i=0;
        register char *a,*b;
        register long son;
        register ushort *space;
        register int j;
        char *ppg,*xpg,*sepat,*malloc(),*goesafter,nodemode=XBRANCH;
        int sepentlen(),didsplit=0;
        long posn,oldxpgat;

        if (parent == 0L)
                return(FAIL);

        /* get workspace for parent page */

        if ((ppg = malloc(x -> xpgsize)) == (char *)(NULL)) {
                db_err(52,UPD_PAREN,errno,"ppg");
                return(FAIL);
        }

getparent:
        if (loadpage(rptr -> fd,x,ppg,parent) == FAIL) {
                db_err(60,UPD_PAREN,errno,"get parent");
                free(ppg);
                return(FAIL);
        }

        /* find entry in parent for curpg */

        a = ppg + XHDRLEN;
        son = x -> xpgat;

        /* treat P0 as a special case */

        if (*(long *)(a) == son) {
                ++i;
                goesafter = a;
        }
        else {
                a +=sizeof(long);
                a = (char *)(*(ushort *)(a));
                if (a == (char *)(0))
                        --i;
                else {
                        /* construct below required because the compiler */
                        /* is too FB to add two IDENTICALLY typed items */
                        /* note, however, that it **will** let you subtract them */
                        a = (char *)((unsigned long)(a) + (unsigned long)(ppg));
                        goesafter = a;
```

```
                        a += sepentlen(a);
                }
        }

        while(!i) {
                if (*(long *)(a) == son)
                        ++i;
                else {
                        a +=sizeof(long);
                        a = (char *)(*(ushort *)(a));
                        if (a == (char *)(0))
                                --i;
                        else {
                                a = (char *)((unsigned long)(a) + (unsigned long)(ppg));
                                goesafter = a;
                                a += sepentlen(a);
                        }
                }
        }

        if (i > 0)  /* found it */
                a += sizeof(long); /* pt at offset to use in linking */
        else { /* error: not found */
                db_err(29,UPD_PAREN,-1,"no P");
                free(ppg);
                return(FAIL);
        }

        /* is there space for the new entry? */

        i = (sp.te_len >> 3) + 3;
        i += sizeof(long) + sizeof(ushort) << 1;
        if (i < (int)(*(ushort *)(ppg + X_SPCOFF))) { /* insert now */
                /* put the code here for now...change the inequality and */
                /* move this to after the if later on */
                /* PAGES MUST NOT HAVE HOLES ! */
                space = (ushort *)(ppg + X_SPCOFF);
                b = ppg + *space;
                *(ushort *)(b) = sp.te_len;
                b += sizeof(ushort);
                *space -= sizeof(ushort);
                sepat = b;                      /* don't move this line */
                for (i = 0; i < sp.te_len >> 3; i++)
                        *b++ = *(sp.te_item + i);
                j = ((j = (long)(sepat + i) & 0x3) ? (4 - j) : 0);
                for (i = 0; i < j; i++)
                        *b++ = 0xff;
                *space -= j + (sp.te_len >> 3);
                *(long *)(b) = sp.te_ptr;
                *space -= sizeof(long);
                b += sizeof(long);
                *(ushort *)(b) = *(ushort *)(a);
                *space -= sizeof(ushort);
                *(ushort *)(a) = (ushort)((sepat - sizeof(ushort)) - ppg);

                /* new entry linked into list on the page...done! */
```

```
                /* write out updated parent page */

                if ((i = flushpage(ppg,rptr -> fd,parent,x -> xpgsize)) == FAIL)
                        db_err(59,UPD_PAREN,errno,"upd parent");

                free(ppg);
                if (didsplit) free(xpg);

                return((i == FAIL) ? FAIL : SUCCESS);
        }
        else { /* no room at the inn */
                if (didsplit) { /* something's very wrong */
                        db_err(30,UPD_PAREN,-1,"parent");
                        free(ppg); free(xpg);
                        return(FAIL);
                }

                /* fake parms for recursive call to split */

                if ((xpg = malloc(x -> xpgsize)) == (char *)(NULL)) {
                        db_err(52,UPD_PAREN,errno,"xpg cache");
                        free(ppg);
                        return(FAIL);
                }
                sepat = a;
                a = xpg;
                b = x -> xbufat;
                j = x -> xpgsize;
                for (i = 0; i < j; i++)
                        *a++ = *b++;
                a = x -> xbufat;
                b = ppg;
                for (i = 0; i < j; i++)
                        *a++ = *b++;
                oldxpgat = x -> xpgat;
                x -> xpgat = parent;
                posn = -(((long)(goesafter - ppg) + oldxpgat) - 2L);
                if (split(rptr,&(sp.te_item),posn,nodemode,sp.te_ptr) == FAIL){
                        db_err(68,UPD_PAREN,-1,"split");
                        free(ppg); free(xpg);
                        return(FAIL);
                }
                ++didsplit;
                /* now have to re-load the child page to pick */
                /* up its new predecessor pointer (may be same */

                x -> xpgat = oldxpgat;
                if (loadpage(rptr -> fd,x,x -> xbufat,x -> xpgat) == FAIL){
                        db_err(60,UPD_PAREN,errno,"get gparent");
                        free(ppg); free(xpg);
                        return(FAIL);
                }
                parent = *(long *)(x -> xbufat + X_PREDOFF);
                goto getparent;

        }
```

```
}
```

```
#include "mrds.h"

char *substr(s,z,b,t)
char *s,*z;
int t,b;

{
        register char *ptr = z;
#ifdef XTRACE
fprintf(stdout,"--> substr(|%s|,*z,%d,%d\n",s,b,t);fflush(stdout);
#endif
        if (t <= 0 || t > (MAXSUBSTR - 1))
        {
                db_err(64,11,-1,ptr);
                return(NULL);
        }

        for ( ; t > 0; t--,b++)
                *ptr++ = *(s + b);
        *ptr = NULLC;
#ifdef XTRACE
fprintf(stdout,"<-- substr(|%s|\n",z);fflush(stdout);
#endif
        return(z);
}
```

```
#include "mrds.h"

int syncrel()

{
        extern int errno;
        extern DBSTATUS sys_db;
        extern REL relcore[];
        extern DOM domcore[];
        extern RD rdcore[];
        extern char inbufr[];
        extern struct rstat sysrelstat;
        register int i,fd;
        register long size;
        long relsize,domsize,rdsize;
        int open(),close(),write(),logent();
        int retval = 0;
        char *malloc();

#ifdef XTRACE
fprintf(stdout,"--> syncrel()\n");fflush(stdout);
#endif

        /* write out `rel' entries */
        sprintf(inbufr,"%s.rel",sys_db.dbs_homedir);
        if ((fd = open(inbufr,WRITMODE)) == FAIL) {
                db_err(51,SYNCREL,errno,inbufr);
                retval |= 4;
        }

        relsize = RELWIDTH * sysrelstat.numrelents;
        domsize = DOMWIDTH * sysrelstat.numdoments;
        rdsize  = RDWIDTH * sysrelstat.numrdents;


        size = relsize;
        relcore[0].cursize = size;
        if (!(retval & 4))
                if ((i = write(fd,relcore,(int)(size))) != (int)(size)) {
                        sprintf(inbufr,"%d:%d",size,i);
                        db_err(54,SYNCREL,errno,inbufr);
                        close(fd);
                        retval |= 4;
                }

        close(fd);

        /* write out `dom' entries */
        sprintf(inbufr,"%s.dom",sys_db.dbs_homedir);
        if ((fd = open(inbufr,WRITMODE)) == FAIL) {
                db_err(51,SYNCREL,errno,inbufr);
                retval |= 2;
        }

        size = domsize;
        if (!(retval & 2))
```

```
                if ((i = write(fd,domcore,(int)(size))) != (int)(size)) {
                        sprintf(inbufr,"%d:%d",size,i);
                        db_err(54,SYNCREL,errno,inbufr);
                        close(fd);
                        retval |= 2;
                }

        close(fd);

        /* write out `rd' entries */
        sprintf(inbufr,"%s.rd",sys_db.dbs_homedir);
        if ((fd = open(inbufr,WRITMODE)) == FAIL) {
                db_err(51,SYNCREL,errno,inbufr);
                retval |= 1;
        }

        size = rdsize;
        if (!(retval & 1))
                if ((i = write(fd,rdcore,(int)(size))) != (int)(size)) {
                        sprintf(inbufr,"%d:%d",size,i);
                        db_err(54,SYNCREL,errno,inbufr);
                        close(fd);
                        retval |= 1;
                }

        close(fd);
        if (fplog) {
                logent("CLOSED",sys_db.dbs_name);
                fclose(fplog);
        }
#ifdef XTRACE
fprintf(stdout,"<-- syncrel()\n");fflush(stdout);
#endif

        return(retval ? -retval : SUCCESS);
}
```

```
#include "mrds.h"

long timer()

{
#ifdef BSD
        struct timeval tp,*tptr;
        struct timezone tzp,*tzptr;
        int gettimeofday();
#endif
#ifdef OLDUNIX
        long time();
#endif

#ifdef BSD
        tptr = &tp;   tzptr = &tzp;
        gettimeofday(tptr,tzptr);
#ifdef XTRACE
fprintf(stdout,"time = %d\n",tp.tv_sec);
#endif
        return(tp.tv_sec);
#endif

#ifdef OLDUNIX
        return(time(0));
#endif

}
```

```
#include "mrds.h"

int wrpage(fd,ptr,len)
int fd;
long *ptr;
long len;

{
        extern int errno;
        register int i,j;
        register long pos;
        int read(),openrel(),db_err();
        long lseek();
        char leaf, *buf;

#ifdef XTRACE
fprintf(stdout,"--> wrpage(%d,X%lx,%ld)\n",fd,ptr,len);
fflush(stdout);
#endif

/*      if (!rptr) return (FAIL); */


        /* conservatism since lseek check should catch this error */

        if ((ptr < 0L) || (ptr > rptr -> cursize)) { /* ptr not within rel */
                db_err(85,WRPAGE,-1,rptr -> relname);
                return(FAIL);
        }

        /* could add check to assure 'pos' is on a page boundary here */

        /* go to that page */

        if ((pos = lseek(fd,ptr,FROMTOP)) != ptr) {
                db_err(60,WRPAGE,errno,rptr -> relname);
                return(FAIL);
        }

        if ((i = write(fd, leaf, 1) != 1)) {
                db_err(60,WRPAGE,errno,rptr -> relname);
                return(FAIL);
        }

        if ((pos = lseek(rptr -> fd,-1L,FROMCUR)) != ptr) { /* cannot happen */
                db_err(60,WRPAGE,errno,rptr -> relname);
                return(FAIL);
        }

        leaf &= XLFPG;
        if (leaf) /* is a leaf page */
                buf = x -> lbufat;
        else
                buf = x -> xbufat;

        j = (leaf ? x -> lpgsize : x -> xpgsize);
```

```
        if ((i = read(rptr -> fd, buf, j)) != j) {
                db_err(60,WRPAGE,errno,rptr -> relname);
                return(FAIL);
        }

        /* no pointers to update: rindx should only point at tuples */

#ifdef XTRACE
fprintf(stdout,"<-- rdpage(size = %d, leaf = %d)\n",i,(leaf ? 1 : 0));
fflush(stdout);
#endif

        return((int)(leaf));
}
```

```
#include "mrds.h"

int wrtuple(rptr,from)
REL  *rptr;
char *from;

{
        extern int errno;
        register int i;
        register long pos;
        int read();
        long lseek();

#ifdef XTRACE
fprintf(stdout,"--> wrtuple(%s,X%lx)\n",rptr -> relname,from);
fflush(stdout);
#endif

        if (!rptr) return (FAIL);

        if (rptr -> mode & WRINH) {
                db_err(59,WRTUPLE,-1,rptr -> relname);
                return(FAIL);
        }

        /* conservatism since lseek check should catch this error */

        if (rptr -> windx > rptr -> maxsize) {
                db_err(85,WRTUPLE,-1,rptr -> relname);
                return(FAIL);
        }

        /* appending or overwriting? check if overwriting allowed */

        if ((rptr -> windx < rptr -> cursize) && (rptr -> mode & APONLY)) {
                db_err(59,WRTUPLE,-1,rptr -> relname);
                return(FAIL);
        }

        if (rptr -> fd == FAIL) /* file holding rel not open yet */
                if ((rptr -> fd = openrel(rptr,O_RDWR)) == FAIL) {
                        db_err(51,WRTUPLE,-1,rptr -> relname);
                        return(NULL);
                }
        else {/* may be open, but for reading only: reopen as RDWR */
                closerel(rptr);
                if ((rptr -> fd = openrel(rptr,O_RDWR)) == FAIL) {
                        db_err(51,WRTUPLE,-1,rptr -> relname);
                        return(NULL);
                }
        }

        if ((pos = lseek(rptr -> fd, rptr -> windx,FROMTOP)) != rptr -> windx){
                db_err(60,WRTUPLE,errno,rptr -> relname);
                return(FAIL);
        }
```

```
        if ((i = write(rptr -> fd, from,(int)(rptr -> width))) != (int)(rptr -> width)){
                db_err(60,WRTUPLE,errno,rptr -> relname);
                rptr -> windx = pos; /*reset to tuple start */
                return(FAIL);
        }

        /* write succeeded */
        rptr -> windx += (long)(rptr -> width);
        /* if appended, then file grew */
        if (rptr -> windx == rptr -> cursize + (long)(rptr -> width))
                rptr -> cursize = rptr -> windx;

#ifdef XTRACE
fprintf(stdout,"<-- wrtuple(width = %d)\n",rptr -> width);
fflush(stdout);
#endif

        return(i);
}
```

```
int yesno(string)

   char *string;

   {
      while (*string) { /* while not end of string */
         if (*string == 'Y' || *string =='y') return(2);
         else if(*string =='N' || *string == 'n') return(1);
               else ++string;
      }
      return(0);
   }
```

```
#include "mrds.h"

int z(rptr,fromZmap,toZmap,tplptr)
REL *rptr;
long fromZmap;
long toZmap;
char *tplptr;

{
        extern char inbufr [];
        extern int errno,rdents;
        register int i,j,k;
        register long n;
        char *getuple(),*malloc(),pdomlist[MAXDOMS][MAXDNMLEN],mode,*work,*tpl;
        char bitmask,*gtpl,*wtpl;
        int adtuple(),strcmp(),closerel(),openrel(),attcount[MAXATTS];
        int zatts=0,newzatts=0,goodadts=0,werr=0,numz,numnz,numnewz,numnewnz;
        int domlens[MAXATTS];
        unsigned restoreclist(),freeclist(),freellist();
        ZCLIST *mkwrklist();
        long timer(),readat,writeat,zbits=0L,newzbits=0L;
        long etime,now,deltatm;
        RD *rds[MAXATTS],*findrd();
        DOM *finddom(),*dptr;
        ZCLIST *zptr,*ztmp,*newzptr;
        ZCLIST *newztmp;
        ZLLIST *nzptr,*nztmp;
        ZLLIST *newnzptr,*newnztmp;
        Zslot *zsp;

#ifdef XTRACE
fprintf(stdout,"--> z(%s,<map:%lxX,>map:%lxX,%lxX)\n",rptr->relname,fromZmap,toZmap,tplptr);
fflush(stdout);
#endif
        if (fromZmap == toZmap)
                return(SUCCESS); /* fast way to do them */

        numz = numnz = 0;
        numnewz = numnewnz = 0;

        /* find slot for this relation */

        zsp = (Zslot *)(NULL);
        for (i = 0; i < ZSLOTS; i++)
                if (zslot[i].zrptr == rptr &&
                    zslot[i].frommap == fromZmap &&
                    zslot[i].tomap == toZmap) {
                        zsp = &zslot[i];
                        zslot[i].zactime = timer();
                        break;
                }

#ifdef XTRACE
if (zsp == (Zslot *)(NULL))
fprintf(stdout,"ZZZ:\tneed new slot for %s\n",rptr->relname);
else
```

```
fprintf(stdout,"ZZZ:\tusing previous slot %d\n",i);
fflush(stdout);
#endif
        if (zsp == (Zslot *)(NULL)) { /* LOTS of work to do! */

                /* get rd and dom particulars */

                if ((findrd(rptr -> relname,(char *)(0),0,rds)) == (RD *)(NULL))
                        return(FAIL);
                for (i = 0; i < rdents; i++)
                        if ((dptr = finddom(rds[i] -> domname)) == (DOM *)(NULL)) {
                                db_err(79,Z_ORD,-1,rds[i] -> domname);
                                domlens[i] = 0;
                        }
                        else domlens[i] = dptr -> len;


                /* find least recently used slot */

                deltatm = 0L; now = timer(); etime = 0L;
                for (i = 0; i < ZSLOTS; i++)
                        if ((deltatm = now - zslot[i].zactime) > etime) {
                                etime = deltatm;
                                zsp = &zslot[i];
                        }

                /* begin loading this slot */

                zsp -> zrptr = rptr;
                zsp -> tomap = toZmap;
                zsp -> frommap = fromZmap;
                zsp -> zrdents = rdents;

                /* free previous slot occupant's lists */

                freeclist(zsp -> fromz);  freeclist(zsp -> toz);
                freeclist(zsp -> wkfrom); freeclist(zsp -> wkto);
                freellist(zsp -> froml);  freellist(zsp -> tol);
                free(zsp -> tbuf);
                zsp -> fromz = (ZCLIST *)(0);   zsp -> toz = (ZCLIST *)(0);
                zsp -> wkfrom = (ZCLIST *)(0);  zsp -> wkto = (ZCLIST *)(0);
                zsp -> froml = (ZLLIST *)(0);   zsp -> tol = (ZLLIST *)(0);

                /* build circular and linked lists if needed */

                if (fromZmap) /* if there are now z-ordered domains */
                        for (i = 0; i < zsp -> zrdents; i++)
                                if (zsp -> frommap & (1L << i)) { /* circular list entry */
                                        if (zsp -> fromz) { /* attach another entry */
                                                if ((ztmp = (ZCLIST *)(malloc(sizeof(ZCLIST)))) == NULL) {
                                                        db_err(53,Z_ORD,errno,"zattrdnnd");
                                                        ++werr;
                                                        goto zbomb;
                                                }
                                                zptr -> next = ztmp;
                                                ztmp -> prev = zptr;
```

```
                                        zsp -> fromz -> prev = ztmp;
                                        ztmp -> next = zsp -> fromz;
                                        zptr = ztmp;
                                }
                                else { /* make first entry into list */
                                        if ((zsp -> fromz = (ZCLIST *)(malloc(sizeof(ZCLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"zattrnd");
                                                ++werr;
                                                goto zbomb;
                                        }
                                        zsp -> fromz -> next = zsp -> fromz;
                                        zsp -> fromz -> prev = zsp -> fromz;
                                        zptr = zsp -> fromz;
                                }
                                zptr -> numzbits = (long)(domlens[i]) << 3L;
                                zptr -> outposn = (long)(rds[i] -> pos) << 3L;
                                zbits += zptr -> numzbits;
                                ++numz;
                        }
                        else { /* linked list entry */
                                if (zsp -> froml) { /* attach another item */
                                        if ((nztmp = (ZLLIST *)(malloc(sizeof(ZLLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"nzattrnnd");
                                                ++werr;
                                                goto zbomb;
                                        }
                                        nztmp -> next = nzptr -> next;
                                        nzptr -> next = nztmp;
                                }
                                else { /* make first entry */
                                        if ((zsp -> froml = (ZLLIST *)(malloc(sizeof(ZLLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"nzattrnd");
                                                ++werr;
                                                goto zbomb;
                                        }
                                        nzptr = zsp -> froml;
                                        nzptr -> next = (ZLLIST *)(0);
                                }
                                nzptr -> tounz = rds[i] -> pos;
                                nzptr -> fromz = i;
                                ++numnz;
                        }
        zsp -> fromzbits = zbits;

        /* build circular and linked lists if needed for new tuples */

        if (zsp -> tomap)
                for (i = 0; i < rdents; i++)
                        if (zsp -> tomap & (1L << i)) { /* circular list entry */
                                if (zsp -> toz) { /* attach another entry */
                                        if ((newztmp = (ZCLIST *)(malloc(sizeof(ZCLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"newzatnnd");
                                                ++werr;
                                                goto zbomb;
                                        }
                                        newzptr -> next = newztmp;
```

```
                                        newztmp -> prev = newzptr;
                                        zsp -> toz -> prev = newztmp;
                                        newztmp -> next = zsp -> toz;
                                        newzptr = newztmp;
                                }
                                else { /* make first entry in circular list */
                                        if ((zsp -> toz = (ZCLIST *)(malloc(sizeof (ZCLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"newzathd");
                                                ++werr;
                                                goto zbomb;
                                        }
                                        zsp -> toz -> next = zsp -> toz;
                                        zsp -> toz -> prev = zsp -> toz;
                                        newzptr = zsp -> toz;
                                }
                                newzptr -> numzbits = (long)(domlens[i]) << 3L;
                                newzptr -> outposn = (long)(rds[i] -> pos) << 3L;
                                newzbits += newzptr -> numzbits;
                                ++numnewz;
                        }
                        else { /* linked list entry */
                                if (zsp -> tol) { /* attach another entry */
                                        if ((newnztmp = (ZLLIST *)(malloc(sizeof(ZLLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"newnz");
                                                ++werr;
                                                goto zbomb;
                                        }
                                        newnztmp -> next = newnzptr -> next;
                                        newnzptr -> next = newnztmp;
                                }
                                else { /* make fist linked list entry */
                                        if ((zsp -> tol = (ZLLIST *)(malloc(sizeof(ZLLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"newnzhd");
                                                ++werr;
                                                goto zbomb;
                                        }
                                        newnzptr = zsp -> tol;
                                        newnzptr -> next = (ZLLIST *)(0);
                                }
                                newnzptr -> tounz = rds[i] -> pos;
                                newnzptr -> fromz = i;
                                ++numnewnz;
                        }
        zsp -> tozbits = newzbits;

        /* now go through nzlist and fix up `from z' values */

        nzptr = zsp -> froml;
        i = 0;
        j = 0;
        k = (int)(zbits >> 3L); /* remainders?????? */
        while (nzptr) {
                i = nzptr -> fromz;
                nzptr -> fromz = j + k;
                j += domlens[i];
                nzptr = nzptr -> next;
```

```
                }

                newnzptr = zsp -> tol;
                i = j = 0;
                while (newnzptr) {
                        i = newnzptr -> fromz;
                        newnzptr -> fromz = j + k;
                        j += domlens[i];
                        newnzptr = newnzptr -> next;
                }

                /* get work buffer for tuples */

                if ((zsp -> tbuf = malloc((int)(rptr -> width))) == NULL) {
                        db_err(53,Z_ORD,errno,"zwork");
                        ++werr;
                        goto zbomb;
                }
                wtpl = zsp -> tbuf;
                for (i = 0; i < rptr -> width; i++) *(wtpl + i) = NULLC;

                /* make working copy of zclists */

                if (zsp -> frommap) /* if any tuples now in z order */
                    if ((zsp -> wkfrom = mkwrklist(numz)) == (ZCLIST *)(0)) {
                        db_err(53,Z_ORD,errno,"wzat");
                        ++werr;
                        goto zbomb;
                }
                if (zsp -> tomap)
                    if ((zsp -> wkto = mkwrklist(numnewz)) == (ZCLIST *)(0)) {
                        db_err(53,Z_ORD,errno,"newwzat");
                        ++werr;
                        goto zbomb;
                }

        } /* if have to add rel to new slot */

        /* make sure not trying to change z map mid-relation */
        if (toZmap != zsp -> tomap)
                db_err(87,Z_ORD,-1,rptr -> relname);

        if (zsp -> frommap)
                restoreclist(zsp -> fromz,zsp -> wkfrom,(short)(1));
        if (zsp -> tomap)
                restoreclist(zsp -> toz,zsp -> wkto,(short)(0));

        gtpl = tplptr;
        wtpl = zsp -> tbuf;
        for (i = 0; i < rptr -> width; i++) *(wtpl + i) = NULLC;
                if (zsp -> frommap){ /* must unshuffle */
                        zptr = zsp -> wkfrom;
                        for (n = 0L; n < zsp -> fromzbits; n++) { /* for all participating bits */
                                bitmask = (*gtpl & (char)(1L << (n % 8L)) ? 1 : 0);
                                ztmp = zptr;
                                while (! zptr -> numzbits) {
```

```
                                        zptr = zptr -> next;
                                        if (zptr == ztmp)
                                            break; /* should never happen; flakey */
                                    }
                                    bitmask = bitmask << (zptr -> outposn % 8L); /* put bit in its place */
                                    *(wtpl + (zptr -> outposn / 8L)) |= bitmask;
#ifdef XTRACE
fprintf(stdout,"ZZZ: unbitmask = %x using bit to posn %d\n",
        bitmask,zptr->outposn);
fflush(stdout);
#endif
                                    --(zptr -> numzbits);
                                    ++(zptr -> outposn);
                                    zptr = zptr -> next;
                                    if (!((n + 1) % 8L)) ++gtpl;
                            }

                            /* now must reposition any unshuffled bits */

                            if (rptr -> mode & PZREL) {
                                    nzptr = zsp -> froml;
                                    for (i = 0; i < zsp -> zrdents; i++)
                                            if ((~(zsp -> frommap)) & (1L << i)) { /* ith attr not in zatts */
                                                    /* where to put it */
                                                    wtpl = zsp -> tbuf + nzptr -> tounz;
                                                    /* where from */
                                                    gtpl = tplptr + nzptr -> fromz;
                                                    for (j = 0; j < domlens[i]; j++)
                                                            *(wtpl +j) = *(gtpl + j);
                                                    nzptr = nzptr -> next;
                                            }
                            }
/*                          restoreclist(zattrlist,wzattrlist,(short)(1)); */
                    }

                    /* tuple is now completely unshuffled; is there shuffling to do? */

                    if (zsp -> tomap) { /* shuffling to do */
                            gtpl = tplptr;
                            wtpl = zsp -> tbuf;
                            if (zsp -> frommap) /* if unshuffling happened */
                                    for (i = 0; i < rptr -> width; i++)
                                            *(gtpl + i) = *(wtpl + i);
                            for (i = 0; i < rptr -> width; i++) *(wtpl + i) = NULLC;

                            newzptr = zsp -> wkto;
                            zptr = zsp -> toz;
                            for (n = 0L; n < zsp -> tozbits; n++) { /* for all pariticpating bits */
                                    newztmp = newzptr;
                                    ztmp = zptr;
                                    while (newzptr -> numzbits >= zptr -> numzbits) { /* flakey */
                                            newzptr = newzptr -> next;
                                            zptr = zptr -> next;
                                            if (newzptr == newztmp) break;
                                    }
                                    bitmask = ((*(gtpl + (newzptr -> outposn >> 3L)) &
```

```
                                    ((char)(1L << newzptr -> numzbits % 8L))) ? 1 : 0);
                                bitmask = bitmask << (n % 8L);
                            *(wtpl + n / 8L) |= bitmask;
#ifdef XTRACE
fprintf(stdout,"ZZZ: bitmask = %x using bit from posn %d\n",
        bitmask,newzptr->outposn);
fflush(stdout);
#endif
                                ++(newzptr -> numzbits);
                                ++(newzptr -> outposn);
                                newzptr = newzptr -> next;
                                zptr = zptr -> next;
                        }

                        /* now reposition attributes not in z order */

                        if (rptr -> mode & PZREL) {
                                newnzptr = zsp -> tol;
                                for (i = 0; i < rdents; i++) /* this attr not in newzatts */
                                        if ( ~(zsp -> frommap) & (1L << i)) {
                                                /* where it goes */
                                                wtpl = zsp -> tbuf + newnzptr -> fromz;
                                                /* where it is from */
                                                gtpl = tplptr + newnzptr -> tounz;
                                                for (j = 0; j < domlens[i]; j++)
                                                        *(wtpl + j) = *(gtpl + j);
                                                newnzptr = newnzptr -> next;
                                        }
                        }
/*                      restoreclist(newzattrlist,newwzattrlist,(short)(0)); */
                }

                /* tuple ready to be added to output from `work' */

                wtpl = zsp -> tbuf;
                gtpl = tplptr;
                for (i = 0; i < rptr -> width; i++)
                        *(gtpl + i) = *(wtpl + i);

        /*}  UNMATCHED !!!!!! */

        /* close up shop */
zbomb:
        /* release dynamically allocated storage */
        if (werr) {
                freeclist(zsp -> fromz);
                freeclist(zsp -> toz);
                freeclist(zsp -> wkfrom);
                freeclist(zsp -> wkto);
                freellist(zsp -> froml);
                freellist(zsp -> tol);
                free(zsp -> tbuf);
                werr = 0;
        }
        else ++werr;
```

```
#ifdef XTRACE
fprintf(stdout,"<-- z(%d)\n",(werr ? werr : FAIL)); fflush(stdout);
#endif

        return(werr ? werr : FAIL);

] /* end z */
```

```
#include "mrds.h"

int Z(rptr,map,rds,out)
REL *rptr;
long map;
RD *rds[];
char *out;

{
        extern char inbufr [];
        extern int errno,rdents;
        register int i,j,k;
        register long n;
        char *getuple(),*malloc(),pdomlist[MAXDOMS][MAXDNMLEN],mode,*work,*tpl;
        char bitmask,*gtpl,*wtpl;
        int adtuple(),strcmp(),closerel(),openrel(),attcount[MAXATTS];
        int zatts=0,newzatts=0,goodadts=0,werr=0,numz,numnz,numnewz,numnewnz;
        int domlens[MAXATTS];
        unsigned restoreclist(),freeclist(),freellist();
        ZCLIST *mkwrklist();
        long readat,writeat,zbits=0L,newzbits=0L;
        DOM *finddom(),*dptr;
        REL *mkrel(),*outrel;
        ZCLIST *zattrlist,*zptr,*ztmp,*wzattrlist,*newzattrlist,*newzptr;
        ZCLIST *newztmp,*newwzattrlist;
        ZLLIST *nzattrlist,*nzptr,*nztmp,*wnzattrlist,*newnzattrlist;
        ZLLIST *newnzptr,*newnztmp,*newwnzattrlist;

#ifdef XTRACE
fprintf(stdout,"--> Z(%s,%ld,%lxX,%s)\n",rptr->relname,map,rds,out);
fflush(stdout);
#endif

        outrel = (REL *)(0);
        numz = numnz = 0;
        numnewz = numnewnz = 0;
        zattrlist = newzattrlist = (ZCLIST *)(0);
        wzattrlist = newwzattrlist = (ZCLIST *)(0);
        nzattrlist = newnzattrlist = (ZLLIST *)(0);

        /* make sure there is something to do */
        if (map == rptr -> Zmap) { /* no work */
                return(SUCCESS);
        }

        /* determine mode for output relation */
        if (!map) {
                mode = ~CONREL | ~ZORD ;
                newzatts = 0;
        }
        else if (((1L << rdents) - 1L) == map){
                mode = ZORD;
                newzatts = rdents;
        }
        else {
                mode = ZORD | PZREL ;
```

```
        for (i = 0; i < rdents; i++)
                if (map & (1L << i))
                        newzatts++;
}

/* in place replacement or new output relation ? */

if(!strcmp(out,rptr->relname)) { /* in place */
        /* badindex(rptr) */
        if (rptr -> fd != FAIL)
                closerel(rptr);
        if ((openrel(rptr,RDWRMODE)) == FAIL) {
                db_err(51,Z_ORD,errno,rptr->relname);
                return(FAIL);
        }
        rptr -> rindx = 0L;
        rptr -> windx = 0L;
        outrel = rptr;
}
else { /* output is a new relation */
        /* build domain list */
        for (i = 0; i < rdents; i++)
                strcpy(pdomlist[i],rds[i]->domname);
        if ((outrel = mkrel(out,mode,map,rdents,pdomlist,
        (rptr->maxsize / (long)(rptr -> width)))) == NULL) {
                db_err(57,Z_ORD,-1,out);
                return(FAIL);
        }
}

/* get lengths of each domain */
for (i = 0; i < rdents; i++)
        if ((dptr = finddom(rds[i] -> domname)) == NULL) {
                db_err(79,Z_ORD,-1,rds[i] -> domname);
                domlens[i] = 0;
        }
        else domlens[i] = dptr -> len;

/* how many domains now in z order ? */
if (rptr -> mode & ZORD)
        if (rptr -> mode & PZREL) {
                for (i = 0; i < rdents; i++)
                        if (rptr -> Zmap & (1L << i)) zatts++;
        }
        else zatts = rdents;
else zatts = 0;

/* build circular and linked lists if needed */

if (zatts) /* if there are now z-ordered domains */
        for (i = 0; i < rdents; i++)
                if (rptr -> Zmap & (1L << i)) { /* circular list entry */
                        if (zattrlist) { /* attach another entry */
                                if ((ztmp = (ZCLIST *)(malloc(sizeof(ZCLIST)))) == NULL) {
                                        db_err(53,Z_ORD,errno,"zattrdnnd");
                                        goto zbomb;
```

```
                                        }
                                        zptr -> next = ztmp;
                                        ztmp -> prev = zptr;
                                        zattrlist -> prev = ztmp;
                                        ztmp -> next = zattrlist;
                                        zptr = ztmp;
                                }
                                else { /* make first entry into list */
                                        if ((zattrlist = (ZCLIST *)(malloc(sizeof(ZCLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"zattrnd");
                                                goto zbomb;
                                        }
                                        zattrlist -> next = zattrlist;
                                        zattrlist -> prev = zattrlist;
                                        zptr = zattrlist;
                                }
                                zptr -> numzbits = (long)(domlens[i]) << 3L;
                                zptr -> outposn = (long)(rds[i] -> pos) << 3L;
                                zbits += zptr -> numzbits;
                                ++numz;
                        }
                        else { /* linked list entry */
                                if (nzattrlist) { /* attach another item */
                                        if ((nztmp = (ZLLIST *)(malloc(sizeof(ZLLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"nzattrnnd");
                                                goto zbomb;
                                        }
                                        nztmp -> next = nzptr -> next;
                                        nzptr -> next = nztmp;
                                }
                                else { /* make first entry */
                                        if ((nzattrlist = (ZLLIST *)(malloc(sizeof(ZLLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"nzattrnd");
                                                goto zbomb;
                                        }
                                        nzptr = nzattrlist;
                                        nzptr -> next = (ZLLIST *)(0);
                                }
                                nzptr -> tounz = rds[i] -> pos;
                                nzptr -> fromz = i;
                                ++numnz;
                        }

        /* build circular and linked lists if needed for new relation */

        if (newzatts)
                for (i = 0; i < rdents; i++)
                        if (map & (1L << i)) { /* circular list entry */
                                if (newzattrlist) { /* attach another entry */
                                        if ((newztmp = (ZCLIST *)(malloc(sizeof(ZCLIST)))) == NULL) {
                                                db_err(53,Z_ORD,errno,"newzatnnd");
                                                goto zbomb;
                                        }
                                        newzptr -> next = newztmp;
                                        newztmp -> prev = newzptr;
                                        newzattrlist -> prev = newztmp;
```

```
        closerel(outrel);
        closerel(rptr);

        /* release dynamically allocated storage */

        freeclist(zattrlist);
        freeclist(newzattrlist);
        freeclist(wzattrlist);
        freeclist(newwzattrlist);
        freellist(nzattrlist);
        freellist(newnzattrlist);
        free(work);

        return(werr ? werr : FAIL);
} /* end z */
```

```
unsigned freeclist(cptr)
ZCLIST *cptr;

{
        register unsigned num = 0;
        register ZCLIST *a, *b;
        int free();

#ifdef XTRACE
fprintf(stdout,"--> freeclist(%lxX)\n",cptr); fflush(stdout);
#endif

        if (!cptr)
                return(0);
        a = b = cptr;
        while (a -> next != cptr) {
                free(a);
                a = b -> next;
                b = a;
                ++num;
        }
        free(a);
#ifdef XTRACE
fprintf(stdout,"<-- freeclist(%d)\n",num + 1); fflush(stdout);
#endif
        return(num + 1);
}



unsigned freellist(lptr)
ZLLIST *lptr;

{
        register unsigned num = 0;
        register ZLLIST *a, *b;
        int free();

        if (!lptr)
                return(0);
        a = b = lptr;
        while (a -> next != (ZLLIST *)(0)) {
                free(a);
                a = b -> next;
                b = a;
                ++num;
        }
        free(a);
        return(num + 1);
}
```

```
int unshuffle(rname,domlist,outname)
char *rname;                    /* name of relation to shuffle   */
char domlist[][MAXDNMLEN];      /* list of domains to unshuffle  */
char *outname;                  /* name of output relation       */

{
        extern int errno,rdents;
        extern char inbufr[];
        register int i,j,k;
        REL *findrel(),*rptr;
        RD *findrd(),*rdtab[MAXATTS];
        long newmap=0L,mapmask;
        int strcmp(),domfnd,domerr=0;

        /* find relation to unshuffle */
        if ((rptr = findrel(rname)) == NULL) {
                db_err(58,Z_ORD,-1,rname);
                return(FAIL);
        }

        if ((findrd(rname,(char *)(0),0,rdtab)) == FAIL)
            return(FAIL);

        /* how many participating domains? 0 ==> all */
        i = 0;
        while (*domlist[i] && i < (MAXATTS + 2)) ++i;
        if (i > MAXATTS ) {
                db_err(77,Z_ORD,-1,rname);
                i = rdents;
        }

        /* if rname is already non z and all domains are asked */
        /* to participate here, no change is being asked for  */
        if ( !(rptr -> mode & ZORD) && !i){
                db_err(98,Z_ORD,-1,"no change");
                return(-2);
        }

        if (i) { /* a list of participating domains was supplied */
                for (j = 0; j < i; j++) { /* for each domain on list */
                        /*find jth item in domlist in rdtab */
                        domfnd = 0;
                        for (k = 0; k < rdents; k++)
                                if (!strcmp(domlist[j],rdtab[k]->domname)) {
                                        ++domfnd;
                                        break;
                                }
                        if (!domfnd) {
                                db_err(79,Z_ORD,-1,domlist[j]);
                                ++domerr;
                        }
                        else
                                mapmask = (1L << k);
                        if (mapmask & newmap)
                            db_err(78,Z_ORD,-1,domlist[j]);
```

```
                        newmap |= mapmask;
                } /* for each domain */

                if (domerr) {
                        sprintf(inbufr,"%d of %d missing",&domerr,&rdents);
                        db_err(79,Z_ORD,-1,inbufr);
                        return(FAIL); /* for now: safe way out */
                }
        } /* if a domlist was supplied */
        else /* no list: unshuffle everything */
        newmap = (1L << rdents) - 1L;

        /* newmap is now set for all attributes to appear unshuffled */
        /* is it worth it? */
        mapmask = newmap;
        if (! mapmask & rptr -> Zmap) {
                db_err(98,Z_ORD,-1,"no change");
                return(-2);
        }

        /* unshuffle */
        return (Z(rptr,~newmap,rdtab,outname));
}
```