CREATING AND MAINTAINING

A DATABASE USING INGRES

by

Robert Epstein

Memorandum No. UCB/ERL M77/71

16 December 1977

CREATING AND MAINTAINING A DATABASE USING INGRES

by

Robert Epstein

Memorandum No. UCB/ERL M77/71

16 December 1977

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Research sponsored by the National Science Foundation Grant MCS75-03839-A01, U.S. Army Research Office Grant DAAG29-76-G-0245, the Naval Electronic Systems Command Contract N00039-78-G-0013, and the Joint Services Electronics Program Contract F44620-76-C-0100.

CREATING AND MAINTAINING A DATABASE USING INGRES

1. INTRODUCTION

In this paper we describe how to create, structure and maintain relations in INGRES. It is assumed that the reader is familiar with INGRES and understands QUEL, the INGRES query language. It is strongly suggested that the document "A Tutorial on INGRES" (ERL M77/25) be read first.

This paper is divided into six sections

1. Introduction
2. Creating a Relation
3. Using Copy
4. Storage Structures
5. Secondary Indices
6. Recovery and Data Update

To create a new data base you must be a valid INGRES user and have "create data base" permission. These permissions are granted by the "ingres" superuser. If you pass those two requirements you can create a data base using the command to the Unix shell:

```
% creatdb mydata
```

where "mydata" is the name of the data base. You become the "data base administrator" (DBA) for mydata. As the DBA you have certain special powers.

1. Any relation created by you can be accessed by anyone else using "mydata". If any other user creates a relation it is strictly private and cannot be accessed by the DBA or any other user.

2. You can use the "-u" flag in ingres and printr. This enables you to use ingres on "mydata" with someone else's id. Refer to the INGRES reference manual under sections ingres(unix) and users(files) for details.

3. You can run sysmod, restore and purge on "mydata".

4. The data base by default is created to allow for multiple concurrent users. If only one user will ever use the data base at a time, the data base administrator can turn off the concurrency control. Refer to creatdb(unix) in the INGRES reference manual.

Once a data base has been created you should immediately run

```
% sysmod mydata
```

This program will convert the system relations to their "best" structure for use in INGRES. Sysmod will be explained further in section 4.

As a DBA or as a user you can create and structure new relations in any data base to which you have access. The remainder of this paper describes how this is done.

2. CREATING NEW RELATIONS IN INGRES

There are two ways to create new relations in INGRES.

```
     create
     retrieve into
```

"Retrieve into" is used to form a new relation from one or more existing relations. "Create" is used to create a new relation with no tuples in it.

example 1:

```
     range of p is parts
     range of s is supply
     retrieve into newsupply(
                   number = s.snum,
                   p.pname,
                   s.shipdate)
     where s.pnum = p.pnum
```

example 2:

```
     create newsupply(
            number = i2,
            pname = c20,
            shipdate = c8)
```

In example 1 INGRES creates a new relation called "newsupply", computing what the format of each domain should be. The query is then run and newsupply is modified to "cheapsort". (This will be covered in more detail in section 4.)

In example 2 "newsupply" is created and the name and format for each domain is given. The format types which are allowed are:

i1	1 byte integer
i2	2 " "
i4	4 " "
f4	4 byte floating point number
f8	8 " " " "
c1,c2,..,c255	1,2,..,255 byte character

In example 2, the width of an individual tuple is 30 bytes (2 + 20 + 8), and the relation has three domains. Beware that INGRES has limits. A relation cannot have more than 49 domains and the tuple width cannot exceed 498 bytes.

UNIX allocates space on a disk in units of 512 byte pages. INGRES gets a performance advantage by doing I/O in one block units. Therefore relations are divided into 512 byte pages. INGRES never splits a tuple between two pages. Thus some space can be wasted. There is an overhead of 12 bytes per page plus 2 bytes for every tuple on the page. The formulas are:

```
number tuples per page = 500/(tuple width + 2)

wasted space = 500 - number of tuples per page
*(tuple width +2)
```

For our example there are

```
22 = 500/(20 + 2)

16 = 500 - 22 * (20 + 2)
```

22 tuples per page and 16 bytes wasted per page. These computations are valid only for uncompressed relations. We will return to this subject in section 4 when we discuss compression.

If you forget a domain name or format, use the "help" command. For example if you gave the INGRES command:

```
help newsupply
```

the following would be printed:

```
Relation:               newsupply
Owner:                  bob
Tuple width:            30
Saved until:            Thu Nov 10 16:17:06 1977
Number of tuples:       0
Storage structure:      paged heap
Relation type:          user relation

 attribute name    type  length  keyno.

 number             i       2
 pname              c      20
 shipdate           c       8
```

Notice that every relation has an expiration date. This is set to be one week from the time when it was created. The "save" command can be used to save the relation longer. See "save(quel)" and "purge(unix)" in the INGRES reference manual.

3. COPYING DATA TO AND FROM INGRES

Once a relation is created, there are two mechanisms for inserting new data:

```
     append command
     copy command
```

Append is used to insert tuples one at a time, or for filling one relation from other relations.

Copy is used for copying data from a UNIX file into a relation. It is used for copying data from another program, or for copying data from another system. It is also the most convenient way to copy any data larger than a few tuples.

Let's begin by creating a simple relation and loading data into it.

Example:

```
     create donation (name = c10, amount = f4, ext = i2)
```

Now suppose we have two people to enter. The simplest procedure is probably to run the two queries in INGRES using the append command.

```
     append to donation (name="frank",amount = 5,ext = 204)

     append to donation (name="harry",ext = 209,amount = 4.50)
```

Note that the order in which the domains are given does not matter. INGRES matches by recognizing attribute names and does not care in what order attributes are listed. Here is what the relation "donation" looks like now:

donation relation

```
|name      |amount    |ext     |
|------------------------------|
|frank     |5.000     |204     |
|harry     |4.500     |209     |
|------------------------------|
```

We now have two people entered into the donation relation. Suppose we had fifty more to enter. Using the append command is far too tedious since so much typing is involved for each tuple. The copy command will better suit our purposes.

Copy can take data from a regular Unix file in a variety of formats and append it to a relation. To use the copy command first create a Unix file (typically using "ed") containing the data.

For example, let's put five new names in a file using the editor.

```
% ed
a
bill,3.50,302
sam,10.00,410
susan,,100
sally,.5,305
george,4.00,302
.
w newdom
68
q
%
```

The format of the above file is a name followed by a comma, followed by the amount, then a comma, then the extension, and finally a newline. Null entries, for example the amount for susan, are perfectly legal and default to zero for numerical domains and blanks for character domains.

To use copy we enter INGRES and give the copy command.

```
     copy donation (name = c0, amount = c0, ext = c0)
             from "/mnt/bob/newdom"
```

Here is how the copy command works:

```
     copy relname (list of what to copy) from "full pathname"
```

In the case above we wrote:

```
     copy donation (. . .) from "/mnt/bob/newdom"
```

Although amount and ext are stored in the relation as f4 (floating point) and i2 (integer), in the Unix file they were entered as characters. In specifying the format of the domain, copy accepts:

```
     domain = format
```

where domain is the domain name and the format in the UNIX file is one of

i1, i2, i4	(true binary integer of size 1, 2, or 4)
f4, f8	(true binary float point of size 4 or 8)
c1, c2, c3,...c255	(a fixed length character string)
c0	(a variable length character string delimited by a comma, tab or new line)

In the example we use

```
     name = c0, amount = c0, extension = c0
```

This means that each of the domains was stored in the Unix file as variable length character strings. Copy takes the first comma, tab, or new line character as the end of the string. This by far is the most common use of copy when the data is being entered into a relation for the first time.

Copy can also be used to copy data from a relation into a Unix file. For example:

```
     copy donation (name = c10, amount = c10, ext = c5)
              into "/mnt/bob/data"
```

This will cause the following to happen:

1. If the file /mnt/bob/data already exists it will be destroyed.

2. The file is created in mode 600 (read/write by you only)

3. Name will be copied as a 10 character field, immediately followed by amount, immediately followed by ext. Amount will be converted to a character field 10 characters wide. Ext will be converted to a character field 5 characters wide.

The file "/mnt/bob/data" would be a stream of characters looking like this:

```
frank          5.000   204harry            4.500   209bill
          3.500   302sam            10.000   410susan
      0.000   100sally           0.500   305george
   4.000   302
```

The output was broken into four lines to make it fit on this page. In actuality the file is a single line. Another example:

```
     copy (name = c0, colon = d1, ext = c0, comma = d1
             amt = c0, nl = d1) into "/mnt/bob/data"
```

In this example "c0" is interpreted to mean "use the appropriate character format". For character domains it is the width of the domain. Numeric domains are converted to characters according to the INGRES defaults (see ingres(unix)).

The statements:

```
     colon = d1
     comma = d1
     nl = d1
```

are used to insert one colon, comma, and newline into the file. The format "d1" is interpreted to mean one dummy character. When copying into a Unix file, a selected set of characters can be inserted into the file using this "dummy domain" specification. Here is what the file "/mnt/bob/data" would look like:

```
frank      :   204,      5.000
harry      :   209,      4.500
```

```
bill      :   302,      3.500
sam       :   410,     10.000
susan     :   100,      0.000
sally     :   305,      0.500
george    :   302,      4.000
```

If you wanted a file with the true binary representation of the numbers you would use:

```
     copy (name = c10, amount = f4, ext = i2)
```

This would create a file with the exact copy of each tuple, one after the other. This is frequently desireable for temporary backup purposes and it guarantees that floating point domains will be exact.

TYPICAL ERRORS

There are 17 different errors that can occur in copy. We will go through the most common ones.

Suppose you have a file with

```
bill,3.5,302
sam,10,410,
susan,3,100
```

and run the copy command

```
     copy donation (name = c0, amount = c0, ext = c0)
             from "/mnt/bob/data"
```

You would get the error message

```
5809: COPY: bad input string for domain amount. Input was "susan".
There were 2 tuples sucessfully copied from /mnt/bob/data into
donation.
```

What happened is that line 2 had an extra comma. The first two tuples were copied correctly. For the next tuple, name = "" (blank), amount = "susan", and ext = "3". Since "susan" is not a proper floating point number, an error was generated and processing was stopped after two tuples.

If you tried to copy the file with a file such as

```
nancy,5.0,35000
```

you would get the error message

```
5809: COPY: bad input string for domain ext. Input was "35000".
There were 0 tuples successfully copied from /mnt/bob/data into
donation.
```

Here, since ext is an i2 (integer) domain, it cannot exceed the value 32767.

There are numerous other error messages, most of which are self-explanatory.

In addition there are three, non-fatal warnings which may appear on a copy "from".

If you are copying from a file into a relation which is ISAM or hash, a count of the number of duplicate tuples will appear, (if there were any). This will never appear on a "heap" because no duplicate checking is performed.

INGRES does not allow control characters (such as "bell" etc.) to be stored. If copy reads any control characters, it converts them to blanks and reports the number of domains that had control characters in them.

If you are copying using the c0 option, copy will report if any character strings were longer than their domains and had to be truncated.

SPECIAL FEATURES

There are a few special functions that make copy a little easier to use

1. Bulk copy

If you ask for:

```
    copy relname () from "file"
        or
    copy relname () into "file"
```

copy expands the statement to mean:

> copy each domain in its proper order according to its proper format.

So, if you said

```
    copy donation () into "/mnt/bob/donation"
```

it would be the same as asking for:

```
    copy donation (name = c10, amount = f4, ext = i2)
        into "/mnt/bob/donation"
```

This provides a convenient way to copy whole relations to and from INGRES.

2. Dummy Domains

If you are copying data from another computer or program, frequently there will be a portion of data that you will want to ignore. This can be done using the dummy domain specifications d0, d1, d2 ... d511. For example

```
    copy rel (dom1 = c5, dummy = d2, dom2 = i4,
        dumb = d0) from "/mnt/me/data"
```

The first five characters are put in dom1, the next two characters are ignored. The next four bytes are an i4 (integer) and go in dom2, and the remaining delimited string is ignored. The name given to a dummy specifier is ignored.

As mentioned previously, dummy domains can be used on a copy "into" a Unix file for inserting special characters. The list of recognizable names includes:

nl	newline
tab	tab character
sp	space
nul	a zero byte
null	a zero byte
comma	,
dash	-
colon	:
lparen	(
rparen	)

3. Truncation

It is not uncommon to have a mistake occur and need to start over. The simplest way to do that is to "truncate" the relation. This is done by the command:

```
    modify relname to truncated
```

This has the effect of removing all tuples in relname, releasing all disk space, and making relname a heap again. It is the logical equivalent of a destroy followed by a create (but with a lot less typing).

Since formatting mistakes are possible with copy, it is not generally a good idea to copy data into a relation that already has valid data in it. The best procedure is to create a temporary relation with the same domains as the existing relation. Copy data into the temporary relation and then append it to the real relation. For example:

```
        create tempdom(name=c10,amount=f4,ext=i2)

        copy tempdom(name=c0,amount=c0,ext=c0)
        from "/mnt/bob/data"
        range of td is tempdom
        append to donation(td.all)
```

4. Specifing Delimitors.

Sometimes it is desirable to specify what the delimiting character should be on a copy "from" a file. This can be done by specifing:

```
        domain = c0delim
```

where "delim" is a valid delimitor taken from the list of recognizable names. This list was summarized on the previous page under "dummy domains". For example:

```
        copy donation (name = c0nl) from "/mnt/me/data"
```

will copy names from the file to the relation. Only a new line will delimit the names so any commas or tabs will be passed along as part of the name.

When copying "into" a Unix file, the "delim" is actually written into the file, so on a copy "into" the specification:

```
        copy donation (name = c0nl) into "/mnt/me/file"
```

will cause "name" to be written followed by a new line character.

4. CHOOSING THE BEST STORAGE STRUCTURES

We now turn to the issue of efficiency. Once you have created a relation and inserted your data using either copy or append, INGRES can process any query on the relation. There are several things you can do to improve the speed at which INGRES can process a query.

INGRES can store a relation in three different internal structures. These are called "heap", "isam", and "hash". First we will briefly describe each structure and then later expand our discussion.

HEAP

When a relation is first created, it is created as a "heap". There are two important properties about a heap: duplicate tuples are not removed, and nothing is known about the location of the tuples. If you ran the query:

```
     range of d is donation
     retrieve (d.amount) where d.name = "bill"
```

INGRES would have to read every tuple in the relation looking for those with name "bill". If the relation is small this isn't a serious matter. But if the relation is very large, this can take minutes (or even hours!).

HASH

A relation whose structure is "hash" can give fast access to searches on certain domains. (Those domains are usually referred to as "keyed domains".) In addition, a "hashed" relation contains no duplicate tuples. For example, suppose the donation relation is stored hashed on domain "name". Then the query:

```
     retrieve (d.amount) where d.name = "bill"
```

will run quickly since INGRES knows approximately where on disk the tuple is stored. If the relation contains only a few tuples you won't notice the difference between a "heap" and a "hash" structure. But as the relation becomes larger, the difference in speed becomes much more noticeable.

ISAM

An isam structure is one where the relation is sorted on one or more domains, (also called keyed domains). Duplicates are also removed on "isam relations". When new tuples are appended they are placed "approximately" in their sorted position in the relation. (The "approximately" will be explained a bit later.)

Suppose donation is isam on name. To process the query

```
     retrieve (d.amount) where d.name = "bill"
```

INGRES will determine where in the sorted order the name "bill" would be and read only those portions of the relation.

Since the relation is approximately sorted, an isam structure is also efficient for processing the query:

```
     retrieve (d.amount) where d.name >= "b" and d.name < "g"
```

This query would retrieve all names beginning with "b" through "f". The entire relation would not have to be searched since it is isam on name.

SPECIFYING THE STORAGE STRUCTURE

Any user created relation can be converted to any storage structure using the "modify" command. For example

```
     modify donation to hash on name
```

or

```
     modify donation to isam on name
```

or even

```
     modify donation to heap
```

PRIMARY AND OVERFLOW PAGES

At this point it is necessary to introduce the concepts of primary and overflow pages on hash and isam structures. Both hash and isam are techniques for assigning specific tuples to specific pages of a relation based on the tuple's keyed domains. Thus each page will contain only a certain specified subset of the relation.

When a new tuple is appended to a hash or isam relation, INGRES first determines what page it belongs to, and then looks for room on that page. If there is space then the tuple is placed on that page. If not, then an "overflow" page is created and the tuple is placed there.

The overflow page is linked to the original page. The original page is called the "primary" page. If the overflow page became full, then INGRES would connect an overflow page to it. We would then have one primary page linked to an overflow page, linked to another overflow page. Overflow pages are dynamically added as needed.

SPECIFYING FREE SPACE

The modify command also lets you specify how much room to leave for the relation to grow. As was mentiond in "create", relations are divided into pages. A "fillfactor" can be used to specify how full to make each primary page. This decision should be based only on whether more tuples will be appended to the relation. For example:

```
     modify donation to isam on name where fillfactor = 100
```

This tells modify to make each page 100% full if at all possible.

```
     modify donation to isam on name where fillfactor = 25
```

This will leave each page 25% full or, in other words, 75% empty. We would do this if we had roughly 1/4 of the data already loaded and it was fairly well distributed about the alphabet.

Keep in mind that if you don't specify the fillfactor, INGRES will typically default to a reasonable choice. Also when a page becomes full, INGRES automatically creates an "overflow" page so it is never the case that a relation will be unable to expand.

When modifying a relation to hash, an additional parameter "minpages" can be specified. Modify will guarantee that at least "minpage" primary pages will be allocated for the relation.

Modify computes how may primary pages will be needed to store the existing tuples at the specified fillfactor assuming that no overflow pages will be necessary originally. If that number is less than minpages, then minpages is used instead.

For example:

```
     modify donation to hash on name where fillfactor = 50,
          minpages = 1

     modify donation to hash on name where minpages = 150
```

In the first case we guarantee that no more pages than are necessary will be used for 50% occupancy. The second case is typically used for modifying an empty or near empty relation. If the approximate maximum size of the relation is known in advance, minpages can be used to guarantee that the relation will have its expected maximum size.

There is one other option available for hash called "maxpages". Its syntax is the same as minpages. It can be used to specify the maximum number of primary pages to use.

COMPRESSION

The three storage structures (heap, hash, isam) can optionally have "compression" applied to them. To do this, refer to the storage structures as cheap, chash, and cisam. Compression reduces the amount of space needed to store each tuple internally. The current compression technique is to suppress trailing blanks in character domains. Using compression will never require more space and typically it can save disk space and improve performance. Here is an example:

```
     modify donation to cisam on name where fillfactor = 100
```

This will make donation a compressed isam structure and fill every page as full as possible. With compression, each tuple can have a different compressed length. Thus the number of tuples that can fit on one page will depend on how successfully they can be compressed.

Compressed relations can be more expensive to update. In particular if a replace is done on one or more domains and the compressed tuple is no longer the same length, then INGRES must look for a new place to put the tuple.

TWO VARIATIONS ON A THEME

As mentioned, duplicates are not removed from a relation stored as a heap. Frequently it is desirable to remove duplicates and sort a heap relation. One way of doing this is to modify the relation to isam specifying the order in which to sort the relation. An alternative to this is to use either "heapsort" or "cheapsort". For example

```
     modify donation to heapsort on name, ext
```

This will sort the relation by name then ext. The tuples are further sorted on the remaining domains, in the order they were listed in the original create statement. So in this case the relation will be sorted on name then ext and then amount. Duplicate tuples are always removed. The relation will be left as a heap. Heapsort and cheapsort are intended for sorting a temporary relation before printing and destroying it. It is more efficient than modifying to isam because with isam INGRES creates a "directory" containing key information about each page. The relation will NOT be kept sorted when further updates occur.

Examples:

Here are a collection of examples and comments as to the efficiency of each query. The queries are based on the relations:
parts(pnum, pname, color, weight, qoh)
supply(snum, pnum, jnum, shipdate, quan)

```
range of p is parts
range of s is supply

modify parts to hash on pnum
modify supply to hash on snum,jnum

     retrieve (p.all) where p.pnum = 10
```

INGRES will recognize that parts is hashed on pnum and go directly to the page where parts with number 10 would be stored.

```
retrieve (p.all) where p.pname = "tape drive"
```

INGRES will read the entire relation looking for matching pnames.

```
retrieve (p.all) where p.pnum < 10  and p.pnum > 5
```

INGRES will read the entire relation because no exact value for pnum was given.

```
retrieve (s.shipdate) where s.snum = 471 and s.jnum = 1008
```

INGRES will recognize that supply is hashed on the combination of snum and jnum and will go directly to the correct page.

```
retrieve (s.shipdate) where s.snum = 471
```

INGRES will read the entire relation. Supply is hashed on the combination of snum and jnum. Unless INGRES is given a unique value for both, it cannot take advantage of the storage structure.

```
retrieve (p.pname, s.shipdate) where
p.pnum = s.pnum and s.snum = 471 and s.jnum = 1008
```

INGRES will take advantage of both storage structures. It will first find all s.pnum and s.shipdate where s.snum = 471 and s.jnum = 1008. After that it will look for all p.pname where p.pnum is equal to the correct value.

This example illustrates the idea that it is frequently a good idea to hash a relation on the domains where it is "joined" with another relation. For example, in this case it is very common to ask for p.pnum = s.pnum

To summarize:

To take advantage of a hash structure, INGRES needs an exact value for each key domain. An exact value is anything such as:

```
s.snum = 471
s.pnum = p.pnum
```

An exact value is not

```
s.snum >= 471
(s.snum = 10 or s.snum = 20)
```

Now let's consider some cases using isam

```
modify supply to isam on snum,shipdate
retrieve (s.all) where s.snum = 471
and s.shipdate > "75-12-31"
```

```
     and s.shipdate < "77-01-01"
```

Since supply is sorted first on snum and then on shipdate, INGRES can take full advantage of the isam structure to locate the portions of supply which satisfy the query.

```
     retrieve (s.all) where s.snum = 471
```

Unlike hash, an isam structure can still be used if only the first key is provided.

```
     retrieve (s.all) where s.snum > 400 and s.snum < 500
```

Again INGRES will take advantage of the structure.

```
     retrieve (s.all) where s.shipdate >= "75-12-31" and
     s.shipdate <= "77-01-01"
```

Here INGRES will read the entire relation. This is because the first key (snum) is not provided in the query.

To summarize:

Isam can provide improved access on either exact values or ranges of values. It is useful as long as at least the first key is provided.

To locate where the tuples are in an isam relation, INGRES searches the isam directory for that relation. When a relation is modified to isam, the tuples are first sorted and duplicates are removed. Next, the relation is filled (according to the fillfactor) starting at page 0, 1, 2... for as many pages as are needed.

Now the directory is built. The key domains from the first tuple on each page are collected and organized into a directory (stored in the relation on disk). The directory is never changed until the next time a modify is done.

Whenever a tuple is added to the relation, the directory is searched to find which page the new tuple belongs on. Within that page, the individual tuples are NOT kept sorted. This is what is meant by "approximately" sorted.

HEAP v. HASH v. ISAM

Let's now compare the relative advantages and disadvantages of each option. A relation is always created as a heap. A heap is the most efficient structure to use to initially fill a relation using copy or append.

Space from deleted tuples of a heap is only reused on the last page. No duplicate checking is done on a heap relation.

Hash is advantageous for locating tuples referenced in a qualification by an exact value. The primary page for tuples with a specific value can be easily computed.

Isam is useful for both exact values and ranges of values. Since the isam directory must be searched to locate tuples, it is never as efficient as hash.

OVERFLOW PAGES

When a tuple is to be inserted and there is no more room on the primary page of a relation, then an overflow page is created. As more tuples are inserted, additional overflow pages are added as needed. Overflow pages, while necessary, decrease the system performance for retrieves and updates.

For example, let's suppose that supply is hashed on snum and has 10 primary pages. Suppose the value snum = 3 falls on page 7. To find all snum = 3 requires INGRES to search primary page 7 and all overflow pages of page 7 (if any). As more overflow pages are added the time needed to search for snum = 3 will increase. Since duplicates are removed on isam and hash, this search must be performed on appends and replaces also.

When a hash or isam relation has too many overflow pages it should be remodified to hash or isam again. This will clear up the relation and eliminate as many overflow pages as possible.

UNIQUE KEYS

When choosing key domains for a relation it is desirable to have each set of key domains as unique as possible. For example, employee id numbers typically have no duplicate values, while something like color is likely to have only a few distinct values, and something like sex, to the best of our knowledge, has only two values.

If a relation is hashed on domain sex then you can expect to have all males on one primary page and all its overflow pages and a corresponding situation with females. With a hash relation there is no solution to this problem. A trade-off must be made between the most desirable key domains to use in a qualification versus the uniqueness of the key values.

Since isam structure can be used if at least the first key is provided, extra key domains can sometimes be added to increase uniqueness. For example, suppose the supply relation has only 10 unique supplier numbers but thousands of tuples. Choosing an isam structure with the keys snum and jnum will probably give many more unique keys. However, the directory size will be larger and consequently it will take longer to search. When providing additional keys just for the sake of increasing uniqueness, try to use the smallest possible domains.

SYSTEM RELATIONS

INGRES uses three relations ("relation", "attribute", and "indexes") to maintain and organize a data base. The "relation" relation has one tuple for each relation in the data base. The "attribute" relation has one tuple for each attribute in each relation. The "indexes" relation has one tuple for each secondary index.

INGRES accesses these relations in a very well defined manner. A program called "sysmod" should be used to modify these relations to hash on the appropriate domains. To use sysmod the data base administrator types

```
% sysmod data-base-name
```

Sysmod should be run initially after the data base is created and subsequently as relations are created and the data base grows. It is insufficient to run sysmod only once and forget about it. Rerunning sysmod will cause the system relations to be remodified. This will typically remove most overflow pages and improve system response time for everything.

5. SECONDARY INDICES

Using an isam or hash structure provides a fast way to find tuples in a relation given values for the key domains. Sometimes this is not enough. For example, suppose we have the donation relation

```
     donation(name, amount, ext)
```

hashed on name. This will provide fast access to queries where the qualification has an exact value for name. What if we also will be doing queries giving exact values for ext?

Donation can be hashed either on name or ext, so we would have to choose which is more common and hash donation on that domain. The other domain (say ext) can have a secondary index. A secondary index is a relation which contains each "ext" together with the exact location of where the tuple is in the relation donation.

The command to create a secondary index is:

```
     index on donation is donext (ext)
```

The general format is:

```
     index on relation_name is secondary_index_name (domains)
```

Here we are asking INGRES to create a secondary index on the relation donation. The domain being indexed is "ext". Indices are formed in three steps:

1. "Donext" is created as a heap.
2. For each tuple in donation, a tuple is inserted in "donext" with the value for ext and the exact location of the corresponding tuple in donation.
3. By default "donext" is modified to isam.

Now if you run the query

```
     range of d is donation
     retrieve(d.amount) where d.ext = 207
```

INGRES will automatically look first in "donext" to find ext = 207. When it finds one it then goes directly to the tuple in the donation relation. Since "donext" is isam on ext, search for ext = 207 can typically be done rapidly.

If you run the query

```
     retrieve(d.amount) where d.name = "frank"
```

then INGRES will continue to use the hash structure of the relation "donation" to locate the qualifying tuples.

Since secondary indices are themselves relations, they also can be either hash, isam, chash or cisam. It never makes sense to a secondary index a heap.

The decision as to what structure to make them on involves the same issues as were discussed before:

Will the domains be referenced by exact value?
Will they be referenced by ranges of value?
etc.

In this case the "ext" domain will be referenced by exact values, and since the relation is nearly full we will do:

```
     modify donext to hash on ext where fillfactor = 100
     and minpages = 1
```

Secondary indices provide a way for INGRES to access tuples based on domains that are not key domains. A relation can have any number of secondary indices and in addition each secondary index can be an index on up to six domains of the primary relation.

Whenever a tuple is replaced, deleted or appended to a primary relation, all secondary indices must also be updated. Thus secondary indices are "not free". They increase the cost of updating the primary relation, but can decrease the cost of finding tuples in the primary relation.

Whether a secondary index will improve performance or not strongly depends on the uniqueness of the values of the domains being indexed. The primary concern is whether searching through the secondary index is more efficient than simply reading the entire primary relation. In general it is if the number of tuples which satisfy the qualification is less than the number of total pages (both primary and overflow) in the primary relation.

For example if we frequently want to find all people who donated less than five dollars, consider creating

```
     index on donation is donamount (amount)
```

By default donamount will be isam on amount. IF INGRES processes the query:

```
     retrieve(d.name) where d.amount < 5.0
```

it will locate d.amount < 5.0 in the secondary index and for each tuple it finds will fetch the corresponding tuple in donation. The tuples in donamount are sorted by amount but the tuples in donation are not. Thus in general each tuple fetch from donation via donamount will be on a different page. Retrieval using the secondary index can then cause more page reads than simply reading all of donation sequentially! So in this example it would be a bad idea to create the secondary index.

6. RECOVERY AND DATA UPDATE

INGRES has been carefully designed to protect the integrity of a data base against certain classes of system failures. To do this INGRES processes changes to a relation using what we call "deferred update" or "batch file update". In addition there are two INGRES programs "restore" and "purge" that can be used to check out a data base after a system failure. We will first discuss how deferred updates are created and processed, and second we will discuss the use of purge and restore.

DEFERRED UPDATE (Batch update)

An append, replace or delete command is run in four steps:

1. An empty batch file is created.
2. The command is run to completion and each change to the result relation is written into the batch file.
3. The batch file is read and the relation and its secondary indices (if any) are actually updated.
4. The batch file is destroyed and INGRES returns back to the user.

Deferred update defers all actual updating until the very end of the query. There are three advantages to doing this.

1. Provides recovery from system failures

If the system "crashes" during an update, the INGRES recovery program will decide to either run the update to completion or else "back out" the update, leaving the relation as it looked before the update was started.

2. Prevents infinite queries

If "donation" were a heap and the query

```
    range of d is donation
    append to donation(d.all)
```

were run without deferred update, it would terminate only when it ran out of space on disk! This is because INGRES would start reading the relation from the beginning and appending each tuple at the end. It would soon start reading the tuples it had just previously appended and continue indefinitely to "chase its tail".

While this query is certainly not typical, it illustrates the point. There are certain classes of queries where problems occur if WHEN an update actually occurs is not precisely defined. With deferred update we can guarantee consistent and logical results.

3. Speeds up processing of secondary indices

Secondary indices can be updated faster if they are done one at a time instead of all at once. It also insures protection against the secondary index becoming inconsistent with its primary relation.

TURNING DEFERRED UPDATE OFF

If you are not persuaded by any of these arguments, INGRES allows you to turn deferred update off! Indeed there are certain cases when it is appropriate (although certainly not essential) to perform updates directly, that is, the relation is updated while the query is being processed.

To use direct update, you must be given permission by the INGRES super user. Then when invoking INGRES specify the "-b" flag which turns off batch update.

```
    % ingres mydate -b
```

INGRES will use direct update on any relation without secondary indices. It will still silently use deferred update if a relation has any secondary indices. By using the "-b" flag you are sacrificing points 1 and 2 above. In most cases you SHOULD NOT use the -b flag.

If you are using INGRES to interactively enter or change one tuple at a time, it is slightly more efficient to have deferred update turned off. If the system crashes during an update the person entering the data will be aware of the situation and can check whether the tuple was updated or not.

RESTORE

INGRES is designed to recover from the common types of system crashes which leave the Unix file system intact. It can recover from updates, creates, destroys, modifies and index commands.

INGRES is designed to "fail safe". If any inconsistancies are discovered or any failures are returned from Unix, INGRES will generate a system error message (SYSERR) and exit.

Whenever Unix crashes while INGRES is running or whenever an INGRES syserr occurs, it is generally a good idea to have the date base administrator run the command

```
    % restore data_base_name
```

The restore program performs the following functions:

1. Looks for batch update files. If any are found, it examines each one to see if it is complete. If the system crash occured while the batch file was being read and the data base being updated, then restore will complete the update. Otherwise the batch file was not completed and it is simply destroyed; the effect is as though the query had never been run.

2. Checks for uncompleted modify commands. This step is crucial. It guarantees that you will either have the relation as it existed before the modify, or restore will complete the modify command. Modify works by creating a new copy of the relation in the new structure. Then when it is ready to replace the old relation, it stores the new information in a "modify batch file". This enables restore to determine the state of uncompleted modifies.

3. Checks consistency of system relations. This check is used to complete "destory" commands, back out "create" commands, and back out or complete "index" commands that were interrupted by a system crash.

4. Purges temporary relations and files. Restore executes the "purge" program to remove temporary relations and temporary files created by the system. Purge will be discussed in more detail a bit later.

Restore cannot tell the user which queries have run and which have not. It can only identify those queries which were in the process of being run when the crash occured. When batching queries together, it is a good idea to save the output in a file. By having the monitor print out each query or set of queries, the user can later identify which queries were run.

Restore has several options to increase its usability. They are specified by "flags". The options include:

-a	ask before doing anything
-f	passed to purge. used to remove temporary files.
-p	passed to purge. used to destory expired relations.
no database	restores all data bases for which you are the dba.

Of these options the "-a" is the most important. It can happen that a Unix crash can cause a page of the system catalogues to be incorrect. This might cause restore to destory a relation. In fact, you might want to "patch" the system relations to correct the problem. No restore program can account for all possibilities. It is therefore no replacement (fortunately) for a human.

If "-a" is specified, restore will state what it wants to do and then ask for permission. It reads standard input and accepts "y" to mean go ahead and anything else to mean no. For example, to have restore ask you before doing anything

```
restore -a mydatabase
```

To have it take "no" for all its questions

```
restore -a mydatabase </dev/null
```

Using the -a flag, restore might ask for permission to perform some cleanup; for example, if it finds an attribute for which there is no corresponding relation, or if it finds a secondary index for which there is no primary relation, etc.

To date, we have never had a system crash which INGRES could not recover from. This does not mean that it will never happen, but rather that it shouldn't be too great a concern for you. It should be mentioned that restore is not a substitution for doing periodic backing up, nor does it ever perform such a function.

PURGE

Purge can be used to report expired relations, destroy temporary system relations, remove extraneous files, and destory expired relations. To use purge you must be the DBA for the data base.

```
% purge mydatabase
```

Purge has several options which are specified by flags which are worth noting:

-f (default is off) remove all extraneous files. Each file is reported and then removed. If "-f" is not specified then the file is only reported.

-p (default is off) destroy all expired relations. Each expired relation is reported and if "-p" was specified the relation is destroyed.

Purge always destroys relations and files which are known to be INGRES system temporaries. When processing multi-variable queries and queries with aggregate functions, INGRES will usually create temporary relations with intermediate results. These relations always begin with the characters "_SYS". Other INGRES commands create temporary files which also begin with "_SYS". Under normal processing they are always destroyed. If a system crash occurs, they might be left. Purge will always clean up the temporary system files. It cleans up the user relations only when specifically asked to.