GETTING STARTED IN INGRES - A TUTORIAL

by

M. Stonebraker

Memorandum No. ERL-M518

23 April 1975

GETTING STARTED IN INGRES - A TUTORIAL

by

Michael Stonebraker

Memorandum No. ERL-M518

23 April 1975

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Research sponsored by the National Science Foundation Grant GK-43024x, U.S. Army Research Office -- Durham Contract DAHC04-74-G0087, the Naval Electronic Systems Command Contract N00039-75-C-0034, and a Grant from the Sloan Foundation.

GETTING STARTED IN INGRES
A
TUTORIAL

by

MICHAEL STONEBRAKER

This document contains an introduction to the data base management system, INGRES, and in particular stresses its user language QUEL. It is meant to be read while interacting with the INGRES system at a terminal.

One's first encounter with INGRES is to type the UNIX shell command

```
ingres data-base-name
```

which has the effect of turning you over to INGRES for subsequent interactions. The actual sequence is:

```
%ingres demo
```

Here, we have entered INGRES and specified that we are interested in the data base "demo" which will have in it the relations of interest to this document. After a momentary pause the following will be returned to your terminal.

```
INGRES vers 2.3 login
Sat Mar 15 14:16:04 1975
Set operators, Aggregate functions and X.ALL are not yet implemented.
go
```

The first three lines constitute the current "dayfile" which gives relevant information on the status of INGRES. The statement "go" indicates INGRES is waiting for your input.

Now type

```
help
\g
```

"help" is an INGRES command which can deluge you with information

about the system. In this case, you will receive the page from the INGRES reference manual which describes the help command. "\g" is a statement to INGRES to execute the "help" command without waiting for additional input from the terminal. The response from INGRES is:

```
query formulation complete

     HELP(X)                        4/22/75                        HELP(X)


     NAME
          help - get information about how to use INGRES

     SYNOPSIS
          HELP ["item-in-question"]

     DESCRIPTION
          HELP may be used to obtain information about any section of
          this manual, the content of the current data base, or a
          specific relation in the data base, depending on the item-
          in-question.  Omission of that argument is functionally
          equivalent to HELP "help" .  The other legal forms are as
          follow:

          HELP "section" - Produces a copy of the specified section of
               the INGRES Programmer's Manual, and prints it on the
               standard output device.

          HELP "" -  Gives information about all relations that exist
               in the current database.

          HELP "relname" - Gives information about the specified rela-
               tion, but in greater detail than would HELP "" .

     EXAMPLE
          HELP
          HELP "quel"
          HELP ""
          HELP "emp"

     SEE ALSO

     DIAGNOSTICS
          Unknown name - The item-in-question could not be recognized.

     BUGS
          Alphabetics appearing within the item-in-question must be
```

lower-case to be recognized.

```
continue
```

The final line contains the word "continue". This indicates INGRES is ready to listen to you again.

At this point it is important for you to realize that INGRES maintains a workspace in which you formulate your interactions. This workspace is desirable so that you can correct spelling errors and other mistakes which you may from time to time make without having to type in your entire interaction again.

At the present time your workspace contains

```
help
```

If you type in "\g" once more, INGRES will simple execute your workspace which will give you a second printout of what you have just seen above.

In order to clear out our workspace we use the command "\r"as follows:

```
\r
go
```

Our workspace now is empty. It is still possible to type in "\g" as follows. However, it has no effect.

```
\g
query formulation complete
continue
```

We will now try to exercise the "retrieve" command and will do so on the data that now follows. To print the contents of any relation (or table if you are more confortable with that terminology), simply type:

```
print relation-name
```

If we type help"" we can obtain a list of relations in the data base demo. One relation from this list is called "parts". We can print this relation as follows:

```
print parts
\g


query formulation complete

parts relation

|pnum  |pname                |color   |weight|qoh   |
|------------------------------------------------|
|    1|central processor    |pink    |    10|     1|
|    2|memory               |gray    |    20|    32|
|    3|disk drive           |black   |   685|     2|
|    4|tape drive           |black   |   450|     4|
|    5|tapes                |gray    |     1|   250|
|    6|line printer         |yellow  |   578|     3|
|    7|l-p paper            |white   |   151|    95|
|    8|terminals            |blue    |    19|    15|
|   13|paper tape reader    |black   |   107|     0|
|   14|paper tape punch     |black   |   147|     0|
|    9|terminal paper       |white   |     2|   350|
|   10|byte-soap            |clear   |     0|   143|
|   11|card reader          |gray    |   327|     0|
|   12|card punch           |gray    |   427|     0|
continue
```

Notice that the "parts" relation has information about the components in a hypothetical computer installation. Each row of this table (or tuple in this relation) contains information on a given part including its part number, its part name, its color, its weight, and the quantity that are on hand.

Using a "retrieve" command we will be able to obtain portions of this table which are of interest to us. (There is almost no limit on how large the tables can be; these examples are done on small ones so that this tutorial does not become too large. In fact, the actual limit on the size of a table is approximately 30,000,000 bytes for those who are interested.)

To obtain information, we must first tell INGRES what table it is that we wish to interogate. One way to do this might be the command

```
I WANT TO TALK ABOUT parts
```

Although this is natural to the beginner, INGRES makes you do something slightly more complicated. This added complexity is necessary so that one does not get into trouble with more complicated interactions.

The statement required in INGRES is

range of variable-name is relation-name

The variable-name is indicated to be a surrogate for the relation name specified. We can declare p to be this surrogate for "parts" as follows:

```
\r
go
range of p is parts
```

Notice that we first cleared our workspace so that the whole parts relation would not be printed again.

Now, we can add a "retrieve" command which can be the following

```
retrieve p.pname
```

The interpretation is that we wish to obtain the pname column of the relation specified by the variable "p".

In order to ensure that we have typed our interaction correctly we may use the special command "\p". This will simply print the contents of our workspace as follows:

```
\p
range of p is parts
retrieve p.pname
```

Since it appears to be a correct query we can execute it by the "\g" command as follows:

```
\g
query formulation complete
PERIOD = '.' : line 3, syntax error
continue
```

Unfortunately, we have made a syntax error. What is more unfortunate is that INGRES is not always overly helpful in showing us what it is.

The problem with this interaction is an arbitrary convention in INGRES that whatever you wish to retrieve must be enclosed in "( )" . We will correct our mistake by retyping the query as follows:

```
\r
go
range of p is parts
retrieve (p.pname)
\g
query formulation complete
|pname                |
|---------------------|
|central processor    |
|memory               |
|disk drive           |
|tape drive           |
|tapes                |
|line printer         |
|l-p paper            |
|terminals            |
|paper tape reader    |
|paper tape punch     |
|terminal paper       |
|byte-soap            |
|card reader          |
|card punch           |
continue
```

Everything has now worked out all right and we have obtained the column of the parts table which contains the names of the parts.

We can retrieve more than one column at once by simply indicating a sequence of column names separated by a comma. Hence we could obtain part names and colors as follows.

```
\r
go
range of p is parts
retrieve (p.pname, p.color)
\g
query formulation complete
|pname                |color   |
|---------------------|--------|
|central processor    |pink    |
|memory               |gray    |
|disk drive           |black   |
|tape drive           |black   |
|tapes                |gray    |
|line printer         |yellow  |
|l-p paper            |white   |
|terminals            |blue    |
```

```
|paper tape reader    |black    |
|paper tape punch     |black    |
|terminal paper       |white    |
|byte-soap            |clear    |
|card reader          |gray     |
|card punch           |gray     |
continue
```

Notice in the printout each column contains the name of the column so we do not get confused. Sometimes we require more complex results than simply the names of columns. Suppose, for example, we require the computation "1000-qoh". In other words, we wish to know for each part how many less than 1000 we possess. This can be stated as follows:

```
\r
go
range of p is parts
retrieve (p.pname, computation=1000-p.qoh)
\g
query formulation complete
|pname                 |comput|
|----------------------|
|central processor     |   999|
|memory                |   968|
|disk drive            |   998|
|tape drive            |   996|
|tapes                 |   750|
|line printer          |   997|
|l-p paper             |   905|
|terminals             |   985|
|paper tape reader     |  1000|
|paper tape punch      |  1000|
|terminal paper        |   650|
|byte-soap             |   857|
|card reader           |  1000|
|card punch            |  1000|
continue
```

Note that the heading on our printout is the first six characters of the name "computation" which we have given to the computed quantity "1000-p.qoh".

In order for INGRES to accept computed quantities you must always give them a name. This is simply done by picking a name and putting it to the left of an equals sign in the retrieval.

Note also that the presence or absence of blanks makes no difference in between the "( )".

It is important that you spell correctly any column names which you use in an interaction, since INGRES has no spelling correcter at the present time.

Note lastly that you need not put interactions on three lines as we have been doing. The following works equally well.

```
\r
go
range of p is parts retrieve (p.pname, computation = 1000-p.qoh) \g
query formulation complete
|pname                    |comput|
|-------------------------|
|central processor        |   999|
|memory                   |   968|
|disk drive               |   998|
|tape drive               |   996|
|tapes                    |   750|
|line printer             |   997|
|l-p paper                |   905|
|terminals                |   985|
|paper tape reader        |  1000|
|paper tape punch         |  1000|
|terminal paper           |   650|
|byte-soap                |   857|
|card reader              |  1000|
|card punch               |  1000|
continue
```

It is usually wise to space your interactions so they are as readable as possible.

So far we have produced interactions which give us columns of the "parts" relation. We now indicate how to obtain only portions of columns. The basic mechanism is a "where" clause which can be added onto the end of the interactions we have been doing. If we wanted the previous query only performed for those parts whose color is pink we would do the following:

```
\r
go
range of p is parts
retrieve (p.pname, computation=1000-p.qoh) where p.color = "pink"
\g
query formulation complete
```

```
|pname                |comput|
|---------------------|
|central processor    |   999|
continue
```

The "where" clause limits the number of rows which are examined to only those which satisfy the qualification given i.e. to those which satisfy "p.color="pink". Only the central procesor has this property so it is the only entry in the output.

We are now to the point where we are typing enough information so that errors in typing are likely. It is very annoying to have to reset the workspace and try again every time an error is encountered. Two mechanisms are supported in INGRES to help with this problem.

1) INGRES accepts the symbol # to mean "backspace". Consequently, one can simply backspace and retype errors which occur. One can backspace as many times as one wishes; INGRES will continue to back up until it reaches the beginning of the current line. Subsequent backspaces will have no effect. If a line has become so garbled that the user wishes to simply erase it and start typing again one can use the symbol @ which means "erase the whole line"

2) More complicated corrections are often necessary than can be done easily using mechanism 1). These are supported by calling on the features of the UNIX program called the editor. A tutorial on the editor is available in the UNIX programmer's manual. Here, we will simply discuss two features of this program. Since it is a very powerful program, the serious INGRES programmer would be wise to study that tutorial in more detail than the few exerpts we present here.

Suppose we type in an incorrect query as follows:

```
\r
go
ranhe of p is perts
retrieve p.pname
where p.pcolor = "pink="
```

This query has many errors and we might do better to start over, but for the exercise we will use the editor which we obtain by typing \e as follows:

```
\e
>>ed
```

The statement ">>ed" says now we are in the hands of the UNIX editor and our workspace has been sent to it.

We can sequence through our program by typing a line number followed by a carriage return i.e.

```
1
ranhe of p is perts
2
retrieve p.pname
3
where p.pcolor = "pink="
1
ranhe of p is perts
2
retrieve p.pname
3
where p.pcolor = "pink="
```

We have now looked at each line twice and are ready to fix each one.

We do this with a substitute command. This has the form:

```
s/this character string/that character string/
```

The editor goes through the current line of our command and finds the first instance of "this character string" and replaces it with "that character string". In this way we can find offending portions of our interaction and fix them.

First we do it for line 1.

```
1
ranhe of p is perts
s/ranhe/range/
s/perts/parts/
1
range of p is parts
```

After two substitutions, everything is fine.

Notice that you only need to specify enough of "this character string" so that the editor can correctly make the substitution.

Also, if you simply put a "p" after the last "/" , the current line will be automatically printed.

Notice lastly, that # and @ work the same way in the editor as in INGRES.

We now proceed to fix the rest of our statement without further comments.

```
2
retrieve p.pname
s/p/(p/
s/me/me)/p
retrieve (p.pname)

3
where p.pcolor = "pink="
s/pc/c/
s/k=/k/p
where p.color = "pink"
```

We have now fixed all lines and use the command "w" to send the corrected statement back to INGRES as follows:

```
w
```

We now issue a "q" command to quit the editor and return to INGRES as follows:

```
q
<<monitor
```

The echo "<<monitor" is to remind you that you have returned to INGRES.

It is usuallly wise to make sure INGRES got your corrected interaction back from the editor correctly by typing "\p" i.e.

```
\p
range of p is parts
retrieve (p.pname)
where p.color = "pink"
```

A "\g" will now execute the corrected command.

```
\g
query formulation complete
|pname                |
|---------------------|
|central processor    |
```

```
continue
```

We now indicate the boolean opperators which may be used. For example, the interaction that follows is accepted by INGRES.

```
\r
go
range of p is parts
retrieve (p.pname)
where p.color ="pink" or p.color = "gray"
\g

query formulation complete
|pname                |
|---------------------|
|central processor    |
|memory               |
|tapes                |
|card reader          |
|card punch           |
continue
```

The operators "not", "and" and "or" are supported in INGRES. Users may simply use the operators remembering only to put a space on either side of them. It is sometimes essential to remember that the precedence of boolean operators is "not" then "and" then "or". Users who wish to alter this precedence (or who do not remember it) may use parentheses to precisely specify their meaning. The following interaction gives an example of multiple boolean operators.

```
\r
go
range of p is parts
retrieve (p.pname)
where (p.color="pink"  or  p.color = "gray") and p.pnum < 10
\g
query formulation complete
|pname                |
|---------------------|
|central processor    |
|memory               |
|tapes                |
continue
```

Three points should be carefully noted about the above interaction:

1) Character strings must be enclosed in double quote marks while numbers may be typed with no special delimiters.

2) Note the arithmetic operator "<" in the above interaction. Valid relational operators include:

```
    =  (equals to)
    <  (less than)
    >  (greater than)
    <= (less than or equal to)
    >= (greater than or equal to)
    != (not equal to)
```

3) There is no limit to the complexity of the expressions which can be constructed using relational and boolean expressions.

We now do one last example concerning arithmetic operators in QUEL. This example finds the total weight (weight times qoh) for each part with a part number less than 10.

```
\r
go
range of p is parts
retrieve (p.pname, tot= p.weight*p.qoh)
where p.pnum < 10
\g
query formulation complete
|pname                |tot   |
|-----------------------|
|central processor    |    10|
|memory               |   640|
|disk drive           |  1370|
|tape drive           |  1800|
|tapes                |   250|
|line printer         |  1734|
|l-p paper            |  1425|
|terminals            |   285|
|terminal paper       |   700|
continue
```

It should be noted that arithmetic operators can be used in the qualification portion of an interaction as well as in the portion indicating the desired information. Valid arithmetic operators include:

```
  +  (addition)
  -  (subtraction)
```

```
  *  (multiplication)
  /  (floating point division)
  ** (exponentiation)
  mod (integer division)
```

It should also be noted that any user can save any result of an interaction by simply specifying the name of a relation into which the answer should be placed. The following suggests an equivalent way of obtaining the previous result. First a relation is created with the answer then the print command is used to display the result.

```
\r
go
range of p is parts
retrieve into local(p.pname, tot=p.weight*p.qoh)
where p.pnum < 10
\g
query formulation complete
continue
\r
go
print local
\g
query formulation complete

local relation

|pname                |tot    |
|---------------------------|
|central processor    |    10|
|memory               |   640|
|disk drive           |  1370|
|tape drive           |  1800|
|tapes                |   250|
|line printer         |  1734|
|l-p paper            |  1425|
|terminals            |   285|
|terminal paper       |   700|
continue
```

Notice that local remains as a relation in the data base and may be used in any future interactions by simply declaring a range variable for it.

We turn now to interactions which involve more than one relation at a time. It is in these interactions that QUEL is especially useful because of its ability to connect information in different relations.

First we print a second relation that will be used in the sequel.

```
\r
go
print supply\g
query formulation complete

supply relation
```

snum	pnum	jnum	shipdate	quan
475	1	1001	73-12-31	1
475	2	1002	74-05-31	32
8	1	1003	74-12-31	1
8	1	1004	75-01-15	1
475	3	1001	73-12-31	2
475	4	1002	74-05-31	1
8	2	1003	74-12-29	128
8	2	1004	75-01-15	256
122	7	1003	75-02-01	144
122	7	1004	75-02-01	48
122	9	1004	75-02-01	144
440	6	1001	74-10-10	2
131	8	1004	74-11-22	4
241	4	1001	73-12-31	1
62	3	1002	74-06-18	3
475	2	1001	73-12-31	32
475	1	1002	74-07-01	1
8	6	1003	74-12-25	2
8	6	1004	75-02-01	4
5	4	1003	74-11-15	3
5	4	1004	75-01-22	6
20	5	1001	75-01-10	20
20	5	1002	75-01-10	75
41	5	1003	75-01-02	50
9	5	1004	75-02-05	400
241	1	1005	75-06-01	1
241	2	1005	75-06-01	32
241	3	1005	75-06-01	1
67	5	1005	75-07-31	20
67	4	1005	75-07-01	1
999	1	1006	76-01-01	1
999	2	1006	76-01-01	32
999	3	1006	76-01-01	1
999	4	1006	76-01-01	1
999	5	1006	76-01-01	20
999	6	1006	76-01-01	1
999	7	1006	76-01-01	10
999	8	1006	76-01-01	1

```
|    999|      9|   1006|76-01-01|    100|
|    999|     10|   1006|76-01-01|    144|
|    999|     11|   1006|76-01-01|      1|
|    999|     12|   1006|76-01-01|      1|
|    999|     13|   1006|76-01-01|      1|
|    999|     14|   1006|76-01-01|      1|
|    241|      8|   1005|75-07-01|      1|
|    241|      9|   1005|75-07-01|    144|
|    131|      8|   1001|75-03-15|      2|
|    131|      8|   1002|75-03-15|      1|
|    131|      9|   1001|75-04-31|    200|
|    131|      9|   1002|75-03-31|    100|
|      8|     11|   1004|75-01-01|      2|
|      8|     12|   1004|75-04-31|      3|
|      8|     11|   1007|76-02-01|      3|
|      8|     12|   1007|76-02-01|      2|
|      8|      8|   1004|74-12-20|      5|
|      8|      9|   1004|74-12-31|    500|
|      8|      1|   1007|76-02-01|      1|
|      8|      2|   1007|76-02-01|   1024|
continue
```

This relation gives information on conditions under which the hypothetical computer installation can buy more parts. It indicates the supplier number from whom each part is available, the quantity in which it can be ordered, the date such an order could be shipped and the job number to which such an order could be charged. Notice that the column pnum appears in both the parts relation and this relation. Using this information we can "connect" the two relations. For example, we might want to know the supplier numbers of suppliers who sell central processors.

One way to proceed is to interrogate the parts relation to find the part number of central processors as follows:

```
\r
go
range of p is parts
retrieve (p.pnum) where p.pname = "central processor"
\g
```

The answer returned is:

```
query formulation complete
|pnum  |
|------|
|     1|
continue
```

Hence, part number 1 is the central processor. Then we could interrogate the supply relation seeking the suppliers of part number 1 as follows:

```
\r
go
range of s is supply
retrieve (s.snum) where s.pnum =1
\g
query formulation complete
|snum  |
|------|
|   475|
|     8|
|     8|
|   475|
|   241|
|   999|
|     8|
continue
```

Notice that suppliers 8,241, 475 and 999 supply central processors.

Notice also that suppliers 8 and 475 are repeated more than once. Because of the internal way that INGRES is organized, much faster response time can be supported if the "answer" is printed on the terminal with duplicate values sometimes present. In this case, the user must look at the response and note the duplications. On the other hand, should the user wish the system to detect and delete the duplicates, the user need only retrieve his answer into a temporary relation and then print that relation. The instructions are the following:

```
\r
go
range of s is supply
retrieve into cpu(s.snum) where s.pnum = 1
print cpu
\g
```

cpu relation

```
|snum  |
|------|
|   475|
|     8|
```

```
|   241|
|   999|
continue
```

In any case, it is rather inconvenient to have to issue two retrieve commands to get the information we require.

What is even more inconvenient is the necessity of obtaining the first output, namely the number 1, and then manually substituting this into the second query. It would have been extremely inconvenient if the central processor had had several part numbers; we would have had to substitute them all.

The following indicates one way around this inconvenience.

```
\r
go
range of p is parts
retrieve into cpu(p.pnum) where p.pname = "central processor"
range of c is cpu
range of s is supply
retrieve (s.snum) where s.pnum =c.pnum
\g
```

Here, we have executed the first half of the query as before obtaining in cpu the answer "1". Then the second half of the query is executed with a variable declared over the cpu relation. In the second retrieve statement the c.pnum simply has the value "1" and the statement should work correctly.

Unfortunately, we get the following response:

```
 In the CREATE of "cpu          " a duplicate relation name
 "cpu          " caused execution to halt.
```

INGRES takes the attitude that it should warn you when you are about to destroy information in a relation by putting new information in it. Hence, it will not let you execute the above statement until you either:

1) destroy cpu (which was created earlier) indicating you do not need the old information any more or

2) change the name of the cpu relation in the interaction so it does not conflict with a relation that exists.

We take the latter course and change cpu to cpunum by entering

the editor and using the substitute command. When we return to INGRES we should have the following:

```
range of p is parts
retrieve into cpunum(p.pnum) where p.pname = "central processor"
range of c is cpunum
range of s is supply
retrieve (s.snum) where s.pnum =c.pnum
\g
```

A more precise way to think about queries with more than one variable is the following. We will indicate a conceptual way that INGRES MIGHT process such a query in a step by step fashion.

We deal with the second half of the above query namely

```
range of s is supply
range of c is newcpu
retrieve (s.snum) where s.pnum=c.pnum
```

The first step of processing this query might be:

```
\r
go
range of s is supply
range of c is cpunum
retrieve into partanswer(snum=s.snum, spnum=s.pnum,cpnum=c.pnum)
print partanswer
\g
```

The relation partanswer contains one row for each and every possible pair of rows in supply and newcpu. The printout is the following. Examine it carefully so you understand what is happening.

```
query formulation complete

partanswer relation

|snum  |spnum |cpnum |
|--------------------|
|   475|     1|     1|
|   475|     2|     1|
|     8|     1|     1|
|     8|     1|     1|
|   475|     3|     1|
|   475|     4|     1|
```

8	2	1
8	2	1
122	7	1
122	7	1
122	9	1
440	6	1
131	8	1
241	4	1
62	3	1
475	2	1
475	1	1
8	6	1
8	6	1
5	4	1
5	4	1
20	5	1
20	5	1
41	5	1
9	5	1
241	1	1
241	2	1
241	3	1
67	5	1
67	4	1
999	1	1
999	2	1
999	3	1
999	4	1
999	5	1
999	6	1
999	7	1
999	8	1
999	9	1
999	10	1
999	11	1
999	12	1
999	13	1
999	14	1
241	8	1
241	9	1
131	8	1
131	8	1
131	9	1
131	9	1
8	11	1
8	12	1
8	11	1
8	12	1
8	8	1

```
|      8|      9|      1|
|      8|      1|      1|
|      8|      2|      1|
continue
```

The second portion of the processing of this query now involves the partanswer relation. Notice that the original qualification statement

```
s.snum=c.pnum
```

which involved two different relations (cpunum and supply) can be stated using only the partanswer relation as follows:

```
\r
go
range of a is partanswer
retrieve (a.snum) where a.spnum=a.cpnum
\g
```

The response to this interaction is the correct answer as follows:

```
|snum  |
|------|
|   475|
|     8|
|     8|
|   475|
|   241|
|   999|
|     8|
continue
```

Notice what has been printed is each row of the partanswer relation that has identical entries in its second and third columns.

Whenever you are in doubt concerning the meaning of a query with more than one variable in it, always think of the two step process described above and you will not go wrong. With this in mind, convince yourself that the correct answer to our interaction above can also be found using the following code.

```
\r
go
range of s is supply
range of p is parts
retrieve (s.snum) where s.pnum=p.pnum  and p.pname="central processor"
\g
```

So far in this document we have considered how to retrieve portions of a relation (or relations) that are of interest. The examples have indicated the power of QUEL for retrieval purposes. The only feature which has not yet been considered is aggregation.

We now illustrate the use of this construct in two examples. The following command finds the number of part names from the parts relation which are black.

```
\r
go
range of p is parts
retrieve (total= count(p.pname where p.color = "black"))
\g
query formulation complete
|total |
|------|
|     4|
continue
```

The next command finds the sum of quantities of part number 6 able to be supplied before October 1, 1976.

```
\r
go
range of s is supply
retrieve (s = sum(s.quan where s.pnum=6 and s.shipdate<"76-10-1"))
\g
query formulation complete
|s     |
|------|
|     9|
continue
```

The following points should be noted about aggregates:

a) aggregates have the form
agg-op(target-list where qualification).

agg-op can be
- min
- max
- count
- sum
- avg (sum/count)

The target list is the quantity for which the aggregate is desired using those tuples which satisfy the qualification.

b) There is no limit on the number of variables which can appear in an aggregate.

c) Aggregates can be nested, i.e. the target list and qualification may themselves contain aggregates.

d) The "QUEL" section of the reference manual indicates certain illegal aggregations. For example, avg is only allowed for quantities which are numeric. An attempt to find the average of a quantity that is alphanumeric (for example pname) will result in an error.

e) An aggregate can appear anywhere in a QUEL interaction.

We now turn to the other features of QUEL.

First, a user may put comments anywhere in his QUEL statements in order to make them more readable. This feature is especially useful when interactions are saved and reexecuted at a later time.

INGRES considers any text string bounded by "/*" on the front and "*/" on the rear to be a comment. It simply deletes the comment during processing as illustrated below.

```
\r
go
range of s is supply
/* This is a comment to indicate the format acccepted by QUEL for
comments*/
retrieve (s.snum) where s.pnum = 1
\g
```

Another command which proves useful is the exit command which is "\q", i.e.

```
\r
go
\q
```

This command will type a friendly greeting on your terminal and return you to the care of UNIX for any further processing you may wish to do. The current greeting is the following:

```
query formulation complete
INGRES vers 2.3 logout
Tue Mar 18 13:39:01 1975
goodbye - come again
```

The only other way to "bail out" of INGRES is to hit the "del" key. This should only be used in emergency (for example to abort a printout which is much too long). It has the effect of returning you directly to UNIX.

The other command which you should know about at this time is the destroy command. It has the following syntax:

```
\r
go
destroy cpu
\g
```

It "wipes away" the cpu relation entirely. It should be used when you are finished with the information in a relation or when you want to reuse the name of a relation for new information. The only response from INGRES which you receive is:

```
query formaulation complete
continue
```

We will now discuss the three update commands that are available in QUEL; respectively delete, replace and append.

The delete command is especially simple and has the following format:

```
delete variable-name where qualification
```

The following illustrates the effect of a delete statement.

```
\r
go
range of s is supply
delete s where s.snum = 8
\g
```

All that INGRES will echo is:

```
query formulation complete
continue
```

The effect of a delete statement is that all rows of supply are found which satisfy the qualification "s.snum =8" and instead of being returned to the user's terminal are instead deleted.

To convince yourself that this is indeed the case try printing the supply relation.

The qualification of a delete statement may be as complicated as it can be for retrieve statements. Therefore, it is a simple matter to delete from the supply relation the rows corresponding to those suppliers who supply the part called "central processor". Try to formulate this delete statement and convince yourself that it worked correctly.

Unfortunately, there is currently no facility in INGRES for the rows which are getting deleted to be echoed on the user's terminal.

Also, you may only delete rows from ONE relation at a time using the delete command. Therefore, only one variable can appear between the delete command and the "where" statement. There are several reasons for enforcing this restriction which are beyond the scope of this manual.

Note finally that a delete statement which has no "where" statement is allowed. It has the effect of deleting all the rows in a relation. What remains is a perfectly legal relation which has nothing in it.

We turn now to the effect of replace commands. They have the following general format:

```
replace variable-name(column-name = result,...,column-name = result)
      where qualification
```

Before formally explaining this command we do some examples. First, we will change supplier number 475 to 495 in the supply relation as follows:

```
\r
go
range of s is supply
replace s(snum=495) where s.snum = 475
\g
```

Again all that is echoed is:

```
query formulation complete
```

```
continue
```

You must again print the supply relation if you do not beleive that INGRES did what you wanted. We will now change the supplier number to 400 of all suppliers who supply the part "central processor" as follows:

```
\r
go
range of s is supply
range of p is parts
replace s(snum=400) where s.pnum=p.pnum and
                          p.pname="central processor"
\g
```

Again the only echo is an indication of completion of the command.

More formally, one can think about replace statements in the following way.

1) All the rows in the relation specified by the variable directly after the "replace" are found which satisfy the qualification. (In this last example it will be those rows which have s.pnum=1).

2) For all such rows, the information inside the parentheses is examined and whatever is on the left of each equals sign is replaced by whatever is on the right of it.

The following points should be noted concerning replace statements:

a) INGRES echos only a "continue" or any error messages which may be present in the command.

b) The qualification may involve any number of variables and may be as complex as desired.

c) The quantity on the right of any equals sign may be any computation possible in a retrieve statement.

d) The equals sign may be replaced by any of the words, "is", "by".

e) there is no requirement that any of the rows be changed by a replace statement; if no rows qualify, then none are changed.

f) It may happen that you try to replace a data item in a relation by more than one value. This represents a situation of "non

functionality". The issue of non functionality will not be pursued further in this manual.

We now turn to the issue of getting new information into INGRES. There are two mechanisms which can be used. One is to use the append command.

This command allows the user to add information to a relation which already exists. In its simplist form it looks like the following:

```
\r
go
append to parts(pnum=18,pname="disk rewinder",color="blue",
weight=7,qoh=1)
\g
```

Again the only message you get from INGRES is:

```
query formulation complete
continue
```

Again you must print the parts relation if you do not believe your update had the correct effect. After you do this try the command to delete the row you just put in.

In this simple form an append command has the form of

```
append to relation-name(column=function,...,column=function)
```

Each column must appear inside the parentheses and must be set equal to something (in the example above various constants). These constants are put into their appropriate places in a new row of the relation indicated by relation-name. Note clearly that a new row can be added to any relation in this fashion.

If one wishes to enter data into a new relation, he must first create the relation using the INGRES create command. This has the effect of creating an empty relation with a given relation name and given column names. In a create statement the format of each column must formally be specified. An example of a create statement is the following.

```
\r
go
create example(character = c10, integer = i2, float = f4)
\g
```

This statement creates a new relation called example with columns character, integer and float. These columns are respectively a character string of length 10 bytes, an integer of length 2 bytes and a floating point number of length 4 bytes. This format information enables INGRES to correctly store and retrieve data of various types. The types currently supported are the following:

```
i1, i2, i4          (integers)
f4, f8              (floating point numbers)
c1, c2, ..., c255   (character strings)
```

Try printing the example relation to see what happens.

You can now execute append statements to add rows to the example relation since it now exists.

Successive application of append statements can add any number of rows to a relation. However, if one has many additions to make it may be easier to use the second update mechanism.

INGRES supports a facility to copy a relation into INGRES from a given user file in UNIX. The general form of a copy statement is the following:

```
copy relation-name(column = format,..., column = format)
    (from, to)  "UNIX-file-name"
```

We do an example of the copy operation at this time.

```
\r
go
copy example(character = c10, integer = i2, float = f4)
       from "/mnt/mike/example"
\g
```

This example finds the file "/mnt/mike/example" and reads the first 16 bytes into row one of the example relation . It then reads the next 16 bytes into row 2 and continues until an end of file. In this way a user who has a tape in a given fixed length format can copy it into a UNIX file and then use the INGRES copy command to form a relation from his data. Likewise, a user who wishes to take information away from INGRES for processing under control of UNIX may use the INGRES copy command to a UNIX file (instead of from a UNIX file).

There are several points to be remembered about copy:

a) the relation name to be copied into or from must exist prior to the copy command.

b) The format statements in the copy command specify the data format of the UNIX file. This format need not be the same as the one used for the INGRES relation being copied.

c) The columns in the copy command need not be in the same order as they appeared in the create command which formed the relation involved. INGRES correctly reorders columns where necessary.

d) If the length of the column in the copy command does not equal the length of the column from the create statement but the data types are the same, the following operations take place:

for character string- they are padded with blanks if a larger field is required. If a shorter field is desired, an error message results

for integers- they are converted to the appropriate length. The result is unpredictable if this conversion causes an overflow.

for floating point- they are converted to the appropriate length

d) If the data format of a column is not the same in the UNIX file and the INGRES relation, appropriate conversions are made using standard conventions.

Often one wants conversion to take place from character strings of a variable length to either integer or floating point format.

Suppose, for example, one creates using the UNIX editor a file called /mnt/mike/sample with contents:

```
123, 46.5
402, 34.1
20, 7.3
2000, 700.0
```

In the editor it is a quick operation to perform this task. What one would like now is for INGRES to convert the first field to an integer and the second to a floating point number for each of the four desired rows during the copy operation. Moreover, one would like INGRES to recognize the comma and carriage return as delimiters between the variable length fields.

This is done as follows:

```
\r
go
create example2(int = i2, float = f4)
copy example2(int = c0, float = c0) from "/mnt/mike/sample"
\g
```

This will correctly copy and convert the four rows. The format c0 says simply look for a character string delimited by a comma, a carriage return or other non numeric character and convert it to the type specified in the create statement. Unfortunately, you cannot put a decimal point into fields which you wish converted to integers.

Of course, the user could have done the same transfer by correctly alligning the information in /mnt/mike/sample so each column was of fixed length. However, c0 format spares the user this hassle.

The last notion we discuss in this manual is how to discover what format a relation is stored in. This is sometimes necessary when we have to know whether to put quote marks around strings that we use in an interaction.

For example in the parts relation discussed above there is a column called pnum. In one interaction we required part names that (among other things) had the property that pnum was less than 10. If pnum was stored as a character string we would have been required to put quotes around the 10 in order for the interaction to work correctly. However, we knew it was an integer and the interaction worked correctly as stated.

To find the format of a relation simply type

```
help "relation-name"
```

and the various columns, their formats and other information will be returned to your terminal.