Paper 046-31

DIGITS and DATES – The SQL Procedure Goes "Loopy"

Jack Hamilton, Kaiser Foundation Health Plan, Oakland, California

ABSTRACT

Although PROC SQL, with its descriptive approach to programming, is very good at manipulating data, it lacks an important feature that is found in traditional procedural programming languages--the ability to loop over a range of values, some of which might not exist in an input data set.

INTRODUCTION

This paper describes how to create and use utility data sets to simulate looping in PROC SQL. The DIGITS data set can be used to simulate general loops, which is the equivalent of "do i = 37 to 43", and the DATES data set can be used to generate date-based records, which is the equivalent of "all weekend dates in October 2005" or "all days in 2006 which are the first Tuesday in the month."

THE DATA STEP APPROACH

Looping in a data step is easy – you simply use the DO statement. For example, if you wanted to process the numbers 15 through 20, you would code:

```
do i = 15 to 20;
   /* code goes here */
End;
```

THE PROBLEM WITH LOOPS IN SQL

If you wanted to do the equivalent of the loop above in SQL, you'd have a problem: SQL doesn't support looping. You always need an input data set (or table, in SQL lingo) to power your processing (this is not true in all implementations of SQL, but it is true in base SAS).

The most straightforward way around this problem is to pre-generate a table containing the numbers you want to loop through:

```
data mynumbers;
   do i = 15 to 20;
      output;
   end;
run;

NOTE: The data set WORK.MYNUMBERS has 6 observations and 1
      variables.


proc sql;
   select   i as myvalue
   from     mynumbers;
quit;
```

prints

```
myvalue
--------
      15
      16
      17
      18
      19
      20
```

That 's an acceptable solution, as long as you know what the range of numbers is going to in advance. If you don't, you'd have to create a table containing all the numbers you'd ever need, and use a WHERE clause to control which numbers are selected:

```
data mynumbers;
   do i = 1 to 100;
      output;
   end;
run;

NOTE: The data set WORK.MYNUMBERS has 100 observations and 1
      variables.


proc sql;
   select   i as myvalue
   from     mynumbers
   where    i between 15 and 20;
quit;
```

That's OK, but what if you don't know in advance what the loop values will be? Suppose sometimes they'll be 1 to 10, and sometimes they'll be 1,000,000 to 1,000,010, and sometimes they'll be -3E10 to -3E10+12? You'd have to create a very large data set; just creating it would take a long time, and it would require a lot of space. Remember the *Frivolous Law of Arithmetic*: Almost all natural numbers are very, very, very large. An alternative to creating a list of all integers might be a good idea.

THE DIGITS DATA SET

The key concept behind this paper is that you can create a small table containing only the one-digit numbers, and use that table to create any other range of numbers you might need.

You could create the table using a data step, but we might was well stick with SQL:

```
proc sql;
   create table digits
      (digit integer);
NOTE: Table WORK.DIGITS created, with 0 rows and 1 columns.
   insert into digits
      values (0) values(1) values(2) values(3) values(4)
      values (5) values(6) values(7) values(8) values(9);
NOTE: 10 rows were inserted into WORK.DIGITS.

quit;
```

You might want to save the table in a SAS library you often use, in your SASUSER library, or even (if you have the appropriate authority) into your installations SASHELP library.

USING THE DIGITS DATA SET

Now that you have a data set containing all the signle digits, you can do a self-join to obtain a range of numbers. For example, to obtain 0 through 99, you could code:

```
proc sql;
   select   tens.digit*10 + ones.digit as myvalue
   from     digits as tens,
            digits as ones;
NOTE: The execution of this query involves performing one or
      more Cartesian product joins that can not be optimized.
   %put INFO: Rows written: &SQLOBS.;
INFO: Rows written: 100
quit;
```

I won't show all the resulting rows, but here's a sample:

```
myvalue
--------
       0
       1
       2
…
      96
      97
      98
      99
```

Here's what happens:

The values in the DIGITS dataset are used to create the one's place numbers 0 through 9, aliased as ONES.

The values in the DIGITS dataset are multiplied by 10 to create the ten's place numbers 0, 10, 20, and so forth, aliased as TENS.

A Cartesian join between the two data sets is performed, and the one's place numbers are added to the ten's place numbers, creating all the numbers between 0 and 99.

The order in which the numbers are returned is implementation-dependent, and not defined by SQL. The version of SQL in base SAS usually works from top to bottom (or right to left, if you prefer to think of it that way), so I coded

```
from     digits as tens,
         digits as ones;
```

and not

```
from     digits as ones,
         digits as tens;
```

In another database, you might have to place the tables in the opposite order. But in the version of SAS we all know and love, reversing the placement would produce these values as the first 11 results:

```
myvalue
--------
       0
      10
      20
      30
      40
      50
      60
      70
      80
      90
       1
```

If you want the digits between 15 and 20, just add a WHERE clause:

```
proc sql;
   select   ones.digit + tens.digit*10 as myvalue
   from     digits as tens,
            digits as ones
   where    calculated myvalue between 15 and 20;
NOTE: The execution of this query involves performing one or
      more Cartesian product joins that can not be optimized.
   %put INFO: Rows written: &SQLOBS.;
INFO: Rows written: 6
quit;
```

If the desired numbers aren't small, you might want to modify the query about so that unnecessary numbers won't be generated only to be discarded. So instead of

```
proc sql;
   select   e0.digit + e1.digit*1e1 + e2.digit*1e2
            + e3.digit*1e3 + e4.digit*1e4 + e5.digit*1e5
            + e6.digit*1e6 as myvalue
   from     digits as e6,
            digits as e5,
            digits as e4,
            digits as e3,
            digits as e2,
            digits as e1,
            digits as e0
   where    calculated myvalue between 1000000 and 1000010;
NOTE: The execution of this query involves performing one or more
      Cartesian product joins that can not be optimized.
   %put INFO: Rows written: &SQLOBS.;
INFO: Rows written: 11
quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           2.60 seconds
      cpu time            2.60 seconds
```

You could code:

```
proc sql;
   select   1e6 + ones.digit + tens.digit*10 as myvalue
   from     digits as ones,
            digits as tens
   where    calculated myvalue between 1000000 and 1000010;
NOTE: The execution of this query involves performing one or more
      Cartesian product joins that can not be optimized.
   %put INFO: Rows written: &SQLOBS.;
INFO: Rows written: 11
quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.04 seconds
      cpu time            0.04 seconds
```

You can, of course, use more complicated expressions in the WHERE clause. The following code selects all even numbers between 0 and 100:

```
991  proc sql;
992     select   ones.digit + tens.digit*10 as myvalue
993     from     digits as tens,
994              digits as ones
995     where    mod(calculated myvalue, 2) = 0;
NOTE: The execution of this query involves performing one or more
      Cartesian product joins that can not be optimized.
996     %put INFO: Rows written: &SQLOBS.;
INFO: Rows written: 50
997  quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds
```

Another way to express the same set of numbers is

```
1048  proc sql;
1049     select   ones.digit + tens.digit*10 as myvalue
1050     from     digits as tens,
1051              digits as ones
1052     where    ones.digit in (0, 2, 4, 6, 8);
NOTE: The execution of this query involves performing one or more
      Cartesian product joins that can not be optimized.
1053     %put INFO: Rows written: &SQLOBS.;
INFO: Rows written: 50
quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.03 seconds
      cpu time            0.03 seconds
```

But there's a trap here. I said above that "[t] The order in which the numbers are returned is implementation-dependent, and not defined by SQL", and in this case the WHERE clause causes SAS to join in a different order, giving you:

```
myvalue
--------
       0
      10
      20
      30
      40
      50
      60
      70
      80
      90
       2
```

If the order is important to you, you'd have to use

```
proc sql;
   select   ones.digit + tens.digit*10 as myvalue
   from     digits as tens,
            digits as ones
   where    ones.digit in (0, 2, 4, 6, 8)
   order    by myvalue;
NOTE: The execution of this query involves performing one or more
      Cartesian product joins that can not be optimized.
   %put INFO: Rows written: &SQLOBS.;
INFO: Rows written: 50
quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.10 seconds
      cpu time            0.06 seconds
```

This gives the result you want, but at the cost of increased CPU and elapsed times. Sometimes it's better to try to fool the optimizer into doing what you want.

A REAL-LIFE EXAMPLE

One of the problems with giving a Coders Corner paper is that you need an example, and it's hard to come up with an example that's either so trivially simple that there's hardly in point in doing it, as in the examples above, or so complicated that it's impossible to explain in the time alloted. I've given some trivially simple examples above, so here's my impossibly complicated example, which I will leave you explore on your own time.

At my previous employer, I made extensive use of SAS/Intrnet. Two of the programming tools available in SAS/Intrnet are htmSQL, which is based on SQL, and the Application Broker, which is based on the data step, procedures, and ODS. Many tasks are easier in htmSQL, and I tried, whenever possible, to use it rather than the Application Broker. This gave me better control over what HTML output looked like, but I was restricted to what I could do in SQL. That meant I got to spend time figuring out how to do seemingly impossilble tasks with SQL.

One of those tasks was displaying a file directory. There are various ways to do that with a data step, including the use of the data set information functions (sometimes call SCL functions) and the use of pipes with OS commands. I couldn't use pipes in htmSQL, and the use of the information functions seemed blocked by the need to keep the directory IDs and file IDs needed by those functions.

Here's the data step code to produce a list of the files in a directory on Windows;

```
data dsfiles (keep=filename);
   basedir = "D:\MyFiles\KP\";
   length filename $256;
   rc = filename("datadir", basedir);
   dir_id = dopen("datadir");
   numfiles = dnum(dir_id);
   do i = 1 to numfiles;
      filename = dread(dir_id, i);
      output;
   end;
   rc = dclose(dir_id);
run;

NOTE: The data set WORK.DSFILES has 4 observations and 1 variables.
```

The output dataset looks like this:

```
           filename

MembershipMacroGETMBSH.doc
MembershipMacroGETMBSH.html
MembershipMacroGETMBSH.odt
MembershipMacroGETMBSH.pdb
```

Here's the equivalent in SQL, using the looping capabilities provided by the DIGITS table to iterate through the filenames:

```
proc sql;
   select   dread(dir_id, filenum) as filename
   from     (select  filename("datadir", 'D:\MyFiles\KP\') as
filename_rc,
                     dopen("datadir") as dir_id,
                     dnum(calculated dir_id) as numfiles
             from    digits
             where   digit = 0) as d,
            (select  one.digit + 10*ten.digit as filenum
              from   digits as one,
                     digits as ten)
   where    filenum between 1 and numfiles;
NOTE: The execution of this query involves performing one or more
      Cartesian product joins that can not be optimized.
quit;
```

Please note that the code contains an assumption that there will be no more than 99 files in the directory; in the case of my htmSQL page, I was sure that was the case.

LOOPING WITH DATES

LOOPING IN THE DATA STEP

If you wanted to loop through a date range, such as all days in January, 2006, in a data step, you could code a loop with SAS date values:

```
do mydate = '01jan2006'd to '31jan2006'd;
   put mydate=;
end;
format mydate date9.;
```

A loop with non-sequential dates is slightly more complicated, but still easily do-able. Suppose you wanted all the Tuesdays in January, 2006:

```
data _null_;
   do mydate = '01jan2006'd to '31jan2006'd;
      if weekday(mydate) = 3 then
         put mydate=weekdatx.;
   end;
   format mydate date9.;
run;

mydate=Tuesday, 3 January 2006
mydate=Tuesday, 10 January 2006
mydate=Tuesday, 17 January 2006
mydate=Tuesday, 24 January 2006
mydate=Tuesday, 31 January 2006
```

In some cases, you could use the INTNX function to help with date intervals, but that can be tricky. That's outside the topic of this paper, but see the *Hazards* section at the end for an example that doesn't work.

LOOPING IN SQL

There are two approaches to date looping in SQL. The first is to use the DIGITS table, and restrict the range to the dates you want. The second approach is to create a DATES table in advance.

LOOPING THROUGH DATES WITH THE DIGITS TABLE

To use the DIGITS table, you have to combine the DIGITS table with SAS date values or functions. This example finds the days in January:

```
228  proc sql;
229     select   '01jan2006'd + ones.digit + 10*tens.digit as mydate
229! format=date9.
230     from     digits as tens,
231              digits as ones
232     where    calculated mydate <= '31jan2006'd;
NOTE: The execution of this query involves performing one or more
      Cartesian product joins that can not be optimized.
233     %put INFO: Rows written: &SQLOBS.;
INFO: Rows written: 31
234  quit;
```

Finding Tuesdays in January can be done in a similar fashion:

```
250  proc sql;
251     select   '01jan2006'd
252                 + ones.digit + 10*tens.digit as mydate format=date9.
253     from     digits as tens,
254              digits as ones
255     where    calculated mydate <= '31jan2006'd
256              and weekday(calculated mydate) =3;
NOTE: The execution of this query involves performing one or more
      Cartesian product joins that can not be optimized.
257     %put INFO: Rows written: &SQLOBS.;
INFO: Rows written: 5
258  quit;
```

This technique works, but for several reasons it may be better to create a DATES utility table similar to the DIGITS table.

LOOPING WITH A DATES UTILITY TABLE

Although a utility table containing all the integers you might ever want to use is impractical, a data set containing all the dates you might want to use is not: SAS date values are restricted to a relatively small range, 6,690,873 days between 1582 CE and 19990 CE, and in practice you're unlikely to need values outside the lifetime of people alive today, which optimistically is only 88,023 days between 1886 and 2136 (SAS dates are not useful for historical dates because they do not properly account for calendar changes). You might want to expand the upper date range to December 31, 9999 to cover the SQL standard "unknown date" value.

In a DATES utility table, you probably want not only the actual SAS date , but a few associated values such as the year, the month number, the day of month number, the day of week number, the Julian date, and a flag for the last day of the month. It's easy to produce this data set using the data step:

```
data dates;
   do date = '01jan1886'd to '31dec2126'd;
      year = year(date);
      month = month(date);
      dayofmonth = day(date);
      dayofweek = weekday(date);
      julian = juldate7(date);
      lastdayofmonth = (date=intnx('month', date, 0, 'e'));
      output;
   end;
   format date date9.;
run;

NOTE: The data set WORK.DATES has 88023 observations and 7 variables.
```

You could add additional variables as needed, such as the quarter, a flag for the first and last weekdays in the month, or accounting date information. When I worked for Varian Associates we had a special date number called the manufacturing day which I kept in a SAS data set.

You can then use SQL to select all of the records from another table whose dates match a date in the DATES table; you can use a WHERE clause to select only those rows whose data match a date characteristic in the dates table, such as the first day of the month:

```
proc sql;
   create table dowjonesfirst as
      select   dj.date,
               dj.snydjcm as dowjones format=8.3
      from     sashelp.citiday as dj,
               dates as d
      where    dj.date = d.date
               and d.dayofmonth = 1;
NOTE: Table WORK.DOWJONESFIRST created, with 36 rows and 2 columns.

quit;
```

Here are the first few records from the resulting table:

DATE	dowjones
01JAN1988	.
01FEB1988	731.400
01MAR1988	767.700
01APR1988	.
01JUN1988	766.390
01JUL1988	796.780

Notice that there is no record for May 1, 1988, because there was no record for that date in the CITIDAY table.

If you want records for all dates, whether or not they are in the main table, you can use a right join:

```
621  proc sql;
622     create table dowjonesfirstall as
623        select   d.date,
624                 dj.snydjcm as dowjones format=8.3
625        from     sashelp.citiday as dj
626        right join
627                 dates as d
628        on       dj.date = d.date
629        where    d.dayofmonth = 1
630                 and d.date between '01jan1988'd and '15feb1992'd;
NOTE: SAS threaded sort was used.
NOTE: Table WORK.DOWJONESFIRSTALL created, with 50 rows and 2 columns.

631  quit;
```

which creates:

```
     date    dowjones

01JAN1988        .
01FEB1988     731.400
01MAR1988     767.700
01APR1988        .
01MAY1988        .
01JUN1988     766.390
01JUL1988     796.780
```

HAZARDS

UNRESTRICTED LOOPS WITH INFORMATION FUNCTIONS

The information functions take a numeric argument specifying the nth something. For example, the DREAD function takes as its second argument the sequence number of the file for which it will return information. If you pass it an invalid sequence number, you will get an error message in the log. In the following example, I left off the clause restricting processing to the number of files in the directory (`where filenum between 1 and numfiles`), producing a message and a bunch of blank lines in the output. You could cheat by using a where clause which deletes blank filenames, but you'll still get an error message:

```
2316  proc sql;
2317     select   dread(dir_id, filenum) as filename
2318     from     (select  filename("datadir", 'D:\MyFiles\KP\') as
2318! filename_rc,
2319                       dopen("datadir") as dir_id,
2320                       dnum(calculated dir_id) as numfiles
2321              from     digits
2322              where    digit = 0) as d,
2323              (select  one.digit + 10*ten.digit as filenum
2324               from    digits as one,
2325                       digits as ten)
2326     where    calculated filename ne ' ';
NOTE: The execution of this query involves performing one or more
      Cartesian product joins that can not be optimized.
NOTE: Invalid argument 2 to function DREAD. Missing values may be
      generated.
2327     %put INFO: Rows written: &SQLOBS.;
INFO: Rows written: 4
2328  quit;
```

UNEXPECTED RESULTS FROM THE INTNX FUNCTION

You might think that you could use a data step loop with the INTNX function to get dates separated by an interval. Well, you can, but you have to be careful. Date intervals don't always cover the periods you'd like them to. Consider the following:

```
data _null_;
   mydate = intnx('week.3', '01jan2006'd, 0);
   do while (mydate <= '31jan2006'd);
      put mydate=weekdatx.;
      mydate = intnx('week.3', mydate, 1);
   end;
   format mydate date9.;
run;

mydate=Tuesday, 27 December 2005
mydate=Tuesday, 3 January 2006
mydate=Tuesday, 10 January 2006
mydate=Tuesday, 17 January 2006
mydate=Tuesday, 24 January 2006
mydate=Tuesday, 31 January 2006
```

ACKNOWLEDGMENTS

Many thanks go to Richard de Venezia, who took the trouble to investigate exactly how SAS handles nested tables in joins.

REFERENCES

Eric W. Weisstein. "Frivolous Theorem of Arithmetic." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/FrivolousTheoremofArithmetic.html

RECOMMENDED READING

Everyone who uses SQL should read Joe Celko's book *SQL For Smarties, Second Edition* (Morgan-Kaufmann), 2000, ISBN 1-55860-576-2. It's available in English, French (ISBN 2-84180-141-1), Japanese (ISBN 4-8101-8949-X), Chinese (ISBN 957-442-015-9), and Hungarian Edition (ISBN 963-9301-20-5). His web page is http://www.celko.com.

CONTACT INFORMATION

Jack Hamilton
Kaiser Foundation Health Plan
1950 Franklin Street
Oakland, California 94612
+1 (510) 987-1556
jfh@alumni.stanford.org
http://www.excursive.com/sas/