

Selected Programming Problems from SAS-L

H. Ian Whitlock, Westat Inc., Rockville, MD

Introduction

SAS-L is an electronic network of SAS® users who pose and answer various SAS related problems. Some of my responses over the last two years to questions involving the "how do you code?" type of problem are presented. The examples are concentrated in two major topics introduced in the version 6 SAS/BASE software.

1 - Use of the KEY option to read indexed data with the SET and MODIFY statements.

2 - PROC SQL versus the DATA step.

The common thread for these topics is solutions to the "look-up" problem.

Indexed Data

Example 1 - Multiple Indexes

William Kreuter asked how SAS handles multiple indices. He created test data with two indices.

```
DATA TEST ( INDEX = ( A C ) );
  INPUT A C ;
CARDS ;
1 1
1 2
1 1
3 2
3 3
3 1
RUN ;
```

He was surprised when his first test didn't work.

```
DATA _NULL_ ;
  SET TEST ;
  BY A C ;
RUN ;
```

And even more surprised when his second test did.

```
DATA _NULL_ ;
  SET TEST ;
  BY C A ;
RUN ;
```

Why did the first test fail? SAS does use an index when available. In this case the INDEX A was used, but it contributed little because the data set is already in A order. On the third observation with A = 1, SAS found that the record was out of order since C = 1 is not greater or equal to C = 2. His mistake was that he needed a **composite index** not two single indices. His DATA statement should have been

```
DATA TEST ( INDEX = ( AC = ( A C ) ) ) ;
```

In this case SAS would have used the composite index and not found any problem with the order.

The second test passed by an accident of the order of the file. Here the index C is used, but within C the records happen to be in A order.

Example 2 - Lookup on ZIP

Jim Roberts had 50,000 observations with ZIP codes falling in a list of 1,000 codes. He wanted to get some information determined by the ZIP. His code took over 15 CPU minutes on a large IBM mainframe. His questions were why, and what could be done to improve the performance?

```
DATA LOOKUP ( KEEP = LKUPZIP INFO ) ;
  INFILE IN ;
  INPUT LKUPZIP $CHAR5.
        INFO $CHAR20. ;
RUN ;
```

```
DATA COMPLETE
(DROP=FOUND NOTFOUND);
IF EOF THEN PUT NOTFOUND= ;
SET IN2.BIOSTS END = EOF ;
DO PTR = 1 TO NOBS
  UNTIL (FOUND);
  SET LOOKUP
    POINT = PTR NOBS = NOBS ;
```

```

IF ZIPCODE = LKUPZIP THEN
    FOUND = 1;
END;
IF NOT FOUND THEN NOTFOUND + 1;
RUN;

```

The problem is that on average, 500 records are read from LOOKUP for every observation in IN2.BIOSTS. His program was reading an astounding 28.5 million records.

I suggested indexing the LOOKUP data set and using the index to locate the right ZIP record. Here is the code.

```

DATA LOOKUP ( KEEP = ZIPCODE INFO
              INDEX = ( ZIPCODE ) );
  INFILE IN;
  INPUT  ZIPCODE $CHAR5.
         INFO $CHAR20.;
RUN;

DATA COMPLETE
  ( DROP = NOTFOUND );
  IF EOF THEN PUT NOTFOUND = ;
  SET IN2.BIOSTS END = EOF;
  SET LOOKUP
    KEY = ZIPCODE / UNIQUE;
  IF _IORC_ > 0 THEN
  DO;
    NOTFOUND + 1;
    _ERROR_ = 0;
    INFO = '';
  END;
RUN;

```

He reported that this code executed in 15 CPU seconds. The improvement was a factor of 60.

The key word UNIQUE on the SET statement is essential. Without it, on the second occurrence of a ZIP, SAS looks for a second record in LOOKUP with the same ZIP. Note that on the SET statement, UNIQUE modifies the lookup process; it doesn't mean that the index has unique values.

By experience I have found that there are usually problems when one doesn't force the index to be unique in the LOOKUP data set.

Example 3 - From Code to Lookup

Warren Stinson presented a problem at NESUG 93 Coder's Corner (page 210).

Given a table of SIC codes and key words, split a flat file containing an SIC code and 6 special fields as follows:

- If the SIC code is in the table and the corresponding key word appears in any of the special fields, then the record goes to ACCURATE.
- Otherwise to NOHIT.

The relevant part of his file layout is given by;

```

@10 NAME $90.
@100 T1 $90.
@190 T2 $90.
...
@550 LOB $128.
@678 SIC $8.
800 more bytes

```

Warren saw this problem as a long list of decisions:

```

IF SIC = '01190100' AND
  ((INDEX(T1,'PEA') > 0) OR
   (INDEX(T2,'PEA') > 0) OR
   ....
   (INDEX(LOB,'PEA') > 0) THEN
  FILE ACCURATE;
ELSE
  ....
  ELSE FILE NOHIT;

```

He said it was over 1500 lines of code.

His solution to avoid writing the boring 1500 lines, was to generate the code using a SAS "PUT" program and then %INCLUDE it.

The problem here is similar to that in Example 2. In both cases one had to linearly plow through a long list of decisions. Assuming a uniform distribution of SIC codes, an average of 750 lines of IF-THEN/ELSE

Advanced Tutorials

code must be executed for every record input. It is faster to go directly to the right decision. In other words - view the problem as a lookup problem.

For simplicity, assume there is only one word per SIC code and that the table has been put into a SAS data set indexed by SIC.

```
DATA _NULL_ ;
  LENGTH T1 - T5 $ 90 LOB $128 ;
  ARRAY CK (6) T1-T5 LOB ;
  INFILE IN ;
  INPUT @678 SIC $8. @ ;
  SET SICTABLE KEY = SIC / UNIQUE ;
  IF _IORC_ = 0 THEN
    DO ; /* HAVE SIC LOOK FOR WORD */
      INPUT ... /* CHECK FIELDS */ ;
      DO I = 1 TO 6 UNTIL ( FOUND ) ;
        IF INDEX ( CK(I) , WORD )
          THEN FOUND = 1 ;
      END ;
      IF FOUND THEN
        FILE ACCURATE ;
      ELSE /* NO KEY WORD */
        FILE NOHIT ;
      END ;
    ELSE DO ; /* SIC NOT FOUND */
      FILE NOHIT ;
      _ERROR_ = 0 ;
    END ;
    PUT _INFILE_ ;
  RUN ;
```

This code doesn't compile and process 1500 lines. In addition it doesn't waste extra time on SIC codes that are not in the table.

Now assume that more than one word can be associated with an SIC code. Now we need a unique index.

A general solution to the problem of multiple values in the index is to add a sequence number and make it part of the index.

First create SICTABLE with the fields

```
SIC
N (sequential number)
```

CNT (total for SIC)
WORD (the key word)

indexed by SICH = (SIC N).

For example:

Input from table
Sort by SIC
PROC SUMMARY for counts
Merge and index

Now here is the lookup code.

```
DATA _NULL_ ;
  LENGTH T1 - T5 $ 90 LOB $128 ;
  ARRAY CK (6) T1-T5 LOB ;
  INFILE IN ;
  INPUT @678 SIC $8. @ ;
  N = 1 ;
  LINK GETSIC ;
  IF _IORC_ > 0 THEN
    DO ; /* SIC NOT FOUND */
      FILE NOHIT ;
      PUT _INFILE_ ;
      RETURN ;
    END ;
    /* -- LOOK FOR WORD -- */
    DO N = 1 TO CNT UNTIL ( FOUND ) ;
      IF N > 1 THEN LINK GETSIC ;
      IF _IORC_ = 0 THEN
        DO ;
          INPUT ... /* CHECK FIELDS */ ;
          DO I = 1 TO 6
            UNTIL ( FOUND ) ;
            IF INDEX ( CK(I) , WORD )
              THEN FOUND = 1 ;
          END ;
        END ;
      ELSE DO ; /* INVALID SICH */
        PUT 'CORRUPTED LOOKUP' ;
        ABORT ;
      END ;
    END ; /* -- LOOK FOR WORD -- */
    IF FOUND THEN FILE ACCURATE ;
    ELSE FILE NOHIT ;
    PUT _INFILE_ ;
```

```

RETURN ;

GETSIC:
  SET SICTABLE KEY = SICN / UNIQUE ;
RUN ;

```

Note that the link routine is required to avoid setting up two buffers for SICTABLE.

Example 4 - An Iterative Lookup

Tom Skinner had a client who wanted for each subassembly to look up all components that used the subassembly. Then for a given list of subassemblies, find all components using any of these subassemblies and all components using any of the components found, etc., until all components involving the original subassemblies have been listed.

Andy Norton pointed out that the problem was called the Bill of Materials problem, and that Paul Kent had given an SQL solution at SUGI 17.

Following Tom's example data, I set up the initial list of subassemblies as a data set WANTED, where SUB names the subassembly, PARENT is initially equal to SUB, and USED is initially 0. USED is a variable to indicate when all direct parents of PARENTS had been looked up.

WANTED -	SUB	PARENT	USED
	A	A	0
	B	B	0
	C	C	0

The problem now reduces to adding records to WANTED until every ancestor of every SUB is included. In other words, continue to do the following:

For each PARENT with USED = 0, set USED = 1 and add all parents of PARENT to WANTED making USED = 0 for these records.

Until all values of USED = 1.

For the look up file, I added a sequence number N to make the index NCOMP = (N COMP) unique.

LOOKUP -	N	COMP	PARENT
	1	A	W
	2	A	X
	3	A	Y
	1	X	Z
	1	B	Z
	1	C	X

The final answer then consisted of removing the original records of WANTED from the expanded WANTED, i.e. when SUB = PARENT.

RESULT -	SUB	PARENT
	A	W
	A	X
	A	Y
	A	Z
	B	Z
	C	X
	C	Z

The whole process is housed in a macro REPEAT which consists of one DATA step. At the end of the DATA step, it calls REPEAT via CALL EXECUTE as long as there are more components to be looked up. The process stops when there are no more calls to REPEAT.

```

%MACRO REPEAT ;
  DATA WANTED ( KEEP = SUB PARENT USED
  );
    IF EOF AND CNT > 0 THEN CALL
EXECUTE ( '%REPEAT' );
    SET WANTED END = EOF ;
    IF USED THEN OUTPUT ;
    ELSE
    DO ;
      USED = 1 ;
      OUTPUT ;
      N = 0 ;
      USED = 0 ;
      COMP = PARENT ;
      DO WHILE ( 1 ) ;
        N + 1 ;
        SET REL KEY = NCOMP / UNIQUE ;
        IF _IORC_ = 0 THEN
          DO ;

```

```

        OUTPUT ;
        CNT + 1 ;
    END ;
    ELSE
    DO ;
        _ERROR_ = 0 ;
        LEAVE ;
    END ;
END ;
END ;
RUN ;

```

```
%MEND REPEAT ;
```

```
%REPEAT
```

```

PROC SUMMARY DATA = WANTED NWAY ;
    WHERE SUB ^= PARENT ;
    CLASS SUB PARENT ;
    OUTPUT OUT = SUMMRY ( KEEP = SUB
        PARENT_FREQ_ ) ;
RUN ;

```

```

TITLE "FINAL RESULT" ;
PROC PRINT DATA = SUMMRY ;
RUN ;

```

PROC SQL

The following problems show why the DATA step programmer needs PROC SQL.

Example 5 - Many-to-Many Merge

Stephen P. James had a file of variable names and a file of formats with their values. He wanted to merge them, but had problems because several variables had one format and each format had several values.

```

VARS -  VARIABLE  FORMAT
        V1        YESNO
        V2        YESNO
        V3

```

```

FMTS -  FORMAT  VALUE  LABEL
        YESNO   1     YES
        YESNO   2     NO

```

Stephen wanted

VARIABLE	FORMAT	VALUE	LABEL
V1	YESNO	1	YES
V1	YESNO	2	NO
V2	YESNO	1	YES
V2	YESNO	2	NO
V3			

Many-to-many merges are easily done in SQL.

```

PROC SQL ;
    CREATE TABLE VARFMTS AS
    SELECT  VARIABLE , V.FORMAT ,
            VALUE , LABEL
    FROM    VARS AS V
            LEFT JOIN
            FMTS AS F
    ON V.FORMAT = F.FORMAT
    ORDER BY VARIABLE , VALUE
    ;
QUIT ;

```

Example 6 - Time Line Merge

Robin Yeldell had a file, DIRECT, of hospital directors with the range of dates when they were active, and a file NEEDNAME of hospitals with specific event dates. He wanted the name of the active director at the event for each observation in NEEDNAME.

```

DIRECT
ID    EFFDATE    TRMDATE    NAME
1     01JAN90    31DEC90    Joe
1     01JAN91    31DEC91    Jane
1     01JAN92    31DEC92    John
....

```

```

NEEDNAME
ID    DATE
1     01JUL92
....

```

SQL provides a quick and easy answer.

```
PROC SQL ;
```

```

CREATE TABLE NAMED AS
SELECT N.ID , DATE , NAME
FROM
  NEEDNAME AS N
LEFT JOIN
  DIRECT AS D
ON N.ID = D.ID AND
  DATE BETWEEN
    EFFDATE AND TRMDATE
;
QUIT ;

```

Example 7 - Almost Matching

Robert Mathews had two data sets with Social Security numbers. He wanted to compare records where at least 7 of the 9 corresponding digits matched.

```

proc sql ;
  create table possmtch as
  select   a.ssn as ssn1 , b.ssn as ssn2 ,
          a.name as name1 ,
          b.name as name2
  from w1 as a , w2 as b
  where
    sum (
      substr(a.ssn,1,1) = substr(b.ssn,1,1),
      substr(a.ssn,2,1) = substr(b.ssn,2,1),
      substr(a.ssn,3,1) = substr(b.ssn,3,1),
      substr(a.ssn,4,1) = substr(b.ssn,4,1),
      substr(a.ssn,5,1) = substr(b.ssn,5,1),
      substr(a.ssn,6,1) = substr(b.ssn,6,1),
      substr(a.ssn,7,1) = substr(b.ssn,7,1),
      substr(a.ssn,8,1) = substr(b.ssn,8,1),
      substr(a.ssn,9,1) = substr(b.ssn,9,1)
    ) >= 7
  order by a.ssn , b.ssn
;
quit ;

```

Example 8 - MERGE, KEY, or SQL?

Bill Brady knew he could use a merge to look up his small list of ID's in data set D1 to get information from the large data set D2. But he was worried about

efficiency because in a straight merge, all the records from D2 would have to go through the Program Data Vector (PDV). His question was how could he take advantage of the fact that D1 was small?

To get a feeling for the problem I used SAS 6.10 for Windows with a Compaq Prolinea 4/50. The following code generated the data.

```

DATA D1 ;
  DO ID = 1 TO 5 ;
    OUTPUT ;
  END ;
RUN ;

DATA D2 ( INDEX = ( ID ) ) ;
  ARRAY A (50) A1 - A50 ;
  DO ID = 1 TO 20000 ;
    Y = RANUNI ( 0 ) ;
    OUTPUT ;
  END ;
RUN ;

```

I then compared a merge, an indexed lookup, and PROC SQL.

```

DATA MERGED1 ;
  MERGE
    D1 ( IN = D1 )
    D2 ( IN = D2 )
  ;
  BY ID ;
  IF D1 ;
  IF NOT D2 THEN
    PUT ID = 'NOT ON D2' ;
RUN ;

```

Time 29.6 seconds.

```

DATA KEYD1 ;
  SET D1 ;
  SET D2 KEY = ID ;
  IF _IORC_ > 0 THEN
    DO ;
      PUT ID = 'NOT ON D2' ;
      _ERROR_ = 0 ;
    END ;
  RUN ;

```

Time 1.64 seconds.

```
PROC SQL ;
    CREATE TABLE SQLD1 AS
        SELECT D2.*
        FROM D1 , D2
        WHERE D1.ID = D2.ID
;
QUIT ;
```

Time 1.14 seconds.

Example 9 - Subquery versus Join

David Alerton wanted to select all records in a group when any one of the records in the group had some property. This kind of question often arises in the form, show all observations in each group that has at least one observation in error.

SQL can be used to avoid the tedium of such reports.

I proposed one form of SQL solution and Frank Dilorio proposed another.

Which is faster?

- Subquery
- Construct list of wanted I's and then extract from the join.

To compare times, I used the following code to generate the data.

```
DATA W ;
    DO I = 1 TO 2000 ;
        DO J = 1 TO 3 ;
            R = RANUNI (0) ;
            OUTPUT ;
        END ;
    END ;
RUN ;
```

The problem: Get all records from W for all I's such that some R is less than .1.

```
PROC SQL ;
    /* SUBQUERY */
    CREATE TABLE T1 AS
```

```
SELECT *
FROM W
WHERE I IN ( SELECT UNIQUE I
              FROM W
              WHERE R < .1 )
;
/* JOIN */
CREATE TABLE WANTED AS
    SELECT UNIQUE I
    FROM W
    WHERE R < .1
;
CREATE TABLE T2 AS
    SELECT W.*
    FROM W , WANTED
    WHERE W.I = WANTED.I
;
QUIT ;
```

I was also interested in the effect of indexing and the order of the file.

Times: (Compaq Prolinea 4/50 SAS 6.10
under Windows 3.1)

	SQL1	SQL2
No index, I order	14.72	6.15
No index, random order	11.03	5.33
Indexed by I, in order	12.8	6.3
Indexed by I, random order	12.52	11.25

I originally tested with SAS 6.08, where the results were quite different. Of course the times are longer because the machine was slower, but it is the pattern that shows an improvement in SQL stability.

Times: (Compaq 386/16 SAS 6.08
under Windows 3.1)

	SQL1	SQL2
No index, I order	54.97	27.73
Indexed by I, in order	54.47	22.85
Indexed by I, random order	55.96	100.78

Example 10 - A Messy Report

Robert Fields had a SAS data set like the following:

<u>DR_ID</u>	<u>Patient_name</u>	<u>Drug_name</u>
Dr_one	Patient_one	Drug_A
Dr_one	Patient_one	Drug_B
Dr_one	Patient_one	Drug_C
Dr_one	Patient_three	Drug_D
Dr_two	Patient_two	Drug_A
Dr_two	Patient_one	Drug_A
Dr_three	Patient_one	Drug_B

He wanted to make a report showing for each doctor, his patients, what he prescribed for them, what other doctors they saw, and what other drugs they were taking.

The following patient/drug combinations were given by "Dr_one":

Patient_one	Drug_A
Patient_one	Drug_B
Patient_one	Drug_C
Patient_three	Drug_D

The following patients, seen by "Dr_one", have outstanding RXs with the following prescribers:

Patient_one	Drug_A	Dr_two
Patient_one	Drug_B	Dr_three

This request is a good example of why SAS programmers should learn PROC SQL.

Assume the data set is W with variables DOC, PAT, and DRUG is sorted by DOC PAT DRUG.

```
PROC SQL ;
  CREATE TABLE W2 AS
    SELECT DISTINCT A.DOC , A.PAT ,
                  B.DOC AS OTHRDOC ,
                  B.DRUG
  FROM W AS A , W AS B
  WHERE  A.PAT = B.PAT AND
        A.DOC ^= B.DOC
  ORDER BY A.DOC , A.PAT
;
QUIT ;

DATA _NULL_ ;
```

```
FILE PRINT ;

DO UNTIL ( LAST.DOC ) ;
  SET W ;
  BY DOC ;
  IF FIRST.DOC THEN PUT DOC ;
  PUT @10 PAT @25 DRUG ;
END ;

PUT / @10 '————' / ;

LAST.DOC = 0 ;
DO UNTIL ( LAST.DOC ) ;
  SET W2 ;
  BY DOC ;
  PUT @10 PAT @25 OTHRDOC @40 DRUG
;
END ;

PUT // ;

RUN ;
```

Example 11 - Neighborhood Problem

The neighborhood problem can be stated as: Match all records in data set A with records in data set B such that A.X is near B.Y, where near means within some distance D.

An elementary solution is given by:

```
PROC SQL ;
  CREATE TABLE C AS
    SELECT A.* , B.*
    WHERE A.X - B.X IS BETWEEN -D AND +D
;
QUIT ;
```

The problem lies in the number of comparisons needed. For even a modest 5,000 records in A and B, this is 25 million comparisons.

When the problem was posed on SAS-L for the case where B is A, Y is X, and D = 2.3, Howard Schreier suggested:

Create data set OFFSET with one variable OFFSET having values 0, 1, 2, and 3. Then use

```
PROC SQL ;
  CREATE TABLE C AS
    SELECT A.*
    FROM A, OFFSET, A AS B
    WHERE
      INT(A.X) + OFFSET = INT(B.X)
      AND
      A.X - B.X IS BETWEEN 0 AND 2.3
;
QUIT ;
```

At the time I dismissed it as an ad hoc solution only of interest to the poser.

- What if D = .001?
- What if D = 10000?

When Paul Kent (SESUG 94, NESUG 94, SUGI 20) presented the example noting how it pays to search for an equality condition, I reconsidered and looked for a general form that I would recognize.

Create OFFSET with values 0, D.

```
PROC SQL ;
  CREATE TABLE C AS
    SELECT A.*, B.*
    FROM A, OFFSET, B
    WHERE
      ABS (ROUND(A.X,D) - ROUND(B.Y,D))
      = OFFSET
      AND
      ABS (A.X - B.Y) <= D
;
QUIT ;
```

In fact Paul mentioned the ROUND function in his paper, so he probably had this solution in mind.

Nabil M Ali recently raised a sort of time lime merge question (Example 6). He wanted to merge data sets A with 40,000 and B with 100,000 observations when X from A is between Y and Z from B. Now the simple code means 4 billion comparisons.

I suggested that if he could put an upper bound to Z - Y, then he could apply the above neighborhood technique.

Suppose for all Y, Z in observations from B that

$$Z - Y \leq D$$

Then the code is almost the same as above.

```
PROC SQL ;
  CREATE TABLE C AS
    SELECT A.*, B.*
    FROM A, OFFSET, B
    WHERE
      ROUND(Y,D) + OFFSET =
      ROUND(Z,D)
      AND
      X IS BETWEEN Y AND Z
;
QUIT ;
```

Conclusion

The newer lookup techniques, involving the KEY option or PROC SQL, are well worth learning. In addition, PROC SQL is important to the DATA step programmer for replacing some tricky DATA step processing.

SAS-L provides many interesting programming problems to help learn these techniques.

The author can be contacted by mail or e-mail

H. Ian Whitlock
Westat
1650 Research Boulevard
Rockville, MD 20850-3129

Whitlo1@westat.com

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.