Data Structure Determines Programming Effort

Ian Whitlock, Westat, Rockville, MD

Abstract

Data are often stored in a rather inconvenient form. Thus a lot of report writing and data manipulation programs contain large chunks of code used to rebuild the data in the correct structure, followed by relatively simple report writing or data maintenance code.

Typical problems will be identified and discussed in terms of examples found while helping students, analyzing my colleagues' code, and answering questions on SAS-L.

Often the programmer does not have the control needed to correct these problems, so efficient coding methods (typically involving macro code) to work around problems that cannot be corrected will also be discussed.

In this manner, I hope to make the programmer aware of:

1. How data storage influences the difficulty of the programming task
2. How to judge the cost in programming time for poor data structures
3. How to best work around these problems when they cannot be corrected.

Introduction

I have studied a lot of SAS code from students, colleagues, and people who ask or answer questions on SAS-L. It seems like a lot of the code is unnecessarily difficult or tedious. As the title and abstract suggest, I think the single most significant reason for this is the way the data are stored. The basic purpose of this paper is to explore and understand to what extent the difficulties are self-imposed and how best to deal with them when they are externally imposed.

In general one should learn the way SAS works and then program to take advantage of the SAS way rather than fight it.

SAS Data Sets

Data should be stored as SAS data wherever possible. Otherwise one must expend effort in transforming the data into SAS data sets.

Unfortunately, the programmer rarely has the power to control this decision. When you have no control, you should decide how to best deal with it. There are two basic possibilities:

- Create a standard set of views so that the SAS programs can be written as if the data were stored in SAS data sets.
- Create standard programs to move the data into SAS data sets.

In the first case, one will pay a transformation price, every time the base data is touched. In the second case the cost is once per program, but there may be a large cost in space for holding an extra copy of the data in SAS.

The first possibility makes more sense if most of the programs for the project work with the underlying data only once. The second makes more sense when many steps must refer to the underlying data.

In any case it is best to isolate this transition process from the analysis programs. One could do this with a macro, %SETUP, or an include file. The advantage is that it is coded and debugged once and does not take up a lot of space as copies in subsequent programs.

For example, if one has an external flat file, then you could make a permanent DATA step view.

```
libname proj "c:\project\views" ;
filename dat "c:\project\ext" ;

data proj.basedata
     / view = proj.basedata;
   infile proj(basedata.dat) truncover;
   input id $char10.
         /* more variables */
   ;
   /* data transformations */
run ;
```

Now each program using this data set must contain the LIBNAME and FILENAME statements, but it need not contain a copy of the code. When a copy is maintained it obscures the real purpose of the program and invites maintenance changes. The problem here is that each PROC and DATA step that refer to PROJ.BASEDATA actually executes the INPUT DATA step.

The alternative is to write the code

```
filename dat "c:\project\ext" ;

data basedata ;
   infile proj(basedata.dat) truncover;
   input id $char10.
         /* more variables */
   ;
   /* data transformations */
run ;
```

and place it in a setup directory. Now each program

required to read these data could have

```
file setup "c:\project\setup" ;

%inc setup(basedata.sas) ;
```

In this case the data are made only once per program, but a large WORK directory may be needed to hold them.

In either case one avoids the problems of maintaining multiple copies of the code and having very long programs where the code for the real purpose of the program is obscured by long introductory steps. It is also fairly easy to evaluate the cost in time and/or space created by not working directly with permanent SAS data.

Version 8 with SAS/ACCESS has made it particularly easy to store data in some external data base system, thus avoiding the cost in terms of code. However, one will still have the cost of extra processing to convert the data into the form that SAS expects.

Data Values

Now let's assume the data are stored in a SAS database. What can go wrong with the values? The value 9 was used to represent a missing value. Since a SAS missing value was not used, work will have to be done fighting against the standard SAS meaning for 9. Here is a piece of code from a long program.

```
/* 5. COLOR NAMES SCORE */
LENGTH COLORS 4;
LABEL COLORS = 'COLOR NAMES SCORE';

ARRAY _D (*) A_D1A--A_D1J;
COL9 = 0;
DO I = 1 TO DIM(_D);
   IF _D(I) EQ 9 THEN
   DO;
      COL9 = COL9 + 1;
      _D(I) = 0;
   END;
END;

IF COL9 >= 2 THEN COLORS = .;
ELSE COLORS = SUM(OF A_D1A--A_D1J);
```

The code isn't long, but the program had thirteen pieces like this one, and it looked as if the same code would also be found in other programs from that project group. How would the code look if the data had been stored the SAS way with a missing value?

```
/* 5. COLOR NAMES SCORE */
LENGTH COLORS 4;
LABEL COLORS = 'COLOR NAMES SCORE';

IF nmiss(of a_d1--a_d1J) >= 2 THEN
   COLORS = .;
ELSE
   COLORS = SUM(OF A_D1A--A_D1J);
```

Sometimes several values were treated as missing and then converted to one of them. Assuming that the underlying data values could not be changed, I suggested the following macro.

```
%macro chngmiss ( from = 0 . 9,
                  /* change values    */
                   to   = 0,
                  /* to this one      */
                   vars =
                  /* for vars in list */
                 ) ;
   /* --------------------------------
      data step code to modify special
      values and take count of number
      modified in counter __chng
      --------------------------------
   */
   array __a&sysindex (*) &vars ;
   drop __: ;

__chng = 0 ;
   do __i&sysindex = 1
               to dim( __a&sysindex );
      if __a&sysindex (__i&sysindex)
                     in ( &from ) then
      do ;
         __a&sysindex (__i&sysindex) =
              &to ;
         __chng + 1 ;
      end ;
   end ;

%mend  chngmiss ;
```

The rule here would be, if you must suffer a repeated complexity, then it is best to isolate the code in a general macro for handling the problem.

Each observation of the data gave the results of a test that a student took. The data included information about when the test began and ended. Here is the portion of the program to determine how long it took the student to take the test.

```
/* 1.TIME FOR ASSESSMENT IN MINUTES */
LENGTH ALENGTH 3;
LABEL ALENGTH = 'LENGTH (MINUTES) OF
ASSESSMENT';

BEGHRS = INT(BEGTIME / 100);
ENDHRS = INT(ENDTIME / 100);
BEGMIN = INT(BEGTIME - (BEGHRS * 100));
ENDMIN = INT(ENDTIME - (ENDHRS * 100));

IF BEGTIME NE 9999 AND ENDTIME NE 9999
THEN DO;
   IF BEGHRS GT ENDHRS THEN
     ALENGTH=(((ENDHRS+12)*60) +
   ENDMIN)
           - ((BEGHRS * 60) + BEGMIN);
```

```
    ELSE
      ALENGTH=((ENDHRS * 60) + ENDMIN)
            - ((BEGHRS * 60) + BEGMIN);
END;
ELSE
    ALENGTH = .;

IF ALENGTH < 0 & ALENGTH NE . THEN
    PUT CHILDID= ALENGTH=
        BEGTIME= AMPM= ENDTIME= AMPM2=;
if ALENGTH < 10 or ALENGTH > 75 then
   ALENGTH = .;
```

Several things make the code more difficult than necessary. First SAS time values were not used - instead separate numbers for hours and minutes were used. Then a twelve-hour clock was used, thus making it necessary to consider the transition from AM to PM.

SAS is very rich in date and time manipulating functions. It is always easier to let the system work for you rather than fight it. Use SAS data and time values. Here is the code to calculate the length of the test assuming that BEGTIME and ENDTIME are SAS time values marking the beginning and ending of the test.

```
ALENGTH = intck ("minute",
                 begtime, endtime) ;
```

In general, avoiding SAS functions, SAS dates or time, and SAS missing values all provide an easy means to making programs harder to read and more complex than necessary. However, the damage is usually limited because data values are inherently a local issue. To create a really complex problem, it is probably necessary to use a poor choice of data structure for storing the data.

Data Structure

You can find many articles on relational data base theory and the use of normal forms elsewhere; however, I would like to simplify and keep it on a common sense basis. When storing data in a SAS data set:

1. One variable should hold an atomic piece of information, i.e. it cannot be split into simpler elements relative to usage.
2. There should be no repeating entities in an observation.
3. Data should be stored in variable values, not in variable names.
4. An observation should hold information about an entity, i.e. a thing of interest to the project.
5. The same entity information should not be stored in more than one SAS data set.

Some programmers see rules 2 and 3 as two views of the same thing; however one can have arrays where order is completely meaningless hence storing information in the names is not an issue. One can also envision storing information in the variable names without thinking of them as arrayed, so I list two separate rules, but recognize that they often are two views of the same thing.

One could also argue that rule 2 is not needed since it may be inferred from rule 4, however I find it so important that I think it better to consider 2 as a separate rule rather than just a corollary to 4.

Atomic Information

If one violates the principle of atomic information, then either the SAS code will contain special constructions for tearing apart the value of a variable, or the data will contain duplicate information to provide the parts. Of course, the problem with duplicate information is that one of the pieces can be changed, without changing the other. The problem with combining parts is that a relationship is implied that may not always be true.

A common problem occurs with observation identifiers. The identifiers should identify a single entity. For example, suppose we are dealing with survey data collected from households where each household (HH) belongs to the primary sampling unit (PSU) from which it was selected. Typically there will be several people in a household. Then we might have three different entities - PSU, household, person. It is tempting to make the first part of the household identifier, HH be the PSU from which the household was selected, and to make the identifier for a person, PERS, contain both the corresponding PSU and HHID. This is convenient because the one can get both PSU and HH information from PERS, but one should resist this temptation.

It is better to have three separate variables - PSU, HH, PERS, identify the three separate entities. Here a household observation might include a value PSU for matching to PSU observations and person record might include both PSU and HH value, but they are separate variables. The informational content is the same; it just takes more variables to get the same information.

It might appear that the problem of non-atomic fields is simply a local one affecting a small piece of code, since the SUBSTR could take apart a combined identifier, and concatenation could put them together.

Now what happens when a household moves to a new PSU? Either we are faced with the ugly prospect of changing the identifier or having the identifier lie about the location of its household. With either decision we are faced with lots of exceptional code pervading the whole project.

On the other hand, when each identifier is not part of the other, it is simple to add a new data set keeping track of a household over time that contains PSU HH, and the date

when the PSU value became the correct one.

In this case, the entities don't change identifiers, and the identifiers do not mislead one about relationships between the entities because these relationships are held in attributes of the entities, not their identifiers. Now, over time the attributes may change without causing identification problems.

Repeating Fields

One of the more common questions on SAS-L is how to restructure data so that one observation might hold repeating information. For example, in the person file of the survey mentioned above, we might have variables for holding the age and sex of the person.

Instead we could have put this information in the household file as repeating information, AGE1 - AGE10 and SEX1 - SEX10 for up to 10 people in a household. Unfortunately the allowed number is almost always wrong. For most households 10 is way too big, and for some it may be too small. Thus it is not only wasteful of space, but it may also be inadequate. Note that repeating fields always indicate an entity problem. When sex and age were kept in the person file, the household observations were about households, not people. By adding the sex and age data to the household file, we no longer can be simple about what entity a household observation represents. Households don't have either a sex or an age. People do; hence we can no longer say that all of the variables provide attributes about a household.

Note that the informational content has not changed. We can still talk about the age or sex of any person in the survey, so it is not information that has been lost. But, look, some of the information is now held in the variable names.

Suppose we identify the head of the household with the index 1, so that AGE1 is the age of the household head. AGE2 might represent the spouse's age, AGE3 the oldest child etc. What we lost was the ability to use variable values to indicate relationships. Now these relationships have moved to the names of the variables. What happens when a child is adopted? Do we move the younger children down the line to make the correct index for this adopted child? If we just add him on the end, then what happens to the relationship that used to be expressed in the order of the indices?

In addition to the relationship problem, we have virtually guaranteed that almost any program dealing with the data will need to get at the indices. Either arrays will be used to make the index values data; or worse, macro code will be required to break apart the variable names so that we can get at the indices. In either case, all of the code dealing with either age or sex must get much more complex than it would have been without repeating values.

Note, I do not say that we can never have repeating values. Some analysis procedures may require it, or you may want it to make a more readable report. It is quite all right to restructure the data in this manner for a temporary purpose. The problem was in storing the data this way so that every program dealing with this data would have to deal information half hidden in the variable names.

No Entity Identification with Observation

Repeating values is one of the most common ways to obscure the entities represented in the database, but it is not the only one. For example, we might want to store the age of the head of the household on the household file. Now there is no repeating value, but it is still a poor idea, because the attribute is not really about the household. It is about a person in that household.

Multiple Data Sets

So what can be wrong with have multiple data sets with the same structure? Now we have to manage them, and that again will probably require macro code to keep the thing manageable. Note if it makes any sense to store the same kind of data in different files, there will be something to differentiate the files. For example, we might store each month's data in a separate file with names like Jan2000, Feb2000, etc. Once again we find that information has been hidden in names instead of retained in data values. Now it is the name of the data set, instead of the variable value, that knows the date of the information.

What if we want a maximum value? Now we either have to write a concatenation program, which is continually in need of modification, or we need to write macro code to loop over the collection of data sets. Either possibility is less desirable than a simple PROC SUMMARY to locate the maximum and a merge to locate the month in which it is found or a double SUMMARY to locate this information.

One may have to bend the rule here because of the sheer volume of data and the inability of current equipment to handle it. But, whenever the rule is broken you should be sure to have a very good reason for doing so, because the complexity must increase for the programs that process the data.

Examples

In a beginning macro class one student presented a 2,500 line program. Here is her first step. She reads in a set of transactions from an external file to be applied to a database where the data have been spread over 36 different data sets. Each of those data sets contains a subset of repeating variables. In other words, both the following rules were broken.

Don't store repeating values.

Don't spread the information about an entity over

many data sets.

Consequently the program is much more difficult to follow than the simple request to make some changes in data values would lead one to expect.

Two variables, ROOMCDE and COMP are read in to identify a variable. Transformation code is then used to obtain the name of the corresponding variable name. The name is then used to determine to which subset the transaction will be applied. The logic is fairly straightforward but the code is long and tedious as shown below.

```
data tochange
     d101 d102 d103 d104 d105 d106
     <skip many data sets>
     d307 d308 d309 d310 d311 d312
     dext01 dext02 dext03 dext04 dext05 ;
   infile mustchg ;
   input id $char9. roomcde $ comp $
         oldpb newpb ;

   length compnbr 8.;
   compnbr = comp;

   /* find the variable to change */
   if compnbr = 26 then
   do;
      if roomcde in ("A" "F" "D" "E") then
         varchg =
            trim(roomcde)||trim(comp)||"B";
      else
      if roomcde in ("B" "C") then
         varchg =
            trim(roomcde)||trim(comp)||"C";
      else
      if roomcde = "Q" then
         varchg =
            trim(roomcde)||trim(comp)||"E";
   end;
   else
      varchg =
         trim(roomcde)||trim(comp)||"E" ;

   /*split based on roomcde & compnbr range
   */
   if roomcde = "A"
      and (1 <= compnbr <= 5) then
         output d101;
   else
   if roomcde = "A"
      and (6 <= compnbr <= 10) then
         output d102;
   else
   <etc. for many lines>
   else
   if roomcde = "E"
      and (21 <= compnbr <= 25) then
         output d311;
   else
   if roomcde = "E" and (compnbr = 26) then
         output d312;
   Else
   if roomcde = "Q"
      and (1 <= compnbr <= 5) then
          output dext01;
   else
   if roomcde = "Q"
      and (6 <= compnbr <= 10) then
          output dext02;
   else
   if roomcde = "Q"
      and (10 <= compnbr <= 15) then
          output dext03;
   else
   if roomcde = "Q"
       and (16 <= compnbr <= 18) then
          output dext04;
   else
   if roomcde = "Q"
       and (24 <= compnbr <= 25) then
          output dext04;
   else
   if roomcde = "Q" and (compnbr = 26) then
      output dext05;
   else
      output tochange;
run ;
```

Now that the transactions have been split up into their respective data sets they can be applied one at a time to their corresponding data set in the database. Each application uses two DATA steps.

The first collapses the information to one record per id using a flag with values 1 or -1 to indicate a change in the corresponding variable named by VARCHG. Here is one example.

```
/* Array data to one record per id */
data d101c (drop = roomcde comp oldpb newpb
                   compnbr varchg);
   set d101;
   by id notsorted;

  retain flg_a1 flg_a2 flg_a3 flg_a4 flg_a5

  ;

   if first.id then do;
      flg_a1 = 1;
      flg_a2 = 1;
      flg_a3 = 1;
      flg_a4 = 1;
      flg_a5 = 1;
   end;

   /* change flg_a1 - flg_a5 to -1 for
      corresp varchg */
```

```
   select (varchg) ;
      when ("A1E") flg_a1 = -1;
      when ("A2E") flg_a2 = -1;
      when ("A3E") flg_a3 = -1;
      when ("A4E") flg_a4 = -1;
      when ("A5E") flg_a5 = -1;
      otherwise put _all_;
   end;

   if last.id;

run;
```

The second step then merges with the corresponding data set in the database and changes the corresponding values by prepending a minus sign when the flag is -1.

```
/* Merge and update matching records */
data xrfquex1.d01u
    (drop = flg_a1 flg_a2 flg_a3 flg_a4
            flg_a5);
   merge
      xrfquex1.d01 ( in = olddata)
      d101c (in = newval)
   ;
   by id;

   /*prepend negative sign to matching data
   */
      if olddata and newval then do;
         if flg_a1 = -1 then
            A1E = "-" || trim(A1E);
         if flg_a2 = -1 then
            A2E = "-" || trim(A2E);
         if flg_a3 = -1 then
            A3E = "-" || trim(A3E);
         if flg_a4 = -1 then
            A4E = "-" || trim(A4E);
         if flg_a5 = -1 then
            A5E = "-" || trim(A5E);
      end;

   if olddata;

run;
```

Since the first step split the transactions into 36 data sets there were 36 corresponding pairs of steps for a total of 72 DATA steps. Altogether, the original code took up 2,436 lines. As you can see this code took a great deal of effort to plan and write.

Sure, some macro code can help. For example, I reduced the pair of steps to a general macro. The code appears more difficult than necessary. However, I have only shown you a typical step having 5 flags; not the atypical steps that have just 1 flag. The problem arises from dividing 26 flags into 6 data sets.

```
%macro cmp_mrg
   ( data = d101,
     l1 = A,
     l2 = E ,
     beg = 0    /* 0, 5, 10, 15, 20, 25 */
) ;

%local i rng lib mem ;

%if &beg = 25 %then %let rng = 1 ;
%else
    %let rng = 5 ;
%let lib = %substr(&data,2,1) ;
%let mem = %substr(&data,3) ;

/* ------------------------------------
   array transactions to one obs per id
   ------------------------------------
*/

data &data.c (drop = roomcde comp oldpb
                     newpb compnbr varchg);
  set &data;
  by id notsorted;

  retain
     %do i = &beg + 1  %to &beg + &rng ;
         flg_&l1&i
     %end ;
  ;

     if first.id then do;
       %do i = &beg+1 %to &beg+&rng ;
          flg_&l1&i = 1;
       %end ;
     end;

     /* change flag to -1 for
        corresponding. varchg */
     select (varchg) ;
        %do i = &beg+1 %to &beg+&rng ;
           when ("&l1&i&l2")
              flg_&l1&i = -1;
        %end ;
        otherwise put _all_;
     end;

     if last.id;

run;

/* ------------------------------------
   Merge and update matching records
   ------------------------------------
*/

data xrfquex&lib..
     d&mem.u (drop =
          %do i = &beg+1 %to &beg+&rng ;
              flg_&l1&i
          %end ;
              );
```

```
      merge
         xrfquex&lib..d&mem ( in = olddata)
         &data.c (in = newval)
      ;
      by id;
      if olddata;

      /* if match, prepend minus sign to
         old value */
      if olddata and newval then do;
         %do i = &beg + 1 %to &beg + &rng ;
            if flg_&l1&i = -1 then
              &l1&i&l2="-"||trim(&l1&i&l2);
         %end ;
      end;

   run;

%mend  cmp_mrg ;
```

As you can see macro helped to reduce the amount of code, but it was still rather tedious to write.

Now what made it so difficult to update these data? Three rules were broken:

1. Data were stored at the ID level necessitating the arraying of the data.
2. These arrayed data were then split into numerous subsets.
3. Variable names were constructed to hold information - the room code, comp number and a subsequent letter.

In a sense, breaking the third rule was a simple consequence of arraying the data. Originally there were room codes, comp numbers, and end letters. When the data were forced to the ID level, these values had to become part of the variable name structure. The information had to go somewhere. It went into the variable names. The numerous subsets just added to the problem making either complex macro code necessary or an extremely long program. To assess how much damage was actually done we should look at the code it takes to maintain this information when properly stored.

First note that with all of this code we did not actually change values from OLDPB to NEWPB, we only prepended minus signs on the appropriate variables. Presumably that was all that was necessary in this case. Suppose we had actually wanted to change values as indicated by the values of OLDPB to NEWPB, where the change might be anything. Then the problem would have been even more difficult.

The student was amazed and extremely pleased, when she saw how learning macro would allow her to reduce the amount of code needed to maintain this database, but she merely shrugged her shoulders, when I suggested that the storage structure was at fault. The situation was not unusual; the programmer often does not have control of how the data are stored. But that should not prevent asking how much damage was done.

We have no such restrictions. Let us assume the data are not stored at the ID level, but rather at the ID, ROOMCDE, COMPNBR, LETTR level. Now the data can be stored in one data set. The transactions can be read into one transaction data set, sorted, and then applied in an UPDATE data step.

Here is the code.

```
/*---------------------------------------
  get transactions and assign lettr
  use oldpb & newpb to update xrfquex.umstr
  ---------------------------------------
*/
data trans ;

   infile mustchg ;
   input id :$char9.
         roomcde :$char1.
         comp :$char2.
         oldpb
         newpb
   ;
   /*assign lettr based on roomcde & comp*/
   if comp = "26" then do;
     if roomcde in ("A" "F" "D" "E") then
        lettr = "B" ;
     else
     if roomcde in ("B" "C") then
        lettr = "C" ;
     else
     if roomcde = "Q" then lettr = "E" ;
   end;
   else
     lettr = "E" ;

run ;

proc sort data = trans ;
   by id roomcde comp lettr ;
run ;

/* ---------------------------------------
   update master with checking
   ---------------------------------------
*/
data xrfquex.umstr ( drop = oldpb newpb )
     errs
   ;
   update xrfquex.mstr ( in = mstr )
          trans  ( in = trans )
   ;
   by id roomcde comp lettr ;
   if mstr then do ;
      if trans then do ;
```

```
        if oldpb = val then val = newpb ;
        else
           output errs ;
      end ;
      output xrfquex.umstr ;
   end ;
   else
       output errs ;
run ;
```

All it takes is two short DATA step and a sort; no macro code, no 2,500 lines of tedious code. So now, you see; we have been looking at a simple database system gone incredibly wild because of the way the data were stored.

Another Case

This time let's look at a problem raised on SAS-L. The question was how to read data crated by a C program that reported data taken from an instrument reading. The data had the form

```
4 3
4.39 1 2 3 4 2 3 4 5 9 w 0 4 8 9 2
2.12 4 3 2 1 9 8 7 6 9 c 3 4 9 2 1
3.45 6 7 8 9 1 2 3 4 9 b 2 1 3 8 9
3 2
2.68 6 7 2 5 4 8 6 c 8 9 2 4 6
1.35 6 8 9 3 5 8 6 f 2 8 3 4 6
```

Each record with two numbers indicated the beginning of a new month. The first number indicated the size of a T and an X array. The second number indicated how many observations were in that month. Altogether there were 120 months of data and many observations per month. Each observation began with Y followed by the T values, then the X values, and some other information not required.

The person raising the question offered a macro that she was unsatisfied with because of the amount of data and the number of months. She felt there had to be a better way. Her macro generated a separate data set for each month.

Here is her macro.

```
%macro mondata(HOWMANY);
/* HOWMANY is how many months to do */
FILENAME datafile "data98.prn";

%LET LINE=0;
%LET OBS=0;

%DO i=1 %TO &HOWMANY;
  %LET LINE=%eval(&LINE+&OBS+1);
  DATA _NULL_;
    INFILE datafile MISSOVER LRECL=6000
      FIRSTOBS=&LINE OBS=&LINE;
    INPUT varnum obnum;
    CALL SYMPUT
       ('VAR',trim(left(put(varnum,8.))));
    CALL SYMPUT
       ('OBS',trim(left(put(obnum,8.))));
  RUN;

  DATA data&i;
    ARRAY t[&VAR];
    ARRAY x[&VAR];
    INFILE datafile MISSOVER LRECL=6000
           FIRSTOBS=%eval(&LINE+1)
           OBS=%eval(&LINE+&OBS)
    ;
    INPUT y t1-t&VAR x1-x&var;
  RUN;

%END;

%MEND mondata;
```

Note pattern here. By now you should begin feeling uneasy whenever anyone mentions an array of data and you should be ready to fight when many data sets are created. In this case, there is no great quantity of code, and what there is, is not overly complex. Of course, the analysis on this output might lead to difficulties, or it might be easy to handle with more macro code.

Suppose we change the output and make one data set with the variables

MON	- month
SEQ	- observation number within a month
V	- index to preserve order within original obs
Y	- the data
T	- the data
X	- the data

Note that MON,SEQ, and V are used to identify a data element, while Y, T, and X form the data elements identified. We have broken a rule. Clearly, Y does not depend on V, so it should be stored in a separate data set because it represents a different entity. At present, we can leave it because we don't know what is to be done with data, and we are not in a position to judge how bad the decision is. But at the slightest hint of a problem we should be prepared to make a second data set for Y entities.

I have added one more simplifying assumption. Instead of having all T's come before all X's I have assumed the order T followed by X followed by the next T and the next X etc. The assumption is reasonable because we are asking what causes the code to get difficult, and the order is one thing. It should have been no harder for the preceding program to present the data this way instead of the way it did.

Now here is the code to read the data into one data set.

```
data mon( keep = mon v seq y t x ) ;
   infile jw truncover ;
   input nv nobs test $char8. ;
   if test ^= "" then
   do ;       /* probe for wrong record */
      error "unexpected value" ;
      stop ;
   end ;
   mon + 1 ;
   do seq = 1 to nobs ; /* loop over obs */
      input y @ ;       /* holding line  */
                        /* until finished*/
      do v = 1 to nv ;  /* array loop    */
         input t x @ ;
         output ;
      end ;
      input ;          /* release line */
   end ;
run ;
```

Now there is no macro, and the data are read only once instead of 120 times. What about our assumption? We could have output the T values to one set and the X values to another and then merged them. So we saved an extra data pass and little code, but it was not a big problem either way.

Conclusion

I have tried to state in informal terms, common sense rules that control how complex the SAS programs that maintain a SAS database will be.

Of necessity the program examples have been long, tedious, and messy. That was the point, to see the consequences of poor data storage, so it could not have been otherwise.

If you can understand the principles and get control of how the data are stored or at least explicate to those in power the consequences of storage design, then you have a chance at writing simpler SAS code. In any case you should be in a better position to judge how much of your effort is because the problem is difficult and how much of it is because of the way the data are stored. Hopefully you will gain experience by asking this question every time you write a program, and eventually will influence the environment in which you work.

Once, when I brought the topic up on SAS-L, I asked for an estimate of how big a wasted effort was expended in maintaining poorly designed SAS databases. Two people responded. We came from different industries working for different companies, but I was pleased to see that our average estimates were all in the same ballpark - somewhere between 60 and 90 percent with exceptional cases going far higher.

The author may be contacted by mail at

Ian Whitlock
Westat
1650 Research Boulevard
Rockville, MD 20850

or by e-mail

whitloi1@westat.com