PROC SQL and Arrays:
The power-house behind data processing

Meera G. Kumar
sanofi aventis Bridgewater Crossings, NJ

ABSTRACT
This paper attempts to explain the power behind the combined usage of PROC SQL and Arrays. The task at hand was to loop through all the datasets in two different directories and to identify common variables across datasets and to unify the attributes (variable label, type, length and format) of the common variables to ensure consistency in the data submitted to the FDA.

Challenge: Conventional approach would require revisiting all labels already seen along with their formats and types. This could get very cumbersome, complex and time-consuming to code.

Solution (problem redefinition): A variable comes with multiple labels, types and format definitions. But a variable when visited can have only two values: New or Old. If the visited variable is old (already visited) then inherit old label, type and format definitions. If variable is new (not visited before), make its label, type and format definitions the standard for future instances of this variable.

The paper demonstrates the use of PROC SQL to count the number of datasets and the number of variables within each dataset. The key task of unifying the attributes is achieved by reading in the data into a multi-dimensional array and flagging the array elements to identify common variables with differing attributes. The use of multiple arrays facilitates the design of an efficient algorithm to minimize the number of comparisons made.

INTRODUCTION
In the pharmaceutical industry, there is quite often a need for looping through many variables across multiple datasets to check for missing values, evaluate consistency of parameters, and look for valid value ranges or even to calculate efficacy parameters using values from multiple domains. When there is such a need for iterative processing the best tool to know and to use is arrays.

WHY ARRAYS
The use of arrays immensely simplifies the task of looping through and checking a series of related variables or array elements. An alternative approach would be to do a bunch of if-then-else conditions, which in-turn would make the code unnecessarily long and cumbersome. Using a multi-dimensional array is very intuitive in that it helps you visualize the data as a grid and go through the motions of identifying and tagging the values of interest to you. Keeping all the required values within the grid helps minimize the number of comparisons made and thereby makes the code highly efficient. The code below shows how the dataset is read into a multi-dimensional array with the help of the POINT=option and NOBS=option. Please note that the entire code is not presented in this paper. Using the POINT=option creates a variable that indicates which observation is read in from the existing dataset. Using NOBS=option helps us tap into value of the number of observations in the dataset being read. This is used to set the bound for the outer loop. The input pointer moves to the next observation with each loop through and it stays there while the next loop runs through.

```plaintext
do i=1 to nobs;
   set trcol point=i nobs=nobs;
   do j=1 to dim(vars);
      bigname(i,j)=vars(j);
      bigtyp(i,j)=typs(j);
      bigfmt(i,j)=fmts(j);
   end;
   do j=1 to dim(lbls);
      biglbl(i,j)=lbls(j);
   end;
end;
```

Coders’ Corner
NESUG 2007
WHY SQL
PROC SQL, among its other functionalities, also provides an easy solution for processing a large number of variables by creating macro variables that can be used with arrays. It provides the flexibility to dynamically set the bounds for array processing. In this instance, there were over 70 datasets and the number of variables within each dataset varied from single digit to over 200. Using SQL makes the code transportable to other studies without much change. The following code demonstrates the use of macro variables to dynamically count and change the required attributes to the variable.

```sas
proc sql noprint;
  select left(put(count(memname),4.0)) into :dscnt from varlbl;
  select memname into :mem1 - :mem&dscnt from varlbl;
quit;

%macro lbl_vars;
%do j = 1 %to &dscnt;
  proc sql noprint;
    select left(put(count(name),4.0)) into :varcnt from  varlbl
    where memname="&&mem&j";
    select name into :name1 - :name&varcnt from varlbl
    where memname="&&mem&j";
    select label  into :lbl1  - :lbl&varcnt  from varlbl
    where memname="&&mem&j";
  quit;
  proc datasets library=&lib nolist;
    modify &&mem&j;
    label  %do  i=1 %to &varcnt;
      &&name&i = "&&lbl&i"
    %end;
  quit;
%end;
%mend lbl_vars;
%lbl_vars;
```

COMPLEXITY ANALYSIS
Once the groundwork, mentioned above, has been put in place, the next question in the mind is to figure out a computationally efficient algorithm. We were dealing with a database greater than one gigabyte.

- **Original Problem Definition**
  Let \( p(i) \) be the probability for parameter \( i \). Let them be ordered in such a way that \( p(i) \geq p(i+1) \) for all values of \( i \). So, on an average when the \( i^{th} \) parameter first appears, there would have been \( 1/p(i) \) already visited parameters. Therefore the number of comparisons needed on the average for the \( i^{th} \) parameter is \( 1/p(i) \).
• **Problem Re-definition:**
  When the \( i^{th} \) parameter is first visited we need to compare it only with distinct parameters that were visited (without duplications of earlier parameters) – that is parameters that are more probable than the parameter \( i \). So the number of comparisons is only \( i-1 \). The maximum comparisons would be for the least probable parameter and that is \( N-1 \), when the last entry to distinct parameter (\( N^{th} \) parameter) is made. After that the number of comparisons for every additional instance of parameter, \( i \), is simply \( i \).

![Diagram showing Next Parameter and 4 Comparisons Needed]

Already visited distinct parameters are a set of parameters that have been already visited not considering duplicate visitations. Once a parameter is visited all further visitations to that parameter will inherit properties from the parameter set of the first visit.

**CONCLUSIONS**

The main conclusion of the paper is to illustrate the immense power of arrays. Attempting a solution based on the primary definition of the problem meant that too many conditions had to be checked. Use of arrays provided an elegant way to solve the problem. Computational efficiency is always a trade-off between memory and processing. But, optimality of an algorithm hopes to minimize both memory as well as processing to achieve the goal. Since this algorithm stores only the visited parameters via the use of arrays its memory requirements are optimal. The benefit of this methodology is especially felt when dealing with very large datasets which in our case was in the order of Giga bytes.

**ACKNOWLEDGMENTS**

SAS is a Registered Trademark of the SAS Institute, Inc. of Cary, North Carolina.

I would like to thank my colleague, Jeffrey Cortez, for his valuable feedback and exchange of ideas.

**CONTACT INFORMATION**

Your comments and questions are valued and encouraged. Contact the author at:

Meera G. Kumar  
Sanofi-Aventis  
200 Crossing Boulevard, PO Box 6890  
Bridgewater, NJ 08807-0890  
Work Phone: (908)304-6478  
Email: meera.kumar@sanofi-aventis.com