Skinny to Fat: In search of the “Ideal” dataset
Jeanina Worden, Pacific Data Designs, Inc., San Francisco, CA

ABSTRACT
Imagine if you will, a data manager’s table, one that is tall, skinny, and beautifully normalized. It’s the super model of data and can easily be maintained with a healthy diet of DATA steps. But when analyzing data you might find the ideal lends more towards a different “model”, specifically short and fat tables. With SAS, given a “tall” table with numerous observations for a patient, it can be difficult to analyze data between rows. While creating a “short” dataset with all observations for a patient in one row can be accomplished with PROC TRANSPOSE, in multivariable datasets the number of TRANSPOSE statements can quickly become too much of a good thing. In this paper a macro is discussed that uses a dataset created from DICTIONARY.COLUMNS and CALL SYMPUT to identify and loop through each variable in the original dataset creating variable specific datasets for each one. Then the new datasets are merged together thus transforming your data from “skinny” to “fat”. The macro was written in version 9 and just needs some understanding of the dictionary tables, CALL SYMPUT and the SAS macro language to use.

INTRODUCTION
It may not be in fashion to advocate “fat” over “skinny” but, when it comes to data, the ideal depends on what you will do with the data. Tall, slender tables have several rows per patient with each representing a single event making them easy to store, update, and perform descriptive statistics. The short, plump tables have several events within a single row per patient so the data are easier to determine relationships between the events. While there are numerous ways to achieve this, this paper will focus on PROC TRANSPOSE. With PROC TRANSPOSE the structure of the dataset can be converted in either direction; fat to skinny, skinny to fat. However, a limitation of this procedure is, when working with a dataset with numerous variables, it becomes cumbersome. This paper was the result of a reoccurring need to transform the data from one shape to the other in cases of that type. It starts with showing a standard way, through numerous TRANSPOSE procedures and a MERGE, of reformatting a relatively simple, skinny dataset into a fat one. In contrast, it shows how a macro uses the DICTIONARY.COLUMNS dataset, CALL SYMPUT, one TRANSPOSE procedure within a loop and an “automated” MERGE that uses the EVAL function, to quickly and more efficiently do the same thing.

GETTING THE SKINNY
The sample dataset resulted from the importing of a spreadsheet that tracked the glucose levels for a patient during a surgery.

<table>
<thead>
<tr>
<th>ID</th>
<th>Date</th>
<th>Time</th>
<th>BG level</th>
<th>Drip rate</th>
<th>Drip Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>332200594</td>
<td>11/19/2003</td>
<td>12:45</td>
<td>169</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>332200594</td>
<td>11/19/2003</td>
<td>14:00</td>
<td>136</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>332200594</td>
<td>11/19/2003</td>
<td>15:00</td>
<td>136</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>332200595</td>
<td>11/19/2003</td>
<td>16:00</td>
<td>127</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>332200595</td>
<td>11/19/2003</td>
<td>17:00</td>
<td>128</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Each row represents an event at which the glucose was taken, noting the insulin drip at the time and if there had been a change since the previous recording. The dataset can easily be queried and frequencies and means obtained for the BG_LEVEL, DRIP_RATE, and DRIP_CHANGE. But what if you wanted that information for a single patient? Or you wanted to calculate the time patients stayed within a designed glucose range? With this format it would be difficult; therefore the dataset needs reformatted so all the information for a patient would be on one row.

A SUNDAE A DAY (OR SLOW AND STEADY)
One way to reshape the data is to use PROC TRANSPOSE. In its simplest form it can be used to change the observations into variables. So if you run:

```sas
proc transpose data=sugardata out=sugardata2;
by patid;
```
var surgdate surgtime bg_level drip_rate drip_change;
run;

The result would be:

<table>
<thead>
<tr>
<th>PatID</th>
<th>NAME_</th>
<th>COL1</th>
<th>COL2</th>
<th>COL3</th>
</tr>
</thead>
<tbody>
<tr>
<td>332200594</td>
<td>SurgDate</td>
<td>11/19/2003</td>
<td>11/19/2003</td>
<td>11/19/2003</td>
</tr>
<tr>
<td>332200594</td>
<td>SurgTime</td>
<td>12:45</td>
<td>14:00</td>
<td>15:00</td>
</tr>
<tr>
<td>332200594</td>
<td>BG_level</td>
<td>169</td>
<td>136</td>
<td>136</td>
</tr>
<tr>
<td>332200594</td>
<td>Drip_rate</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>332200594</td>
<td>Drip_Change</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>332200595</td>
<td>SurgDate</td>
<td>11/19/2003</td>
<td>11/19/2003</td>
<td></td>
</tr>
<tr>
<td>332200595</td>
<td>SurgTime</td>
<td>16:00</td>
<td>17:00</td>
<td></td>
</tr>
</tbody>
</table>

... and so on

This is fine for single variable analysis, but it still doesn’t get all the information on a single row. Typically to “fatten up” a dataset for analysis requires a three-step process. First we sort the dataset by the identifying variable, in this case the PATID.

proc sort;
by patid;
run;

Then we use the TRANSPOSE procedure for each variable that are to be included in the final dataset. By using the PREFIX option the variable can be distinguished in the final dataset, and the OUT option creates a dataset for each variable:

For the date variable:

proc transpose data=sugardata2 prefix=surgdate out=dt;
by patid;
var surgdate;
run;

For the time variable:

proc transpose data=sugardata2 prefix=surgtime out=tm;
by patid;
var surgtime;
run;

For the glucose level:

proc transpose data=sugardata2 prefix=bg_level out=bgl;
by patid;
var bg_level;
run;

For the drip rate:

proc transpose data=sugardata2 prefix=drip_rate out=driprt;
by patid;
var drip_rate;
run;

For the change variable:

proc transpose data=sugardata2 prefix=drip_change out=dripchg;
by patid;
var drip_change;
run;
The final step would be to MERGE each of the individual datasets together by the identifying variable used in the TRANSPOSE procedures.

```sas
data sugar3;
merge dt bgl drip dt dripchg;
by patid;
drop _name_
run;
```

With just a few variables it’s apparent this process will get to the “Great Rectangle of Truth” though it is time consuming and hard to maintain. Each variable must have its own dataset created, and the dataset names are hard-coded in the MERGE statement requiring new variables to be added in numerous places.

**SUPER SIZED DATA!**

Using this macro is like indulging in a daily super sized calorie load: it quickly plumps up the dataset with little additional effort. Before the macro gets started, a dataset is created using the `DICTIONARY.COLUMNS` table. `DICTIONARY` tables are special read-only PROC SQL tables. They retrieve information about all the SAS data libraries, SAS data sets, SAS system options, and external files that are associated with the current SAS session. The `DICTIONARY.COLUMNS` table has information about the columns that is similar to the result of running a `PROC CONTENTS`. The following variables are available in this table:

- `LIBNAME` : Library Name
- `MEMNAME` : Member Name
- `MEMTYPE` : Member Type
- `NAME` : Column Name
- `TYPE` : Column Type
- `LENGTH` : Column Length
- `NPOS` : Column Position
- `VARNUM` : Column Number in Table
- `LABEL` : Column Label
- `FORMAT` : Column Format
- `INFORMAT` : Column Informat
- `IDXUSAGE` : Column Index Type
- `SORTEDBY` : Order in Key Sequence
- `XTYPE` : Extended Type
- `NOTNULL` : Not NULL
- `PRECISION` : Precision
- `SCALE` : Scale
- `TRANSCODE` : Transcoded

For more specific definitions, please see the `DICTIONARY.COLUMNS` documentation. Some of these variables are used within the SELECT of the SQL statement in order to create the VARFLDS dataset that holds the column information within the SUGARDATA dataset. Like this:

```sas
proc sql;
create table varflds as
select libname, memname, name, type, length, label
from dictionary.columns
where libname='WORK' and memname = upcase('Sugardata');
quit;
```

The resulting dataset, VARFLDS, would contain:

<table>
<thead>
<tr>
<th>LIBNAME</th>
<th>MEMNAME</th>
<th>NAME</th>
<th>TYPE</th>
<th>LENGTH</th>
<th>LABEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORK</td>
<td>SUGARDATA</td>
<td>PATID</td>
<td>char</td>
<td>8</td>
<td>Patient ID</td>
</tr>
<tr>
<td>WORK</td>
<td>SUGARDATA</td>
<td>SurgDate</td>
<td>char</td>
<td>8</td>
<td>Surgery Date</td>
</tr>
<tr>
<td>WORK</td>
<td>SUGARDATA</td>
<td>SurgTime</td>
<td>char</td>
<td>8</td>
<td>Surgery Time</td>
</tr>
<tr>
<td>WORK</td>
<td>SUGARDATA</td>
<td>BG_level</td>
<td>num</td>
<td>8</td>
<td>BG Level</td>
</tr>
<tr>
<td>WORK</td>
<td>SUGARDATA</td>
<td>Drip_rate</td>
<td>num</td>
<td>8</td>
<td>Drip Rate</td>
</tr>
<tr>
<td>WORK</td>
<td>SUGARDATA</td>
<td>Drip_Change</td>
<td>num</td>
<td>8</td>
<td>Drip Change</td>
</tr>
</tbody>
</table>
THE MEAT OF THE MATTER
Now the “meat” of the macro, the CALL SYMPUT routine uses the VARFLDS to create a macro variable. CALL SYMPUT allows the creation of SAS variables within a data step. Unlike the LET function, which is static, the CALL SYMPUT routine creates the variable dynamically from the dataset. The syntax for the routine is:

```
CALL SYMPUT(macro-variable, value);
```

MACRO-VARIABLE is the name of the macro variable to be initialized, and VALUE is the variable or data step expression to be used. From our VARFLD dataset we create a macro variable from NAME:

```
CALL SYMPUT('var'||left(_n_), trim(name));
```

The _N_ is the observation number of each NAME variable, so since PATID is the first NAME in the VARFLD dataset then VAR1=PATID, and VAR2=SURGDATE, and so on. Instead of creating just one macro variable, we are initializing a list of variables, one macro variable for each individual NAME. Without the _N_, however, each NAME would overwrite the previous one (resulting with VAR equal to the last variable name only).

With the initialized variables, instead of multiple PROC TRANSPOSE statements, each variable can be looped through one statement, one at a time. The original variable name is used to prefix the transposed variable names, as well as, to name the output dataset. The code looks like this:

```
%do i=1 %to &count;
  proc transpose data=sugardata prefix=&var&i out=fat.&var&i;
  by patid;
  var &&var&i;
  run;
%end;
```

One juicy tidbit you might take note of is that the new datasets are put into a new library. The answer to the question of why brings us to the next PROC SQL statement.

FAT LOADING
Once again, the DICTIONARY table COLUMNS is used but this time we keep the MEMNAME to get the dataset names in the FAT library. If we had used the same library as the original datasets those would be included in the MODNAME dataset, instead it will only contain the names of the transposed datasets.

```
proc sql;
create table modname as
  select distinct libname, memname
  from dictionary.columns
  where libname='FAT';
quit;
```

Again, too, we use CALL SYMPUT to create a list of variables called, MODULE(_n_), to hold the dataset names for the final loop.

```
data _null_; 
set modname end=fnl;
call symput('module'||left(_n_),trim(memname));
if fnl then call symput('max',_n_);
run;
```

The final loop merges the datasets together, but rather than using a MERGE statement each transposed dataset is “joined” into the new dataset, NEWSUGAR, using UPDATE. To do this NEWSUGAR must exist, so the first step is to verify it exists with the EXIST function. The EXIST function is similar to the FILEEXIST function in SAS, but while FILEEXIST checks for the presence of external data, EXIST verifies the presences of datasets in current libraries. The syntax is:

```
EXIST(member-name<,member-type<,generation>>)
```

Member-name represents the dataset name, member-type is the type of member it is (the default being DATA), and the generation refers to the generation number of the dataset (if type is not DATA then this is
The EXIST function returns 1 if the library member exists, or 0 if member-name does not exist. As the module names are processed through the loop, if it does not yet exist using SET with the current module will create it. Once it is created this dataset is used as the base to which the other transposed datasets are added. By using UPDATE, the NEWSUGAR dataset retains its contents while the next data is added.

%do j=1 %to &max;
%if %sysfunc(exist(sgf.newsugar)) = 0 %then %do;
data sgf.newsugar;
set fat.&module&j;
run;
%end;
%else %do;
proc sort data=fat.&module&j; by patid; run;
data sgf.newsugar;
update sgf.newsugar fat.&module&j;
by patid;
drop _label_ _name_; run;
%end;
%end;

CONCLUSION
And that’s it… by applying a rich diet of the SAS DICTIONARY table, COLUMNS, with sprinklings of some functions, like; CALL SYMPUT and EXIST, and a dash of statements like UPDATE, you have a perfect recipe to turn your dataset from skinny to fat. This paper introduces a macro built by getting; first, variable names, NAMES, from the DICTIONARY table COLUMNS. These are looped through a PROC TRANSPOSE statement after using CALL SYMPUT to create a macro variable for each variable. The PROC TRANSPOSE uses the variable name as a prefix for the transposed variables and for the name of the output dataset. The datasets are output to a new library to keep them separate from the original dataset. Then the DICTIONARY:COLUMNS dataset is used for a second time to get the names of the transposed datasets, MEMNAME. The final loop first checks for the existence of the final dataset, NEWSUGAR, and if it does not exist, sets the current dataset as NEWSUGAR. If the final dataset does exist the current dataset is used to UPDATE the NEWSUGAR dataset.

REFERENCES


ACKNOWLEDGMENTS
The programming team at Pacific Data Designs, Inc. contributed extensively to the development of this paper. Their support is greatly appreciated.

CONTACT INFORMATION
Your comments and questions are valued and encouraged. Contact the author at:
Jeanina Worden
353 Sacramento, Ste 800
San Francisco, CA 94111
Phone: 415-776-0660
E-mail: jworden@pdd.net
proc datasets library=WORK kill nolist; quit;
proc datasets library=SGF nolist; delete newsugar; quit;

data sugardata;
  set SGF.sugardata;
run;

proc sql;
create table varflds as
  select libname, memname, name, type, length, label
  from dictionary.columns
  where libname='WORK' and memname = upcase('Sugardata');
quit;

%macro Skinny_2Fat;
* --- Create a transposed dataset for each variable other;
data _null_;   
set varflds end=last;
call symput('var'||left(_n_),trim(name));
if last then call symput('count',_n_);
run;

%do i=1 %to &count;
proc transpose data=sugardata prefix=&var&i out=fat.&var&i;   
  by patid;
  var &var&i;
run;
%end;
* --- Put the transposed datasets together into a new dataset;
proc sql;
create table modname as
  select distinct libname, memname
  from dictionary.columns
  where libname='FAT';
quit;
data _null_;   
set modname end=fnl;
call symput('module'||left(_n_),trim(memname));
if fnl then call symput('max',_n_);
run;

%do j=1 %to &max;
%if %sysfunc(exist(sgf.newsugar)) = 0 %then %do;
data sgf.newsugar;
  set fat.&module&j;
run;
%end;
%end;
%else %do;
proc sort data=fat.&&module&&j; by patid; run;
   data sgf.newsugar;
      update sgf.newsugar fat.&&module&&j;
      by patid;
      drop _label_ _name_
   run;
%end;
%end;

%mend Skinny_2Fat;
options mprint mlogic;
%Skinny_2Fat;