SESUG Paper 211-2019

When in Doubt, Shell It Out

Timothy Egan, IQVIA

ABSTRACT

Sometimes a table shell appears simple at first glance, but can quickly grow complicated when it comes time to program. Inconsistent blank lines that are needed within and between sections of the table, unpopulated categories that will not be produced with the FREQ procedure but still need to appear in the final table, future visits that still need to be presented, and inconsistent indentations are all subtle details that can challenge a programmer when they sit down to knock out a table. While the DATA step and REPORT procedure are perfectly valid approaches to create these tables, there are simpler methods in SAS® that will allow a programmer to easily build these obscure or repetitive table structures. A key advantage to using these methods is that if the table structure needs to be updated at a later time, the program is already in a ready-to-update state for either the original programmer or someone else who has inherited the program.

This paper will show how to utilize the DATA step, DATALINES, or the SQL procedure to create a complicated or repetitive table structure with a simple and easy to read program. It will also examine strengths and weakness of the different methods which will enable programmers to select the more efficient choice on a case-by-case basis.

INTRODUCTION

The main goal of writing any program is to simply create something that works. Once a programmer is able to achieve this task, other goals present themselves. Is this program robust? Could it require updates if the data changes? If updates are required, is it clear where the updates are needed? The list goes on. Creating a table applies to many if not all SAS® programmers in any field. Regardless of the complexity of the table, a majority of the effort put into writing the program should focus on presenting the correct numbers. While spacing, line breaks, and other formatting can be immensely important to the final product, there is no need to over-complicate a table program when the correct numbers are just one or two SAS procedures away.

This paper will cover various simple methods to create table shells that will allow you to focus on the important part of a table, the numbers. These tables are created by creating the appropriate shell, pulling the desired numbers from the data, and then merging or joining the numbers onto the shell at the end of the program. The main engines that will be used to create these shells will be the DATALINES statement, the SQL procedure, and the FREQ procedure. It should be noted that the purpose of this paper is not to teach the syntax of these engines. These methods can be used by programmers of any skill level, in any field. The examples in this paper will be pulled from the clinical trial.

THE MOTIVATION BEHIND PROGRAMMING TABLE SHELLS

When considering tables, there are two main types: static and dynamic. A static table is a table whose structure will never change regardless of the data cut. The number of rows and columns will be the same for each run of the table. An example of a static table is a demographic table that presents descriptive statistics on the population of interest. If the number of subjects in the table increases, the number of rows and columns will not increase. Instead, the subjects will just be counted into the rows and columns that already exist. On the other hand, a dynamic table can change with each run of the table. Most of the time the number of columns will not change, but it is very common for the number of rows to increase as more data gets recorded throughout the study. Dynamic tables very often present event-based data, meaning the data is only recorded when the event takes place. Take for example a table that is supposed to display descriptive statistics for laboratory parameters. If the table is programmed at the beginning of the study, it is likely that only a fraction of the scheduled visits will have occurred at that point in time. However, a well-written program should not need to be updated as more visits appear in the

data. Using a dynamic shell for these tables is an effective way to ensure that a program written at the beginning of the study will still work correctly by the end of the study.

One very powerful way to program any table is to start by creating a table shell. The purpose of this shell is to create a dataset that contains the total number of sections and rows that are expected in the final table. Once this shell is created in one dataset and the numbers of the table are stored in one or more additional datasets, the actual numbers of the table can be merged or joined onto this table shell and your table will be ready for the REPORT procedure. When you take the techniques from this paper and apply them to your table programming, the main steps will be:

1. Program the table shell with the methods from this paper.
2. Program the actual numbers of the table.
3. Merge or join the numbers of the table with the table shell.

CREATING A STATIC TABLE SHELL

The first example in this paper will show how to create a static table shell. Take for example Table 1 below. This table needs to display a number of statistics for an individual parameter. Although more data will be collected throughout the course of the study, the number of rows and columns within this table will never change based on the data. As a result, a static table shell would be the shell of choice for creating this table. Two methods for creating a static shell are the DATALINES statement and PROC SQL. Each method has strengths and weaknesses that will be covered.

Table 1 needs to display multiple statistics for a single parameter.

Table X.X.X
Title 1

Population

Parameter Measure	Final Model		Estimate (95% C.I.)*	P-value**
Parameter Score	AIC: XXXX.XX			
	Slope for:			
	Baseline Parameter Score		X.XX (X.XX, X.XX)	X.XXX
	Covariates in the Final Model			
	Age (Years)		X.XX (X.XX, X.XX)	X.XXX
	Gender		X.XX (X.XX, X.XX)	X.XXX
	Center	AUS vs USA	X.XX (X.XX, X.XX)	X.XXX
		CHE/DEU vs USA	X.XX (X.XX, X.XX)	X.XXX
		GBR vs USA	X.XX (X.XX, X.XX)	X.XXX

Table 1. Table of statistics for one parameter

CREATING A SHELL WITH THE DATALINES STATEMENT

The first way to create a static shell is with the DATALINES statement. While some programmers view DATALINES as a primitive way of inputting data into SAS, it can be used as a sneaky way to create a static shell with ease. The code below will create the shell that you will need for this table:

```
data shell1_nortf;
  length col2 col3 $200;
  infile datalines delimiter='|';
  input sort subsort col2 $ col3 $;
  datalines;
1 | 1 | AIC =                          |
2 | . | Slope for:                     |
2 | 1 | Baseline Score                 |
3 | . | Covariates in the Final Model |
3 | 1 | Age (Years)                    |
3 | 2 | Gender                         |
3 | 3 | Center                         | AUS vs USA
3 | 4 |                                | CHE/DEU vs USA
3 | 5 |                                | GBR vs USA
;
run;

data shell1;
    set shell1_nortf;
    col2=tranwrd(col2,"Baseline","&_LineIndent1.Baseline");
    col2=tranwrd(col2,"Age","&_LineIndent1.Age");
    col2=tranwrd(col2,"Gender","&_LineIndent1.Gender");
    col2=tranwrd(col2,"Center","&_LineIndent1.Center");
run;
```

The final shell, `work.shell1`, contains two numeric variables and two character variables. The numeric variables will not appear in the final table and have only been added for sorting purposes. It is worth noting that the sorting of this final table can easily be accomplished with only one numeric variable. The reason that two variables are used is because Table 1 has multiple sections that are separated by line breaks. In the code above, the first sort variable `sort` differentiates the sections of the table and will simplify the code needed to create line breaks in the final output. While `sort` establishes the different sections of the table, `subsort` creates an additional sort within each section of the table. Again, it should be noted that while some rows have `subsort` set to missing, setting it to zero would be just as effective for the final sorting of the table.

As mentioned, using DATALINES has its strengths and weaknesses. An advantage of this method is that it is easy to visualize the final shell as you build it. While loops can be used to build repetitive shells, this particular table shell is short but complex so the DATALINES method is an excellent way to picture the final shell as you build it. A disadvantage of this method is that the DATALINES statement cannot be used within a macro. Additionally, if you are trying to include macro variables within your table shell, such as the RTF indentation code above, they will not resolve as expected. Instead, the macro variables must be added in later, which is why this piece of code requires a DATA step after `work.shell1_nortf`.

CREATING A SHELL WITH THE SQL PROCEDURE

You can also use PROC SQL in a manner similar to DATALINES to create the same table shell. This method uses PROC SQL to create an identical table shell that again includes two numeric variables and two character variables:

```
proc sql;
    create table shell2 (sort num, subsort num, col2 char(200), col3
    char(200));
    insert into shell2
          values(1,1,"AIC =",                          "")
          values(2,.,"Slope for:",                     "")
          values(2,1,"&_LineIndent1.Baseline Score",   "")
          values(3,.,"Covariates in the Final Model","")
          values(3,1,"&_LineIndent1.Age (Years)",      "")
          values(3,2,"&_LineIndent1.Gender",           "")
          values(3,3,"&_LineIndent1.Center",           "AUS vs USA")
          values(3,4,"",                               "CHE/DEU vs USA")
          values(3,5,"",                               "GBR vs USA")
          ;
quit;
```

Similar to the DATALINES method, PROC SQL is an excellent way to picture the final shell as you build it. An improvement from the DATALINES method is that this piece of code can be included in a macro, and your macro variables will resolve as expected. The main disadvantage to this method is that most programmers are less comfortable with PROC SQL, so the syntax may be unfamiliar. However, for less experienced SQL users who are looking to use more SQL code, this method can be a great starting point.

The two previous methods will both create the table shell found in Output 1. As mentioned earlier in this paper, this shell will be responsible for creating the structure of the table. Once the actual statistics are created appropriately, you can easily merge or join them onto this table shell with the `sort` and `subsort` variables to create the final output. It should be noted that each row has a distinct combination of `sort` and `subsort`. This is very important because if the row combinations are not distinct, the statistics will not merge onto the table shell properly.

Output 1 shows a print of the final shell created by the previous two methods.

The SAS System

Obs	SORT	SUBSORT	COL2	COL3
1	1	1	AIC =	
2	2		Slope for:	
3	2	1	¤R'\li340 'Baseline Score	
4	3		Covariates in the Final Model	
5	3	1	¤R'\li340 'Age (Years)	
6	3	2	¤R'\li340 'Gender	
7	3	3	¤R'\li340 'Center	AUS vs USA
8	3	4		CHE/DEU vs USA
9	3	5		GBR vs USA

Output 1. Static table shell created with DATALINES or PROC SQL

CREATING A DYNAMIC, DATA-DRIVEN TABLE SHELL

The following two examples will cover how to create a dynamic table shell that will not require updates throughout the study as new data continues to grow. By nature, these dynamic shells will need to be data-driven because unlike static shells, you do not always know how many rows you will need in the final table.

Consider Table 2 below. It does not indicate the number of expected parameters or the number of expected visits. In the clinical trial setting, this information can be found in other documentation which can clarify the ambiguity. In some cases you may want to present all parameters, but only a prespecified

number of visits. At other times, you may want to write a program that will present all parameters and all visits that are found within the data.

Table 2 needs to display multiple statistics for each parameter-visit combination.

Table X.X.X
Title 1

Population

Parameter	Analysis Visit	Statistic	Treatment A (N = XXX)	Treatment B (N = XXX)
<Parameter 1>	Baseline	N	XXX	XXX
		Mean (STD)	XX.X (XX.XX)	XX.X (XX.XX)
		Min - Max	XX - XX	XX - XX
	Week 2	N	XXX	XXX
		Mean (STD)	XX.X (XX.XX)	XX.X (XX.XX)
		Min - Max	XX - XX	XX - XX
	<Continue for remaining visits>	...	...	...
<Parameter 2>	Baseline	N	XXX	XXX
		Mean (STD)	XX.X (XX.XX)	XX.X (XX.XX)
		Min - Max	XX - XX	XX - XX
	Week 2	N	XXX	XXX
		Mean (STD)	XX.X (XX.XX)	XX.X (XX.XX)
		Min - Max	XX - XX	XX - XX
	<Continue for remaining visits>	...	...	...
<Continue for remaining parameters>	...	...	...	...

Table 2. Table of multiple statistics for unknown number of parameters/visits

CREATING A SHELL WITH THE FREQ PROCEDURE AND A DATA STEP

One way to create a dynamic shell for this table is to use PROC FREQ paired with a DATA step. The following code demonstrates this approach:

```
proc freq data=adam.advs noprint;
    tables paramcd*param / list missing out=shell3_par(drop=count percent);
run;

data shell3;
    set shell3_par;

    do i=1 to 3;
          avisitn=i;
          output;
    end;
run;
```

This method is a simple but elegant way to capture all parameters found within the data and then create a prespecified number of visits for each parameter. Using PROC FREQ, all distinct parameters are output into the dataset `work.shell3_par`. In the ensuing DATA step, a DO loop containing an OUTPUT statement is used to create one row for each visit, for each parameter. The total number of rows will be the number of parameters found by PROC FREQ multiplied by the `i` visits that are defined in the DO loop.

The method above has several strengths. For one, it is very data-driven. Each time this program is run, the table will reflect which parameters are found within the data. Another advantage is that it is very easy to control the number of visits that are displayed. This is helpful because throughout the life-cycle of a study, it is beneficial to be able to control the number of visits displayed in tables. With the code above, it is a matter of updating one character to increase or decrease the number of visits displayed. Finally, the code is very succinct and does not have many moving parts, making it simple for another programmer to confidently update.

CREATING A SHELL WITH THE SQL PROCEDURE AND A NATURAL JOIN

With the previous example in mind, this next method will operate similarly. However, the exception is that this method will be completely based on the data, and the number of visits will not be explicitly defined:

```
proc sql;
    create table paramcds as
          select distinct(paramcd), param
          from adam.advs;

    create table avisitns as
          select distinct(avisitn)
          from adam.advs
          ;

    create table shell4 as select a.*, b.*
          from paramcds as a natural join avisitns as b
          order by a.paramcd, b.avisitn
          ;
quit;
```

The basic idea behind this code is to select all parameters found in the dataset, select all visits found in the dataset, and then join them together with a NATURAL JOIN to create a row for every possible combination of parameter and visit. This method produces almost the same results as the following:

```
proc freq data=adam.advs noprint;
    tables paramcd*param*avisitn / list missing out=shell4(drop=count
    percent);
run;
```

With this in mind, you may ask, 'Instead of using PROC SQL to create two datasets and then join them together, why not just use PROC FREQ? It is a fraction of the code.' For many cases these 2 pieces of code will indeed produce the same results. However, the whole point of creating table shells in general is to create rows for the events that may not have occurred yet. If you use only PROC FREQ, it is possible that there will be a parameter-visit combination that does not exist within the data, but it still should be presented in the table. If this is the case for your data, using the longer section of SQL code will be the better option.

Again, the previous two methods will create the shell displayed in Output 2. This shell contains one record per visit, per parameter.

Output 2 shows a print of the final shell created by the previous two methods. Additional processing is required to add the additional rows for statistics found in Table 2.

The SAS System

Obs	PARAMCD	PARAM	AVISITN
1	DIABP	Diastolic Blood Pressure (mmHg)	1
2	DIABP	Diastolic Blood Pressure (mmHg)	2
3	DIABP	Diastolic Blood Pressure (mmHg)	3
4	PULSE	Pulse Rate (beats/min)	1
5	PULSE	Pulse Rate (beats/min)	2
6	PULSE	Pulse Rate (beats/min)	3
7	RESP	Respiratory Rate (breaths/min)	1
8	RESP	Respiratory Rate (breaths/min)	2
9	RESP	Respiratory Rate (breaths/min)	3
10	SYSBP	Systolic Blood Pressure (mmHg)	1
11	SYSBP	Systolic Blood Pressure (mmHg)	2
12	SYSBP	Systolic Blood Pressure (mmHg)	3
13	TEMP	Temperature (C)	1
14	TEMP	Temperature (C)	2
15	TEMP	Temperature (C)	3
16	WEIGHT	Weight (kg)	1
17	WEIGHT	Weight (kg)	2
18	WEIGHT	Weight (kg)	3

Output 2. Table shell created with previous pieces of code

Looking back at Table 2, you will notice that each visit and parameter combination will need 3 rows of statistics. This is where creativity comes into play, as there are several approaches available to turn the shell in Output 2 into the final shell. For instance, using another DATA step, you can read in the shell and use a DO loop combined with an OUTPUT statement (covered previously in this paper) to create three additional rows for each parameter-visit combination. Another way to add these statistic rows would be to create a static shell (covered in the first section of this paper) with three statistic rows, and then use a NATURAL JOIN with `work.shell4`.

MAKING A STATIC SHELL DYNAMIC

The end of the last example motivates the idea that at times, you may want to begin with a static shell and then turn it into a dynamic, data-driven shell. Take for example Table 3. If you notice, it is almost the same as Table 1, with the exception that instead of displaying statistics for one parameter, Table 3 needs to display the statistics for all parameters in the dataset.

Table 3 needs to display multiple statistics for multiple parameters.

Table X.X.X
Title 1

Population

Parameter Measures	Final Model		Estimate (95% C.I.)*	P-value**
<Parameter 1> Score	AIC: XXXX.XX			
	Slope for:			
	Baseline <Parameter 1> Score		X.XX (X.XX, X.XX)	X.XXX
	Covariates in the Final Model			
	Age (Years)		X.XX (X.XX, X.XX)	X.XXX
	Gender		X.XX (X.XX, X.XX)	X.XXX
	Center	AUS vs USA	X.XX (X.XX, X.XX)	X.XXX
		CHE/DEU vs USA	X.XX (X.XX, X.XX)	X.XXX
		GBR vs USA	X.XX (X.XX, X.XX)	X.XXX
<Parameter 2> Score	AIC: XXXX.XX			
	Slope for:			
	Baseline Attention score		X.XX (X.XX, X.XX)	X.XXX
	Covariates in the Final Model			
	Age (Years)		X.XX (X.XX, X.XX)	X.XXX
	Gender		X.XX (X.XX, X.XX)	X.XXX
	Center	AUS vs USA	X.XX (X.XX, X.XX)	X.XXX
		CHE/DEU vs USA	X.XX (X.XX, X.XX)	X.XXX
		GBR vs USA	X.XX (X.XX, X.XX)	X.XXX
<Continue for remaining parameters>				

Table 3. Table of statistics for multiple parameters

CREATING A SHELL USING A NATURAL JOIN WITH A STATIC SHELL

Conveniently, a fraction of Table 3 was already created at the beginning of this paper (see Output 1). If you knew that n parameters need to be displayed, you could simply use a DATA step to stack `work.shell2` n times. However, the goal of creating these shells is to let the data create the shell. With this in mind, you can combine previous techniques covered in this paper. The following code does just that:

```
proc sql;
    create table paramcds as
          select distinct(paramcd)
          from adam.advs
          ;

    create table shell5 as select a.*, b.*
          from shell2 as a natural join paramcds as b
          order by b.paramcd, a.sort, a.subsort
          ;
quit;
```

As mentioned, the code will reference `work.shell2` that was created at the beginning of this paper. The first SQL query will create the dataset `work.paramcds` that contains one row for each distinct parameter found in `ADAM.ADVS`. The second query will take `work.shell2` and perform a NATURAL JOIN with `work.paramcds` to create `work.shell5`. This final dataset shown in Output 3 is essentially the same result as if you were to stack `work.shell2` n times for each parameter found in `ADAM.ADVS`. The key advantage here is that if new parameters show up in the dataset, future runs on the data will create the necessary rows.

Output 3 shows a print of the final shell created by the previous SQL join.

The SAS System

Obs	PARAMCD	SORT	SUBSORT	COL2	COL3
1	DIABP	1	1	AIC =	
2	DIABP	2		Slope for:	
3	DIABP	2	1	¤R'\li340 'Baseline Score	
4	DIABP	3		Covariates in the Final Model	
5	DIABP	3	1	¤R'\li340 'Age (Years)	
6	DIABP	3	2	¤R'\li340 'Gender	
7	DIABP	3	3	¤R'\li340 'Center	AUS vs USA
8	DIABP	3	4		CHE/DEU vs USA
9	DIABP	3	5		GBR vs USA
10	PULSE	1	1	AIC =	
11	PULSE	2		Slope for:	
12	PULSE	2	1	¤R'\li340 'Baseline Score	
13	PULSE	3		Covariates in the Final Model	
14	PULSE	3	1	¤R'\li340 'Age (Years)	
15	PULSE	3	2	¤R'\li340 'Gender	
16	PULSE	3	3	¤R'\li340 'Center	AUS vs USA
17	PULSE	3	4		CHE/DEU vs USA
18	PULSE	3	5		GBR vs USA
.					
.					
.					
46	WEIGHT	1	1	AIC =	
47	WEIGHT	2		Slope for:	
48	WEIGHT	2	1	¤R'\li340 'Baseline Score	
49	WEIGHT	3		Covariates in the Final Model	
50	WEIGHT	3	1	¤R'\li340 'Age (Years)	
51	WEIGHT	3	2	¤R'\li340 'Gender	
52	WEIGHT	3	3	¤R'\li340 'Center	AUS vs USA
53	WEIGHT	3	4		CHE/DEU vs USA
54	WEIGHT	3	5		GBR vs USA

Output 3. Table shell containing multiple statistics for multiple parameters

CONCLUSION

While programming the numbers for tables may be complicated at times, programming the actual table structure itself never should be. With just a few techniques in your SAS toolkit, you should be able to create almost any table structure with relative ease. These techniques include the DATALINES statement, PROC FREQ, PROC SQL, and the DATA step. When preparing to create a table, first and foremost you need to determine if the table structure is static or dynamic. From there, decide if your table needs to be completely data-driven, or if certain sections will be prespecified such as the number of visits or number of parameters. Once these two questions are answered, the examples and explanations in this paper should help you identify exactly which method or methods you can use to create your table shell. Using these approaches will allow for easy updates by other programmers if the table structure needs to change in future runs. Once you have your robust table shell, you can spend the majority of your time and energy creating the important part of your table, the numbers.

CONTACT INFORMATION

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

Timothy Egan
IQVIA
Timothy.Egan@IQVIA.com
https://www.linkedin.com/in/timothyegan/