PharmaSUG 2013 - Paper BB10

Atypical Applications of the UPDATE Statement

John Henry King, Ouachita Clinical Data Services, Inc., Caddo Gap AR

ABSTRACT

This paper explores uses of the UPDATE statement as it relates to programming clinical trials data. The UPDATE statement is often overlooked as tool for clinical trials programming, indeed it is often difficult to see how its transaction processing paradigm is directly applicable to programming tables, figures, and listings. Most of us know MERGE and SET, and they along with SQL suffice for most applications. This paper attempts to show through examples how the UPDATE can be applied to clinical trials programming problems.

INTRODUCTION

The SAS® UPDATE statement is not often used in day to day clinical trials programming. The master data transaction data processing paradigm is not that common. If we want to write a program to balance a checkbook, or a thousand checkbooks, UPDATE is perfect; for each account process all the debits (subtract) and credits (add) adjusting the balance accordingly.

The SAS online documentation for the UPDATE statement describes the syntax as:

UPDATE
master-data-set<(*data-set-options*)>
transaction-data-set<(*data-set-options*)>
<END=*variable*>
<UPDATEMODE= MISSINGCHECK | NOMISSINGCHECK>;
BY *by-variable*;

There are only two data sets allowed quite different from SET and MERGE. The BY statement can include more than one variable but the master data set should be unique for the levels of the BY variables. The transaction data set can have any number of observations for each BY group. UPDATE has two properties in particular that distinguish it from SET and MERGE,

- missing check
- timing of output

Missing check, a property we will exploit to our advantage, works by checking all variables read from the transaction data for missing values and NOT overwriting existing values of the same variable with missing. By comparison MERGE will overwrite with any data, missing or not, from the data set listed to the right on the MERGE statement. OUTPUT is timed to process all observations from the transaction data and output a single observation for each unique BY group in the master. You might think of this as a sub-setting if statement “IF LAST.BYVAR;”. We can override the output timing with an explicit OUTPUT; statement and we will exploit that feature as well.

IMPORT EXCEL SHEET WITH MERGED CELLS

This example demonstrates using the UPDATE statement to fix up a data set imported from EXCEL that has merged cells.

	A	B	C	D	E
1	**Region**	**Subsidiary**	**Product**	**Stores**	**Sales**
2	Asia	Bangkok	Boot	1	1996
3			Men's Dress	1	3033
4			Women's Casual	1	5389
5			Boot	17	60712
6			Men's Casual	1	11754
7		Seoul	Men's Dress	7	116333
8			Sandal	3	4978
9			Slipper	21	
10			Sport Shoe	9	9745
11			Women's Dress	2	12601
12		Tokyo	Boot	25	40213
13	Canada	Calgary	Men's Casual	3	53929
14			Men's Dress	11	112009
15			Women's Casual	2	24497
16		Montreal	Women's Dress	12	132638
17			Boot	5	7892
18			Men's Casual	1	19210
19			Sandal	2	2600

Consider the EXCEL spread sheet shown to the left. It has two columns Region and Subsidiary with merged cells. This looks nice as a spread sheet but does not work well as a data base. When this sheet is read with PROC IMPORT

```
proc import
   datafile='shoes.xlsx'
   dbms=xlsx out=shoes;
   run;
```

the SAS data set looks just like the spreadsheet which is usually desired but in this case we need the values of REGION and SUBSIDIARY to be repeated for the merged rows.

We need to find a way to process this data and have the non-missing values of REGION and SUBSIDIARY fill in the missing values as we move from one observation to the next. We want the observations coalesced vertically.

One way to approach this problem would be to create new variables to receive the coalesced values of REGION and SUBSIDIARY and this is a very straight forward solution especially with the introduction of the COALSECEC and COALESCEN functions.

```
data shoesFixed1;
   if 0 then
      set shoes(keep=region subsidiary);
   set shoes(rename=(
      region=_region
      subsidiary=_subsidiary));

   region=coalesceC(_region,region);
   subsidiary=
      coalesceC(_subsidiary,subsidiary);
   drop _:;
   run;
```

This method is fine and works well enough; the technique involves these steps;

- Create variables using unexecuted SET
- Rename all of the variables to be coalesced in the executed SET statement
- Write assignments statements for each variable to be coalesced.
- Drop the old versions of the coalesced variables.

While not specific to this example we have to pay attention to the data type of each variable and use the appropriate coalesce function. With only two variables not a big deal but it is still a bit tedious.

We can take advantage of the features of the UPDATE statement to make this a bit less tedious. Because the default action of the UPDATE statement is to check transaction observations for missing values and NOT replace the current value of any variable with a missing value we can use that feature to "coalesce" the values of REGION and SUBSIDIARY. We will have to pay attention to variables that we don't want coalesced as in our example where SALES in observation 8 is missing and we don't want to it changed.

```
data shoesV / view=shoesV;
   retain one 1;
   set shoes;
   run;

data shoesFixed(drop=one);
   update shoesV(obs=0) shoesV;
   by one;
   output;
   call missing(of product--sales);
   run;
```

- Since the UDPATE requires a BY statement but we don't have one for this data we will need to add a variable that is constant on every observation. A SAS data step view is a convenient tool for this because it doesn't actually create any data until the view if referenced.
- For this operation we don't really a have master data so we will use the transaction data SHOEV as a placeholder master but not read any observations (OBS=0).
- The BY statement uses the dummy variable ONE created to act as the required BY variable. It is also dropped from the new data set.
- We don't want the default output from UPDATE which would be to output when the last observation for the BY variable is processed so an OUTPUT statement is added to output every observation read from the transaction action data.
- The missing check is applied to all variables but we don't want it applied quite so broadly, CALL MISSING is added to explicitly set to missing the variables we don't want coalesced. In this example a name range list is used to specify all variables between PRODUCT and SALES.

Region	Subsidiary	Product	Stores	Sales
Asia	Bangkok	Boot	1	1996
Asia	Bangkok	Men's Dress	1	3033
Asia	Bangkok	Women's Casual	1	5389
Asia	Bangkok	Boot	17	60712
Asia	Bangkok	Men's Casual	1	11754
Asia	Seoul	Men's Dress	7	116333
Asia	Seoul	Sandal	3	4978
Asia	Seoul	Slipper	21	.
Asia	Seoul	Sport Shoe	9	9745
Asia	Seoul	Women's Dress	2	12601
Asia	Tokyo	Boot	25	40213
Canada	Calgary	Men's Casual	3	53929
Canada	Calgary	Men's Dress	11	112009
Canada	Calgary	Women's Casual	2	24497
Canada	Montreal	Women's Dress	12	132638
Canada	Montreal	Boot	5	7892
Canada	Montreal	Men's Casual	1	19210
Canada	Montreal	Sandal	2	2600

As you can see from the table on the left the missing values have been coalesced into a properly formatted data set. Also note that SALES has not been affected because we explicitly set it to missing with the missing function.

LAST OBSERVATION CARRIED FORWARD

Obs	Subjid	Visit	X	Y	Z	Assessment
1	001	0	156	100	62	Normal
2	001	1	164	93	65	High
3	001	2	.	.	76	Undet
4	001	4	148	89	56	Normal
5	001	5	.	.	.	
6	002	0	176	120	62	High
7	002	1	164	93	65	Normal
8	002	2	156	.	.	
9	002	4	.	.	.	
10	002	5	.	.	.	
11	003	0	163	67	76	High
12	003	1	.	.	.	
13	003	2	.	.	.	
14	003	4	.	.	.	
15	003	5	.	.	.	
16	004	0	.	.	.	
17	004	1	190	90	89	High
18	004	2	.	.	99	
19	004	4	.	.	.	
20	004	5	.	.	.	

Think of the data shown here on the left as a representation of clinical trials data where the missing values of X, Y, Z and ASSESSMENT are to be treated with the LOCF technique. With an exception for VISIT=0 where we don't want to carry forward any value. The program follows here.

```
data locf;
   update observed(obs=0) observed;
   by subjid;
   output;
   if visit eq 0 then call missing(of _all_);
   run;
```

- The UDPATE statement is defined with a dummy master with zero observations.
- The required BY variable is SUBECT and when the BY variable changes the transaction processing is reset so no values are carried over from one subject to the next, e.g. SUBJID=004 does not get values for VISIT=0 from SUBJID=003.
- OUTPUT every observation; remember the default action would be to output only on LAST.SUBJECT.
- To accommodate the specification that VISIT=0 not be carried forward execute CALL MISSING to set all values to missing so that no non missing value will be carried forward.

Obs	Subjid	Visit	X	Y	Z	Assessment
1	001	0	156	100	62	Normal
2	001	1	164	93	65	High
3	001	2	164	93	76	Undet
4	001	4	148	89	56	Normal
5	001	5	148	89	56	Normal
6	002	0	176	120	62	High
7	002	1	164	93	65	Normal
8	002	2	156	93	65	Normal
9	002	4	156	93	65	Normal
10	002	5	156	93	65	Normal
11	003	0	163	67	76	High
12	003	1	.	.	.	
13	003	2	.	.	.	
14	003	4	.	.	.	
15	003	5	.	.	.	
16	004	0	.	.	.	
17	004	1	190	90	89	High
18	004	2	190	90	99	High
19	004	4	190	90	99	High
20	004	5	190	90	99	High

The new data shown here on the right has been successfully transformed us last of carried forward. Each non-missing value for each visit, sans VISIT=0, has been carried forward to the next observation if and only if the value in the next observation is missing. Again we have exploited the missing check feature of the UPDATE statement and the code does not explicitly reference to the variables being LOCFed. No tedious renaming or assigning needed.

COLLAPSING A DATA SET

This example is from a question that was recently asked on SAS-L with subject "collapsing a data set". Apparently this is a common problem because it has been asked many times over the years that I have participated in SAS-L.

Obs	LDB	ASM	NAC	y1	e1	y2	e2	y3	e3
1	1	10	100	9000	5	.	.	.	.
2	1	10	100	.	.	8500	4	.	.
3	1	10	100	.	.	.	.	9700	3
4	2	10	100	15000	3	.	.	.	.
5	2	10	100	.	.	24000	3	.	.
6	2	10	100	.	.	.	.	38000	4

The data on the left is the starting point and the desire is to collapse three rows into one for each level of LDB. The desired data set can be had with UPDATE.

```
data want;
   update have(obs=0) have;
   by ldb;
   run;
```

Notice that we do not include an explicit OUTPUT statement as in the previous examples, in this situation we want the default action of output on LAST.LDB. The result is one observation for each LBL as show below.

Obs	LDB	ASM	NAC	y1	e1	y2	e2	y3	e3
1	1	10	100	9000	5	8500	4	9700	3
2	2	10	100	15000	3	24000	3	38000	4

CONCLUSION

We've looked at some examples of using the UPDATE statement maybe they are helpful examples to inspire you to try UPDATE.

CONTACT INFORMATION

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

John King
Ouachita Clinical Data Services, Inc.
1769 Highway 240 West
Caddo Gap, AR 71935
870-356-3033
datanull@gmail.com: