

A FIRST. (but not LAST.) Word on BY-Group Processing

Christine A. Smiley, J.C. Penney Company, Plano, TX

ABSTRACT

This paper details the use of BY-group processing within the SAS® system. Learn details about using BY statements with procedures. Discover how to use the FIRST. and LAST. variables. Learn some common mistakes to avoid. See a comparison between BY-group processing and PROC SQL. Many examples are provided, highlighting how to summarize purchase data to a customer level. This paper is appropriate for users who are familiar with SAS DATA steps but who have limited exposure to the BY statement and the corresponding special grouping variables.

OUTLINE

What is a BY-group

BY Statement in Procedures

- DESCENDING & NOTSORTED Options
- System Option BYLINE
- Multiple BY statements
- Compared to CLASS Statement

BY Statement in DATA Step

- With SET Statement

BY Statement to Roll Up Data

- DATA Step Iteration
- RETAIN Statement
- Sum Statement
- FIRST. and LAST. Variables
- Creating a Sum for a BY-group

Other BY-group Processing Tips

- Beware the Subsetting IF
- Identifying Duplicates
- Multiple BY-groups

Comparison to PROC SQL

Conclusions

WHAT IS A BY-GROUP

A BY-group is a subset of observations within a SAS data set that share the same value on a particular variable (called the BY variable). The data set must be sorted, grouped, or indexed by the BY variable (or variables). Unless otherwise specified, SAS expects the data to be in ascending numeric or alphabetic order. As an example, if data set *cities* is sorted by the variable *state*, then

the observations for each separate state form a BY-group.

BY STATEMENT IN PROCEDURES

A BY statement can be used in many SAS procedures, generally to perform computations separately for each BY-group. The BY statement in PROC SORT specifies the variable(s) to sort by. Using a BY statement in PROC PRINT will produce a header line at the start of each BY-group.

```
proc print data=cities;
    by state;
```

state=AR

Obs	statename	city	population
1	Arkansas	Conway	43000
2	Arkansas	Sunset	348

state=AZ

Obs	statename	city	population
3	Arizona	Flagstaff	53000
4	Arizona	Phoenix	1300000

state=TX

Obs	statename	city	population
5	Texas	Carrollton	109000
6	Texas	Lewisville	78000
7	Texas	Plano	222000

Notice how the BY variable itself is printed only in the header and not with the other variables. If you want to force it to print with the other variables, use a VAR statement.

```
proc print data=cities;
    by state;
var state statename city population;
```

Here is a portion of the listing:

```
state=AR
Obs state statename city population
1 AR Arkansas Conway 43000
2 AR Arkansas Sunset 348
```

Here we use a PROC MEANS with a BY statement to compute figures separately for each state.

```
proc means data=cities;
  by state;
  var population;
```

Here is a portion of the output.

```
state=AR
Analysis Variable : population
```

N	Mean	Std Dev	Minimum	Maximum
2	21674	30159.52	348	43000

A BY statement can be used with the following BASE SAS procedures: CALENDAR, CHART, COMPARE, CORR, FORMS, FREQ, MEANS, PLOT, PRINT, RANK, REPORT, STANDARD, TABULATE, TIMEPLOT, TRANSPOSE, and UNIVARIATE. For some procedures, each BY-group will start on a new page.

DESCENDING & NOTSORTED OPTIONS

If a data set is sorted in descending or reverse order, the DESCENDING option must be placed in the BY statement before the variable name. Sometimes observations are grouped by the BY-group variable, but are not in a strict ascending or descending order. In such a case, use a BY statement with the NOTSORTED option at the end.

For example, the data set *cities* might be sorted by the two character state postal abbreviation (variable *state*), but for the PROC MEANS output we want to display the full state name (variable *statename*). Statenames are in a different order than *state* (AR is before AZ, but Arizona sorts before Arkansas). Without the NOTSORTED option, this PROC MEANS would produce an error.

```
proc means data=cities;
  by statename notsorted;
```

Here is a portion of the output:

```
statename=Arkansas
Analysis Variable : population
```

N	Mean	Std Dev	Minimum	Maximum
---	------	---------	---------	---------

2	21674	30159.52	348	43000
---	-------	----------	-----	-------

```
statename=Arizona
Analysis Variable : population
```

N	Mean	Std Dev	Minimum	Maximum
2	76500	881762.16	53000	1300000

With the NOTSORTED option, most procedures define a BY-group as a set of contiguous observations with the same value on the BY variable(s). The example below shows the results when neighboring observations do not have the same value on the BY variable.

```
proc sort data=cities; by city;
```

```
proc print data=cities;
  by statename notsorted;
```

```
statename=Texas
Obs  state  city      population
1    TX    Carrollton  109000
```

```
statename=Arkansas
Obs  state  city      population
2    AR    Conway      43000
```

```
statename=Arizona
Obs  state  city      population
3    AZ    Flagstaff  53000
```

```
statename=Texas
Obs  state  city      population
4    TX    Lewisville  78000
```

```
statename=Arizona
Obs  state  city      population
5    AZ    Phoenix    1300000
```

```
statename=Texas
Obs  state  city      population
6    TX    Plano      222000
```

```
statename=Arkansas
Obs  state  city      population
7    AR    Sunset      348
```

Thus, one should be extremely cautious when using the NOTSORTED option, since it may produce confusing results when the data is not ordered in an appropriate manner. Use NOTSORTED only when you are sure that the data is in a logical order. SAS is trusting you and

does not check to see if values of the BY variable are repeated out of order.

SYSTEM OPTION BYLINE

The SAS system option BYLINE splits procedure output from BY statements with lines identifying each BY-group, as shown in the prior examples. The BY line can be turned off with an OPTIONS statement containing NOBYLINE.

When NOBYLINE is in effect, procedures such as PROC MEANS and PROC PRINT will have a page break instead of a BY line, and the BY-group will not be identified with the value of the BY variable. In this case, you may wish to create a customized title that mimics a BY line, using the BY substitution keywords #BYVALn, #BYVARn, or #BYLINE in a TITLE or FOOTNOTE statement.

```
options nobyline;

proc sort data=cities; by state;

title1 "Report for #BYLINE";
proc means data=cities;
  by statename notsorted;
```

A portion of the listing is shown here.

Report for statename=Arkansas
Analysis Variable : population

N	Mean	Std Dev	Minimum	Maximum
2	21674	30159.52	348	43000

For both #BYVAL and #BYVAR the name of the BY variable can be listed in parenthesis or a number can be used to refer to which BY variable from the BY statement (the first BY variable, second, so on). Use a period if you want the substituted text to be immediately followed by another character with no space in between.

```
options nobyline;
title1 "Report only for
#BYVAR(statename) of
#BYVAL1.=#BYVAL2";

proc means data=cities;
  by statename state notsorted;
```

The first title line for the listing is:

Report only for statename of
Arkansas=AR

MULTIPLE BY STATEMENTS

When more than one BY statement is applied to a procedure, SAS will use the BY variables that appear on the last BY statement. Previous BY statements will be ignored, except when a BY statement contains a variable that does not exist in the data set.

These two PROC PRINT examples both produce the same output.

```
proc print data=cities;
  by state;
  by statename notsorted;
```

```
proc print data=cities;
  by city;
  by statename notsorted;
```

A portion of the listing is shown here.

statename=Arkansas

Obs	statename	city	population
1	Arkansas	Conway	43000
2	Arkansas	Sunset	348

This PROC PRINT will fail.

```
proc print data=cities;
  by x;
  by state;
```

ERROR: Variable X not found.

COMPARED TO CLASS STATEMENT

PROC TABULATE, MEANS, and UNIVARIATE allow the use of a CLASS statement. With a CLASS statement the data need not be sorted in any particular order. With PROC MEANS you can display the results in an order of your choosing using the ASCENDING, DESCENDING, and ORDER= options. The ORDER= option is also available in PROC UNIVARIATE and TABULATE. Unlike with a BY statement, observations that have a missing value on the BY variable will be excluded with a CLASS statement. To include MISSING values

use the MISSING option with the CLASS statement. The CLASS statement has a variety of additional options; refer to the SAS PROCEDURES guide for more information.

BY STATEMENT IN DATA STEP

In a DATA step, a BY statement is used to control how the data is handled. BY-group processing can be used only for data within existing SAS data sets. There can be only one BY statement corresponding to a prior SET, MERGE, UPDATE, or MODIFY statement.

WITH SET STATEMENT

When two data sets need to be stacked together a SET statement can be used with or without the BY statement. When no BY statement is used, then the data sets are concatenated, with all the observations from the first data set being added, followed by the observations from the second data set. When a BY statement is used, then the data sets are interleaved in order of the BY variable, and the output data set will be in order.

```
proc print data=a;
```

Obs	acct	age
1	10	42
2	30	56

```
proc print data=b;
```

Obs	acct	age
1	20	62

```
data concat;
  set a b;
```

```
proc print data=concat;
```

Obs	acct	age
1	10	42
2	30	56
3	20	62

```
data inter;
  set a b;
  by acct;
```

```
proc print data=inter;
```

Obs	acct	age
1	10	42
2	20	62

3 30 56

BY STATEMENT TO ROLL UP DATA

A BY statement is also used in a DATA step to set up temporary variables that indicate the start and end of each BY-group. These variables can be used as starting and stopping points for various calculations. To understand these variables, one needs a clear picture of data step iteration. Data set *sample* is used in the examples.

```
proc print data=sample;
```

ACCT	DATE	MERCH	PRICE
84	12/20/00	W	23.99
84	05/15/01	W	14.98
84	05/15/01	M	12.45
798	09/25/01	C	7.98
798	09/25/01	C	15.05
798	12/19/01	C	19.75
852	02/20/02	W	49.98
852	03/14/02	W	27.15
852	03/14/02	W	14.58
852	05/11/02	H	74.98

DATA STEP ITERATION

The program data vector is initialized with all variables set to missing values. The automatic variable `_N_` counts the number of times the data step has iterated. Variables coming from the input data set are automatically retained across iterations.

PUT statements with the keyword `_ALL_` print the values of all variables in the log, allowing us to view what happens during data step iteration. These PUT statements are used for illustration purposes and are not a needed part of the code. Throughout these examples, we use the `OBS=` option to limit the number of observations processed (just to keep things short) and the `KEEP=` option to select only the desired variables.

```
data one;
  put _ALL_;
  set sample
      (obs=2 keep=acct price);
  put _ALL_;
```

The log output generated by the PUT statements is shown here.

```

ACCT=.  PRICE=.  _ERROR_=0  _N_=1
ACCT=84 PRICE=23.99 _ERROR_=0  _N_=1

ACCT=84 PRICE=23.99 _ERROR_=0  _N_=2
ACCT=84 PRICE=14.98 _ERROR_=0  _N_=2

ACCT=84 PRICE=14.98 _ERROR_=0  _N_=3

```

To begin, all variables are initialized with missing values. As data values are read in from the incoming data set, the missing values are filled in. The values are carried over to the next iteration of the DATA step and then replaced with values from the next observation in the incoming data set.

Variables created within a DATA step are set back to missing on each iteration.

```

data one;
put _ALL_;
  set sample
  (obs=2 keep=price);
put _ALL_;
  tax = price * 0.1;
put _ALL_;

```

A portion of the log output generated by the PUT statements is shown here.

```

PRICE=.      tax=.      _N_=1
PRICE=23.99  tax=.      _N_=1
PRICE=23.99  tax=2.399  _N_=1

PRICE=23.99  tax=.      _N_=2
PRICE=14.98  tax=.      _N_=2
PRICE=14.98  tax=1.498  _N_=2

PRICE=14.98  tax=.      _N_=3

```

RETAIN STATEMENT

The RETAIN statement causes a variable to retain its value across iterations of the data step. An initial value can be set by listing a number after the variable name in the RETAIN statement.

In the example below, we keep a running total of the number of observations by retaining the variable *count*, initializing it at 0, and then adding 1 with each data step iteration.

```

data one;
  set sample (obs=3 keep=price);

```

```

put _ALL_;
  retain count 0;
  count = count + 1;
put _ALL_;

```

A portion of the log output is shown here.

```

PRICE=23.99  count=0  _N_=1
PRICE=23.99  count=1  _N_=1
PRICE=14.98  count=1  _N_=2
PRICE=14.98  count=2  _N_=2
PRICE=12.45  count=2  _N_=3
PRICE=12.45  count=3  _N_=3

```

RETAIN is a non-executable statement, meaning it does not matter where it appears in the data step, since it provides information that takes effect before DATA step iteration. It can be the first line after the DATA statement or the last line. However, an attempt to use RETAIN in an executable statement will produce an error.

```

data one;
  set sample;
  if _N_ = 1 then retain count;

```

ERROR 180-322:Statement is not valid or it is used out of proper order.

SUM STATEMENT

A Sum statement can also be used to accumulate variable values. This syntax is used to automatically retain variables, initializing them at 0 before the first observation is read in. If an initial value other than 0 is needed, a RETAIN statement can be added. A Sum statement treats missing values like 0. The prior example can be accomplished using the Sum statement.

```

data one;
  set sample (obs=3 keep=price);
put _ALL_;
  count + 1;
put _ALL_;

```

A portion of the log output is shown here.

```

PRICE=23.99  count=0  _N_=1
PRICE=23.99  count=1  _N_=1
PRICE=14.98  count=1  _N_=2
PRICE=14.98  count=2  _N_=2
PRICE=12.45  count=2  _N_=3

```

```
PRICE=12.45 count=3 _N_=3
```

The Sum statement must be written as the base variable plus another expression. If you need to subtract something then use syntax like this:

```
A + (-B);
```

FIRST. AND LAST. VARIABLES

Using a BY statement in a DATA step invokes BY-group processing. SAS identifies the beginning and end of each BY-group by creating two temporary variables for each BY variable. Temporary variables are available during the DATA step but are not added to the output data set. The variable called *FIRST.VARIABLE* will be equal to 1 on the first record of each BY-group and 0 for all other records. The variable called *LAST.VARIABLE* will be equal to 1 on the last record of each BY-group and 0 for all other records.

```
data one;
  set sample (keep=acct);
  by acct;
  put _ALL_;
```

A portion of the log output is shown here.

```
ACCT=84 FIRST.ACCT=1 LAST.ACCT=0
ACCT=84 FIRST.ACCT=0 LAST.ACCT=0
ACCT=84 FIRST.ACCT=0 LAST.ACCT=1
ACCT=798 FIRST.ACCT=1 LAST.ACCT=0
ACCT=798 FIRST.ACCT=0 LAST.ACCT=0
ACCT=798 FIRST.ACCT=0 LAST.ACCT=1
ACCT=852 FIRST.ACCT=1 LAST.ACCT=0
ACCT=852 FIRST.ACCT=0 LAST.ACCT=0
ACCT=852 FIRST.ACCT=0 LAST.ACCT=0
ACCT=852 FIRST.ACCT=0 LAST.ACCT=1
```

CREATING A SUM FOR A BY-GROUP

With BY-group processing, you can keep a running total for each BY-group by making use of the FIRST. and LAST. variables. Simply retain the variable that will contain the running total, reset it to 0 at the start of each BY-group, increment the variable, and output the total at the end of each BY-group.

```
data one;
  set sample (keep=acct);
  by acct;

  retain rows;
```

```
  if first.acct then rows = 0;
  put _ALL_;
```

```
  rows = rows + 1;
  put _ALL_;
```

```
  if last.acct then output;
A portion of the log output is shown here.
```

```
ACCT=84 rows=0 _N_=1
ACCT=84 rows=1 _N_=1
ACCT=84 rows=1 _N_=2
ACCT=84 rows=2 _N_=2
ACCT=84 rows=2 _N_=3
ACCT=84 rows=3 _N_=3
ACCT=798 rows=0 _N_=4
ACCT=798 rows=1 _N_=4
ACCT=798 rows=1 _N_=5
ACCT=798 rows=2 _N_=5
ACCT=798 rows=2 _N_=6
ACCT=798 rows=3 _N_=6
ACCT=852 rows=0 _N_=7
ACCT=852 rows=1 _N_=7
ACCT=852 rows=1 _N_=8
ACCT=852 rows=2 _N_=8
ACCT=852 rows=2 _N_=9
ACCT=852 rows=3 _N_=9
ACCT=852 rows=3 _N_=10
ACCT=852 rows=4 _N_=10
```

```
proc print data=one;
```

Obs	ACCT	rows
1	84	3
2	798	3
3	852	4

OTHER BY-GROUP PROCESSING TIPS

BEWARE THE SUBSETTING IF

FIRST. and LAST. flags are created as observations come into the program data vector from the incoming data set. Thus, they are filled in before any executable statements take effect within the data step. In our *sample* data set, let's count the number of rows for each *acct* where the *price* is under \$20. Look back at the data set printout and note that we would expect to get 2 rows, 3 rows, and 1 row respectively for the three *accts*.

The next example fails to get this result. When the first observation is read in, the FIRST.ACCT flag is set to 1. But, that observation is then dropped by the subsetting IF statement. So the variable *rows* does not get set to 0 at this point, and when the DATA step reaches the LAST.ACCT for acct 84, the variable *rows* is still missing. For acct 798, since all records have price < \$20, then FIRST. and LAST. records are not dropped and the calculations are correct. Nothing gets output for acct 852 since the LAST. record for this account is dropped before it is output.

```
data bad;
  set sample (keep=acct price);
  by acct;

  retain rows;
  if price < 20;

  if first.acct then rows = 0;

  rows = rows + 1;

  if last.acct then output;
  keep acct rows;
```

NOTE: There were 10 observations read from the data set WORK.SAMPLE.

```
proc print data=bad;
```

Obs	acct	rows
1	84	.
2	798	3

There are two main alternatives to the subsetting IF statement. First, use a WHERE statement. The WHERE statement eliminates observations before they enter the program data vector, so the FIRST. and LAST. flags get set after unneeded observations are dropped. The second option is to increment the counter conditionally when the price criteria is met, without actually dropping any rows. Both examples are shown below and produce the same result.

```
data good;
  set sample (keep=acct price);
  by acct;
  where price < 20;

  retain rows;
  if first.acct then rows = 0;
```

```
rows = rows + 1;
if last.acct then output;
keep acct rows;
```

NOTE: There were 6 observations read from the data set WORK.SAMPLE.

```
data good;
  set sample (keep=acct price);
  by acct;

  retain rows;
  if first.acct then rows = 0;
  if price < 20 then rows=rows + 1;
  if last.acct then output;
  keep acct rows;
```

NOTE: There were 10 observations read from the data set WORK.SAMPLE.

```
proc print data=good;
```

Obs	acct	rows
1	84	2
2	798	3
3	852	1

As a general rule, one should avoid any subsetting IF statements in data steps that use FIRST. and LAST. flags.

IDENTIFYING DUPLICATES

Sometimes you think a data set has only one observation for each BY-group, but you realize there must be an exception. In such an instance, there may be a need to quantify the number of duplicates or understand why there are duplicates. The next example will use the data set *dups*.

```
proc print data=dups;
```

Obs	acct	age
1	10	42
2	10	50
3	30	56
4	50	34

A PROC SORT with the NODUPKEY option can show you the number of duplicates, but it does not tell you which observations have duplicates.

```
proc sort data=dups
  out=test nodupkey;
```

```
by acct;
```

NOTE: 1 observations with duplicate key values were deleted.

The FIRST. and LAST. flags can be used in a DATA step to output the records for any *acct* that has more than one record per BY-group. In this example one can see that *acct* #10 has a discrepancy in that two different ages were listed. Note that this IF statement cannot be changed to a WHERE statement, since the FIRST. and LAST. flags do not exist in the incoming data set.

```
data find;
    set dups;
    by acct;
```

```
if first.acct + last.acct ne 2;
```

```
proc print data=find;
```

Obs	acct	age
1	10	42
2	10	50

MULTIPLE BY-GROUPS

When two variables are listed in the BY statement, then second variable is grouped within the first variable. There will be two sets of FIRST. and LAST. variables. In the next example, both *acct* and *date* are used as BY variables. Since the *date* 12/20/2000 appears only once for *acct* 84, the first observation is marked as both the FIRST. and LAST. date.

```
data one;
    set sample
      (keep=acct date obs=6);
    by acct date;
    put _ALL_;
```

A portion of the log output is shown here.

```
acct=84 date=12/20/2000
FIRST.acct=1 LAST.acct=0
FIRST.date=1 LAST.date=1 _N_=1
acct=84 date=05/15/2001
FIRST.acct=0 LAST.acct=0
FIRST.date=1 LAST.date=0 _N_=2
acct=84 date=05/15/2001
FIRST.acct=0 LAST.acct=1
FIRST.date=0 LAST.date=1 _N_=3
```

```
acct=798 date=09/25/2001
FIRST.acct=1 LAST.acct=0
FIRST.date=1 LAST.date=0 _N_=4
```

```
acct=798 date=09/25/2001
FIRST.acct=0 LAST.acct=0
FIRST.date=0 LAST.date=1 _N_=5
```

```
acct=798 date=12/19/2001
FIRST.acct=0 LAST.acct=1
FIRST.date=1 LAST.date=1 _N_=6
```

Suppose you wanted to compute the number of different dates for each account. Using the FIRST. flag for variable *date*, a counter can be incremented each time a new date appears.

```
data one;
    set sample (keep=acct date);
    by acct date;

    retain numdate;
    if first.acct then numdate = 0;

    if first.date then
        numdate = numdate + 1;

    if last.acct then output;

    keep acct numdate;
```

```
proc print data=one;
```

Obs	acct	numdate
1	84	2
2	798	2
3	852	3

COMPARISON TO PROC SQL

For data set *sample* we want to compute the following five variables for each *acct*:

- Total sales
- Number of different dates
- Number of different dates with at least one item having *merch=W*
- Total sales / number of different dates

This can be accomplished using a DATA step with FIRST. and LAST. flags or with PROC SQL. The next two examples both produce the same output. The DATA step is more wordy than the PROC SQL. However, even on this data set with only 10 rows, the PROC SQL tends to run 3 times as long as the DATA step.

```
proc sql;
create table one as

    select acct,
```

```

sum(price) as sales,
count(distinct date) as ords,
count(distinct case when merch='W'
then date end) as w_ords,
calculated sales / calculated ords
as avgorder
from sample

group by acct;

data one;
  set sample ;
  by acct date;
retain sales ords w_ords wflag;
  if first.acct then do;
    sales = 0;
    ords = 0;
    w_ords = 0;
  end;
  if first.date then do;
    wflag = 0;
    ords = ords + 1;
  end;
  sales = sales + price;
  if merch = 'W' then wflag = 1;
  if last.date then w_ords =
    w_ords + wflag;
  if last.acct then do;
    avgorder = sales / ords;
    output;
  end;
keep acct sales ords w_ords avgorder;

```

Obs	acct	sales	ords	w_ords	avgorder
1	84	51.42	2	2	25.7100
2	798	42.78	2	0	21.3900
3	852	166.69	3	2	55.5633

I have found that with large data sets (300,000+ observations) computing 50+ summary variables, a DATA step almost always runs significantly faster than a corresponding PROC SQL.

CONCLUSIONS

BY-groups provide a convenient way to break apart data based on the values of key variables. Options such as DESCENDING, NOTSORTED, and BYLINE provide flexibility when defining and displaying BY-groups. BY statements are

crucial when combining data sets with SET or MERGE statements. BY-group processing provides a very powerful and effective way to summarize data in a DATA step using the FIRST. and LAST. flags. These flags can also be used to identify and study duplicate records. BY-group processing is often more efficient than a PROC SQL. I've given you some FIRST. words on the BY statement and BY-group processing. I hope you will keep learning until you know the LAST. word on the subject.

REFERENCES

SAS Institute Inc., SAS OnlineDoc®, Version 8, Cary, NC: SAS Institute Inc., 1999.

SAS Institute Inc., *SAS Language: Reference, Version 6, First Edition*, Cary, NC: SAS Institute Inc., 1990. 1042 pp.

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies.

ACKNOWLEDGMENTS

Thank you to these fellow SAS users for reviewing my paper and providing many useful suggestions: Dan Watson, Kenneth Smiley, Vasanthi Rao, Ken Larkin, Irene Huang, Andrew Cartwright, and John Boddie.

CONTACT INFORMATION

Christine Smiley
JCPenney, Mailstop 7322
6501 Legacy Drive
Plano, TX 75024

972-431-3803
csmil1@jcpenny.com