

The Use and Abuse of the Program Data Vector (The Procs)

Jim Johnson, Covance Periapproval Services, Inc., Radnor, PA

ABSTRACT

Have you ever wondered why SAS does the things it does, or why your programs “get away with” the things that they do, or why SAS would not do what you wanted it to? A key operational component of SAS is the program data vector. Without it SAS would not function, as we know it. With knowledge of the program data vector, programmers can better understand how SAS works. This paper will help you understand how the program data vector works, how the Procedures use it, and how you can exploit, manipulate, and trick it.

INTRODUCTION

This paper will pick up where my 2003 paper “The Use and Abuse of the Program Data Vector” left off. In the previous paper the Program Data Vector was discussed in some detail. How data is passed to and from the Program Data Vector, some tricks that work in SAS and why they work relative to the Program Data Vector were discussed. The previous paper dealt specifically the SAS data step and deliberately avoided the topic of SAS Procedures.

This paper will look at how the same tools discussed in the previous paper apply to SAS Procedures. SAS procedures are compiled code, and though they are officially part of SAS, they do not always appear to adhere to the accepted SAS standards. This paper will not be as exciting or robust as the previous paper because the Program Data Vector cannot be followed the same way, but it will still shed some light on how things work in the Procedures and may help explain why the Procedures are not doing what you expect.

“There are a hundred different ways to do everything in SAS.” The techniques shown in this paper are only one way to do things. There are many more. These techniques have been chosen to demonstrate specific functions of SAS.

All examples given are based on the Microsoft® Windows® environment using SAS system for Personal Computers® version 8.2.

REVIEW

SAS STATEMENTS appear in the body of data step code. The action of SAS statements occur more globally within the data step, they function at different times, and may have different effects than their SAS data set option counterpart.

SAS DATA SET OPTIONS allow you to specify actions on a specific SAS data set as the data are read or written. Data set options are specified in parentheses following the SAS data set name to which they apply and can only appear on the DATA statement or input statements such as SET, MERGE, or UPDATE.

DROP, KEEP, and RENAME statements

1. take effect as the data is written to the output data sets,
2. apply to all output data sets,
3. multiple of each are allowed.

WHERE statement

1. takes effect as the data is read from the source data sets,
2. apply to all input data sets,
3. multiple WHERE statements are allowed, however the conditions are augmented as if written with an AND operator.

DROP=, KEEP=, RENAME=, and WHERE= data set options

- when used on **input** data sets
 1. take effect as the data is read from the source data sets,
 2. apply only to the data sets to which they are attached,
 3. only one of each is allowed for any one data set.
- when used on **output** data sets
 1. take effect as the data is written to the output data sets,
 2. apply only to the data sets to which they are attached,
 3. only one of each is allowed for any one data set.

WHAT IS A SAS PROCEDURE?

The SAS documentation says:

“SAS procedures are software tools for data analysis and reporting.”

“SAS procedures analyze data in SAS data sets to produce statistics, tables, reports, charts, and plots, to create SQL queries, and to perform other analyses and operations on your data. They also provide ways to manage and print SAS files.”

The documentation tells us what Procedures do, but never really tells us what a SAS Procedure is. Ever since we were just little SAS programmers, we have been told that Procedures are “compiled code written to provide a consistent, validated means of doing repetitive operations, having a focused purpose, defined interface, and a predictable output.”

Essentially, SAS Procedures are pre-existing compiled code, written by one or more programmers, that have been packaged into the SAS system. The key information here is “pre-existing compiled code”. Since they are compiled code, one can never be completely certain how they work behind-the-scenes without a lot of trial and error experimentation. The SAS Procedures, though supported by SAS, may or may not appear to operate the same way as similar data step operations.

In the data step, there are both data set options and data statements that can be used to manipulate the program data vector. In SAS Procedures, the same data set options are available for use, but many of the statement counter parts are not. SAS Procedures do not recognize the DROP, KEEP, or RENAME statements. The WHERE statement is functional in most of the SAS Procedures.

PRINT PROCEDURE

The PRINT Procedure is pretty benign. There is not a lot for it to do except to read the data in and spill it out on the page.

	ONE			
Obs	a	b	c	d
1	1	2	1	4
2	1	2	2	4
3	1	2	2	4
4	1	2	2	4
5	1	2	3	4
6	1	3	3	4

This example uses the DROP= and KEEP= data set options with the same variable. Because SAS provided no message indicating the variable A could not be found, this suggests that the KEEP= data set option functions first. The variable A was kept using the KEEP=, then dropped using the DROP=, therefore the message that there are no variables in data set ONE.

```
proc print data=one(drop=a keep=a);  
run;
```

NOTE: No variables in data set WORK.ONE.

~~~~~

This example adds the WHERE= data set option to subset the data. An error message is generated because the variable A is no longer available. The variable A was kept by the KEEP= data set option, then dropped with the DROP= data set option, therefore unavailable for the WHERE= data set option, proving the WHERE= executes after the DROP= data set option.

```
proc print data=one(drop=a keep=a where=(a=1));  
run;
```

ERROR: Variable a is not on file WORK.ONE.  
NOTE: The SAS System stopped processing this step because of errors.

~~~~~

In this example, the DROP= data set option has been removed and the RENAME= data set option has been added. The failure tells us that the RENAME= occurs before the WHERE= because there is no variable A for the WHERE=. The order the data set options are specified does not matter.

```
proc print data=one(keep=a where=(a=1) rename=(a=aa));  
run;
```

ERROR: Variable a is not on file WORK.ONE.
NOTE: The SAS System stopped processing this step because of errors.

The examples above show us that in the PRINT Procedure, the order of execution of the data set options is KEEP=, DROP=, RENAME=, and WHERE=.

~~~~~

The DROP and KEEP statements are officially not supported as described in the NOTES, the statements are ignored and SAS continues to function.

```
proc print data=one;  
  drop b;  
  var a b c d;  
run;
```

NOTE: The DROP and KEEP statements are not supported in procedure steps in this release of the SAS System. Therefore, these statements are ignored.

~~~~~

The RENAME statement causes an error and SAS stops.

```
proc print data=one;  
  rename b=bb;  
run;
```

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

~~~~~

The following examples show that the WHERE statement and WHERE= data set option both work.

```
proc print data=one(where=(c=0));  
run;
```

NOTE: No observations were selected from data set WORK.ONE.

NOTE: There were 0 observations read from the data set WORK.ONE.  
WHERE c=0;

```
proc print data=one;  
  where c=3;  
run;
```

NOTE: There were 2 observations read from the data set WORK.ONE.  
WHERE c=3;

~~~~~

When the WHERE statement and WHERE= data set option are used together, they augment each other as if written with an AND operator. When the conditions are augmented, SAS provides a message showing the composite WHERE condition. In the DATA step, when the WHERE= data set option is used on input and a WHERE statement is used, the WHERE statement is ignored.

```
proc print data=one(where=(b=3));  
  where c=3;  
run;
```

NOTE: Where clause has been augmented.
NOTE: There were 1 observations read from the data set WORK.ONE.
WHERE (b=3) and (c=3);

~~~~~

If the augmented WHERE clauses produce an obviously impossible result, SAS will provide a message that is not totally clear. The message does not provide the composite clause, so it makes you figure out what it means and what you did wrong. In this example the composite expression would be **WHERE C=3 AND C=1**, obviously impossible.

```
proc print data=one(where=(c=3)); /** where on input works **/  
  where c=1;  
NOTE: Where clause has been augmented.  
NOTE: No observations were selected from data set WORK.ONE.  
NOTE: There were 0 observations read from the data set WORK.ONE.  
WHERE 0 /* an obviously FALSE where clause */
```

~~~~~

If multiple WHERE statements are specified, only the last one specified will be used.

```
proc print data=one; /** where stmt works **/  
  where c=3;  
  where c=1;  
run;
```

NOTE: Where clause has been replaced.
NOTE: There were 1 observations read from the data set WORK.ONE.
WHERE c=1;

~~~~~

If multiple WHERE statements are used with a WHERE= data set option, the messages suggest that all the WHERE clauses are augmented. In fact, only the last WHERE statement is augmented with the WHERE= data set option.

```
proc print data=one(where=(b=3)); /** where stmt works **/
  where c=3;
  where c=1;
run;
```

**NOTE: Where clause has been augmented.**

**NOTE: Where clause has been augmented.**

NOTE: No observations were selected from data set WORK.ONE.

NOTE: There were 0 observations read from the data set WORK.ONE.

**WHERE (b=3) and (c=1);**

## FREQ PROCEDURE

Testing similar to that done for the PRINT Procedure show that the order of execution of the data set options, both on input data sets and output data sets is KEEP=, DROP=, RENAME= and WHERE=. This is the same order as for the PRINT Procedure. Testing also shows the DROP and KEEP statements are officially not supported in the FREQ Procedure. The RENAME statement also causes a SAS error in the FREQ Procedure.

```
proc freq data=one;
  table a/out=freqa(keep=a drop=a);
run;
```

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

NOTE: The data set WORK.FREQA has 1 observations and **0 variables**.

The example above leaves ZERO variables, not even those generated by the FREQ Procedure are kept. The KEEP= kept only the variable A, effectively dropping COUNT and PERCENT that were generated by the FREQ Procedure. The DROP= then dropped the only remaining variable, A, leaving none to output to the data set. The Procedure generated the expected printed output.

The FREQ Procedure

| a | Frequency | Percent | Cumulative<br>Frequency | Cumulative<br>Percent |
|---|-----------|---------|-------------------------|-----------------------|
| 1 | 6         | 100.00  | 6                       | 100.00                |

~~~~~

WHERE statement and WHERE= data set option both work. When used, they apply to all TABLE statements in the Procedure. Location of the statement does not matter. Output for both steps below is exactly the same.

```
proc freq data=one(where=(c=1));
  table a;
  table b;
run;
```

```
proc freq data=one;
  table a;
  where c=1;
  table b;
run;
```

The FREQ Procedure

a	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	1	100.00	1	100.00

b	Frequency	Percent	Cumulative Frequency	Cumulative Percent
2	1	100.00	1	100.00

As with the PRINT Procedure, if both the WHERE statement and WHERE= data set option are used, they augment each other as if written with an AND operator. If the WHERE clauses produce an obviously impossible result, SAS will provide a message that is not totally clear, just as in the PRINT Procedure section. When multiple WHERE statements are specified, only the last one is used.

The WHERE statement and WHERE= data set option used on the input data set limit the data read into the program data vector. The WHERE= data set option on output limits the data being written from the program data vector to the output data set.

In this example, the WHERE statement limits input from the data set ONE. The WHERE= data set option limits output to the data set FREQA. Only those observations in ONE where B=2 will be subjected to the FREQ Procedure, and only the observations from that subset where C=1 will be written to the output data set FREQA.

```
proc freq data=one;
  table a * c / list out=freqa(when=(c=1));
  where b=2;
run;
```

NOTE: There were 6 observations read from the data set WORK.ONE.
WHERE b=2;
NOTE: The data set WORK.FREQA has 1 observations and 4 variables.

~~~~~

The WHERE= data set option on output can use the variables generated by the FREQ Procedure. In this case only those observations in ONE where B=2 will be subjected to the FREQ Procedure. Only the observations generated by FREQ Procedure where the COUNT is greater than one will be written to the output data set FREQA.

```
proc freq data=one;
  table a * c / list out=freqa(when=(count > 1));
  where b=2;
run;
```

NOTE: There were 6 observations read from the data set WORK.ONE.  
WHERE b=2;  
NOTE: The data set WORK.FREQA has 2 observations and 4 variables.

---

## **SORT PROCEDURE**

The SORT Procedure gets a little more interesting.

Testing similar to that done for the PRINT Procedure show that the order of execution of the data set options, both on input data sets and output data sets is KEEP=, DROP=, RENAME= and WHERE=. This is the same order as for the PRINT Procedure. Testing also shows the DROP and KEEP statements are officially not supported in the SORT Procedure. The RENAME statement also causes a SAS error in the SORT Procedure.

WHERE statement and WHERE= data set option work. When used together they augment each other as if written with an AND operator. If the WHERE clauses produce an obviously impossible result, SAS will provide a message that is not totally clear, just as in the PRINT Procedure section. When multiple WHERE statements are specified, only the last one is used.

WHERE clauses used without the OUT= option will change the source data. In this example, the data set ONE will be subset WHERE B=2, sorted by A, then the subset data will be written back into the data set ONE.

| ONE |   |   |   |
|-----|---|---|---|
| Obs | a | b | c |
| 1   | 1 | 1 | 3 |
| 2   | 0 | 1 | 3 |
| 3   | 0 | 2 | 3 |
| 4   | 0 | 2 | 2 |
| 5   | 1 | 2 | 1 |
| 6   | 1 | 3 | 1 |

```
proc sort data=one(where=(b=2));  
  by a;  
run;
```

NOTE: There were 3 observations read from the data set WORK.ONE.

WHERE b=2;

NOTE: The data set WORK.ONE has 3 observations and 3 variables.

~~~~~

When WHERE= data set option is used on input and output and the WHERE statement is used, the input data set option and statement are augmented and the output WHERE= data set option remains independent. In this example the data input to the SORT Procedure is subset using WHERE C=3 AND B=1 (2 observations). Once sorted, the output data is limited to only those observations where A=1 (only 1 of the 2 sorted observations).

```
proc sort data=one(where=(c=3))  
  out=two(where=(a=1));  
  where b=1;  
  by a;  
run;
```

NOTE: Where clause has been augmented.

NOTE: There were 2 observations read from the data set WORK.ONE.

WHERE (c=3) and (b=1);

NOTE: The data set WORK.TWO has 1 observations and 3 variables.

~~~~~

SAS version 8 is smart enough not to sort already sorted data.

```
proc sort data=one;  
  by a;
```

```
run;
```

NOTE: Input data set is already sorted, no sorting done.

Introduction of OUT= will produce a different message. The new message does not explicitly say the data will not be sorted, but knowing SAS did not resort the data above would leave you to believe that SAS would not sort the data under these conditions. Testing with a large data set proved that SAS does not resort the data. The execution time for the SORT was similar to that of a simple data step setting one data set equal to the other.

```
proc sort data=one
  out = two;
  by a;
run;
```

NOTE: Input data set is already sorted; it has been copied to the output data set.

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

NOTE: The data set WORK.TWO has 6 observations and 3 variables.

~~~~~

One of the powerful features of the SORT Procedure is its ability to sort ignoring duplicates. It does so using NODUPRECS (also known as NODUP) and NODUPKEY. Interestingly, under certain conditions, they do not act as expected.

From SAS Procedure Guide:

NODUPKEY checks for and eliminates observations with duplicate BY values. If you specify this option, PROC SORT compares all BY values for each observation to those for the previous observation written to the output data set. If an exact match is found, the observation is not written to the output data set.

NODUPRECS (NODUP) checks for and eliminates duplicate observations. If you specify this option, PROC SORT compares all variable values for each observation to those for the previous observation that was written to the output data set. If an exact match is found, the observation is not written to the output data set. Because NODUPRECS checks only consecutive observations, some nonconsecutive duplicate observations may remain in the output data set. You can remove all duplicates with this option by sorting on all variables.

The example below proves that the KEEP= data set option on an input data set works. It has effectively dropped the variables B and C. When the WHERE= data set option tries to execute, it cannot find the variable C and an error occurs. This proof of the function of the KEEP= data set option is important for the next example.

ONE			
Obs	a	b	c
1	1	1	3
2	0	1	3
3	0	2	3
4	0	2	2
5	1	2	1
6	1	3	1

```
proc sort data=one(keep=a where=(c=3))
  out = two
  nodup;
  by a;
run;
```

ERROR: Variable c is not on file WORK.ONE.

NOTE: The SAS System stopped processing this step because of errors.

WARNING: The data set WORK.TWO may be incomplete. When this step was stopped there were 0 observations and 0 variables.

~~~~~

The step below is written to count the number of distinct values of A in the data set ONE. By keeping only A on input, the NODUP option should provide the desired results. Given the data set above, you would expect this step to produce a data set with 2 observations and one variable. However, what one expects and what one gets are two different things.

```
proc sort data=one(keep=a)
  out = two
  nodup;
  by a;
run;
```

NOTE: **0 duplicate observations were deleted.**

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

NOTE: The data set WORK.TWO has **6 observations** and 1 variables.

| TWO |   |
|-----|---|
| Obs | a |
| 1   | 0 |
| 2   | 0 |
| 3   | 0 |
| 4   | 1 |
| 5   | 1 |
| 6   | 1 |

The earlier example above proved that the KEEP= data set option works on input. Therefore the only variable available to the SORT Procedure should be A. The SORT Procedure is acting as if the input KEEP= data set option is not present, as shown in the example below which produces the exact same output.

```
proc sort data=one
  out = two(keep=a);
  by a;
run;
```

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

NOTE: The data set WORK.TWO has 6 observations and 1 variables.

The expected result can be obtained using the NODUPKEY option ... but why?

```
proc sort data=one(keep=a)
  out = two
  nodupkey;
  by a;
run;
```

NOTE: 4 observations with duplicate key values were deleted.

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

NOTE: The data set WORK.TWO has **2 observations** and 1 variables.

| TWO |   |
|-----|---|
| Obs | a |
| 1   | 0 |
| 2   | 1 |

The actions of the SORT Procedure are related to the little known SAS system option SORTDUP. When SORTDUP is set to PHYSICAL, the default, the SORT Procedure will remove duplicates based on all the variables present in the data set. When set to LOGICAL, the SORT Procedure removes duplicates based on only the variables remaining after the DROP= and KEEP= data set options are processed.

The LOGICAL setting drops or keeps the variables, then does the sort, and provides the output expected.

```
options sortdup=logical; /* default value is PHYSICAL */

proc sort data=one(keep=a)
  out = two
  nodup;
  by a;
run;
```

NOTE: 4 duplicate observations were deleted.

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

NOTE: The data set WORK.TWO has **2 observations** and 1 variables.

| TWO |   |
|-----|---|
| Obs | a |
| 1   | 0 |
| 2   | 1 |

---

## SQL PROCEDURE

Data set options do not work in the SQL Procedure when the source data is a SAS view.

```
proc sql;
  create view view_one as
  select *
  from one;
```

NOTE: SQL view WORK.VIEW\_ONE has been defined.

```
create table a as
  select *
  from view_one(drop=a);
```

ERROR: Unable to process SAS data set options for SQL view WORK.VIEW\_ONE.

~~~~~

Testing similar to that done for the PRINT Procedure show that the order of execution of the data set options, both on input data sets and output data sets is KEEP=, DROP=, RENAME= and WHERE=. This is the same order as for the PRINT Procedure. Testing also shows the DROP and KEEP statements are officially not supported in the SQL Procedure. The RENAME statement also causes a SAS error in the SQL Procedure.

The SQL Procedure is essentially a language of its own. The DROP, KEEP, and RENAME statements are not part of the SQL Procedure and will generate errors.

The DROP=, KEEP=, RENAME= and WHERE= data set options can be used on input data sets in the FROM clause and on the output data set in the CREATE TABLE statement.

ONE			
Obs	a	b	c
1	1	1	3
2	0	1	3
3	0	2	3
4	0	2	2
5	1	2	1
6	1	3	1

```
proc sql;
  create table one_a(keep=a rename=(a=aa) where=(aa=1)) as
  select a
  from one;
```

NOTE: Table WORK.ONE_A created, with 3 rows and 1 columns.

ONE_A	
Obs	aa
1	1
2	1
3	1

~~~~~

Looking at the code below, you might think the two steps are the same.

```
proc sql;
  create table three as
  select one.*
         ,two.*
  from one (where=(a=1))
  FULL JOIN
         two
  on one.b = two.digit;
```

```
proc sql;
  create table three as
  select one.*
         ,two.*
  from one
  FULL JOIN
         two
  on one.b = two.digit
  where one.a = 1;
```

A FULL JOIN will keep everything from both tables in the join whether it matches the other data set or not.

| ONE |   |   | TWO |       |       |
|-----|---|---|-----|-------|-------|
| Obs | a | b | Obs | digit | word  |
| 1   | 1 | 1 | 1   | 0     | zero  |
| 2   | 0 | 1 | 2   | 1     | one   |
| 3   | 0 | 2 | 3   | 2     | two   |
| 4   | 0 | 2 | 4   | 3     | three |
| 5   | 1 | 2 |     |       |       |
| 6   | 1 | 3 |     |       |       |

```
proc sql;
  create table three as
  select one.*
         ,two.*
  from one (where=(a=1))
  FULL JOIN
         two
  on one.b = two.digit;
```

NOTE: Table WORK.THREE created, with 4 rows and 5 columns.

| THREE |   |   |       |       |
|-------|---|---|-------|-------|
| Obs   | a | b | digit | word  |
| 1     | . | . | 0     | zero  |
| 2     | 1 | 1 | 1     | one   |
| 3     | 1 | 2 | 2     | two   |
| 4     | 1 | 3 | 3     | three |

In the example above, the data set ONE is subset to three records as the data are read into the program data vector. The FULL JOIN with data set TWO provides the fourth output record (digit=0) which did not match a record in data set ONE.

In the example below, the data sets are joined and then subset WHERE A=1 providing a different result.

```
proc sql;
  create table three as
  select one.*
         ,two.*
  from one
  FULL JOIN
         two
  on    one.b = two.digit
  where one.a=1;
```

NOTE: Table WORK.THREE created, with 3 rows and 5 columns.

| THREE |   |   |       |       |
|-------|---|---|-------|-------|
| Obs   | a | b | digit | word  |
| 1     | 1 | 1 | 1     | one   |
| 2     | 1 | 2 | 2     | two   |
| 3     | 1 | 3 | 3     | three |

~~~~~

Use of wildcard (*) may cause unexpected results, but knowing how the SQL Procedure processes in the program data vector, this issue can be used to your advantage in spite of the messages received. This example is trying to attach the variable LETTER to the data set ONE where the letter number matches the value of B in ONE.

ONE			THREE		
Obs	a	b	Obs	a	letter
1	1	1	1	1	A
2	0	1	2	2	B
3	0	2	3	3	C
4	0	2			
5	1	2			
6	1	3			

```
proc sql;
  create table four as
  select one.*
         ,three.*
  from   one
         ,three
  where one.b = three.a;
```

WARNING: Variable a already exists on file WORK.FOUR.

NOTE: Table WORK.FOUR created, with 6 rows and 3 columns.

This warning tends to bother some people, until they know what the message is saying. If you use the PRINT Procedure to display the output of FOUR, it may or may not contain the data you expected.

FOUR			
Obs	a	b	letter
1	1	1	A
2	0	1	A
3	0	2	B
4	0	2	B
5	1	2	B
6	1	3	C

Perhaps you were expecting

FOUR			
Obs	a	b	letter
1	1	1	A
2	1	1	A
3	2	2	B
4	2	2	B
5	2	2	B
6	3	3	C

The message "WARNING: Variable a already exists on file WORK.FOUR." is telling you that the variable A has been encountered more than once in the select clause. What it does not tell you is that all the variables A encountered after the first will be ignored when the data is output.

Instead of using CREATE TABLE, simply SELECT the same data so that it will be printed to the output window, and you will see the program data vector and the multiple occurrences of the variable A.

```
proc sql;          /* SHOWS TRUE PDV */
  select one.*
         ,three.*
  from   one
         ,three
 where  one.b = three.a;
```

a	b	a	letter
1	1	1	A
0	1	1	A
0	2	2	B
0	2	2	B
1	2	2	B
1	3	3	C

Knowing that SAS will use the first encountered version of the variable in the SQL Procedure opens a door that allows programmers to exploit this feature. As long as you know what the message is telling you and how the program data vector is responding, you can use it to your advantage.

In the example below, the data set E1 contains the variable ALPHA. This step calculates a new value for ALPHA. Because the data set E1 also contains dozens of other variables it is desirable to use the * wildcard rather than typing each variable name. Because the ALPHA we want to keep, the newly calculated one, was listed first, the message "WARNING: Variable alpha already exists on file WORK.FOUR." will occur when the second occurrence of ALPHA is encountered, by virtue of the wild card, but the second occurrence will not be written to the output data set.

```
proc sql;
  create table e1 as
```

```

select alpha + beta as alpha
      ,*
from   e1;

```

In this example, the select statement is reordering the program data vector by specifically naming choice variables from the source data, then selecting all the variables using the asterisk. When the named variables are encountered during the processing of the asterisk, a message will be generated for each, and those subsequent occurrences of the variables will not be written to the destination data set.

```

proc sql;
  create table work_crf as
  select patient
         ,random
         ,complete
         ,*
  from   crfdata.&dsn b;

```

~~~~~

| ONE |   |   | TWO |       |   |
|-----|---|---|-----|-------|---|
| Obs | b | d | Obs | digit | d |
| 1   | 1 | A | 1   | 1     | A |
| 2   | 1 | B | 2   | 1     | B |
| 3   | 2 | C | 3   | 2     | C |
| 4   | 2 | D | 4   | 2     | D |

```

proc sql;
  create table first as
  select distinct
         one.*
         ,two.*
  from   one
         ,two
  where  one.b = two.digit;

```

The DISTINCT in the step above will eliminate exact duplicate rows. We know that the SQL Procedure will see two variables named D, give us a message, and keep only the first one, so the output we expect should look something like this:

| FIRST |   |   |       |  |
|-------|---|---|-------|--|
| Obs   | b | d | digit |  |
| 1     | 1 | A | 1     |  |
| 2     | 1 | B | 1     |  |
| 3     | 2 | C | 2     |  |
| 4     | 2 | D | 2     |  |

However, the SQL Procedure will come up with something different. The duplicate rows have not been removed.

| FIRST |   |   |       |  |
|-------|---|---|-------|--|
| Obs   | b | d | digit |  |
| 1     | 1 | A | 1     |  |
| 2     | 1 | A | 1     |  |
| 3     | 1 | B | 1     |  |
| 4     | 1 | B | 1     |  |
| 5     | 2 | C | 2     |  |
| 6     | 2 | C | 2     |  |
| 7     | 2 | D | 2     |  |
| 8     | 2 | D | 2     |  |

Run the SQL Procedure again without the CREATE statement to reveal the contents of the program data vector and we get the following:

| b | d | digit | d |
|---|---|-------|---|
| 1 | A | 1     | A |
| 1 | A | 1     | B |
| 1 | B | 1     | A |
| 1 | B | 1     | B |
| 2 | C | 2     | C |
| 2 | C | 2     | D |
| 2 | D | 2     | C |
| 2 | D | 2     | D |

It is from this data set that the SQL Procedure will select the distinct records, and since none of the records above are duplicates, none are deleted.

Run the Procedure again, this time dropping D from the second data set on the way in, the results will be as expected and this time there will be no message.

```
proc sql;
  create table first as
  select distinct
    one.*
  ,two.*
  from   one
  ,two(drop=d)
  where one.b = two.digit;
```

| FIRST |   |   |       |
|-------|---|---|-------|
| Obs   | b | d | digit |
| 1     | 1 | A | 1     |
| 2     | 1 | B | 1     |
| 3     | 2 | C | 2     |
| 4     | 2 | D | 2     |

~~~~~

ONE			
Obs	a	b	c
1	1	1	3
2	0	1	3
3	0	2	3
4	0	2	2
5	1	2	1
6	1	3	1
7	1	3	1

Get the count of records in ONE. The output is as expected.

```
proc sql;
  select count(*) as count
  from   one;
```

count

7

Get the count of records in ONE where A=1. The output is as expected.

```
select count(*) as count
from one(where=(a=1));
```

count
4

Get B and the count of records in ONE where A=1. Since the GROUP BY clause is not being used, the output is as expected. There is one record for every record where A=1, its corresponding value of B, and the count of records where A=1.

```
select b
, count(*) as count
from one(where=(a=1));
```

b	count
1	4
2	4
3	4
3	4

Now, for the unanswered question: Given the three examples above, why do we get the output we do here? There are 7 records in ONE, but only 4 have the variable A = 1.

```
select 'one' as dsn
, count(*) as count
from one(where=(a=1));
```

dsn	count
one	7

Move the WHERE= data set option from the input data set and place it after the FROM clause as a WHERE statement and we get the expected results.

```
select 'one' as dsn
, count(*) as count
from one
where a=1;
```

dsn	count
one	4

But why do we get the results we did above? Perhaps that is yet another paper.

CONCLUSION

I started out expecting to prove that the SAS Procedures did not act as advertised when it came to the use of certain statements and data set options. This was my thesis based on some personal experience. The research for this paper definitely enlightened me. Some of my theories were proven, some disproved, and others were explained.

The SAS statement DROP, KEEP, and RENAME simply do not work in the tested Procedures. The DROP=, KEEP=, RENAME= and WHERE= data set options work in the tested Procedures, but in a slightly different order than in the DATA step, they have different effects depending on system option settings, and most interestingly, they act differently in combination than in the DATA step. Some of the data set options can be used in places one would not normally think of, like in the CREATE TABLE and FROM clauses in the SQL Procedure, and perhaps even more powerfully in the output statement of the FREQ Procedure. Basically, anywhere there is a data set name, you can use data set options.

Use of WHERE conditions, wild cards, and joins in the SQL Procedure should be used with caution, but at the same time, do not be afraid to experiment.

Experimentation is a very powerful tool. Always experiment, you never know what you might find. I have told my students every semester, "Do not let someone tell you that you cannot do something, because then you will never try". There are a hundred different ways to do everything in SAS, don't be afraid to experiment.

REFERENCES

SAS Institute, Inc. (1999) *SAS Procedures Guide, Version 8*, Cary, NC : SAS Institute, Inc.

SAS Institute, Inc. (2000) *SAS Language Reference: Dictionary, Version 8*, Cary, NC : SAS Institute, Inc.

SAS Institute, Inc. (1999) *SAS Language Reference: Concepts, Version 8*, Cary, NC : SAS Institute, Inc.

SAS Institute, Inc. (2004) *SQL Processing with the SAS System Course Notes*, Cary, NC : SAS Institute, Inc.

SAS Institute, Inc. (February 2000) *SAS OnlineDoc, Version 8*, Cary, NC: SAS Institute, Inc.

Johnson, Jim (May 2003), "**The Use and Abuse of the Program Data Vector**", *Proceedings of the 2003 Conference of the Pharmaceutical Industry SAS Users Group*, Cary, NC: SAS Institute, Inc., 601-610.

RECOMMENDED READING

Johnson, Jim (May 2003), "**The Use and Abuse of the Program Data Vector**", *Proceedings of the 2003 Conference of the Pharmaceutical Industry SAS Users Group*, Cary, NC: SAS Institute, Inc., 601-610. This paper is available on the PharmaSUG website at <http://pharmasug.org/2003/BestPapers/tt070.pdf>.

TRADEMARKS

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.

Other brand and product names are trademarks of their respective companies.

CONTACT INFORMATION

Jim Johnson
Principal Scientific Programmer, Biometrics
Covance Periapproval Services, Inc.
One Radnor Corporate Center, Suite 300
Radnor, PA 19087
jb.johnson@covance.com