Output SAS® DATA Step Views: an Experimental Feature

Thomas E. Billings, Kaiser Permanente, Oakland, California

Abstract

Output DATA step views are a specialized experimental feature in the Base SAS system. They are compiled _null_ DATA steps that are executed as output from a PROC or DATA step. Simple examples of output DATA step views are demonstrated and explained. Differences between output views and other types of views supported in SAS are described. The limitations/constraints of output views are specified, e.g., views are compiled hence the PDV (program data vector) must be fully defined and cannot change, and _null_ DATA steps (by themselves) don't create an output SAS data set. Possible approaches to mitigate these limitations are discussed, including macros to create a set of ATTRIB statements to define the PDV, and the possibility of using a hash object to produce an output SAS data set from a view. Potential applications of output views are described.

DATA step views: an overview

SAS DATA step views are compiled SAS DATA step programs that are executed when invoked. Input DATA step views create a SAS data set that is used as input for a PROC or DATA step. Output DATA step views are _null_ DATA steps, programs that are run to provide a logical DATA step layer on the output side of a PROC or DATA step. Output DATA step views include a SET statement on the view name (a proxy for the physical file the view executes with), which allows transformation and testing of the data in an output SAS data set. However, as we shall see in the code that follows, it is ironic that output data step views do not inherently create *any* data output whatsoever. Instead, all output from the view must be explicitly programmed in the output view code. To make matters worse, significant effort may be required to produce an actual SAS data set using an output view.

As a result of being compiled programs, all DATA step views are subject to the constraint that - once compiled - the PDV (program data vector) cannot change. This means that the PDVs of data sets that are input to a DATA step view must conform to the PDV compiled in the view.

Recognizing that most view programs compile very quickly, a way to minimize failure caused by PDV changes is to recompile the view at regular intervals and/or just before it is used (the choice here depending on the frequency of changes to underlying source files/tables). DATA step views are different from, and should not be confused with, views created in PROC SQL.

Input DATA step views are a standard production feature in the SAS system and a valuable tool for programmers. *Output DATA step views are an experimental feature and should not be used for production programs.* They are highly specialized and of limited practical use. Their best use may be educational, as their non-trivial usage requires one to understand the PDV and other important SAS features, which we explore in the code below.

Sample code

Let's review some examples of DATA step views, to better understand how they work and their limitations.

Sample data. A simple 2-row data set containing a character variable with values "world" and "hello" as found in the "hello world" message so common in computer programming training.

```
1    options nocenter dquote;
2
3    * sample data set;
4
5    data test_data;
6    length testvar $8.;
7    testvar = 'world';
8    output;
9    testvar = 'hello';
```

```
10   output;
11   stop;
12   run;
```

PROC PRINT shows the expected contents:

```
Obs    testvar

 1      world
 2      hello
```

Input DATA step view. Let's modify the sample data set in an input view, and print the resultant data:

```
17   * simple example of an input data step view;
18
19   data input_view / view = input_view;  * note syntax;
20   set test_data;
21   *  transformations go here;
22   x = _n_ *10 ;
23   upcase_version = upcase(testvar);
24   run;

NOTE: DATA STEP view saved on file WORK.INPUT_VIEW.
NOTE: A stored DATA STEP view cannot run under a different operating system.
```

The form of the DATA command above identifies this as an input view. PROC PRINT on the view yields the results:

```
                          upcase_
     Obs    testvar     x    version

       1     world     10     WORLD
       2     hello     20     HELLO
```

Create and execute an output DATA step view.

```
29   *  simple example of an output data step view;
30
31   data _null_ / view=test_view;  * note syntax on data + set;
32   length testvar $8.;
33   set test_view end=endflag;     * note syntax on data + set;
34   put testvar=;
35   put endflag=;
36   run;

NOTE: DATA STEP view saved on file WORK.TEST_VIEW.
NOTE: A stored DATA STEP view cannot run under a different operating system.
WARNING: The definition of an output DATA step view is an experimental feature
in this release
         and is not intended for use in the development of production
applications.

38   proc sort data=test_data out=test_view; * execute view;
39   by testvar;
40   run;
```

```
NOTE: There were 2 observations read from the data set WORK.TEST_DATA.
testvar=hello
endflag=0
NOTE: The data set WORK.TEST_VIEW has 2 observations and 1 variables.
testvar=world
endflag=1
```

Note the syntax on the DATA and SET statements above; they illustrate how to define an output view. The view name on the DATA statement and the same name in the SET statement identify the value in the SET statement as a proxy for a data set to be supplied on invocation. Note the LENGTH statement before SET; the PDV must be fully specified for the data set that will be input into the view. Also, a warning message is issued that this is an experimental feature. Finally, when invoked as output from PROC SORT, the sorted data set variables are printed on the log per the code in the view.

A way to avoid explicitly specifying the PDV in output views. Use a statement similar to line 46 below to use the PDV for an already-defined data set. Note that execution of the view produces similar results as above.

```
44   data _null_ / view=test_view2;
45   *length x 8;   * commented out;
46   if (0) then set test_data;
47   set test_view2;
48   put testvar=;
49   x = _n_*2;
50   put x =;
51   run;

NOTE: DATA STEP view saved on file WORK.TEST_VIEW2.
NOTE: A stored DATA STEP view cannot run under a different operating system.
WARNING: The definition of an output DATA step view is an experimental feature
in this release
         and is not intended for use in the development of production
applications.

52
53   proc sort data=test_data out=test_view2;
54   by testvar;
55   run;

NOTE: There were 2 observations read from the data set WORK.TEST_DATA.
testvar=hello
x=2
testvar=world
x=4
NOTE: The data set WORK.TEST_VIEW2 has 2 observations and 1 variables.
```

If the output view is to operate on a complex data set produced via a PROC, one can create a prototype data set (with no observations but providing the requisite PDV) using code similar to the following:

```
options obs=0;
proc _name_here /* options */ ;
* any additional statements for proc;
run;
* note: the option works with data steps too. ;
options obs=max;    * reset ;
```

Views fail to execute if the (input) PDV does not conform to the compiled PDV. Let's create a new data set from the original test_data file and add 1 variable, then use it as input to the previously compiled output view.

```
59   data test_data_mod;
60   set test_data;
61   y = 1;   * add an extra variable;
62   run;

NOTE: There were 2 observations read from the data set WORK.TEST_DATA.
NOTE: The data set WORK.TEST_DATA_MOD has 2 observations and 2 variables.

64   proc sort data=test_data_mod out=test_view2;
65   by testvar;
66   run;

ERROR: The variable y is not defined in the OUTPUT view WORK.test_view2.
ERROR: Failure loading view WORK.TEST_VIEW2.VIEW. Error detected during View
Load request.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.TEST_VIEW2 was only partially opened and will not be
saved.
```

Can views be used to update data sets? Using a data step to update a data set and overwrite the original file is a common practice in SAS programs. Let's see if we can use a view to perform that process; note the syntax on the DATA and SET statements below:

```
70   data my_update / view=my_update;
71   length testvar $8.;
72   set my_update;
73   * optional: additional transformations as appropriate;
74   put testvar=;
75   run;

ERROR: UPDATE views are not supported.
NOTE: View not saved due to errors.
NOTE: The SAS System stopped processing this step because of errors.
```

It did not work, and understanding the underlying processing provides an explanation. If the code above compiled, it would yield a view, a compiled data step with the name my_update. If the view code executed as listed above, then on the first execution of the view, the view itself would be overwritten by a physical data set. On first execution, the view would apply any transformations in its code, and it would yield the transformed (desired) result. On a second or later run, the view no longer exists, only the physical file. It would be overwritten with a new file version that lacks any transformations performed in the view. Such a setup would create opportunities for confusion.

Can output views be used to perform updates? Let's try:

```
82   data my_update2/ view=my_update;
83   length testvar $8.;
84   set my_update;
85   put testvar=;
86   run;

NOTE: DATA STEP view saved on file WORK.MY_UPDATE.
NOTE: A stored DATA STEP view cannot run under a different operating system.
```

```
WARNING: The definition of an output DATA step view is an experimental feature
in this release
         and is not intended for use in the development of production
applications.
```

An invocation of PROC DATASETS shows that the view my_update is created, but no view or data set named my_update2 is created. However, the data set my_update2 is created when the output view is executed in a PROC or DATA step. So update functionality is provided, but only under a different file name. Giving the updated version a different name avoids the problems described above. However this comes at the cost of potential confusion, e.g., programmers wondering why the code above did not create the file my_update2 when the data step was compiled.

Use ATTRIB statement to describe PDV. A variety of statements can be used to define and describe the variables in the PDV: LENGTH, LABEL, FORMAT, INFORMAT, RETAIN, and ATTRIB. The use of ATTRIB is suggested as it is multi-functional and supports neater/better code. Here is the sample output view, recreated using an ATTRIB statement:

```
95   data _null_ / view=test_view3;
96   attrib
97     testvar format=$8. informat=$char8.
98       label="test character variable" length =$8
99     x format=4. informat=4.
100      label="test numeric variable" length=8;
101  set test_view3;
102  put testvar=;
103  x = _n_*3;
104  put x =;
105  run;

NOTE: DATA STEP view saved on file WORK.TEST_VIEW3.
NOTE: A stored DATA STEP view cannot run under a different operating system.
WARNING: The definition of an output DATA step view is an experimental feature
in this release
         and is not intended for use in the development of production
applications.

107  proc sort data=test_data out=test_view3;
108  by testvar;
109  run;

NOTE: There were 2 observations read from the data set WORK.TEST_DATA.
testvar=hello
x=3
testvar=world
x=6
```

ATTRIB statements can be generated in code. If you have an existing data set that is a prototype for your view, the “if (0) then set filename” approach is usually the easiest way to facilitate creation of the target PDV. However, as an alternative, PROC CONTENTS can produce a file that contains the information needed for an ATTRIB statement. The processing in simplified form is as follows:

- create output file describing target data set using PROC CONTENTS
- use PROC CONTENTS file and code to construct the ATTRIB statement
- load ATTRIB statement into a %GLOBAL macro variable for use in the output view code.

The following proof-of-concept code illustrates the processing:

```
126  %global attrib_stmt;
127  data _null_;
128  length codeline $10000.;
129  retain codeline;
130  length suppline $200. format_suf informat_suf $6 var_type $1;
```

```
131  set contents_3 end=ctfl_end;
132  if (_n_ = 1) then codeline='attrib ';
133  suppline =cat(strip(name));
134  put suppline=;
135  if (not missing(length)) then do;
136    var_type = ' ';
137    if (type = 2) then var_type = '$';
138    suppline = cat(strip(suppline),' ',
       'length=',var_type,strip(put(length,4.)),' ');
139  *put suppline=;
140  end;
141  if (not missing(formatl)) then do;
142    format_suf = cat(strip(put(formatl,4.)),'.',strip(put(formatd,4.)));
143    suppline = cat(strip(suppline),' ','format=',
       strip(format),strip(format_suf),' ');
144  *put suppline=;
145  end;
146  if (not missing(informl)) then do;
147    informat_suf = cat(strip(put(informl,4.)),'.',strip(put(informd,4.)));
148    suppline = cat(strip(suppline),' ','informat= ',
       strip(informat),strip(informat_suf),' ');
149  *put suppline=;
150  end;
151  if (not missing(label)) then
152      suppline = cat(strip(suppline),' ','label= "',strip(label),'" ');
153  *put suppline=;
154  codeline = cat(strip(codeline),' ',strip(suppline));
155  put codeline=;
156  if (ctfl_end) then
157   call symputx('attrib_stmt',codeline);
158  run;

codeline=attrib
         testvar length=$8 format= $8.0 informat= $CHAR8.0
                 label= "test character variable"
         x length= 8 format= 4.0 informat= 4.0 label= "test numeric variable"
NOTE: There were 2 observations read from the data set WORK.CONTENTS_3.

160  * test generated attrib statement;
161
162  options symbolgen;
163  data _null_ / view=test_view4;
164  &attrib_stmt;
SYMBOLGEN:  Macro variable ATTRIB_STMT resolves to [*manually reformatted*]
attrib testvar length=$8 format= $8.0 informat= $CHAR8.0 label= "test character variable"
       x       length= 8 format= 4.0 informat= 4.0 label= "test numeric variable"
165  set test_view4;
166  put testvar=;
167  x = _n_*4;
168  put x =;
169  run;

NOTE: DATA STEP view saved on file WORK.TEST_VIEW4

171  * test the view;
172  proc sort data=test_data out=test_view4;
173  by testvar;
174  run;
```

```
NOTE: There were 2 observations read from the data set WORK.TEST_DATA.
testvar=hello
x=4
testvar=world
x=8
```

Production-quality code needs to handle special characters. The code above could be easily encapsulated in a macro, if desired. The code is described as proof-of-concept because it is not production quality:

- output views are experimental features that should not be used for production jobs,
- variable labels may contain characters with special meaning for the macro processor, making their handling more complex,
- under the appropriate SAS system option, variable names can contain any character, requiring more complex handling in the macro processor, as illustrated in the following example.

```
176  options validvarname=any;
WARNING: Only Base procedures and SAS/STAT procedures have been tested for use with
         VALIDVARNAME=ANY. Other use of this option is considered experimental and may
cause
         undetected errors.

178  data _null_;
179  * test name literals;
180  ' "&my %strange"'; varname 'n = 1;
181  put ' "&my %strange"'; varname'n =;
182  stop;
183  run;

' "&my %strange"'; varname'n=1

185  options validvarname=v7;  * reset back to normal;
```

Note the warning message above, and the unusual variable names that are possible (but very bad programming practice) when the validvarname=any option is activated.

It is possible to write a sophisticated macro here that could be a single-line invocation in the DATA step used to create the output view, i.e., an ATTRIB statement could be created without the need for an external _null_ DATA step. Such a macro would use %SYSFUNC and the relevant SAS file I/O functions to read the PROC CONTENTS file and construct the ATTRIB statement. Given that output views are an experimental feature with specialized and limited uses, it follows that developing an extremely complex macro is probably not cost-effective (in this context).

Hash objects to create an output SAS data set in an output view, part 1. The only way to produce a SAS data set from an output view is by use of hash objects. As a compiled DATA step, output views can send output to a wide range of output destinations: log, list, and external files. But _null_ DATA steps do not produce native SAS files the way regular (i.e., other than _null_) DATA steps do. This is a major limitation of output views.

Recall that the typical usage of input views is to take input file(s) and apply transformations to produce a modified SAS file. This can be done with output views, but it requires significant work, i.e., to describe the PDV, create a hash object to hold the data, then finally create an output SAS data set from the hash object. Programmers should weigh the effort required against simple alternatives that are often available, i.e., just create a standard SAS data set as output and then modify/test it in a successor DATA step or PROC invocation.

The code that follows shows how to create a hash object from an existing SAS data set (or view), load it, update the entries, and finally write the results out as a standard SAS data set. First, an input data step view is created as a way to specify the PDV, then the view is used to load the hash object in the form of the desired output SAS data set.

```
208  data test_data_inview/ view=test_data_inview;
209  set test_data;
210  length x 8.;
211  my_index = _n_;
212  run;

NOTE: DATA STEP view saved on file WORK.TEST_DATA_INVIEW.

216  data _null_ / view=test_view3;
217  if (0) then set test_data_inview; * get PDV for variable definitions;
218  if (_n_ = 1) then do;
219      declare hash ODV(dataset:"test_data_inview");
220      ODV.definekey("my_index");  * need unique index in 9.1;
221      ODV.definedata("testvar","x");
222      call missing(x);
223      ODV.definedone();
224  end;
225  set test_view3 end=end_of_file;
226     x = _N_ *2 ;
227     testvar = upcase(testvar);
228     ODV.replace();
229     put x=;
230     put testvar=;
231  if (end_of_file) then ODV.output(dataset:"updated_version");
232  run;

NOTE: DATA STEP view saved on file WORK.TEST_VIEW3.
NOTE: A stored DATA STEP view cannot run under a different operating system.

234  proc sort data=test_data_inview out=test_view3;
235  by testvar;
236  run;

NOTE: Variable x is uninitialized.
NOTE: There were 2 observations read from the data set WORK.TEST_DATA_INVIEW.
NOTE: There were 2 observations read from the data set WORK.TEST_DATA.

NOTE: Variable x is uninitialized.
NOTE: There were 2 observations read from the data set WORK.TEST_DATA_INVIEW.
NOTE: There were 2 observations read from the data set WORK.TEST_DATA.
x=2
testvar=HELLO
NOTE: The data set WORK.TEST_VIEW3 has 2 observations and 3 variables.
x=4
testvar=WORLD
NOTE: The data set WORK.UPDATED_VERSION has 2 observations and 2 variables.

241  proc print data=updated_version;
242  run;
```

PROC PRINT produced the expected result, but this time from an actual physical SAS data set that was created in an output view:

```
Obs    testvar    x

 1      HELLO     2
 2      WORLD     4
```

Let's review the processing above:

1. An input data view was created to take the sample data set and add a) a unique index variable (needed in 9.1 hash objects, optional in 9.2), b) placeholder for the extra variable "x' that we want to include in the output data set produced from the hash object. (Attempts to add the "x" variable directly to the hash object, using only the definedata method produced error messages.)
2. The output data step view creates and loads the hash object from the target data set.
3. The output data step reloads the same data via SET, transforms it, and replaces the row in the hash object using the replace method.
4. Finally, the transformed data are written from the hash object into an output SAS data set.
5. A test of the output view shows that it runs correctly and produces the target output SAS data set.

Hash objects to create an output SAS data set in an output view, part 2. We should be able to modify the code above to create the hash object from scratch, i.e., without need for an input SAS data set (or view) as template. The code was revised to not use an input data set to create the target hash object and tested as _null_ DATA steps; it worked correctly. The same code, when minor changes were made to convert to output views, produced the error message:

```
ERROR:  Read Access Violation In Task [ DATASTEP )
Exception occurred at (642D5463)
```

which is to say that the modified output view did not even compile. This should serve as a reminder that output views are still experimental features, not for production use.

Possible applications for output views

The significant constraints on output views severely limit their usefulness. A few applications to consider are as follows.

- Complex reports produced in _null_ DATA steps are a possible application, but you should be using Base SAS PROCs (TABULATE, REPORT, FREQ, PRINT, SUMMARY) or SAS Business Intelligence tools instead for reporting, whenever possible
- Use output data step view(s) to force an ABORT under some circumstances, e.g. if there are duplicates in a sort, i.e., PROC SORT DUPS= output_view and the view executes the ABORT statement. Timing and order of data set update in the PROC may be an issue here, so be careful.
- Use output data step view(s) to perform filtering on the output side of a PROC or DATA step as the data are being written, e.g., looking for data that exceed some pre-set limits.

In many – but not all – cases, it may be easier to simply create an output data set and filter or test it after the fact in a conventional DATA step or PROC. SAS output DATA step views are interesting and educational but have limited applications.

Acknowledgements:

The author wishes to thank Jack Hamilton of Kaiser Permanente, Oakland, California for valuable suggestions during the research for this paper. Any mistakes herein are the responsibility of the author.

Contact Information

Thomas E. Billings
Kaiser Permanente
1950 Franklin; 14th floor
Oakland, CA 94612

Phone: 510-987-1320
Email: tebillings@yahoo.com