Paper 070-2012

Techniques for Generating Dynamic Code from SAS® DICTIONARY Data

Jingxian Zhang, Quintiles, Overland Park, Kansas

ABSTRACT

Integrating the information from SAS DICTIONARY tables into programming helps create dynamic and efficient scripts to manage data sets. The purpose of this paper is to provide such techniques for generating dynamic code from SAS DICTIONARY tables. The author uses three macros to demonstrate how DICTIONARY tables-driven code is dynamically constructed via three approaches: SQL select into macro-variable method, call execute method, and generate-and-include an external file method. These macros manage all data sets at the library level: capitalize all character data, dump all data into an excel file, and query all character data with certain length. Using the basic techniques discussed in the paper, SAS programmers can develop their own dynamic scripts to accomplish other tasks.

INTRODUCTION

SAS programmers in Data Management of pharmaceutical industry program and format CDMS (for example, Phase Forward Inform, Medidata Rave, Oracle Clinical) database data and third-party vendor data such as laboratory test results and EKG data to STDM-like data sets per CDISC or sponsors' specification. A clinical trial or study typically has 20 to 30 SAS data sets. During data validation and review processes, SAS programmers often face the challenging questions from the data reviewers: Can you provide a list of all character data whose length >= 190? Can you change all character data to upper case? Can you present the data in excel format instead of data sets for the ease of data review? If you had only one or two small data sets, you might accomplish these tasks by updating each individual program used to generate these data sets. But if you have 30 data sets with hundreds of variables, are you going to use the above, tedious approaches?

The purpose of this paper is to find a better solution that can accomplish the above tasks. I will first discuss how SAS DICTIONARY tables and their corresponding SASHELP views are accessed and what information is available to SAS users. Then I will show how to dynamically construct SAS DICTIONARY tables–driven code via three approaches – SQL select into macro- variable method, call execute method, and generate-and-include an external file method.

SAS DICTIONARY LIBRARY

The SAS System read-only DICTIONARY tables and corresponding SASHELP views provide valuable information about SAS libraries, data sets, columns and attributes, catalogs, indexes, macros, system options, titles, views, and more. This information has been used to verify data set structures and attributes (Teng and Wang, 2006) and data inconsistencies in multiple data sets (Murphy, 2011).

DICTIONARY is a special case of a SAS library. DICTIONARY tables are special read-only PROC SQL tables or views. DICTIONARY tables are accessed only by PROC SQL. Based on the DICTIONARY tables, SAS provides PROC SQL views that can be used in other SAS procedures and in the DATA step. These views are stored in the SASHELP library and are commonly called SASHELP views.

To see what tables the DICTIONARY has, one can run the following code and will get a list of 29 tables (Table 1). Table names in Table 1 are meaningful so that you know what they are about.

```
PROC SQL;
   SELECT UNIQUE memname
   FROM DICTIONARY.dictionaries;
QUIT;
```

Abbreviations: CDISC, Clinical Data Interchange Standards Consortium, a non-profit group that defines clinical data standards for the pharmaceutical industry; CDMS, Clinical Data Management System, a tool used in clinical research to manage the data of a clinical trial. The clinical trial data gathered at the investigator site in the case report form are stored in the CDMS. EKG, Electrocardiogram (e-lek-tro-KAR-de-o-gram), also called ECG, a simple, painless test that records the heart's electrical activity; STDM, Study Data Tabulation Model, defines a standardized structure for data tabulations that are to be sent to the FDA as part of a regulatory submission.

Table 1. DICTIONARY Tables and Their Corresponding SASHELP Views (SAS 9.2)

Dictionary Table	*SASHELP Views*	*Dictionary Table*	*SASHELP Views*
CATALOGS	VCATALG	INFOMAPS	VINFOMP
CHECK_CONSTRAINTS	VCHKCON	LIBNAMES	VLIBNAM
COLUMNS	VCOLUMN	MACROS	VMACRO
CONSTRAINT_COLUMN_USAGE	VCNCOLU	MEMBERS	VMEMBER
CONSTRAINT_TABLE_USAGE	VCNTABU	OPTIONS	VOPTION
DATAITEMS	VDATAIT	PROMPTS	VPROMPT
DESTINATIONS	VDEST	PROMPTSXML	VPRMXML
DICTIONARIES	VDCTNRY	REFERENTIAL_CONSTRAINTS	VREFCON
ENGINES	VENGINE	REMEMBER	VREMEMB
EXTFILES	VEXTFL	STYLES	VSTYLE
FILTERS	VFILTER	TABLES	VTABLE
FORMATS	VFORMAT	TABLE_CONSTRAINTS	VTABCON
FUNCTIONS	VFUNC	TITLES	VTITLE
GOPTIONS	VGOPT	VIEWS	VVIEW
INDEXES	VINDEX		

To see what views SASHELP views have, one can use the following syntax:

```
PROC SQL;
          SELECT DISTINCT memname
          FROM SASHELP.vsview
          WHERE libname='SASHELP'AND memname like 'V%';
QUIT;
```

To see how each DICTIONARY table is defined, submit a DESCRIBE TABLE statement. For example, the following code would show the definition of DICTIONARY.columns. The results are written to the SAS log, which has the variable names and their attribute information.

```
PROC SQL;
          DESCRIBE TABLE DICTIONARY.columns;
QUIT;
```

To examine the contents of each view, one can use the following syntax, where view_name is any valid view.

```
PROC CONTENTS DATA = SASHELP.view_name;
RUN;
```

For more information about DICTIONARY information, readers are referred to Eberhardt and Brill (2007), Thornton (2011) and CodeCrafters, Inc. (2010).

SQL SELECT INTO MACRO-VARIABLE APPROACH

It is pretty easy to check data length via length function. To check if a variable has data length >= 190, one could use either "< Data step approach: DATA test; SET dataset_name; IF LENGTH (variable) GE190; RUN;>" or "<SQL approach: PROC SQL; SELECT variable FROM dataset_name WHERE LENGTH (variable) >= 190; QUIT; >".

These two approaches work fine if you have only a few variables. However, it would be tedious or impossible to implement if one needs to deal with hundreds of variables. Using the DICTIONARY.columns and dynamic programming techniques, I have developed the following macro %query_length to accomplish such tasks.

```
%MACRO query_length(mylib=, maxlen=);
   PROC SQL;
         /**** Section I ****/
   CREATE TABLE columns as
     SELECT    memname, name, length, type
     FROM      DICTIONARY.columns
     WHERE     libname = "%upcase(&mylib)" AND
               memtype = "DATA" AND
               type="char";

    %LET cnt = &sqlobs;

         /**** Section II ****/
   SELECT "SELECT UNIQUE '" || STRIP(memname) || "' as DSNAME,'"
         || STRIP(name) || "' as VARNAME, SUBJID, length(strip("
         || STRIP(name) || ")) as MAX_LENGTH, "
         || STRIP(name) || " as DATAVALUE FROM  &mylib.."
         || STRIP(memname) || "
    WHERE length(strip("|| STRIP(name) || ")) >= &maxlen.; "
    INTO :select1-:select&cnt
   FROM columns;

          /**** Section III ****/
   CREATE TABLE  tempds
         (DSNAME                char(10),
          VARNAME               char(10),
          SUBJID                char(20),
          DATALENGTH                    num,
          DATAVALUE             char(200));

   %DO i=1 %TO &cnt;
    INSERT INTO tempds
    &&select&i;
    %END;
  QUIT;
  %MEND query_length;
```

Special Note: *For this specific macro, the variable subjid(subject ID) must exist in all data sets at mylib library because I want to know which subjid has the queried results.*

The macro %query_length queries all character variables that have data length >= a specified number. The DICTIONARY.columns table used in section I shows data set information at the variable level. Section I in the macro is to get the memname (dataset name), name (variable name), length (variable defined length) and type (variable data type) on all character variables at a given library. Using the returned information from section I, section II dynamically constructs SQL query code which will be used to populate a data set tempds in section III. The macro call gives output for the SELECT statement for each variable that has data length >= &maxlen. Table 2 shows what has been returned on the test data I used when maxlen is 190.

Table 2. Returned Records from Macro QUERY_LENGTH call with &maxlen = 190

DSNAME	VARNAME	SUBJID	DATAVALUE	LENGTH
EG	EGORRES	99994-006	ABNORMAL, BUT NOT CRITICAL. NOT DONE PREDOSE NOR BEFORE LAB DRAW. STAFF FORGOT TO HOLD AM DOSE AND PT UNCOOPERATIVE WITH INITIAL ATTEMPT TO COMPLETE EKG. ALL OTHER PROCEDURES COMPLETED THEN REA	194
FA	FACOM	99993-007	THIS IS NOT DONE DUE TO SAMPLE PROBLEM. IMPRESSION: STATUS POST VERTEBRAL BODY AUGMENTATION OF T011 AND T012. STABLE COMPLETE COLLAPSE OF THE T10 VERTEBRAL BODY. INTERVAL LOSS OF VERTEBRAL LAB ONE	197
LB	LBNAM	99993-010	THIS IS MIDWEST SAS LAB TWO FOR TEST DATA. THIS IS THE TEST LAB CENTER USED TO CREATE THE TEST DATA USED FOR THIS PRESENTATION. DUE TO REGULATIONS, THIS LAB MAY OR MAY NOT BE USED FOR PATIENTS. HOW,	199

The key code of the macro is the SELECT INTO statement that has the following syntax:

```
SELECT Column(s)
 INTO :<Macro Variable name1> -:<Macro Variable name9999>
 FROM Table-name | View-Name;
```

The SELECT statement stores returned row values in a list of user-defined macro variables. Only the required number of macro variables will be created. A number large enough to hold the number of observations returned from the SELECT statement must be specified in the INTO statement if system macro variable SQLOBS is not used. In the macro, I have used “INTO :select1-:select&cnt”.

The syntax of string concatenation in Section II needs a little explanation. The following statement

```
SELECT "SELECT UNIQUE '" || STRIP(memname) || "' as DSNAME,'"
        || STRIP(name) || "' as VARNAME, SUBJID, length(strip("
        || STRIP(name) || ")) as MAX_LENGTH, "
        || STRIP(name) || " as DATAVALUE FROM  &mylib.."
        || STRIP(memname) || "
    WHERE length(strip("|| STRIP(name) || ")) >= &maxlen.; "
```

will be dynamically decoded to the following code after execution if memname (DSNAME) = EG and name (VARNAME) = EGORRES and &maxlen = 190:

```
SELECT UNIQUE 'EG' as DSNAME, 'EGORRES' as VARNAME,  SUBJID,
     length(strip(EGORRES)) as MAX_LENGTH,  EGORRES as DATAVALUE
FROM LIBREF.EG  WHERE length(strip(EGORRES)) >=  190;
```

***Special note:** LIBREF is the mylib when the macro is called.*

Similar SELECT statements for other variables will be dynamically generated and are kept under macro variable select*i* (where *i* is between 1 and SQLOBS. For my test data, SQLOBS = 639). This is accomplished via dynamic variables used in the macro. Basic dynamic variables <''' || VARIABLE || '''> and <'" || VARIABLE || "'> are commonly used in dynamic SQL code generation and they are different. When 'variable' is needed in the generated script, ''' || VARIABLE || ''' should be used. When VARIABLE is needed in the generated script, " || VARIABLE || " should be used. For example, if name is EGORRES and you want ' EGORRES ' to show up in the script, then you need to use ''' || name || '''; However, if you want EGORRES to show up as a variable in the script, then you need to use " || name || ".

GENERATE-AND-INCLUDE AN EXTERNAL FILE APPROACH

After I have programmed 23 data sets, I got a request to have all character data to be capitalized. Instead of modifying each individual program, we can use the dictionary metadata to do the capitalization. Here I have used a different approach. Using the DICTIONARY metadata information, the macro %upcase_ds below generates SAS code via “put statement” and then writes it to an external file. To execute the macro %upcase_ds, one can run it and then use the “%include statement” to include the file (i.e., upcase_chardata.sas) generated from the macro call.

```
%MACRO upcase_ds(mylib=, _filepath=);
 PROC SQL;
   CREATE TABLE columns as
     SELECT memname as member, name as variable
     FROM dictionary.columns
     WHERE libname = "%upcase(&mylib)" and
          memtype="DATA" and type="char"
     ORDER by member, variable;
    QUIT;

 DATA _null_;
    SET columns end=EOF;
     BY member variable;
    FILE  "&_filepath.\upcase_chardata.sas";
    dlm = byte(9);
    IF _n_ =1 then do;
        PUT 'PROC SQL;';
     END;
    PUT dlm +(-1) 'UPDATE ' "&mylib.." member ' SET ' variable '
      = UPCASE(' variable +(-1) ');';
     if EOF then do; put 'QUIT; ';
     END;
 RUN;
%MEND upcase_ds;
```

The following is part of the UPDATE statements contained in the dynamically generated file upcase_chardata.sas when the macro is called (FA is one of the data sets present in the mylib LIBREF). For my test data, I have 23 SAS data sets and 639 char variables. That means 639 “update statements” will be generated if the macro is called. You can see how powerful this technique is.

```
PROC SQL;
        UPDATE LIBREF.FA  SET DOMAIN        = UPCASE(DOMAIN);
        UPDATE LIBREF.FA  SET FABLFL        = UPCASE(FABLFL);
        UPDATE LIBREF.FA  SET FACAT         = UPCASE(FACAT);
        UPDATE LIBREF.FA  SET FACOM         = UPCASE(FACOM);

<< Total 639 UPDATE statements will be generated for the test data used>>

QUIT;
```

In case readers wonder how this task can be done via “SQL select into macro-variable method”, here is the code:

```
%MACRO upcase_chardata(mylib);
 PROC SQL;
   CREATE TABLE columns as
   SELECT memname, name, length, type
   FROM DICTIONARY.COLUMNS
   WHERE libname = "%upcase(&mylib)" and  memtype="DATA" and type="char";

   %let cnt = &sqlobs;
   SELECT "update &mylib.."||strip(memname)||" set "||name||"  =
       upcase("||strip(name)||");"
   INTO :update1-:update&cnt
  FROM columns;

  %DO i=1 %TO &cnt;
    &&update&i;
   %END;
 QUIT;
%MEND upcase_chardata;
```

CALL EXECUTE APPROACH

When exporting multiple SAS data sets to Excel files, the traditional method is to write multiple steps as below.

```
PROC EXPORT DATA = LIBREF.AE  DBMS =excel2000
 OUTPUT = "_outpath\ae.xls" REPLACE; SHEET = "AE";
RUN;

PROC EXPORT DATA = LIBREF.CM DBMS =excel2000
  OUTPUT = "_outpath\cm.xls" REPLACE; SHEET = "CM";
RUN;
```

Consider the scenario that one wants to dump all the data sets in the library to one excel file with each tab corresponding to a data set. Instead of writing this block of code many times for exporting each data set, I have developed a sas_to_excel macro (mylib= , _outpath = , _project =) by using SASHELP view VTABLE and CALL EXECUTE routine, where mylib is libref, _outpath specifies output folder and _project is the excel output file name.

```
%MACRO sas_to_excel(mylib= , _outpath = , _project =);
  %Macro printds(libname,dsname) ;
       PROC EXPORT DATA = &libname..&dsname DBMS = excel2000
         OUTFILE = "&_outpath.\&_project..xls" REPLACE; SHEET = "&dsname";
       RUN;
   %MEND printds;

  DATA _null_ ;
  SET sashelp.vtable ;
  WHERE libname = "%upcase(&mylib)";
  CALL EXECUTE('%printds('||strip(libname)||','||strip(memname)||')' ) ;
  RUN;
%MEND sas_to_excel;
```

Note that CALL EXECUTE is used within a data step and has the following syntax: CALL EXECUTE (argument). The macro uses the %printds as the argument for CALL EXECUTE and gets LIBREF and MEMNAME from SASHELP.VTABLE. For other uses of CALL EXECUTE, readers are referred to Ruelle and Moses (2006) and Michel(2005).

CONCLUSION

The author used three case study macros to demonstrate how SAS code is dynamically built based on the DICTIONARY metadata. The techniques presented maximize procedural efficiency and lighten the workload of routine processing. Based on their expertise, SAS programmers may select a proper method and develop their own dynamic scripts to accomplish programming tasks.

REFERENCES

CodeCrafters, Inc., 2010. Summary of SAS DICTIONARY Tables and Views. http://www.codecraftersinc.com/pdf/DICTIONARYTablesRefCard.pdf

Eberhardt, P. and Brill, I. SAS Global Forum 2007. How Do I Look it Up If I Cannot Spell It:An Introduction to SAS® Dictionary Tables. http://www2.sas.com/proceedings/forum2007/235-2007.pdf

Michel, D. SUGI 2005. CALL EXECUTE: A Powerful Data Management Tool. http://www2.sas.com/proceedings/sugi30/027-30.pdf

Murphy, W.C. SAS Global Forum 2011. Who Do You Have? Where Are They? http://support.sas.com/resources/papers/proceedings11/104-2011.pdf

Ruelle, A. and Moses, K. PharmaSUG 2006. CALL EXECUTE: A Primer. http://www.lexjansen.com/pharmasug/2006/technicaltechniques/tt16.pdf

Teng, C and Wang, W. PharmaSUG 2006. Simple Ways to Use PROC SQL and SAS DICTIONARY TABLES to Verify Data Structure of the Electronic Submission Data Sets.. http://www.lexjansen.com/pharmasug/2006/tutorials/tu03.pdf

Thornton, P. SAS Global Forum 2011. SAS® DICTIONARY: Step by Step. http://support.sas.com/resources/papers/proceedings11/264-2011.pdf

ACKNOWLEDGMENTS

I would like to thank Jill Jenia for her support, encouragement and helpful remarks.

CONTACT INFORMATION

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

Name:	Jingxian Zhang
Enterprise:	Quintiles, Inc.
Address:	6700 W 115th St
City, State ZIP:	Overland Park, KS 66223
Work Phone:	(913)-708-6674
Fax:	(913)-871-9569
E-mail:	jing.zhang@quintiles.com
Web:	www.quintiles.com