PharmaSUG 2015 - Paper SS01

Getting Loopy with SAS® DICTIONARY Tables: Using Metadata from DICTIONARY Tables to Fulfill Submission Requirements

Jeanina Worden, Santen Inc., Emeryville, CA

ABSTRACT

Standardizing data allows for a quick understanding of the contents for FDA reviewers however this is only the case when the standards are followed. Lapses in compliance can create delays in the review process, or worse, give the reviewers concerns over the quality of the data. The DICTIONARY tables that are accessible within Base SAS® provide data on your data that can be utilized to check, or create, compliant datasets. This paper will provide examples of requirements and how, with a little looping, issues can be identified and requirements can be met.

INTRODUCTION

Submission requirements are nothing new to pharmaceutical programmers however with the FDA and PMDA announcements late last year; those requirements will include compliance with CDISC standards. Though programming of submission datasets using these standards may also be “old hat”, these announcements may increase the current level of compliance to these standards. Compliance goes beyond ensuring the acceptance of the study by the regulatory agency; it can impact the timeliness of the review. Therefore, as the value of the study, both to sponsors and patients, is only realized once the agency’s review is complete, and decision made, compliance of all requirements cannot be overstated. SAS® can be an invaluable tool in ensuring adherence to the requirements. While the end result of programming efforts can utilize other tools to verify standards have been met, this can result in rework at critical periods. Programmers and managers can however use the metadata available in the SAS® DICTIONARY tables to assess compliance throughout the development process. After a brief description of what these tables are, and how to access them, this paper will provide examples of how they can be used in direct response to submission requirements. While there are several tables to choose from within the system, two specific DICTIONARY tables will be highlighted, COLUMNS and TABLES. Moving through the development process the information contained within these tables will be utilized to address a specific requirement. First they will be used to identify issues going from the raw data to a SDTM structured dataset. Moving then to the data’s content, formatting requirements will be addressed by identifying date fields and making the necessary conversions. Then, identification of actual to planned size differences variables will be assessed as a finalizing step to ensure the dataset is at its smallest size. Finally the metadata will be used to identify all of the final datasets so final transport versions can be created for the submission. The code found within this paper was developed using SAS® 9.2 with the intention of demonstrating the concept being discussed, not as a definitive solution. There are numerous ways to capitalize on the data provided within these, and all of, the DICTIONARY tables. It is the hope of this author that anyone familiar with a DATA step and/or PROC SQL, and familiarity with utilizing “looping” code within SAS® can use the concepts to facilitate compliance with submission requirements.

DEFINING THE DICTIONARY “TABLES”

DICTIONARY tables and their associated SASHELP views are commonly referred to as a “tables” but they are actually read-only SAS® data views. They are automatically populated at the start of any SAS® session with information about the current session such as data about the libraries, data sets, macros, and external files. As work begins within a session, these tables are dynamically updated, reflecting any changes made during the session (i.e. new libraries, contents of WORK, newly created variables). The data within the tables can be utilized like that of any other dataset as source data for queries, merging or within SAS® procedures. While the data in the tables can be valuable to the programmer, it is vital to keep in mind how, and when, the tables are populated. The tables will only contain information regarding the data at the time they are accessed, so information regarding variables or libraries yet to be assigned will not be available. Additionally, if temporary datasets are overwritten as the program runs, only the information of the current state of the data will be represented when the table is accessed. Some examples of the DICTIONARY tables, and their corresponding views, are:

Table/View Name	**Content description**
COLUMNS/VCOLUMN	Contains variable attributes, similar to the output from PROC CONTENTS
FORMATS/VFORMAT	Contains attributes of system and user defined formats
LIBNAMES/VLIBNAM	Contains attributes of allocated libraries, including the path

MACROS/VMACRO	Contains macro variable values, including the variables scope (global, local, automatic)
TABLES/VTABLE	Contains attributes of the tables and views, including number of observations

Table 1. Partial list of available DICTIONARY tables/SASHELP views

A complete list of the tables/views, and a description of their contents, can be found in the SAS® 9.2 SQL Procedure User's Guide.

As noted above, the two tables that will be utilized in the following examples are COLUMNS and TABLES and how they are accessed, and named, is dependent on the procedure used, using a DATA step or PROC SQL. Take, for example, the need to get data on the columns within the CLASS dataset in the SASHELP library. If a DATA step is used then the view would be accessed through the SASHELP library, and the view identified with a leading "V", so column data is contained in the view called VCOLUMN. An example of a program using a DATA step to set the VCOLUMN view is:

```
data columns;
       set sashelp.vcolumn;
       where libname = 'SASHELP‘ and memname=‘CLASS’;
run;
```

The WHERE statement is important as it is included to filter the VCOLUMN view to the desired dataset's information, otherwise the view will contain all libraries, datasets, and columns available at run-time. So if a dataset called CLASS is stored in the library SASHELP and another of the same name is in WORK library, the omission of the library name would return information of the datasets in both libraries.

PROC SQL can also be used to access these tables; however the library name and dataset name are modified to DICTIONARY, for the library, and COLUMNS, for the dataset. The same example as stated above would be written as:

```
proc sql;
       create table columns as
       select *
       from dictionary.columns
       where libname = 'SASHELP‘ and memname = ‘CLASS’;
quit;
```

One way to remember which method to use with which name is by the letter at the start or end of the name:

- SASHELP.v<table> (V = SASHELP View)
- DICTIONARY.<table>s (S = SQL)

A couple of final notes regarding accessing are of performance. Beyond the familiarity, and preference, of the programmer in regards to the method being used, the decision of which method may be dependent of the time it takes for the request to process. For example, the logs that resulted from the above two sets of code indicated a difference of several seconds:

```
       NOTE: Table WORK.COLUMNS created, with 921 rows and 18 columns

       NOTE: DATA statement used (Total process time):
             real time           55.59 seconds
             cpu time            8.70 seconds
…
       NOTE: PROCEDURE SQL used (Total process time):
             real time           0.17 seconds
             cpu time            0.17 seconds
```

Additional testing would be needed to determine if this is a consistent pattern but, during development, it is something to be mindful of. Also, as these tables are dynamically "growing", it's worth noting SAS® does not maintain query information for DICTIONARY tables between runs, therefore as the contents of the session grows performance could also be impacted if they are accessed frequently.

Now that we know what the DICTIONARY tables are, how they are populated, and how we can access them, we can take a look at how we can use them.

DEVELOPING THE SPECIFICATIONS: MAPPING NEEDS (ASSESSMENT)

As previously stated, the FDA and PMDA have both announced the requirement of SDTM compliant domains for submissions, for studies started (FDA) or submitted (PMDA) after 24 months of the announcement's December 2014 publication date. In an ideal world the data collected would be in exact SDTM structure straight so no conversion would be needed for data coming from the database. However factors such as data entry requirements or monitoring needs often make this unrealistic therefore some mapping will likely be required for the submission-ready domains to be compliant. But how can the programmer, or their manager, be certain of the amount of mapping that is needed? The answer could be provided with a merge of a dataset containing the SDTM Implementation Guide (IG) structure (spreadsheet can be downloaded from the CDISC website then converted to a SAS® dataset) with the COLUMNS/VCOLUMNS table, using the data captured in the variables MEMNAME (dataset name) and NAME (variable name). A simple string match example would be:

```
*******************************************************************************;
***    1. Set dataset with SDTM configuration and add a column for the derivation;
***    2. Merge metadata with the SDTM configurations;
***************************************************************************;
%let vsdtm = 313; ***Identifies version of SDTM to be used;
proc sql;
       create table id_mapping as
       select a.*, b.*
       from dictionary.columns (where= (libname = 'RAW')) as a full join
                  sdtm.sdtm_&vsdtm. as b
       on a.memname = b.Domain_Prefix and a.name = b.Variable_Name;
quit;
```

This code returns all variables from both the raw dataset and SDTM structure. The resulting dataset will clearly show where the variables match and where variables, on either side, have no corresponding partner. Below is a sample of some of the records, and variables, the resulting dataset would contain:

memname	name	label	length	format	type	Domain _Prefix	Variable_ Name	Variable_Label
						AE	STUDYID	Study Identifier
AE	AETERM	AETERM	600	$400.	char	AE	AETERM	Reported Term for the Adverse Event
AE	AESTDAT	AESTDAT	8	DATETIME22.3	num	AE		

Table 2. Example of ID_MAPPING records

These records show how a SDTM variable, STUDYID, is not present in the raw dataset, AETERM is in both datasets and AESTDAT is in the raw dataset but does not match a SDTM variable. This dataset can then be reviewed to identify variables in which mapping will be required to produce SDTM compliant datasets. This simple merge does not take into account the fact not all SDTM variables are "Required", or "Expected", therefore additional filtering will need to be applied if a higher level of granularity is desired. This code can be further expanded to dynamically do the merge on an individual dataset basis, using the MEMNAME variable in TABLES to collect all the datasets names, and the NAMES variable in COLUMNS as the string being compared. The resulting output would then be domain-specific and could be used to begin mapping documentation (also required for traceability). It could be coded as:

```
*******************************************************************************;
***    1. The VTABLE view can be sorted to create a dataset containing dataset names
within a library;
***    2. Next, using the newly created dataset, WORK.DSN, create code to loop through
each dataset, and compare to the corresponding SDTM domain structure;
***************************************************************************;
proc sort data=sashelp.vtable out=dsn;
       by memname;
       where libname='RAW';
run;

%macro COMPARE_DSN;
```

```
data _null_;
      set dsn end=last;
      call symput('DSN'||left(_n_),trim(MEMNAME));
      if last then call symput('count',_n_);
run;

%do i = 1 %to &count;
****Sort VCOLUMN to only the raw dataset needed;
proc sort data= sashelp.vcolumn out=columns;
      by libname;
      where libname="RAW" and memname="&&dsn&i.";
run;

****Compare each dataset within RAW *;
proc sql;
      create table &&dsn&i. as
            select a.*, b.*
      from columns as a full join
      sdtm.sdtm_&vsdtm. as b
      on b.Domain_Prefix = "&&dsn&i." and
            a.name = b.Variable_Name;
quit;
%end;
%mend COMPARE_DSN;
%COMPARE_DSN;
```

WORKING WITH THE DATA: ISO DATE FORMATTING

Another area in which the CDISC SDTM standards are needed is in the formatting of the data. One such standard is regarding dates that, though collection may be in multiple formats (i.e. DATE9., MMDDYY10.) for ease of entry or visual understanding, are required by the SDTM standards to be presented in ISO8601 date format. The previous example used a combination of the two tables but here the DICTIONARY.COLUMNS table can be used alone to accomplish this task. The NAME column within the table can be used to identify date fields within each dataset within a library by using the variable name. Once the variables are identified, the information can be further employed to loop back through the datasets to create a corresponding numeric, then ISO, date for each date field. An example of how this can be done is:

```
*******************************************************************************;
***   1. The COLUMNS table is filtered to identify variables across datasets that
have ‘DT’ in the variable name;
***   2. Next, using the newly created dataset, WORK.VARS, create code to loop
through each dataset, create numeric and ISO dates for all date fields identified, and
set the updated datasets to a new library (to maintain original);
****************************************************************************;
proc sql;
      create table vars as
      select distinct memname, name
      from dictionary.columns
            where libname = "RAW" and name like '%DT'
            group by memname;
quit;

data _null_;
      set vars;
      by memname;
      call execute ('data TEMP.'|| strip(memname) ||';');
      call execute ('set RAW.'|| strip(memname) ||';');
… insert code to check for complete dates;
      call execute (strip(name)||'_N = input('||strip(name)||',b8601da.);');
      call execute (strip(name)||'_I = put('||strip(name)||'_N,is8601da.);');
      call execute ('run;');
run;
```

SUBJID	Sex	Age	DOBDT	OTHDT
123	M	14	19701201	19800101
345	F	13	19751201	19850101
567	F	13	19751201	19850101
789	F	14	19751201	19850101

SUBJID	Sex	Age	DOBDT	OTHDT	DOBDT_N	DOBDT_I	OTHDT_N	OTHDT_I
123	M	14	19701201	19800101	3987	1970-12-01	7305	1980-01-01
345	F	13	19751201	19850101	5813	1975-12-01	9132	1985-01-01
567	F	13	19751201	19850101	5813	1975-12-01	9132	1985-01-01
789	F	14	19751201	19850101	5813	1975-12-01	9132	1985-01-01

Table 3. Example of RAW dataset and new TEMP dataset with new numeric and ISO dates

A caveat is that the above solution is dependent on naming conventions being in place to ensure consistency across all datasets in the raw database. If this is not the case, and labeling conventions are consistent (i.e. all variables contain "DATE" in the label), the table also contains the variable LABEL, which contains the label information and could be used as an alternative. (LABEL is also available in the TABLES view and can be useful in checking the dataset labels are applied and accurate according to the SDTM IG, but this will not be covered here)

FINALIZING THE DATASETS: PLANNED LENGTH vs ACTUAL

Once the datasets have been finalized for structure and content appropriately formatted, one additional check must be performed to meet the FDA's requirement of having the dataset size as minimized as possible. Variable size is usually set to the largest possible size during development to accommodate the uncertainty of data collection. Often character fields are created at a size of 200, however when the data is finalized the maximum length actually used is much smaller. Again, the DICTIONARY tables contain information to aid in compliance of this requirement by enabling the identification of these differences so that issues can be addressed. The COLUMNS table contains each variables assigned length in the variable LENGTH. This can be used to compare the "planned" length to what the actual length of the data is for each variable. The example code below can be used to identify the length of each variable, assess the actual length of the variables within the datasets, and then combine the information and flag where there is a difference.

```
*****************************************************************************;
*** 1. The TABLES is used to create a dataset contain all the dataset names
*** 2. The new DSN dataset is used to create temporary datasets for each dataset found
       in ADAM (note: the method below generates UNINITIALIZED warnings)
*** 3. The COLUMNS table is then filtered to create a dataset containing all character
       variables in all datasets within the ADAM library;
*** 4. Using the newly created dataset, WORK.VARS, create code to loop through each
       dataset, creating a temporary dataset to hold the data;
*** 5. Update the temporary dataset by looping through the ADAM datasets and with the
       maximum length of each variable (note: this generates warnings due to it
       recursively referencing the target table);
*** 6. Finally, merge the COLUMNS information in and flag if the actual length is
       different than the assigned length;
**************************************************************************;
proc sql;
       create table dsn as
       select distinct memname
       from dictionary.tables
              where libname = "ADAM"
       group by memname;
quit;

data _null_;
       set dsn;
       call execute ('data TEMP.'|| strip(memname) ||';');
       call execute ('length memname $ 32 name $ 200 varlen 8;');
       call execute ('format varlen best12.;');
       call execute ('format varlen best12.;');
       call execute ('stop;');
       call execute ('run;');
run;

proc sql;
       create table vars as
       select distinct memname, name, type, length
       from dictionary.columns
       where libname = "ADAM" and type="char"
       order by memname;
quit;

data _null_;
       set vars;
       by memname;
       call execute ('proc sql;');
       call execute ('create table TEMP.'||strip(memname)||' as');
       call execute ('select "'||strip(memname)||'" as memname, "'||strip(name)||'" as
name, max(length('||strip(name)||')) as varlen');
       call execute ('from ADAM.'||strip(memname));
       call execute ('union');
       call execute ('select * from TEMP.'||strip(memname));
       call execute ('quit;');

       call execute ('proc sql;');
       call execute ('create table TEMP.'||strip(memname)||'_ck as');
       call execute ('select a.memname, a.name, a.length, b.varlen,');
       call execute ('case');
       call execute ('when a.length ^= b.varlen then 1');
       call execute ('else 0');
       call execute ('end as diffflg');
       call execute ('from vars as a, TEMP.'||strip(memname)||' as b ');
       call execute ('where a.memname = b.memname and');
       call execute ('a.name= b.name;');
       call execute ('quit;');

run;
```

The following is a snapshot of the ADAE data that results from the above code:

memname	name	length	varlen	diffflg
ADAE	AEACN	16	16	0
ADAE	AEBODSYS	200	67	1
ADAE	AEDECOD	200	39	1
ADAE	AEDICT	12	12	0
ADAE	AEDTHFL	1	1	0
ADAE	AEENDTC	20	10	1

Table 4. Example of TEMP.ADAE_CK dataset with differences in length flagged

The differences are flagged either as 0, if the assigned length matches the actual data's maximum length within the variable, or 1 if they are different. This information can be used to create a report so that action can be taken, or the code expanded to address the issues once identified. It is worth noting the requirement also specifies that the same variable must have consistent lengths across all datasets. The determination on how to handle differences across datasets is often company specific (i.e. keep the longest length from all datasets) and therefore not covered here however as the concept is similar, the example code is focused only on single datasets.

SUMISSION READY: XPT DATASETS

The final requirement covered here also corresponds to the final step before submission of the datasets. The FDA requires datasets to be submitted in XPT format, so each dataset has to be converted to an individual XPT file. The MEMNAME variable within DICITONARY.TABLES can once again be taken advantage of. As with the code above, a temporary dataset can be created to identify the dataset names, and then a loop developed to convert each dataset in the library to XPT format.

```
*****************************************************************************;
*** 1. The TABLES is used to create a dataset contain all the dataset names
*** 2. The XPT library is defined using the current dataset name and the dataset is
       converted
**************************************************************************;
proc sort data=sashelp.vtable out=dsn;
       by memname;
       where libname=‘ADAM';
run;

data _null_;
       set dsn;
       call execute ("libname xptds xport 'U:\ADAMOUT\"||strip(memname)||".xpt';");
       call execute ('proc copy in=adam out=xptds;');
       call execute ('select '||memname||';');
       call execute ('run;');
run;
```

CONCLUSION

The SAS® DICTIONARY tables can be accessed via a DATA step or using PROC SQL, only changing the library and naming depending on the method. This paper has journeyed through the development process to reach submission ready datasets. At each step an example of a submission requirement was highlighted with a way for the metadata available in these tables, specifically COLUMNS and TABLES, to be utilized to comply with the standards put forth by regulatory agencies. In the first example a combination of these tables to demonstrate a way to identify structural differences between the raw data and the required SDTM domain structures. Next dates across all datasets within the raw library were identified and converted, first to numeric, then to the required ISO8601 format. Then the size of the datasets were checked by looking at the planned versus actual lengths of each variable, resulting in a dataset that could be reviewed so differences could be made to meet the FDA's requirement of minimizing size prior to submission. And finally, the requirement of delivery of XPT datasets for submission packages was accomplished by creating a loop to convert each individually based on the contents of the library being submitted. These examples, while not the only solution, or the only requirements, are meant to provide the reader with ways to take advantage of the metadata provided within these tables, either as a stand-alone source of data or in combination. It is true that other tools can be used once the datasets are completed so that compliance can be checked, however these tables allow requirements assessment to be included in the development process, thereby minimizing rework later in the process and ensuring issues can be identified and corrected before submission.

REFERENCES

Lafler, Kirk Paul. 2005. “Exploring DICTIONARY Tables and SASHELP Views.” Proceedings of the SUGI 30 Conference. Available at URL: http://www2.sas.com/proceedings/sugi30/070-30.pdf

SAS® 9.3 SQL Procedure User's SAS® 9.3 SQL® Procedure User's Guide. Available at URL: https://support.sas.com/documentation/cdl/en/sqlproc/63043/PDF/default/sqlproc.pdf

ACKNOWLEDGMENTS

I would like to thank Wei Liu for his support in the creation of this paper. I would also like to thank my Santen co-workers who helped review, and/or provide input for, this paper.

CONTACT INFORMATION

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

Name: Jeanina (Nina) Worden
Enterprise: Santen

Address: 2100 Powell Street, 15th Floor
City, State ZIP: Emeryville, CA, 94608
Work Phone: (415) 268-9095
E-mail: jworden@santeninc.com