PharmaSUG 2019 - Paper BP-286

One More Paper on Dictionary Tables and Yes, I Think it Is Worth Reading

Vladlen Ivanushkin, DataFocus GmbH

ABSTRACT

Before writing this paper on dictionary tables I made some research on what was already out there so that I won't duplicate someone else's work. I found quite a number of papers, but I still decided to write my own and to concentrate on how programmers can benefit from using dictionary tables in their everyday life. In this paper I would like to share with you the tasks I actually faced during my work as a statistical programmer and how using dictionary tables makes it so much easier to deal with them. There is quite a variety of them from creating macros to programming STDMs.

INTRODUCTION

First question that arises is 'What are dictionary tables?'. The easiest way to answer it is to look up the SAS® support website which says:
'DICTIONARY tables are special read-only PROC SQL tables or views. They retrieve information about all the SAS libraries, SAS data sets, SAS system options, and external files that are associated with the current SAS session.'
So as the readers probably know, the DICTIONARY tables can be accessed via the SQL procedure and the information is stored in several tables grouped by specific categories.
DICTIONARY tables contain only current actual information, i.e. the information valid at the time of accessing a particular table. Again, on this point the SAS support website says:
'Note: SAS does not maintain DICTIONARY table information between queries. Each query of a DICTIONARY table launches a new discovery process.'
Thus, comprehensiveness, ease of accessing and using make the DICTIONARY tables very valuable source of information about the current SAS session. As the information stored in the tables is quite diverse, starting from the SAS options and finishing with active macro variables and details about all the accessible data sets, they can be used for various purposes. For example, TLFs programming, macro generation, SDTM/ADaM creation etc.
Probably everyone has used DICTIONARY tables at least once to create some list of variables or to retrieve some information about their datasets. But what if we go a bit further and use the information stored in these tables to produce entire pieces of code? In other words, create dictionary tables data driven code. Or what if we try involving DICTIONARY tables more in data set programming or macro creation?
Let's see how we can benefit from using DICTIONARY tables even more.

MANIPULATING VARIABLES AND ATTRIBUTES

Probably the most used and famous DICTIONARY table is called COLUMNS. Surprisingly, it contains information about columns in all known tables. Screenshot 1 below shows a piece of the COLUMNS table for SASHELP.CARS data set.

	libname Library Name	memname Member Name	memtype Member Type	name Column Name	type Column Type	length Column Length	label Column Label	format Column Format	informat Column Informat
1	SASHELP	CARS	DATA	Weight	num	8	Weight (LBS)		
2	SASHELP	CARS	DATA	Wheelbase	num	8	Wheelbase (IN)		
3	SASHELP	CARS	DATA	Length	num	8	Length (IN)		

Screenshot 1. DICTIONARY.COLUMNS for SASHELP.CARS data set

One could think of many ways of utilizing this table but let's focus on the attributes. Updating or assigning attributes, copying or renaming variables, applying formats etc. are fairly common tasks everyone faces all the time. And sometimes doing this could be a real struggle.

DUPLICATING ATTRIBUTES

Imagine a situation when you need to duplicate a variable or a set of variables in your dataset and update the names of those variables. Sounds easy enough, right? You just initialize the variables, create arrays and let the DO loops(or just multiple assignment statements) do the work for you copying data from one variable to another. But what if you need not only the data values to be copied but also all the attributes? That sounds not that simple already. Of course, one could invent numbers of ways to do this, but how much time will it take? And will it work for every case or only for this particular one? As the COLUMNS table contains the metadata of all datasets from all assigned libraries, it is really simple and convenient to use this source of information. Also, a great advantage of this is that you don't need to enter anything by hand. So, no direct referencing of the variable attributes, just selecting the needed variables.
Let's use the CARS data set from the SASHELP. To get access to the DICTIONARY tables it is required to use the SQL which is also very convenient as it is possible to put information from multiple rows directly to a macro variable with help of the SEPARATED BY statement.
The task is to create a copy of Weight, Wheelbase and Length variables along with the attached label.

```
proc sql noprint;
    /*Construct code lines for each variable (i.e. NAME)*/
    select distinct catt(name, '_dupl=', name,
                         '; attrib', ' '||name,'_dupl label="', label,'"')
           into :copy_vars_ds separated by ';'
           from dictionary.columns
           /*Select needed dataset and variables*/
           where libname='SASHELP' and memname='CARS'
                 and upcase(name) in ('WEIGHT' 'WHEELBASE' 'LENGTH');
quit;
```

So basically, the SQL procedure puts the following text (spacing is formatted for better overview):
into the macro variable called COPY_VARS.

```
 Length_dupl=Length; attrib Length_dupl label="Length (IN)";
 Weight_dupl=Weight; attrib Weight_dupl label="Weight (LBS)";
 Wheelbase_dupl=Wheelbase; attrib Wheelbase_dupl label="Wheelbase (IN)
```

Output 1. Copying variables along with the attributes for DATA STEP

Once the macro variable is created, it can be used in a data step:

```
data cars_ds;
    set sashelp.cars;
    &copy_vars_ds;
run;
```

As the ATTRIB statement can be put anywhere within the DATA STEP, it is easy to use the SEPARATED BY statement of SQL to generate lines of code for as many variables as needed (of course there is a limit of macro variable length, i.e. 65,534 characters).
Of course, it is not necessary to include the copying of variables itself but in this case, it only makes the process easier. Also, it does not matter whether to use a DATA STEP or the PROC SQL. Just some adjustments need to be done:

```
proc sql noprint;
    /*Construct code lines for each variable (i.e. NAME)*/
    select distinct catt(name,' as',' '||name,'_dupl label="',label,'"')
           into :copy_vars_sql separated by ', '
           from dictionary.columns
           /*Select needed dataset and variables*/
           where libname='SASHELP' and memname='CARS'
                 and upcase(name) in ('WEIGHT' 'WHEELBASE' 'LENGTH');
quit;
```

And now COPY_VARS_SQL macro-variables resolves to the following:

```
   Length as Length_dupl label="Length (IN)",
   Weight as Weight_dupl label="Weight (LBS)",
   Wheelbase as Wheelbase dupl label="Wheelbase (IN)"
```

Output 2. Copying variables along with the attributes for SQL

So it can be easily used in SQL:

```
proc sql;
    create table cars_sql as
           select *, &copy_vars_sql
           from sashelp.cars;
quit;
```

After executing previous lines of code, the COLUMNS table will hold the newly created data sets CARS_DS and CARS_SQL as Screenshot 2 shows. Note the highlighted lines, variables with _DUPL suffix have the same label.

	libname Library Name	memname Member Name 1	memtype Member Type	name Column Name 2	type Column Type	length Column Length	label Column Label	format Column Format	informat Column Informat
1	WORK	CARS_DS	DATA	Length	num	8	Length (IN)		
2	WORK	CARS_DS	DATA	Length_dupl	num	8	Length (IN)		
3	WORK	CARS_DS	DATA	Weight	num	8	Weight (LBS)		
4	WORK	CARS_DS	DATA	Weight_dupl	num	8	Weight (LBS)		
5	WORK	CARS_DS	DATA	Wheelbase	num	8	Wheelbase (IN)		
6	WORK	CARS_DS	DATA	Wheelbase_dupl	num	8	Wheelbase (IN)		
7	WORK	CARS_SQL	DATA	Length	num	8	Length (IN)		
8	WORK	CARS_SQL	DATA	Length_dupl	num	8	Length (IN)		
9	WORK	CARS_SQL	DATA	Weight	num	8	Weight (LBS)		
10	WORK	CARS_SQL	DATA	Weight_dupl	num	8	Weight (LBS)		
11	WORK	CARS_SQL	DATA	Wheelbase	num	8	Wheelbase (IN)		
12	WORK	CARS_SQL	DATA	Wheelbase_dupl	num	8	Wheelbase (IN)		

Screenshot 2. DICTIONARY.COLUMNS for CARS_DS and CARS_SQL

RENAMING VARIABLES

Another tedious task is renaming variables. Sure, I'm not talking about renaming one or two variables. For example, when merging or joining two almost identical data sets sometimes we want to keep all the variables from both. We run our SQL and get numerous WARNINGS that say

```
WARNING: Variable <var name> already exists on file <your data set>.
```

Output 3. Overlapping variables WARNING

With the MERGE we might also get many problems and unexpected results. We will not get any errors or warnings, but the results might be surprising as all the overlapping variables will be overwritten. Overwriting columns is not a good idea neither in the DATA STEP nor in the SQL and should be avoided. In this case the COLUMNS table allows us to make the renaming really easy, generic and reusable just in a few steps. Another advantage of using this approach is that a programmer doesn't even need to bother checking data sets for overlapping variables.

Merging SASHELP.CARS with itself is just for demonstration purposes. The idea is to create the list for renaming.

```
proc sql noprint;
       select catx('=' , l.name , catx('_',"renamed", l.name))
          into :rename_list separated by ' '
              from
```

```
                  /*Get all variable names from the first domain*/
                  (select distinct name
                      from dictionary.columns
                      where libname="SASHELP" and memname="CARS") as l
                full join
                  /*Get all variable names from the second domain*/
                  (select distinct name
                      from dictionary.columns
                      where libname="SASHELP" and memname="CARS") as r
                on l.name=r.name
                where l.name=r.name;
quit;
```

As data set options like WHERE, KEEP or DROP are not permitted while referring to a DICTIONARY table, two separate queries are used to get the list of variables from each data set before joining. If there is a need to merge more than two data sets, similar approach may also be applied with modifications. The SQL puts the following text into macro variable RENAME_LIST:

```
Cylinders=renamed_Cylinders DriveTrain=renamed_DriveTrain
EngineSize=renamed_EngineSize Horsepower=renamed_Horsepower
Invoice=renamed_Invoice Length=renamed_Length MPG_City=renamed_MPG_City
MPG_Highway=renamed_MPG_Highway MSRP=renamed_MSRP
Make=renamed_Make Model=renamed_Model Origin=renamed_Origin
Type=renamed_Type Weight=renamed_Weight Wheelbase=renamed_Wheelbase
```

Output 4. Code for renaming overlapping variables

Now the macro variable may be used in either SQL or DATA STEP to rename the overlapping variables:

```
proc sql;
    create table work_cars
           as select *
           from sashelp.cars as l
           full join sashelp.cars(rename=(&rename_list)) as r
           on l.make=r.renamed_make and l.model=r.renamed_model
              and l.DriveTrain=r.renamed_DriveTrain;
quit;
```

As a result we get a data set with two sets of same variables, both original names and with prefix “renamed” - as Screenshot 3 shows:

	libname Library Name	memname Member Name	memtype Member Type	name Column Name
1	WORK	WORK_CARS	DATA	Make
2	WORK	WORK_CARS	DATA	Model
3	WORK	WORK_CARS	DATA	Type
4	WORK	WORK_CARS	DATA	Origin
5	WORK	WORK_CARS	DATA	DriveTrain
6	WORK	WORK_CARS	DATA	MSRP
7	WORK	WORK_CARS	DATA	Invoice
8	WORK	WORK_CARS	DATA	renamed_Make
9	WORK	WORK_CARS	DATA	renamed_Model
10	WORK	WORK_CARS	DATA	renamed_Type
11	WORK	WORK_CARS	DATA	renamed_Origin
12	WORK	WORK_CARS	DATA	renamed_DriveTrain
13	WORK	WORK_CARS	DATA	renamed_MSRP
14	WORK	WORK_CARS	DATA	renamed_Invoice

Screenshot 3. DICTIONARY.COLUMNS for WORK_CARS for two sets of same variables

Similar approach may be used for other scenarios depending on the task. For example, it is still handy to use the COLUMNS table for renaming set of variables based on some name features (i.e. prefix or suffix) or renaming all character (numeric) variables.

CREATING DECODES

The similar logic can be applied for creating 'DECODE' variables, i.e. creating new variables with applied formats. For this example, let's again use the same data set CARS. There we have two numeric variables with specified formats. So the task is to get two additional ones but character and with applied format. Also, it might be beneficial to update the labels, so let's add '(C)' to the labels.

```
proc sql noprint;
    select catt(name, "_decod=vvalue(", name, "); label",
           ' '||name, "_decod='", strip(label)||"(C)';")
           into :create_decodes separated by ' '
           from dictionary.columns
           /*Select only numeric variable names with applied format*/
           where libname="SASHELP" and memname="CARS"
                 and upcase(type)='NUM' and ^missing(format);
quit;
```

The SQL creates a macro variable CREATE_DECODES which resolves to the following text:

```
MSRP_decod=vvalue(MSRP); label MSRP_decod='(C)';
Invoice_decod=vvalue(Invoice); label Invoice_decod='(C)';
```

Output 5. Code for creating decode variables

As the original variables MSRP and Invoice don't have any label specified, the new variables will just get label 'C'. Using the macro variable in a DATA STEP gives desired result as Screenshot 4 illustrates.

```
data work_cars;
    set sashelp.cars;
    &create_decodes;
run;
```

	Make	Model	Type	Origin	DriveTrain	MSRP	MSRP_decod (C)	Invoice	Invoice_decod (C)
1	Acura	MDX	SUV	Asia	All	$36,945	$36,945	$33,337	$33,337
2	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$23,820	$21,761	$21,761
3	Acura	TSX 4dr	Sedan	Asia	Front	$26,990	$26,990	$24,647	$24,647
4	Acura	TL 4dr	Sedan	Asia	Front	$33,195	$33,195	$30,299	$30,299

Screenshot 4. WORK_CARS with MSRP_decod and Invoice_decod added

In this or similar way many operations with variables and variable attributes can be generalized and simplified. Of course, if you need to update label or format only for one variable it might not be worth it. However, if we are talking about significant number of variables, why not just run a simple PROC SQL which would do all the work for you instead of typing tens of identical lines of code.

MACRO PROGRAMMING

Clearly, updating variables and variable attributes using DICTIONARY tables could help in the macro programming as well. But is there anything specific for the macro development in the DICTIONARY tables? As one could already guess, the answer is "yes". There is a table called MACROS. It contains information about all resolved macro variables at the time of accessing the table, i.e. scope, name and value.

There are things to keep in mind while working with this table. If you just open the corresponding view, you will only see two types of SCOPE – GLOBAL and AUTOMATIC, i.e. no LOCAL. It can be explained by the nature of DICTIONARY tables and LOCAL macro variables. I.e. the LOCAL macro variables exist

only during macro execution and usually one can review the dictionary table view after the macro executed.

If so, how can we catch those local macro variables while executing a macro? Obviously, during the macro execution. Any DICTIONARY table can be saved in a temporary (or in a permanent) SAS data set. It just has to be done during the macro execution, i.e. within the macro.

Let's run this pretty useless macro

```
%macro get_local_mvars(param1=param1, param2=param2, param3=param5);
    proc sql noprint;
        create table local_mvars_from_dt as select *
               from dictionary.macros;
    quit;
%mend get_local_mvars;

%get_local_mvars;
```

The only thing the macro does is saving MACRO table to a temporary data set in WORK. Screenshot 5 shows how this data set looks like:

	scope Macro Scope	name Macro Variable Name	offset Offset into Macro Variable	value Macro Variable Value
1	GET_LOCAL_MVARS	PARAM1	0	param1
2	GET_LOCAL_MVARS	PARAM2	0	param2
3	GET_LOCAL_MVARS	PARAM3	0	param5
4	GET_LOCAL_MVARS	SQLEXITCODE	0	0
5	GET_LOCAL_MVARS	SQLOBS	0	0
6	GET_LOCAL_MVARS	SQLOOPS	0	0
7	GET_LOCAL_MVARS	SQLRC	0	0
8	GET_LOCAL_MVARS	SQLXOBS	0	0
9	GET_LOCAL_MVARS	SQLXOPENERRS	0	0
10	GLOBAL	SYS_SQL_IP_ALL	0	-1
11	GLOBAL	SYS_SQL_IP_STMT	0	
12	AUTOMATIC	AFDSID	0	0

Screenshot 5. DICTIONARY.MVARS during macro execution

So, in the SCOPE variable we get the macro name.

One of the aspects of good macro programming practice is harmonizing macro variables, and I believe using the MACROS table here is a perfect solution. It is possible to get a list of all LOCAL macro variables defined in the macro without any explicit references. Then one can decide what to do with them - upcase all values, unquote, delete multiple blanks or whatever else is needed. By the way, while selecting only required variables from the table, explicit denotation of the macro name is not necessary either. There is a system macro variable called SYSMACRONAME which contains the name of currently executing macro! Additional thing to note is that the SQL automatically creates some macro variables as it is visible from Screenshot 5. Keep this in mind while selecting all required macro variable names.

```
%macro dictionary_macros_usex(param1=, param2=, param3=);
    /*Get list of local macro variable names*/
    proc sql noprint;
        select name into :macro_param_list separated by ' '
               from dictionary.macros
               where scope="&sysmacroname" and name ^like 'SQL%';
    quit;

    /*Add & to each name to print resolved values to the log*/
    %put 1. Values before: %sysfunc(prxchange(s/^|\s+/ &/, -1,
```

```
                                                    &macro_param_list));

    /*Loop trough all macro variables*/
     %do __i=1 %to %sysfunc(countw(&macro_param_list));
         /*Temporary macro variable which holds macro variable
           Name with &i number*/
         %let __param=%scan(&macro_param_list, &__i);
         /*&__param resolves to actual macro variable name and &&&__param
           resolves to value of that macro variable*/
         %let &__param=%cmpres(%upcase(&&&__param));
     %end;

     %put 2. Values after: %sysfunc(prxchange(s/^|\s+/ &/, -1,
                                                %str(&macro_param_list)));
%mend dictionary_macros_usex;

%dictionary_macros_usex(
    param1 = woRd1
  , param2 = more     WORDS
  , param3 = %nrstr(param1 - &param1%str(,) param2 - &param2));
```

The macro prints two lines in the log. Under the number one are the original parameter values. And under the number two are the parameter values after upcasing, deleting duplicating blanks and unquoting. If unquoting is not desired, then the %Q macro functions may be used instead which suppress unquoting of

```
1. Values before:  woRd1 more     WORDS param1 - &param1%str(,) param2 -
&param2
2. Values after:  WORD1 MORE WORDS PARAM1 - WORD1, PARAM2 - MORE WORDS
```

Output 6. Macro output

previously quoted macro variables. The outcome of the macro execution is the following:

Thus, a small SQL query and a macro cycle with two lines of code save some time. And again, the piece of code can be copied from one macro to another without a single update except the desired actions for the macro variables. And all this thanks to the DICTIONARY tables and MACROS table in particular.

DATASETS UPDATING

For data sets there are also special DICTIONARY tables so no need to use the COLUMNS table for retrieving data set level information. Table 1 below shows description of the tables.

DICTIONARY Table	SASHELP View	Purpose
MEMBERS	VMEMBER	Contains information about all data types (tables, views and catalogs)
TABLES	VTABLE	Contains information about tables/datasets

Table 1. DICTIONARY.MEMBERS and DICTIONARY.TABLES

What can they be used for? As with the previous examples, it depends on programmers' imagination and/or given task. I would like to share a couple of common and useful examples I found helpful during my work as a statistical programmer.

STACKING AND/OR UPDATING DATA SETS

Need to stack all datasets in a libname? What can be easier, just create a macro variable with list of all data sets from that library. Keep in mind that if data sets from some other library than the WORK library are to be stacked then the correct library referencing must be added:

```
proc sql noprint;
    select catx('.', libname, memname) into: ds_list separated by ' '
          from dictionary.members
          where libname='SASHELP' and memtype='DATA'
                and memname like 'R%';
quit;
```

With this SQL query all data set names that start with letter ‘R’ along with the library reference are put into the macro variable DS_LIST:

```
SASHELP.RENT SASHELP.RETAIL SASHELP.REVHUB2 SASHELP.ROCKPIT
```

Output 7. Data sets for stacking

Just in one step the required information is retrieved, the only step remaining is the stacking itself.

```
data sashelp_r_stack;
    set &ds_list;
run;
```

The example could be enhanced/modified by adding options for the data sets to be stacked, like WHERE, KEEP, DROP etc.

Similarly to getting list of the desired data sets, some other examples might be applicable. Want to sort some set of data sets (or do whatever else is needed)? A tiny macro and SQL query could make it.

```
%macro sort_all(inlib=, ds_list=, outlib=WORK, sort_key=);
    %do i=1 %to %sysfunc(countw(&ds_list));
        %let ds=%scan(&ds_list, &i);
        proc sort data=&inlib..&ds out=&outlib..&ds;
            by &sort_key;
        run;
    %end;
%mend sort_all;

%sort_all(
    inlib    = SDTM
  , ds_list  = &sdtm_for_sort
  , outlib   = WORK
  , sort_key = studyid subjid
)
```

WORKING WITH THE TABLES TABLE

Another useful example for dictionary tables crossed my mind when I needed to check my PROD run against DEV run. The first thing one wants to check in such situation is that the number of observations for the same data sets in the two libraries matches. Obviously, I did not want to do it manually, I wanted a really quick and easy approach. Preferably, a piece of code I could rerun many times if needed. So, I was quite happy when I checked DICTIONARY table called TABLES and noticed variable called NOBS – “Number of Physical Observations”. This was exactly what I needed and only a small deal had to be done – i.e. write corresponding SQL queries (by the way, did you know it can be done just in one step? Not really a small one though).

For illustration purpose I used data sets which start with letter ‘A’ from the SASHELP library. They were copied to WORK library and number of observations in one of them was changed.

```
proc sql;
    create table nobs_compare
          as select coalescec(l.memname, r.memname) as memname,
                    l.nobs as base_nobs, r.nobs as compare_nobs,
                    l.nvar as base_nvar, r.nvar as compare_nvar,
```

```
            /*Filled as 1 if difference in number of variables or
              observations is found*/
            case
                when calculated base_nobs ne calculated compare_nobs
                  or calculated base_nvar ne calculated compare_nvar then 1
                else .
            end as difference_found
            from
                 /*Select data set names, number of variables and
                   observations from the first lib*/
                 (select memname, nobs, nvar
                     from dictionary.tables
                     where libname='SASHELP' and memtype='DATA'
                           and memname like 'A%') as l
            full join
                 /*Select the same for joining from the second lib*/
                 (select memname, nobs, nvar
                     from dictionary.tables
                     where libname='WORK' and memtype='DATA'
                     and memname like 'A%') as r
            on l.memname=r.memname;
quit;
```

Basically, the SQL just joins two tables by MEMNAME keeping number of observations and variables. Additional variable is created to indicate if there is any difference. Screenshot 6 shows the result, for the highlighted row there is a difference in number of observations.

	memname	base_nobs Number of Physical Observations	compare_nobs Number of Physical Observations	base_nvar Number of Variables	compare_nvar Number of Variables	difference_found
1	AACOMP	2020	2020	4	4	.
2	AARFM	130	130	4	4	.
3	ADSMSG	426	426	6	6	.
4	AFMSG	1090	1090	6	6	.
5	AIR	144	100	2	2	1
6	APPLIANC	156	156	25	25	.
7	ASSCMGR	402	402	19	19	.

Screenshot 6. Observation and variable number comparison using DICTIONARY.TABLES

As alternative, similar result can be achieved with a DATA step and SASHELP.VTABLE view.

The TABLES table is very useful when data set level information is of interest. Besides information about number of observations and number of variables it also contains other important details like date created, longest label, longest variable name, encoding type etc.

SDTM/ADAM MAPPING

Some SDTM domains contain consolidated information and have to be derived using data from multiple sources. As an example of such domains Comments (CO) and Subject Visits (SV) could be mentioned. SV, for instance, provides a summary of subject's visits also including Unscheduled ones. For Unscheduled visits the visit numbers are often assigned based on closest scheduled visits (i.e. visits according to protocol). To correctly assign all visit numbers, one needs to collect and summarize all the applicable data. And this is where DICTIONARY tables could help as well. As variable information is required, it makes sense to use the COLUMNS table to identify all data sets that contain VISIT and/or VISITNUM. An SQL query below can help to find out what exact data sets have to be combined:

```
proc sql;
    create table ds_with_vis as
           select distinct l.libname, l.memname, l.name
           from dictionary.columns as l
           join dictionary.columns as r
              on l.libname=r.libname and l.memname=r.memname
                 and l.libname='RAW' and l.name like '%DT'
                 and r.name in ('VISIT' 'VISITNUM');
quit;
```

Such SQL query will help to find out not only data sets to be combined, but also will include the information for date variables for each particular data source as Screenshot 7 shows. If the intention is to create a reusable program which would work regardless of specific domains available, this might be really handy.

	libname Library Name	memname Member Name	name Column Name
1	RAW	EX	EXSTDT
2	RAW	EX	EXSTDTC
3	RAW	EX	EXSTDTM
4	RAW	LB	LBDT
5	RAW	LB	LBDTC
6	RAW	LB	LBDTM
7	RAW	PC	PCDTC
8	RAW	PR	PRSTDT
9	RAW	PR	PRSTDTC
10	RAW	QS	QSDT
11	RAW	QS	QSDTC
12	RAW	QS	QSDTM
13	RAW	SU	SUSTDT
14	RAW	SU	SUSTDTC
15	RAW	VS	VSDT
16	RAW	VS	VSDTC
17	RAW	VS	VSDTM

Screenshot 7. RAW data sets with VISIT/VISITNUM

As describing SDTM creation techniques is not purpose of this paper I will not dig into further code details. Based on the information from data set illustrated in Screenshot 7 a set of macro variables can be created for each domain with a corresponding index:

```
%put Data set name - &ds_name1, Vars to keep - &ds_varlist1, Vars to
     rename - &ds_renamelist1;
```

gives in the log:

```
Data set name - OAD.EX, Vars to keep - EXSTDT EXSTDTC EXSTDTM,
Vars to rename - EXSTDT=STDT EXSTDTC=STDTC EXSTDTM=STDTM
```

Output 8. Macro variables for combining VISIT-related data sets

The macro variables themselves can be used further in a DO loop, combining all datasets and performing needed operations, like renaming. Screenshot 8 shows schematic possible outcome:

	DOMAIN Domain Abbreviation	VISITNUM Visit Number	STDT Start Date	STDTM Start Date/Time	STDTC Start Date/Time (C)	DT Date	DTM Date/Time	DTC Date/Time (C)	visit_date Visit Date	visit_date_char Visit Date (C)	dt_type 1 - Complete, 2 - Partial, 3- Missing
306	Exposure	9990.0000	.	.	2015-06-30	.	.		30JUN2015	2015-06-30	1
307	Exposure	9990.0000	.	.	2015-07-03	.	.		03JUL2015	2015-07-03	1
308	Exposure	9990.0000	.	.	2015-07-17	.	.		17JUL2015	2015-07-17	1
309	Exposure	9990.0000	.	.	2015-07-02	.	.		02JUL2015	2015-07-02	1
310	Laboratory ...	9990.0000	.	.		.	.	2014-09-01T08:00	01SEP2014	2014-09-01	1

Screenshot 8. Combined Unscheduled visits data

This would give consolidated information of all VISIT-related domains with a common date variable which could be used afterwards for merge with protocol visits and assigning numbers for the Unscheduled visits.

Similar approach can be applied to CO. Also, in ADaM world quite often we create derived parameters which imply using multiple domains. If this is the case, try to think if a DICTIONARY table could help as it might save some time and efforts. Such example could be deriving the last available date for a subject. Obviously, to find that date, all domains that contain date information and are applicable for consideration will need to be used. So why not use DICTIONARY.COLUMNS table here as well? As benefit, having written the code just once a programmer will be able to use it over and over.

CONCLUSION

The examples shared in this paper utilize COLUMNS, TABLES, MEMBERS and MACROS dictionary tables. From my experience they are most commonly used and usually cover 90% tasks a programmer faces during day-to-day life (especially COLUMNS and MEMBERS). Examples provided in this paper are the ones I found the most interesting, however there are many other ways these tables can be used to optimize your programming. If you want to write data independent, flexible and reusable code? – don't forget about the DICTIONARY tables!

REFERENCES

SAS(R) 9.3 SQL Procedure User's Guide. "Accessing SAS System Information by Using DICTIONARY Tables".
http://support.sas.com/documentation/cdl/en/sqlproc/63043/HTML/default/viewer.htm#n02s19q65mw08gn140bwfdh7spx7.htm

Eberhardt, Peter and Brill, Ilene. „How Do I Look it Up If I Cannot Spell It: An Introduction to SAS® Dictionary Tables"
SAS Conference Proceedings: SUGI 35, Paper 259-31.
https://support.sas.com/resources/papers/proceedings/proceedings/sugi31/259-31.pdf

CONTACT INFORMATION

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

Vladlen Ivanushkin
DataFocus GmbH
vladlen.ivanushkin@gmail.com