SESUG Paper AT-279-2019

Discovering the Power of SAS® Metadata: An Introduction to Dictionary Tables and Views

Frank C. DiIorio, CodeCrafters, Inc.

ABSTRACT

All SAS® programs, regardless of size or complexity, create and populate dozens of metadata files, commonly known as Dictionary Tables. These Tables are filled with information that is often difficult, and sometimes impossible, to obtain through other means. Any programmer who develops even simple general-purpose programs should be familiar with the Tables' organization, content, and potential uses.

This paper describes Dictionary Tables and their associated SASHELP library views. It:

1. Presents scenarios that show how they can be used
2. Gives high-level descriptions of some of the more important Tables
3. Identifies features of SQL and the macro language that are commonly used when writing programs that effectively use the Tables
4. Shows examples of the Tables' use, emphasizing the use of SQL and the macro language interface

The reader should come away from the discussion with an understanding of the Tables as well as a checklist of SQL skills that are required to use the Tables most effectively.

INTRODUCTION

A well-designed, general-purpose SAS utility needs to know the current state of the execution environment. For example, what global macro variables are defined? What are their values? Which datasets are in a library? Are required variables present and populated in a group of datasets? A program that blindly assumes resources are available is bound to fail unexpectedly at some point. It can inadvertently overwrite settings or datasets, produce inaccurate results, and generate errors that could have been avoided had more thought been put into its design.

Much of this information is readily and reliably available, stored in metadata referred to as SAS Dictionary Tables. All that is required for their use is knowledge of how the Tables are stored and the preferred techniques for accessing them. The Tables' structure and usage toolset are the focus of this paper. Topics include:

- A brief discussion of metadata
- An overview of the Tables, identifying common features and describing how they are defined and maintained
- A discussion of content, structure, and quirks of some of the more commonly used Tables
- The case for generalization. While the Tables can be used on an *ad hoc* basis, they are most effective when used in general-use tools such as SAS macros.
- Examples of use. Most of these are simple, straightforward, and supplemented with ways that their scope can be expanded.

These topics, resources discussed in "Recommended Reading," and the detailed reference material in the Appendix will give the reader a clear understanding of the Tables' contents and potential range of applications.

A WORD ABOUT METADATA

Consider anything stored in digital format – photographs, music, Web pages, and so on. We consume these in two ways. The first, and most obvious, is the putative use: looking at the picture, listening to the music, and scrolling through content on the web page.

The second part of the consumption process is more subtle and less direct. You viewed the picture on Instagram because it had an interesting location tag. You listened to the music because you applied a "classic rock" filter to your iTunes library. The web page you selected came from a list suggested by Google. What guided your consumption of these media was descriptive data attached to the JPEG, MP4, and HTML files. These indirectly consumed data points are examples of metadata.

The canonical definition of metadata is "data about data." That's catchy, and was sufficiently descriptive for the mainframe-dominant era in which it was coined. But as will be seen shortly, that definition should be expanded, possibly to "data about data and processes." That is, data that describes traditional data sources as well as the environment in which data is captured, processed, and consumed.

Dictionary Tables are stores of metadata that describe every aspect of the SAS environment. The Tables are transparently updated as options, datasets, and other aspects of the SAS session are added, modified, or deleted. They are an information-rich resource, and form the basis for a wide variety of general-purpose tools.

FEATURES OF THE TABLES

Before delving into the contents of individual Tables, let's review some background and discuss some common features:

- **Notable for Their Longevity.** The Tables are *not* leading-edge technology. Eight Tables were available in the early 1990's, in SAS Version 6.07. Subsequent releases of Base SAS software have added Tables, and variables have been added to existing Tables (to date, Tables have never been removed or renamed). These enhancements mean that being aware of Version "x" Tables is no guarantee you'll be equally savvy in subsequent versions.
- **Always Present.** The Tables are *always* created during initialization of *every* SAS program. This is true whether you run batch, Enterprise Guide®,SAS Studio, or a desktop version. Even if you don't use them, SAS does, so there is no way to not create them.
- **Cannot Be Altered.** The Tables are read-only. You cannot alter their structure or directly modify values for example, in a DATA step. Indeed, that is contrary to the primary function of the Tables – to accurately *and transparently* reflect the current state of the SAS environment.
- **Automatically Refreshed.** Access to the Tables is read-only to the user, but read/write to SAS software. As events occur during the SAS session, the relevant Tables are updated automatically. The user never has to say "I just changed the DATE option, now update the OPTIONS Table." Instead, SAS detects the change and automatically updates the OPTIONS Table.
- **Can Update Multiple Tables.** A single event can, and often does, result in updates to multiple Tables. If, for example, a DATA step creates an indexed dataset and a macro variable, the TABLES, COLUMNS, INDEXES, and MACROS Tables will be updated.
- **Not Always Populated.** Some features such as security, encryption, and data source prevent population of some fields. This is frequently the case when dealing with non-native data sources.
- **Accessible in SQL.** The Tables are accessible only from SQL using a LIBNAME of DICTIONARY (yes, a 10-character LIBNAME). This LIBNAME is visible only to SQL and cannot be successfully reference in other PROCs, DATA steps, or interactive SAS tools.
- **Use Anywhere with Views.** The Tables are indirectly accessed anywhere by views that are defined during session initialization. The views are virtual datasets that contain the location of the Tables and how to read them. Each Table (with the inexplicable exception of VIEW_SOURCES) can be accessed by one or more views stored in LIBNAME SASHELP. Views to the Tables always start with

V, and are *usually* followed by the singular form of its Table (VCOLUMN, for example, is the view to Table COLUMNS).

- **Efficiency Matters.** It is usually more efficient to read a Table rather than its view. Since the Tables are indexed, programs should avoid altering index fields (upper-casing, substringing, etc.) since this effectively removes the index's functionality. If processing scenarios permit, using a snapshot of a Table rather than reading it multiple times can result in noticeable reductions in CPU and elapsed time. This decision to use a static copy of a Table is an experience-based, rather than a rule-based, process.
- **Some Are More Helpful Than Others.** There are 32 Tables in SAS Version 9.4. Some (TABLES, COLUMNS) are extremely useful for a host of applications. Others (ENGINES, STYLES, CHECK_CONSTRAINTS) are, to be kind, obscure. What should be done, however, is taking the time to at least be aware of the content and structure if *every* Table. A good analogy is that of a master mechanic who is familiar with all tools in the shop, even if some of them will rarely, if ever, be used.

DISCUSSION: COMMONLY USED TABLES

An exhaustive discussion of all the Tables would be both quite lengthy and mind-numbing to read. This section presents a selection of Tables that are most likely to be used in general-purpose applications. The presentation of each – content, associated view, granularity, and comments – can also serve as a guide for how to approach learning about the Tables that aren't discussed here.

DESCRIBING THE TABLES

The discussion of each Table below follows a similar format:

- *Content*. A brief summary of the Table's contents.
- *View*. The view associated with the Table.
- *Granularity*. Table fields that uniquely identify a row.
- *Comments*. This section identifies possible uses for the Table, things to know for reliable use (*i.e.*, quirks and features), and fields that may not populate as expected when processing data that is protected or in non-native SAS formats.
- *Reference card excerpt*. The Appendix is a two-page summary of these Tables and additional Tables not discussed in this paper. The Table-specific portion of the Appendix is included here for easy reference. Refer to the first page of the Appendix for an explanation of notation.

DICTIONARIES

Content: "Data about data about data." This Table has attributes of all Dictionary Tables.

View: sashelp.vDctnry

Granularity: memname, name

Comments: In the author's experience, this Table's primary use is to programmatically generate a list of all Tables.

Appendix Excerpt:

dict.**dictionaries** / *sh*.**vDctnry**		Dictionary Table Attributes
► memname	$32	Dictionary table name **[UC]**
memlabel	$256	Dictionary table label **[CP]**
► name	$32	Column name **[UC]**
type	$4	Column type [char|num]
length	num	Column length
npos	num	Column position [offset within observation. >= 0]
varnum	num	Column number in table [1, 2, 3, …]
format	$49	Column format [*may* include width, period]
informat	$49	Column informat [*may* include width, period]

TABLES

Content: TABLES contains information of datasets and views allocated with the LIBNAME statement or function. The Table also includes LIBNAMEs typically allocated during initialization (SASUSER, SASHELP, and maps).

View: sashelp.vTable

Granularity: libname, memName, memType

Comments: TABLES contains some of the information found in CONTENTS output datasets. The advantage is that you don't have to ask for the data to be made available. That is, you don't have to run PROC CONTENTS. Indeed, the nature of *all* Dictionary Tables is that they are available without needing to be surfaced by any coding.

The Table has many uses, notably for generating lists and counts of datasets in a library. The process is straightforward for native SAS datasets, but can be problematic for other data sources. Consider an Excel file with sheet names `First sheet` and `Second` and a filter for a column in `Second`. MEMNAME values are stored as:

```
'First sheet$'
Second$
Second$_xlnm#_FilterDatabase
```

The difference in notation of the first two sheet and the unexpected presence of the third sheet underscore the need for robust coding when handling non-native SAS data sources.

Appendix Excerpt:

*dict.***tables** / *sh.***vTable**		Attributes of Tables and Views	Typical setting for non-native (MDB, XLS, XPT) member ▼
► libname $^{\#}$	$8	Library name **[UC]**	
► memname $^{\#}$	$32	Member name **[UC]** (SAS) **[CP]** (other)	
► memtype $^{\#}$	$8	Member type [DATA|VIEW]	
dbms_memtype	$8	If non-native engine, member type [VIEW|TABLE|LINK|…]. Otherwise, blank	
memlabel **V**	$256	Dataset label **[CP]**	*missing*
typemem	$8	Dataset type [*blank*|DATA|ATTLIST|VIEW|…]	DATA
crdate	num	Date-time created	*missing*
modate	num	Date-time modified	*missing*
nobs **V**	num	Number of observations	*missing*
delobs **V**	num	Number of deleted observations [>= 0]	0
nlobs **V**	num	Number of logical observations [. if view, else positive integer]	*missing*
obslen	num	Observation length	0
nvar	num	Number of variables	
maxvar	num	Length of longest variable name	
maxlabel **V**	num	Length of longest label [>=0]	0
num_character	num	Number of character variables	
num_numeric	num	Number of numeric variables	
protect $^{\#}$ **V**	$3	Password protection [position 1: -|R position 2: -|W position 3: -|A]	---
compress **V**	$8	Compression routine [NO|CHAR|BINARY]	NO
encrypt **V**	$8	Encryption [NO|YES]	NO
filesize **V**	num	File size [>= 0]	0
npage **V**	num	Number of pages [0, 1, …]	0
pcompress **V**	num	Percent compression [0, 1, …] [value stored is truncated integer -can be negative]	*missing*
reuse **V**	$3	Reuse space [no|yes]	no
bufsize **V**	num	Buffer size	0

*dict.***tables** / *sh.***vTable**		Attributes of Tables and Views	Typical setting for non-native (MDB, XLS, XPT) member ▼
indxtype **V**	$9	Index types [*blank*|SIMPLE|COMPOSITE|BOTH]	*missing*
sortname	$8	Name of collating sequence	*missing*
sorttype	$4	Sorting type [S=sort verified SR=sort w. NODUPREC SK=sort w. NODUPKEY]	*missing*
sortchar	$8	Character sorted by [ANSI|ASCII|…]	*missing*
encoding	$256	Data encoding [blank if view]	Default
# If password protected (TABLES.PROTECT position 1 = "R") this field *will* be populated			

COLUMNS

Content: Each row contains attributes for all currently allocated datasets and views.

View: sashelp.vColumn

Granularity: libname, memName, memType, name

Comments: The Table contains the fields you would expect – length, type, label, *et al.* One of its features that seems minor but is helpful when you use it is TYPE. Its values are stored as `char` or `num` (contrasted with CONTENTS datasets, which use the relentlessly confusing 0 or 1). Case sensitivity means that care must be taken when testing for the presence of a variable or building variable lists. Case is preserved for SAS datasets, upper-cased for transport files, and preserved with blanks translated to underscores for Oracle, Excel, and other formats. This is known and consistent behavior, and suggests that when using COLUMNS you should convert names to the same case to ensure reliable handling.

Appendix Excerpt:

*dict.***columns** / *sh.***vColumn**		Variable Attributes
► libname	$8	Library name **[UC]**
► memname	$32	Member name **[UC]**
► memtype	$8	Member type [DATA|VIEW]
► name	$32	Column name **[CP]** (except for transport files, always **[UC]**). Can contain blanks. Non-native engines: blanks translate to underscore; names can be truncated.
type	$4	Column type [char|num]
length	num	Column length
npos	num	Column position [offset within obs., e.g., 0, 1, 20]
varnum	num	Column number in table [1, 2, 3, …]
label **NN**	$256	Column label. **[CP]**
format **NN**	$16	Column format [DATE9. $HEX22.]
informat **NN**	$16	Column informat [MMDDYY10. 8.2]
idxusage **NN**	$9	Column index type [SIMPLE|COMPOSITE|BOTH]
sortedby	num	Key sequence order [0, 1, …] Negative if descending
NN: field whose value may be missing for non-native SAS file types (including SAS Transport files)		

MACROS

Content: Content of macro variables

View: sashelp.vMacro

Granularity: scope, name, offset

Comments: If a variable is longer than 200 characters, it is split into multiple rows in the Table (OFFSET value 0 for the first 200 characters, 200 for next 200, and so on). Therefore, OFFSET is needed to uniquely identify a variable. Handling variables with length over 200 comes with a caveat: the first 200 characters are not always stored with OFFSET=0.

Since `%put _global_;` does not present the variables in alphabetical order, MACROS is a good source for a workaround, filtering with a WHERE or similar statement for OFFSET=0. This is demonstrated in Example 5, later in this paper.

Appendix Excerpt:

*dict.***macros** / *sh.***vMacro** Macro Variable Attributes		
► scope	$9	Macro variable scope [GLOBAL|AUTOMATIC|macro name (if local)]
► name	$32	Macro variable name **[UC]**
► offset	num	Offset into macro variable [0, 200, …]. The beginning of a value spanning observations may not always be stored with OFFSET=0.
value	$200	Macro variable value [case, spacing are preserved]

TITLES

Content: Contains text of current titles and footnotes

View: sashelp.vTitle

Granularity: type, number

Comments: ODS formatting options (height, bold, etc.) are *not* preserved in the title or footnote text. Macro variable references are resolved before being stored (*e.g.*, `18AUG2019` will be stored, not `&sysdate9`).

One use of this Table is to check for the presence of required text such as a copyright notice or program reference in a title or footnote (see Example 9).

Appendix Excerpt:

*dict.***titles** / *sh.***vTitle**Title and Footnote Attributes		
► type	$1	Title location [T|F]
► number	num	Title number [1, …, 10]
text	$256	Text **[CP]** Rendering information (h=1 j=l *etc.*) is removed. Macro variables are resolved.

FORMATS

Content: Attributes of system and user-defined formats

View: sashelp.vFormat

Granularity: Source-dependent – see Reference Card excerpt, below

Comments: Data exchange standards sometimes specify that SAS datasets must not reference user-defined formats or informats. The FORMATS Table is a resource for identifying these objects (filter with `SOURCE=”C”`). Note that FORMATS honors the path defined by the SASAUTOS option, so if user-defined format REGION is defined in two currently allocated format catalogs, the lone reference to REGION will be to the *first* occurrence in the path. To locate *all* instances of REGION, use the CATALOGS Table (described in the Appendix).

Appendix Excerpt:

*dict.***formats** / *sh.***vFormat** Format and Informat Attributes		
► libname	$8	Library name **[UC]** [when SOURCE=’C’]
►memname	$32	Member name **[UC]** [when SOURCE =’C’]
► path	$1024	Path name **[UC]** [when SOURCE =’U’]
► objname	$32	Object name **[UC]** [e.g. GROUP, TYPE] [when SOURCE =’U’, ‘C’]
► fmtname	$32	Format name **[UC]** [e.g., GROUP, $TYPE]
► fmttype	$1	Format type [F (format) I (informat)]
source	$1	Format source [U (system) B (built-in) C (catalog, i.e., created by PROC FORMAT)]
minw	num	Minimum width [>= 0]
mind	num	Minimum decimal width [>= 0]
maxw	num	Maximum width [>= 0]
maxd	num	Maximum decimal width [>= 0]
defw	num	Default width [>= 0]

XATTRS

Content: New to Version 9.4, the XATTRS Table contains values of metadata attached to individual SAS datasets. These values can be assigned at both the dataset and variable levels.

View: sashelp.vXattr

Granularity: libname, memname, name, xattr, xoffset

Comments: Extended attributes are an interesting addition to the SAS programmer's toolbox. They are user-defined metadata (not a new concept) maintained by Base SAS software (definitely a new concept). As is often the case with significant changes to the SAS toolbox, evaluating the innovation's benefits and assessing its costs will take both time and an open mind. Meanwhile, see "Recommended Reading," below, for documentation.

Appendix Excerpt:

*dict.***xattrs** / *sh.***vXattr**		Extended Attributes (*added in V9.4*)
► libname	$8	Library name **[UC]**
► memname	$32	Member name **[UC]**
► name	$32	Variable name **[UC]** Blank if dataset-level.
► xattr	$32	Attribute name **[CP]**
xtype	$4	char|num
► xoffset	num	Offset into XVALUE value [0, 200, …] (Note: no interaction with PROC DATASETS SEGLEN option)
xvalue	$200	Attribute value **[CP]**

OPTIONS, GOPTIONS

Content: Graphics (GOPTIONS) and other (OPTIONS) option settings

Views: sashelp.vGopt (GOPTIONS), sashelp.vOpt (OPTIONS), sashelp.vAllopt (contents of both OPTIONS and GOPTIONS)

Granularity: optName, level, offset

Comments: You can retrieve individual, groups (variable GROUP), or all option settings from these Tables and views. When filtering or retrieving individual values, use the full name of the option rather than its alias (*e.g.*, LINESIZE rather than LS).

SAS provides several tools to save and restore option settings. The GETOPTION function is best suited for retrieval of individual options, although it can be used in programs that restore values as well. The OPTSAVE and OPTLOAD procedures can save and restore all or individual option settings. These PROCs are easy to use, but come with a caveat about coverage: in Version 9.4 Enterprise Guide, OPTSAVE captures values for 289 options. By contrast, VALLPOT, the union of OPTIONS and GOPTIONS, has values for *611* options. The save/restore process is easy to code when using OPTSAVE and OPTLOAD. But keep in mind that using the OPTIONS and GOPTIONS Tables for the same purpose, while requiring more coding, will be more thorough.

Appendix Excerpt:

*dict.***options** / *sh.***vOpt** *dict.***goptions** / sh.**vGopt** / *sh.***vAllopt**		Option Settings
► optname	$32	Full name (not alias) of the option [UC]
setting	$1024	Option setting [CP]
optdesc	$160	Option description
level	$8	Option location GOPTIONS: [GRAPH] OPTIONS: [Portable|Host]
group	$32	OPTIONS only: option group [UC] [MACRO|SORT|ENVFILES|…]
► offset	$200	Offset into SETTING value [0, 200, …].
opttype	$8	Option type [boolean|char|num]

THE CASE FOR GENERALIZING

As noted earlier, the Tables and their views can be used for specific, *ad hoc* inquiries (“how many variables end with _E?”; “is variable SUBJID in every dataset in the library,” etc.). However, the utility of the Tables’ contents is greatly enhanced when generalized. Let’s take a brief look at what can be gained by abstracting specific, hard-coded tasks into macros.

A design consideration in any general-use program is that it can fail gracefully. If a resource is not available, the execution environment is incorrect, or some other condition is not met, the program should react by detecting this, issuing a message, and bypass some or all of the remaining steps. Not programming for failure conditions usually results in SAS errors and warnings. This is unpleasant for the program’s users and doesn’t reflect well on the program’s author.

A common condition to test is whether a dataset is populated. If it isn’t, one or more steps may need to be bypassed. This code snippet in a macro obtains the observation count by querying TABLES:

```
%local dsetN;
proc sql noprint;
select nobs into :dsetN
from dictionary.tables
where libname=’WORK’ & memname=’SUBSET1’ & memType=’DATA’
;
quit;
```

With this simple coding, branching within the program can be based on values of DSETN. It’s useful, and can be used in other programs. But any programmer with a modicum of self-awareness knows that when this snippet is copied to another program, at least one of the following will occur:

- LIBNAME or MEMNAME will not be changed
- MEMTYPE might be omitted, allowing the possibility for views to be evaluated (and, by definition, returning a missing value)
- The %local statement might be omitted, possibly causing &dsetN’s scope to be Global

These and similar missteps can be avoided by making the code a generalized macro. Macro COUNTOBS accepts a one- or two-level dataset name, and allows the user to override the default name of the global macro variable containing the observation count. The result is both an effective use of the Tables and a glimpse into the power and flexibility of the macro language:

```
%macro countObs(data=, n=dsetN);
%global &dsetN.;
%let &dsetN = -1;
%let data = %upcase(&data.);
%if %index(&data., .) > 0 %then %do;
    %let lib = %scan(&data., 1, .);
    %let mem = %scan(&data., 2, .);
    %end;
    %else %do;
          %let lib = WORK;
          %let mem = &data.;
          %end;

proc sql noprint;
select nobs into :&n.
from dictionary.tables
where libname=”&lib.” & memname=”&mem.” & memType=’DATA’
;
quit;
%mend countObs;
```

This code is more robust than the earlier version, and will available to any program if COUNTOBS.SAS resides in an autocall directory. This code snippet demonstrates COUNTOBS' use:

```
… macro statements …
%countObs(data=subset1, n=tCount)
%if &tCount. = 0 %then %do;
    %put Filtering created an empty dataset. Execution terminating.;
    %end
… macro statements …
```

We will revisit this macro in Example 4 in the next section. For now, though, simply reflect on the coding simplicity made possible by using *both* the Tables and the macro language.

PUTTING THE TABLES TO WORK: EXAMPLES

So far, we have presented an overview of the Tables, given details on some of the more frequently used Tables, and demonstrated the benefits of providing access to them via the macro language. This section presents practical applications of the Tables' use.

Each example contains a usage scenario and a code snippet. Most examples also suggest how the snippet could be further developed into a general-purpose utility macro. Note that the focus of the code snippets is using the Table, *not* writing a production-quality macro. They omit coding conventions that you would normally follow: a comment header, parameter checks, cleanup prior to terminating, and so on. See "Recommended Reading," below, for macro design references.

EXAMPLE 1: DESCRIBE EACH TABLE

Background. Earlier, we described the DICTIONARIES Table as "data about data about data." You can get an idea of which Tables are available by building a list of metadata table names, then using PROC SQL's DESCRIBE statement for each Table:

```
%macro dictInfo;
proc sql noprint;
     select distinct memname into :tables separated by ‘ ‘
     from dictionary.dictionaries;
     %let dictN = &SQLobs.;
     %do idx = 1 %to &SQLobs.;
         %let tableName = %scan(&tables., &idx.);
         describe table dictionary.&tableName.;
     %end;
quit;
%mend;
```

Enhancements/Comments: DESCRIBE output is rather meager, only displaying variables in order, along with their type, length, and label. The Appendix has a more content-rich and nuanced description of the Tables.

EXAMPLE 2: DETECT USER-WRITTEN FORMATS AND INFORMATS

Background. Data exchange standards may limit variable formats and informats to those found in Base SAS software. This code snippet creates macro variables FMT and INFMT, blank-delimited lists of user-defined formats and informats, respectively. Once the variables are defined, downstream processing can read variable attributes, possibly from the COLUMNS Table, and flag instances of formats or informats being found in the two lists.

```
%let fmt   = ;
%let inFmt = ;
proc sql noprint;
     select fmtName into :fmt separated by ‘ ‘   /* list of formats */
     from dictionary.formats
```

```
       where source=”C” & fmtType=”F” ;
       select fmtName into :inFmt separated by ‘ ‘  /* list of informats */
       from dictionary.formats
       where source=”C” & fmtType=”I” ;
  quit;
```

Enhancements/Comments: A simple extension would be to make the code a macro that resides in an autocall library. The macro’s parameters would allow the user to change the default output variable names.

EXAMPLE 3: XPT DATASET COMPATABILITY

Background. A vital feature of macro design is identifying and reacting to problematic conditions during execution. As noted earlier, rather than letting a DATA step or PROC fail, a well-designed macro will anticipate and react appropriately when failure conditions are encountered.

This example identifies variable attributes that are incompatible with the SAS XPT specification. The code could be at the end of the program creating the dataset or, with some modification, be part of a generalized XPT creation macro.

Notice PROC SQL’s creation of dataset ATTRIBS. We probably could have used a CASE expression to identify the error conditions, thus keeping all processing within SQL. The extra DATA step coded here allows more flexibility than SQL for messaging. While more verbose than a pure-SQL solution, it is likely easier to read and maintain:

```
  proc sql noprint;
       create table attribs as
       select libname, memname, name, length, label from
       dictionary.columns
       where libname=’ANALYSIS’ & memname=’ADAE’ ;
  quit;

  %let OKforXPT = t;
  data _null_;
       set attribs end=eof;
       retain ok ‘t’;
       if index(name, ‘ ‘) > 0 then do;
          ok = ‘f’;  put “Name contains a blank: “ name;
          end;
       if length > 200 then do;
          ok = ‘f’;  put “Variable length exceeds 200: “ name;
          end;
       if length(name) > 8 then do;
          ok = ‘f’;  put “Variable name exceeds 8 characters: “ name;
          end;
       if length(label) > 40 then do;
          ok = ‘f’;  put “Variable label exceeds 40 characters: “ name;
          end;
       if eof then call symput(‘OKforXPT’, ok);
  run;
```

Enhancements/Comments: Innumerable improvements can be made to this program. It could be converted into a macro with parameters that specify LIBNAME and dataset name. If dataset name is null, all datasets in the library identified by the LIBNAME parameter would be processed. Another parameter could specify whether to test for the presence of user-written formats and informats (a process described in Example 2, above).

EXAMPLE 4: OBSERVATION COUNTER REVISITED

Background. "The Case for Generalizing," above, suggested that the Tables are most effectively used when they are read as part of parameterized macros. The code below builds on that section's %countObs macro:

```
%macro tableStats(data=, prefix=);
%global &prefix.Nobs &prefix.Nvars &prefix.Nchar &prefix.Nnum;
%let &dsetN = -1;
%let data = %upcase(&data.);
%if %index(&data., .) > 0 %then %do;
    %let lib = %scan(&data., 1, .);
    %let mem = %scan(&data., 2, .);
    %end;
    %else %do;
          %let lib = WORK;
          %let mem = &data.;
          %end;

proc sql noprint;
select nobs          into :&prefix.N,
       nvar          into :&prefix.Nvars,
       num_character into :&prefix.Nchar
       num_numeric   into :&prefix.Nnum
from dictionary.tables
where libname=”&lib.” & memname=”&mem.” & memType=’DATA’
;
quit;
%mend tableStats;
```

Several output variables were added to the original macro, and the new name – tableStats – accurately reflects its expanded scope. Rather than creating separate macros for the number of observations, number of variables, etc., all the functionality is bundled into one macro.

Enhancements/Comments: An obvious extension is to add other values found in the TABLES Table, *e.g.* CRDATE and MEMLABEL. A tempting, but counterproductive, modification would be adding a parameter specifying which variables should be created, *e.g.*, CREATE=N NVARS. This theoretically, would ensure that the only variables produced are those that were actually requested by the user. However, this approach would clutter the code due to the parsing of the CREATE parameter. It would also require conditional execution of %global and SQL statements to control variable creation. Finally, this approach would require the user to remember valid values for CREATE. As long as the macro's variable output is clearly documented, it is simpler to create all variables, even if some will not be used.

EXAMPLE 5: CLEAN DISPLAY OF MACRO VARIABLES

Background. Anyone who has run %put _global_; to display values of global macro variables knows that the Log output can be confusing. Variables are not listed in alphabetical order, nor are name-value pairs arranged in columnar format. A good solution is to use the MACROS Table to create a display tool:

```
proc sql noprint;
     create table _macvars_ as
     select name, offset, scope, value
     from dictionary.macros
     where offset=0 and scope='GLOBAL'  /* Pay attention to filters! */
     order by name
     ;
quit;
```

```
data _null_;
     set _macvars_;
     put name $20. +1 value $80.;
run;
```

Enhancements/Comments: The hard-coded widths in the DATA step will inevitably create confusing or erroneous output (*e.g.*, when NAME length exceeds 21). Assigning NAME and VALUE widths can be be determined programmatically (determine the maximum length of NAME, use the LINESIZE option to determine VALUE's width). Another enhancement, of course, is to store the code in a macro autocall library.

EXAMPLE 6: LIST AND COUNT OF DATASETS IN A LIBRARY

Background. A macro often needs to perform a similar set of tasks for each dataset in a library. The basis for this processing, a list and count of the datasets, is created below:

```
proc sql noprint;
     select memname into :datasets separated by ‘ ‘
     from dictionary.members
     where memtype = ‘DATA’ & libname = ‘PROD’
     ;
     %let datasetsN = &SQLobs.;
quit;
```

Enhancements/Comments: This example hard-codes the library and macro variable names. A helpful modification would be to create a macro with a parameter that would accept one or two-level dataset names.

A usage consideration here is giving the user a reliable means to assess results. We could assign a dataset count (variable DATASETSN) of -1 if the library was not allocated. Examining this variable rather than the list itself (variable DATASETS) is more robust since the list would be null regardless of whether the library was not allocated *or* it was allocated but empty.

EXAMPLE 7: PRINT FROM EVERY DATASET IN A LIBRARY

Background. Example 6 and the modifications suggested for Example 4 form the basis for the following macro. It uses PROC REPORT to display a user-controlled number of observations from datasets in a library.

```
%macro printIt(lib=, obs=10);
       %memList(lib=&lib., prefix=list)    /* Example 6 */
       %do idx = 1 %to &listN.;
           %let dset = %scan(&list., &idx.)
           %tableStats(data=&lib..&dset., prefix=dset_)   /* Example 4 */
           proc report data=&lib..&dset. headline nowd style=journal;
           title1 “First &obs. observations from &lib..&dset.”;
           title2 “# obs=&dset_N # vars=&dset_nvars.”;
           run;
       %end;
%mend;
```

Enhancements/Comments: This code does not consider the possibility of empty datasets. Ideally, if there were “n” datasets in the library there would be an equal number of reports written to the LST file. An empty dataset report could be created by adding some branching based on &dset_N: if greater than 1, execute the REPORT step. Otherwise, run a DATA _NULL_ that writes a message to FILE PRINT.

EXAMPLE 8: INCONSISTENT ATTRIBUTES OF LIKE-NAMED VARIABLES

Background. Pharmaceutical (and, likely, other industry) standards require that like-named variables in a data library have identical attributes. This SQL code reads the COLUMNS Table, populating dataset DONTMATCH with information about variables with identical names but differing type and/or length:

```
proc sql noprint;
     create table dontMatch as
     select catX(' - ', type, length) as compare,
            upcase(name) as nameUC, memname
     from dictionary.columns
     where libname="ANALYSIS"
     group by nameUC
     having count(distinct compare) > 1
     order by nameUC, memname
     ;
quit;
```

Comments, Enhancements: Among the many extensions to this snippet: turn it into a macro with a parameter for the LIBNAME filter. A macro parameter could be added, giving the user control over what should be compared (label, format, and informat would be good additions to the attribute list). Notice in the example that variables were upper-cased before grouping. This enabled the comparison of variables “SubjID” and “subjid”. The code could be modified to flag same-named but differently-cased variables.

EXAMPLE 9: COMPLIANT FOOTNOTES

Background. Tables and other displays usually have to comply with title and footnote content requirements. These often include the name of the program that created the display, the date-time of creation, and other identifying information. Such standardized content lends itself to automated creation of titles and footnotes. The TITLES Table can be used to check for compliance, in this case to see if the last footnote contains the data cut date:

```
%macro checkFoot;
       %let compliant = -1;
       proc sql noprint;
            select indexW(text, ‘Data cut date:')
            into   :compliant
            from dictionary.titles
            where type='F'
            having number = max(number)
            ;
       quit;
       … Processing based on &compliant value …
%mend;
```

If COMPLIANT is greater than 0, the last footnote contained the required text. Values of 0 or -1 identify conditions where the text was missing or no footnotes were specified, respectively.

CONCLUSION

The Dictionary Tables are a valuable addition the programmer’s toolbox. They are a reliable and easily accessed source of a wealth of information (some of which in unobtainable by another any means) about the SAS environment. To use them effectively, you must become familiar with their structure and contents, and be aware of conditions that may result in incomplete data or seemingly anomalous results. The utility of the Tables is fully realized when they are used in well-designed macros. The payoff is having a library of robust, general-purpose applications.

ACKNOWLEDGMENTS

Many thanks to Greg Weller, Rho Inc., for his thoughtful review of an early draft of this paper. And, as always, thanks to my wife, April Sansom, for her editing. Thanks to her, the *Final* version of this paper became version *Somewhat* Final, and after much back and forth, version *Finally* Final. The paper is much better off for her efforts.

RECOMMENDED READING

Resources abound for learning more about the Tables and getting examples of their use. Among the best (with some noted caveats) are:

- **www.LexJansen.com.** This site archives decades' worth of papers from SAS Global Forum, PharmaSUG, PhUSE, and regional user group conferences. A search for "dictionary table" will produce links to a host of papers.
- **support.SAS.com.** The SAS web site's Base SAS Language Reference describes the Tables in mostly general terms. It lacks reasons for missing data points, Table quirks, and other usage features needed for application development.
- **SAS Community, SAS-L.** These popular forums are best suited for examples of Table use and problem resolution. They are not a resource for orderly learning about Table structure and contents.
- **Enterprise Guide, SAS Studio, or desktop.** These tools, or any application that will open a SAS view, allow *ad hoc* examination of the Tables' contents.
  Digging even deeper, these are also environments that let you observe the Table maintenance process. You can start a session, examine MEMNAME values in SASHELP.VTABLE, then create a WORK dataset. When you re-open the VTABLE display it will contain rows for the new dataset. These simple steps demonstrate several fundamental about the Tables: the View was predefined, and when you changed the SAS environment by adding a dataset, you could see that TABLES and its view were automatically updated.
- **Tables Reference Card.** The Appendix contains a description of the Tables' contents. It notes case-sensitivity, typical values for fields, and how the Table is populated if a data source is password protected, encrypted, or is not stored in a native SAS format. Note that this is a curated selection; not all Tables are described, and some fields are omitted.

While not the focus of this paper, two related topics are also worth pursuing:

- *Extended Attributes.* Home-grown, user-defined metadata is nothing new. What *is* new is Version 9.4's ability to add user-defined metadata to SAS datasets. Details are the online Base SAS® 9.4 Procedures Guide: documentation of PROC DATASETS (MODIFY statement).
- *Macro Design.* If you have read this far, it should be clear that macros afford the most reliable and effective use of the Tables. This underscores the importance of designing macros that are flexible, robust, extensible, and easily maintained. As with the Tables, www.LexJansen.com is an excellent resource for papers on this topic.

CONTACT INFORMATION

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

Frank DiIorio
CodeCrafters, Inc.
FrankDiIorio@gmail.com

APPENDIX: SUMMARY OF SAS V9.4 DICTIONARY TABLES AND VIEWS

Notation:

- In headers, **dict** indicates SQL reference (dictionary.*tableName*); **sh** indicates SASHELP reference (sashelp.*viewName*); **PW** identifies tables with no data displayed if member is password-protected.
- ► indicates a field used for uniquely identifying an observation
- **V** identifies a field whose value may be missing or 0 for SAS views
- **[UC]** upper-cased, **[CP]** case preserved

Usage Notes:

- MEMNAME: If using native SAS engines, upper-cased. Others preserve case and spacing, possibly replacing blanks with underscores.
- Non-native files: if the data source has a name exceeding 32 characters, cannot be read due to security restrictions, or uses syntax not understood by SAS, it will *not* be presented in *any* Table or View.
- *Not all tables and fields are shown!*

*dict.***dictionaries** / *sh.***vDctnry** Dictionary Table Attributes

► memname	$32	Dictionary table name **[UC]**
memlabel	$256	Dictionary table label **[CP]**
► name	$32	Column name **[UC]**
type	$4	Column type [char|num]
length	num	Column length
npos	num	Column position [offset within observation. >= 0]
varnum	num	Column number in table [1, 2, 3, …]
format	$49	Column format [*may* include width, period]
informat	$49	Column informat [*may* include width, period]

*dict.***catalogs** / *sh.***vCatalg** Catalog Member Attributes

► libname	$8	Library name **[UC]**
► memname	$32	Member name **[UC]**
memtype	$8	Member type [CATALOG]
► objname	$32	Object name **[UC]**
► objtype	$8	Object type [FORMAT|FORMATC|MACRO…]
objdesc	$256	Object description **[CP]**
created	num	Date-time created
modified	num	Date-time modified
alias	$8	Object alias **[UC]**
level	num	Library concatenation level [0, 1, 2, …]

*dict.***columns** / *sh.***vColumn** PW Variable Attributes

► libname	$8	Library name **[UC]**
► memname	$32	Member name **[UC]**
► memtype	$8	Member type [DATA|VIEW]
► name	$32	Column name **[CP]** (except for transport files, always **[UC]**). Can contain blanks. Non-native engines: blanks translate to underscore; names can be truncated.
type	$4	Column type [char|num]
length	num	Column length
npos	num	Column position [offset within obs., e.g., 0, 1, 20]
varnum	num	Column number in table [1, 2, 3, …]
label **NN**	$256	Column label. **[CP]**
format **NN**	$16	Column format [DATE9. $HEX22.]
informat **NN**	$16	Column informat [MMDDYY10. 8.2]
idxusage **NN**	$9	Column index type [SIMPLE|COMPOSITE|BOTH]
sortedby	num	Key sequence order [0, 1, …] Negative if descending

NN: field whose value may be missing for non-native SAS file types (including SAS Transport files)

*dict.***destinations** / *sh.***vDest** Open ODS Destinations

► destination	$100	Destination [PDF|RTF|PS|…]
style	$32	One-level style name [Journal|Printer|…]

*dict.***engines** / *sh.***vEngine** Attributes of *all* available engines

► engine	$8	Engine name **[UC]**
alias	$8	Alias **[UC]**
description	$40	Description **[CP]**
preferred	$3	Preferred? [yes|no]

*dict.***extfiles** / *sh.***vExtfl** User, system external files

► fileref	$8	FILEREF **[UC]** duplicated for concatenated files
► xpath	$1024	Path. Can be [1] directory [2] directory+file name+extension (e.g., *dir*\dm.xpt) [3] command if REF is piped [4] blank. **[CP]**
xengine	$8	Engine name **[UC]** [e.g., DISK, DUMMY, PIPE]
directory	$3	Does FILEREF point to a directory [yes|no]
exists	$3	Does the location exist? [yes|no]
fileSize	num	Size of file, in bytes
► level	num	File concatenation level (0, 1, …)
modDate	num	File modification date-time
temporary	char	Allocated as a temporary location? [yes|no]

*dict.***formats** / *sh.***vFormat** Format and Informat Attributes

► libname	$8	Library name **[UC]** [when SOURCE='C']
►memname	$32	Member name **[UC]** [when SOURCE ='C']
► path	$1024	Path name **[UC]** [when SOURCE ='U']
► objname	$32	Object name **[UC]** [e.g. GROUP, TYPE] [when SOURCE ='U', 'C']
► fmtname	$32	Format name **[UC]** [e.g., GROUP, $TYPE]
► fmttype	$1	Format type [F (format) I (informat)]
source	$1	Format source [U (system) B (built-in) C (catalog, i.e., created by PROC FORMAT)]
minw	num	Minimum width [>= 0]
mind	num	Minimum decimal width [>= 0]
maxw	num	Maximum width [>= 0]
maxd	num	Maximum decimal width [>= 0]
defw	num	Default width [>= 0]

*dict.***functions** / *sh.***vFunc** Function Attributes

► fncname	$32	Function name **[UC]**
fncargs	num	Argument attributes
fncprod	$1	Implementation type
► fnctype	$1	Function type [N|B|C]
maxarg	num	Maximum number of arguments to function
minarg	num	Minimum number of arguments to function
source	$1	Function source [U|B]

*dict.***indexes** / *sh.***vIndex** PW Native Engine Index Attributes

► libname	$8	Library name **[UC]**
► memname	$32	Member name **[UC]**
memtype	$8	Member type [DATA]
► name	$32	Column name **[CP]**
idxusage	$9	Column index type [COMPOSITE|SIMPLE]
► indxname	$32	Index name **[CP]**
► indxpos	num	Column position (offset within obs.) in composite key.
nomiss	$3	NOMISS option [yes|*blank*]
unique	$3	UNIQUE option [yes|*blank*]

*dict.***libnames** / *sh.***vLibnam** Attributes of Allocated Libraries

► libname	$8	Library name **[UC]**
engine	$8	Engine name **[UC]**
► path	$1024	Path name [blank for some engines: URL, …] **[CP]** XPT. non-native engines include file name, extension.
► level	num	Library concatenation level [0, 1, 2, …]
readonly	$3	Read only? [no|yes]
sequential	$3	Sequential? [no|yes]
temp	$3	Allocated with temporary access? [no|yes]

*dict.*macros / *sh.*vMacro

Macro Variable Attributes

► scope	$9	Macro variable scope [GLOBAL|AUTOMATIC|macro name (if local)]
► name	$32	Macro variable name **[UC]**
► offset	num	Offset into macro variable [0, 200, …]. The beginning of a value spanning observations may not always be stored with OFFSET=0.
value	$200	Macro variable value [case, spacing are preserved]

*dict.*members

Member Attributes

► libname	$8	Library name **[UC]**
► memname	$32	Member name **[UC]** (SAS) **[CP]** (other) Generation datasets: *name*#nnnn
memtype	$8	Member type [DATA|VIEW|ITEMSTOR|CATALOG|…]
dbms_memtype	$32	DBMS member type [VIEW|DATA|LINK|…]
engine	$8	Engine name **[UC]**
index	$32	Uses indexes? [yes|no] *[Note:* not *index names!]*
path	$1024	Path name **[CP]** [concatenated paths are quoted and enclosed in parentheses; otherwise, no parentheses or quotes] [For most non-native engines, directory + file name + extension; otherwise, just directory]

Views

*sh.***vMember**	All fields from dictionary.members
*sh.***vsAccess**	LIBNAME, MEMNAME (memtype='ACCESS')
*sh.***vsCatlg**	LIBNAME, MEMNAME (memtype='CATALOG')
*sh.***vsLib**	LIBNAME, PATH (unique values of LIBNAME)
*sh.***vsTable**	LIBNAME, MEMNAME (memtype='DATA')
*sh.***vsView**	LIBNAME, MEMNAME (memtype='VIEW')
*sh.***vsTabvw**	LIBNAME, MEMNAME, MEMTYPE (memtype='DATA', 'VIEW')

*dict.*views / *sh.*vView PW

View Attributes *(SAS file formats only)*

► libname	$8	Library name **[UC]**
► memname	$32	Member name **[UC]**
memtype	$8	Member type [VIEW]
engine	$8	Engine name [SASESQL|SASDSV|…]

*dict.*styles /*sh.*vStyle

Attributes of Allocated Styles

► libname	$8	Library name **[UC]**
► memname	$32	Member name **[UC]**
► style	$32	Style name **[CP]**
crdate	num	Date-time created

*dict.*options / *sh.*vOpt
*dict.*goptions / sh.vGopt
*sh.*vAllopt

Option Settings

► optname	$32	Full name (*not* alias) of the option **[UC]**
setting	$1024	Option setting **[CP]**
optdesc	$160	Option description
level	$8	Option location • GOPTIONS: [GRAPH] • OPTIONS: [Portable|Host]
group	$32	OPTIONS only: option group **[UC]** [MACRO|SORT|-ENVFILES|…]
► offset	$200	Offset into SETTING value [0, 200, …].
opttype	$8	Option type [boolean|char|num]

*dict.*titles / *sh.*vTitle

Title and Footnote Attributes

► type	$1	Title location [T|F]
► number	num	Title number [1, …, 10]
text	$256	Text **[CP]** Rendering information (h=1 j=l *etc.*) is removed. Macro variables are resolved.

*dict.*tables / *sh.*vTable

Attributes of Tables and Views

			Typical setting for non-native (MDB, XLS, XPT) member ▼
► libname #	$8	Library name **[UC]**	
► memname #	$32	Member name **[UC]** (SAS) **[CP]** (other)	
► memtype #	$8	Member type [DATA|VIEW]	
dbms_memtype	$8	If non-native engine, member type [VIEW|TABLE|LINK|…]. Otherwise, blank	
memlabel **V**	$256	Dataset label **[CP]**	*missing*
typemem	$8	Dataset type [*blank*|DATA|ATTLIST|VIEW|…]	DATA
crdate	num	Date-time created	*missing*
modate	num	Date-time modified	*missing*
nobs **V**	num	Number of observations	*missing*
delobs **V**	num	Number of deleted observations [>= 0]	0
nlobs **V**	num	Number of logical observations [. if view, else positive integer]	*missing*
obslen	num	Observation length	0
nvar	num	Number of variables	
maxvar	num	Length of longest variable name	
maxlabel **V**	num	Length of longest label [>=0]	0
num_character	num	Number of character variables	
num_numeric	num	Number of numeric variables	
protect # **V**	$3	Password protection [position 1: -|R position 2: -|W position 3: -|A]	---
compress **V**	$8	Compression routine [NO|CHAR|BINARY]	NO
encrypt **V**	$8	Encryption [NO|YES]	NO
filesize **V**	num	File size [>= 0]	0
npage **V**	num	Number of pages [0, 1, …]	0
pcompress **V**	num	Percent compression [0, 1, …] [value stored is truncated integer -can be negative]	*missing*
reuse **V**	$3	Reuse space [no|yes]	no
bufsize **V**	num	Buffer size	0
indxtype **V**	$9	Index types [*blank*|SIMPLE|COMPOSITE|BOTH]	*missing*
sortname	$8	Name of collating sequence	*missing*
sorttype	$4	Sorting type [S=sort verified SR=sort w. NODUPREC SK=sort w. NODUPKEY]	*missing*
sortchar	$8	Character sorted by [ANSI|ASCII|…]	*missing*
encoding	$256	Data encoding [blank if view]	Default

# If password protected (TABLES.PROTECT position 1 = "R") this field *will* be populated

*dict.*xattrs / *sh.*vXattr

Extended Attributes (*added in V9.4)*

► libname	$8	Library name **[UC]**
► memname	$32	Member name **[UC]**
► name	$32	Variable name **[UC]** Blank if dataset-level.
► xattr	$32	Attribute name **[CP]**
xtype	$4	char|num
► xoffset	num	Offset into XVALUE value [0, 200, …] (Note: no interaction with PROC DATASETS SEGLEN option)
xvalue	$200	Attribute value **[CP]**

Other tables

check_constraints
constraint_column_usage
constraint_table_usage
dataitems
filters
infomaps
locales *(added in V9.4)*
prompts
promptsxml
referential_constraints
remember
table_constraints
view_sources