Dictionary Tables and Views: Essential Tools for Serious Applications

Frank DiIorio, CodeCrafters, Inc., Chapel Hill NC
Jeff Abolafia, Rho, Inc., Chapel Hill NC

INTRODUCTION

Dictionary tables were introduced to the SAS System in the early 1990's, in Version 6.07. Laden with information that is often difficult, and sometimes impossible, to get through other means, they still appear to be on the outside of many programmers' Bag of Tricks. This is both perplexing and unfortunate for as we will see in this paper, once their content and organization is understood, they are readily adapted for a range of applications that, to use an old saw, "are only limited by your imagination."

This paper describes dictionary tables and their associated SASHELP library views. It:

- presents **scenarios** that show how they can be used
- gives high-level **descriptions** of some of the more important (a relative term, to be sure) tables
- identifies features of **SQL** and the **macro language** that are commonly used when writing programs that effectively use the tables
- 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 with a checklist of SQL skills that are required to use the tables most effectively.

CONSIDER THE ALTERNATIVES

SAS programmers, particularly those writing utilities, have always needed high-level information about the SAS environment (option settings, data set characteristics, etc.). Consider a few realistic scenarios and the coding strategy required, both not using and using the dictionary tables:

- You want to insert the name of the current (Windows OS) directory in a footnote.

  Without the tables: use a _NULL_ DATA step to write a temporary file, then parse the SAS Log to identify the file's location. Ugh!

  Using the tables: allocate a file name of '.', then store the EXTFILES table's XPATH value of the filename in a macro variable. See Example 2 for details.

- As part of "resetting" the SAS environment in an interactive session, you want to delete all global macro variables.

  Without the tables: set the MVARSIZE system option to 0 (forcing all global macro variables to be written to a catalog, rather than be held in memory), then delete entries of type MACRO from the catalog. Depending on the number and length of the macro variables, the I/O required by this technique can affect performance. Ugh!

  Using the tables: using the MACROS table, save the names of the global macro variables, then use %SYMDEL. See Example 4 for details.

- A project uses numerous SAS datasets, some of them with like-named variables. You want to identify instances of like-named variables with conflicting data types or lengths. For example, the variable GENDER may be stored as a character, length 1 in one dataset and numeric, length 3 in another. Ugh!

  Without the tables: create an output data set from the CONTENTS procedure, then sort by variable name, and flag inconsistencies in a DATA step.

  Using the tables: use the COLUMNS dictionary table and SQL to create a data set containing the errant variables. See Example 1 for details.

- You want to print the first "n" observations from every data set in a library.

  Without the tables: write a macro parameterized for data set name and number of observations to print. Then, invoke the macro, manually specifying the data sets in the library. Alternately, use the CONTENTS procedure to programmatically acquire a list of data set names in the library, then use CALL EXECUTE to invoke the macro. Again, ugh!

  Using the tables: use the COLUMNS dictionary table and SQL to create a data set ready for a reporting procedure (PRINT, REPORT, *et al.*). See Example 8 for details.

ABOUT THE TABLES

Dictionary tables make these tasks and others feasible with a minimal amount of coding effort. Let's take a look at some of their characteristics. They are:

- **Metadata**, or "data about data." They are data that are one or more steps away from operational or summary data, which is what most people think of when they hear "data." Rather than containing salary levels, event codes, dosage levels, and the like, metadata identify the dataset containing the data, its creation date, the data type of the variables, and a host of other high-level details.
- Available only in **Version 6.07 or later** (hopefully this isn't an issue for anyone!)
- **Always and automatically created** during SAS System startup. There is no system option to suppress their creation or maintenance.
- **Automatically maintained** during the course of the interactive session or batch job.
- **Read-only**. You *cannot* change the table or view organization. You *can* affect their contents by making changes in the SAS environment. Whenever you change a system option, create a dataset, delete a member from a catalog, add a label to a variable, etc. one or more tables are updated.
- **Accessible from SQL** with the reserved LIBNAME of DICTIONARY (yes, that's a 10-letter LIBNAME!)
- **Accessible outside SQL** by using views defined in the LIBNAME of SASHELP. SASHELP is allocated automatically during startup.
- Usually **more efficient when accessed from SQL**. That is, using SQL to access, say, DICTIONARY.TABLES will usually execute faster than SQL, a DATA step, or a procedure reading SASHELP.VTABLE.

So much for what they are. Let's take a look at what's inside them.

WHAT'S OUT THERE?

Before you can use the tables, you have to know what tables are available. In addition to this paper, there are several sources.

SAS Online Doc. Enter "dictionary tables" as the search term.

Viewtable. While in an interactive SAS session, use SAS Explorer to display the contents of the SASHELP library. The views of the dictionary tables begin with 'V'. Double-click on any 'V' object to invoke Viewtable to display its contents.

Alternately, if you know the object's name, enter 'viewtable *object_name*' on the command line. For example, to see the view of the OPTIONS table, enter the following in the command line:

```
viewtable sashelp.voption
```

Viewtable is the ideal tool for gaining familiarity with the contents of the tables. You can browse the data, adjust the order of columns, and utilize the viewer to get an understanding of the data that would be difficult to gain from the documentation or PRINT procedure output alone.

CONTENTS. Provided you know the name of the view, you can use the CONTENTS procedure (or the CONTENTS statement in the DATASETS procedure) as follows:

```
proc contents data=sashelp.view;
```

In the above, *view* is the name of the view of interest (vcolumn, vcatalg, etc.).

SQL. The DESCRIBE statement in the SQL procedure provides some of the same information as CONTENTS. Results are written by default to the SAS Log. Examples for tables and views follow:

```
describe table dictionary.table;
describe view sashelp.view;
```

Sample output from both forms of the statement follow Exhibit 1, below.

Programmatically, via a Macro. You can list the attributes (*not* the contents) of the tables and views programmatically. A macro to do this is shown below:

EXHIBIT 1: PROGRAMMATIC LISTING OF TABLE ATTRIBUTES

```
%macro DictInfo;
       proc sql noprint;
       %if %scan(&sysver., 1) = 9 %then %do;
           select distinct memname,
                  count(distinct memname)
           into :tbl separated by ' ', :ntbl
           from dictionary.dictionaries ;
           %end;
           %else %do;
                 %let tbl = CATALOGS COLUMNS
                            EXTFILES INDEXES
                            MACROS MEMBERS
                            OPTIONS STYLES
                            TABLES TITLES
                            VIEWS;
                 %let ntbl = 11;
                 %end;
       select memname,
              count(distinct memname)
              into :view separated by ' ',
                   :nview
       from dictionary.views
       where memname like "V%" and
             libname = 'SASHELP' and
             memtype = 'VIEW' ;
       %do i = 1 %to &ntbl.;
           %let item = %scan(&tbl., &i.);
           describe table dictionary.&item.;
       %end;
       %do i = 1 %to &nview.;
           %let item = %scan(&view., &i.);
           describe view sashelp.&item.;
       %end;
       quit;
%mend;
```

The macro uses SQL to write a description of each table to the SAS Log. The following, for example, is the description of the EXTFILES table.

```
NOTE: SQL table DICTIONARY.EXTFILES was
created like:

create table DICTIONARY.EXTFILES
  (
   fileref char(8) label='Fileref',
   xpath char(1024) label='Path Name',
   xengine char(8) label='Engine Name'
  );
```

The SQL output for Views is usually a bit more compact. For the EXTFILES table references above, we see:

```
NOTE: SQL view SASHELP.VEXTFL is defined as:

        select *
          from DICTIONARY.EXTFILES;
```

It's important to emphasize that these tools simply identify what is available for use. Actually understanding their usefulness, nuances, and quirks is a bit of an art, most effectively developed while meeting real-world needs.

TABLE AND VIEW ORGANIZATION

SAS Version 9 supports 22 dictionary tables (available in SQL) and 29 views of the tables (available with the reserved LIBNAME of SASHELP). Version 8 supports 11 tables and 17 views. The tables and their relationship to the views are presented in **Exhibit 2**.

EXHIBIT 2: TABLE-VIEW CORRESPONDENCE

DICTIONARY.*table*	SASHELP.*view*	Earliest Version[1]
catalogs	vcatalg	6.07
check_constraints	vchkcon	9.0
columns	vcolumn	6.07
constraint_column_usage	vcncolu	9.0
constraint_table_usage	vcntabu	9.0
dictionaries	vdctnry	9.0
engines	vengine	8.0
extfiles	vextfl	6.07
formats	vformat	9.0
goptions	vgopt, vallopt[2]	9.0
indexes	vindex	6.07
libnames	vlibnam	9.0
macros	vmacro	6.11
members	vmember, vsacces, vscatlg, vslib, vstable, vsview, vstabvw	6.07
options	voption, vallopt[2]	6.07
referential_constraints	vrefcon	9.0
remember	vrememb	9.0
styles	vstyle	8.0
tables	vtable	6.07
table_constraints	vtabcon	9.0
titles	vtitle	6.11
views	vview	6.07

Notes:

[1] This is the first version that had the table and view. Readers familiar with the tables should note that some of the older, familiar tables have new fields in Version 9. Notable among these are TABLES and COLUMNS.
Note, too, that some fields have subtly different content in Version 9 – CRDATE and MODATE in TABLES, for example, is now a date-time value, rather than a simple date value.

[2] VALLOPT has all rows and columns from the GOPTIONS and OPTIONS tables.

The remainder of this section is devoted to descriptions of some of the more popular tables. "Popular" is, of course, a loaded term. In this context, it simply means tables that the authors have used most frequently in their daily work. Descriptions of most of the other tables are found in Appendix A. **Exhibit 3**, below, explains the format of the table descriptions.

EXHIBIT 3: TABLE DESCRIPTION FORMAT

dictionary.*TableName* sashelp.*ViewName*		
➲ column	type/len	Column is in Version 8 Arrow indicates the column helps to uniquely identify a record in the table.
column	type/len	Shaded rows: new to Version 9

COLUMNS

Content: Information about variables in currently allocated data sets and views. See **Exhibit 4** for details.

Granularity: Variable in a data set or view.

Comments:

[1] Users familiar with this table should pay particular attention to new fields in Version 9.

[2] Variable NAME, containing the name of a variable, is stored with the upper-lower case mix used at its creation. Subsetting on NAME should, therefore, either be explicit about case or use a function to fold NAME to consistently upper or lower-case.

[3] Users of CONTENTS output data sets will be relieved to see the inscrutable-but-predictable 1-2 TYPE values replaced by the dictionary table's more straightforward 'num' and 'char'.

[4] There are some situations where using a CONTENTS procedure output dataset is more effective than the dictionary tables and views. This is because these data sets are not normalized, unlike the dictionary tables.
A CONTENTS data set will have information at both the table *and* variable levels in the same observation (e.g., number of observations, number of variables, individual variable name, type, and length). Getting the same information from the dictionary tables requires a join of the MEMBERS, COLUMNS, and INDEXES tables.
This is not a huge problem, but is generally not as convenient as the "old" way. Indeed, there are items in the CONTENTS data sets that are simply not available anywhere in the dictionary data. Among these are whether the NODUPREC and NODUPKEY options were used during creation of the data set.

Used in Examples: 1, 5, 6.

MACROS

Content:.Values of all current macro variables. See **Exhibit 5** for details.

Granularity: Macro scope, name, and offset.

Comments: Long (greater than 200 character) values are broken into multiple observations, then distinguished by their offset value (first observation is offset 0, next is 200, and so on). To work with unique macro variable names, use a WHERE clause like "offset = 0".

Used in Examples: 3, 4.

Exhibit 4: COLUMNS Table

dictionary.columns sashelp.vcolumn		
➲ libname	$8	Library name [upper case]
➲ memname	$32	Member name [upper case]
➲ memtype	$8	Member type [DATA|VIEW]
➲ name	$32	Column name [case as-is from data set creation]
type	$4	Column type [char|num]
length	num	Column length
npos	num	Column position [offset within observation, e.g., 0, 1, 20]
varnum	num	Column number in table [1, 2, 3, …]
label	$256	Column label
format	$16	Column format [DATE9. $HEX22.]
informat	$16	Column informat [MMDDYY10. 8.2]
idxusage	$9	Column index type [SIMPLE|COMPOSITE|-BOTH]
sortedby	num	Order in key sequence [0, 1, 2, …]
xtype	$12	Extended type
notnull	$3	Not NULL? [no|yes]
precision	num	Precision
scale	num	Scale

Exhibit 5: MACROS Table

dictionary.macros sashelp.vmacro		
scope	$9	Macro scope [GLOBAL | AUTOMATIC | macro_name if local]
name	$32	Macro variable name [upper case]
offset	num	Offset into macro variable [0, 200, …]
value	$200	Macro variable value [case and spacing are preserved]

MEMBERS

Content: Information about SAS data stores – data sets, views, template item stores, catalogs, MDDB, etc. See **Exhibit 6** for details.

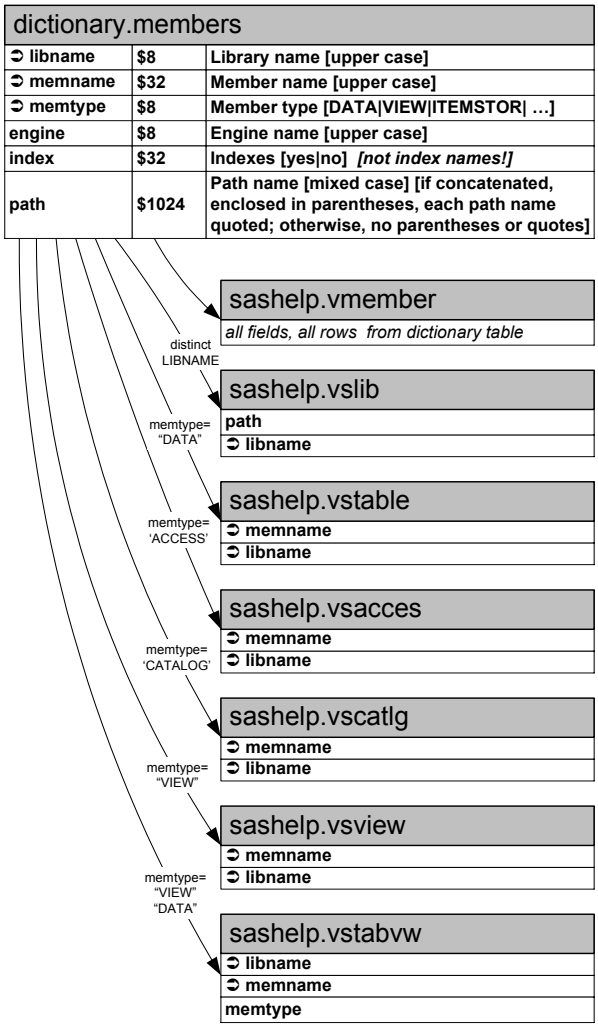
Granularity: Unique combination of library name, entity name, and entity type.

Comments:

[1] The views of this table facilitate subsetting by entity type (VSTABLE selects only data sets) and library (VSLIB is useful when identifying a physical path for a LIBNAME).

[2] See Note 4 in the description of the COLUMNS table, above.

Exhibit 6: MEMBERS Table



OPTIONS / GOPTIONS

Content: Information about Base SAS and SAS/GRAPH options. See **Exhibit 7** for details.

Granularity: Option.

Comments:

[1] The OPTDESC field contains brief, useful descriptions of the option.

[2] The GETOPTION function, used with the KEYWORD parameter, provides a useful and somewhat more powerful way to retrieve option settings.

Used in Example: 12.

Exhibit 7: OPTIONS Table

dictionary.options sashelp.voption		
➲ optname	$32	Option name [upper case]
setting	$1024	Option setting [mixed case]
optdesc	$160	Option description
level	$8	Option location [Portable|Host]
group	$32	Option group [upper case] [MACRO|SORT|ENVFILES| …]

TABLES

Content: Information about data sets and views. See **Exhibit 8** for details.

Granularity: Unique combinations of library name, data set/view name.

Comments: Users familiar with this table should pay particular attention to new fields in Version 9. Also note the DATETIME formats and informats associated with CRDATE and MODATE. These used to be date values. The increased date-time precision is a welcome change, but is also one that will require modification of Version 8 programs that use these values.

Used in Examples: 8, 9, 10, 11.

Exhibit 8: TABLES Table

TITLES

Content: Current titles and footnotes. See **Exhibit 9** for details.

Granularity: Title / footnote

Comments: Only the first 256 characters of a title or footnote are stored.

Exhibit 9: TITLES Table

dictionary.titles sashelp.vtitle		
➲ type	$1	Title location [T|F]
➲ number	num	Title number [1|…|10]
text	$256	Title text

TOOLS TO PUT THE TABLES TO WORK

Ask "n" SAS programmers for a solution to a problem and you'll get at least "n" distinct answers! Almost all SAS users would acknowledge that it is a robust tool. At the same time, though, one would be hard-pressed to argue that effective use of the metadata described in this paper would *not* require familiarity with Structured Query Language (SQL) and the macro language.

It's no accident that each of the examples below employs SQL and most use the macro language. Its table-joining ability, ease of identifying group characteristics, and macro language interface make it the ideal tool for building applications with the SAS metadata.

The macro language's ability to conditionally execute some of, all of, or multiple statements, along with the %SYSFUNC function's on-the-fly use of common functions, provides the programmer with a powerful toolset for building powerful, generalized utilities.

Exhibit 10 identifies some of the features of SQL and briefly describes why they are useful for building applications using the metadata. A complete description of each feature is well beyond the scope of this paper. The Exhibit is simply intended to be a starting point, based on the authors' experience, of key SQL features used in building metadata-based applications.

Exhibit 10: Key SQL Features

Feature	Why Important?
Macro language interface (INTO, SEPARATED BY)	Store counts, lists, etc. in macro variables, which can be used for macro language decision-making and other purposes
SQLOBS macro variable	Automatic, reliable indicator of size of an executed query
SELECT statement	Specifies columns to retrieve; can be used to perform calculations and format retrieved columns
GROUP BY clause, HAVING expression	Data referred to in SELECT can be collapsed into groups meeting one or more criteria
WHERE expression	Filter data from table(s) based on one or more criteria

Exhibit 11 identifies key macro language features used in metadata-based programs. As with the SQL discussion, above, we do not enter the syntax and usage thicket. The table simply identifies parts of the language the authors most frequently use.

Exhibit 11: Key Macro Language Features

Feature	Why Important?
keyword parameters in macro definition	Clarity. The user doesn't have to remember the meaning of numerous positional parameters
Iterative %DO	Repeat an operation "n" times, where "n" might be specified by preceding SQL operations
%IF-%THEN	Conditional execution, possibly based on results from preceding SQL operation on metadata
%SYSFUNC	Use DATA step functions outside the DATA step, in any context acceptable for a call to a macro function. Result can be a macro variable or part of %IF-%THEN decision-making
Quoting functions	Useful for generalizing code (allows program to proceed with unanticipated inputs)

Finally, on the subject of "must have" tools, consider *how* much of the use of the tables takes place. Tables are often used as part of generalized code, or utilities. Rather than hard-code the statements needed for printing from every data set in a library, we generalize the code and print from a library specified via a macro parameter. The best way of providing access to these utilities is to store them in one or more macro libraries and make them available to all programs via the AUTOCALL option. A representative example follows:

```
options mautocall
        sasautos=('j:\common\macros',
                  's:\client1\proj244\macros')
        ;
```

Thus we see the importance of macro language *system options* as well as the language statements themselves.

EXAMPLES

There are *many* applications for the information contained in the dictionary tables. In this section we present a series of examples taken from "real world" applications. These are summarized in **Exhibit 11**. Application background, code and, when appropriate, output are shown. Notice that most of the examples show the tables and views used as part of macros. This is in keeping with the high-level, meta-data nature of the information. Abstracted information and generalized tools such as macros are a logical, complementary fit.

Exhibit 11: Index of Examples

#	Description	Tables Used	Other Features
1	Locate variables with conflicting type or length	columns	GROUP BY HAVING COUNT()
2	Identify current directory	extfiles	macro interface
3	Identify all global macro variables	macros	ORDER BY
4	Delete global macro variables	macros	macro interface
5	Identify variables meeting criteria	columns	macro interface %IF-%THEN
6	Identify features incompatible with transport format	columns	WHERE
7	Generate code from SQL		SELECT
8	Print from every data set in a library	tables	SQLOBS macro interface %DO %UNTIL
9	Extend Example 8	tables	Multiple SELECT
10	Calculation as part of SELECT	tables	SELECT
11	Count observations in a data set	tables	SQLOBS

#	Description	Tables Used	Other Features
12	Option capture and reset	options	GETOPTION SELECT stmt %IF-%THEN COUNT()

Example 1: Conflicting Variable Attributes

A harsh and dreary reality of working with "real world" data is the potential for inconsistent attributes. Data sets that should have identical lengths, type, formats, etc. can have mismatches. A demographic data set, for example, might store GENDER as character, length 1 ("M" and "F"). Outsource, laboratory data can represent the same information as numeric (1 and 2), or character, length 6 ("Male" and "Female"). Havoc ensues when the data sets are combined.

Example 1 uses the COLUMNS table to identify inconsistencies between like-named variables. Regardless of the number of times a variable appears in WORK library data sets, it should have the same attribute (we look at length and type here, but could extend it to format, informat, and label). Thus if we have more than one value, we have an inconsistency. The Example sets up two data sets, A and B, then uses SQL to ferret out the problems.

Example 1 (Part 1)

```
data a;
x = 'a';
y = 'ppp';
run;

data b;
length y 3;
x = 'xx';
y = 1;
z = '2';
run;

proc sql noprint;
     create table temp as
     select libname, name, length, type
     from dictionary.columns
     where libname = 'WORK'
     group by name
     having count(distinct length) > 1
          | count(distinct type) > 1
     ;
quit;
```

PRINT procedure output is shown below. It's not hard to think of extensions to the program: make it a macro, with parameters for LIBNAME selection and attributes to compare.

Example 1 (Part 2)

Obs	memname	name	length	type
1	A	x	1	char
2	B	x	2	char
3	B	y	3	num
4	A	y	3	char

Example 2: Identify Current Directory

Programs often need to know the name of the directory from which they are executing. A program could, for instance, need to identify whether it is running in a test or production environment. This could be determined from the directory name.

In the following Example, run in Windows, we use the EXTFILES table to write the directory name (variable XPATH) to global macro variable CURRENT. We assign a FILENAME before the SQL code, then clean up and deassign it after SQL terminates.

Once macro %CurrDir completes, the macro variable CURRENT will be available to the calling program.

Example 2

```
%macro CurrDir;
       %global Current;
       filename __temp '.';
       proc sql noprint;
            select xpath into :Current
            from dictionary.extfiles
            where fileref = '__TEMP';
       quit;
       filename __temp clear;
%mend
```

Example 3: Identify All Global Macro Variables

If you have read to this point, you're probably sold on the idea of using metadata as often as possible. And if you read this Example's title, you'll say "sure, I can write %put _global_; to list macro variables, but I'd rather exercise the MACROS table instead." With that in mind …

Example 3

```
proc sql noprint;
     create table _macvars_ as
     select *
     from dictionary.macros
     where offset=0 & scope='GLOBAL'
     order by name
     ;
quit;
```

Data set _MACVARS_ can be used by any reporting procedure to display the first 200 characters of all global macro variables (we use only OFFSET=0). The advantage of this approach compared to %put _global_ is that it is cleaner. The %PUT statement spews variable names and values to the SAS Log in a non-obvious manner. This may be fine for a few macro variables, but when "lots" are involved, the more precise and controlled presentation shown in the Example is preferable.

Example 4: Delete Global Macro Variables

Repeated execution of programs in an interactive SAS session can create a mass of unwanted and/or incorrectly assigned macro variables. It would be helpful to have a way clean up the session and delete all global macro variables. Prior to Version 8 and the introduction of the %SYMDEL statement, this was difficult (but not impossible). The simplest way was to end the SAS session and begin a new one. Crude, but effective.

The Example below uses the MACROS table and the macro variable interface to create a list, stored in macro variable __GLOB, of all Global macro variables (the list *will* include __GLOB). We then pass the list to %SYMDEL.

Example 4

```
%macro DelMacVar;
       proc sql noprint;
            select name into
                   :__glob separated by ' '
            from dictionary.macros
            where offset = 0 &
                  scope = 'GLOBAL'
            ;
       quit;
       %symdel &__glob.;
%mend;
```

A simple extension to DelMacVar is the addition of KEEP and/or DROP parameters, which would control the selection of macro variables in __GLOB.

Example 5: Use Variables Meeting Criteria

This Example demonstrates how the dictionary tables can be used in the first small step toward generalized code. Rather than hard-code a list of variable names – AE1, AE2, and so on – we use the COLUMNS table to identify these variables. Thus if one data set has AE1 through AE5 and another has AE1 through AE3, no changes to the program will be needed. In the Example below, we create a list with the names and use it in an array definition in a DATA step.

Example 5 (Original)

```
proc sql noprint;
  select trim(name) into :vars separated by ' '
  from dictionary.columns
  where libname='INDATA' and memname='AE' and
        substr(NAME,1,2)='AE' and type = 'char';
quit;

data test;
     set indata.ae;
     array ae(*) &vars.;
     do i = 1 to dim(ae);
        if ae(i) ^= ' ' then do;
           NonMissingAE = 1;
           leave;
           end;
     end;
     if NonMissingAE then output;
run;
```

A few caveats are warranted. First, this type of coding requires that you know the data – the program assumes that you want *any* character variable starting with AE. Be sure that the program performs the selection correctly. You want to avoid the inclusion of AESEV1, AERSLT1, and the like. Also consider another data-driven possibility, that there are *no* character variables beginning with AE. The ARRAY statement will be handed a null value of macro variable VARS, and will fail. A good practice when making lists is to create a count of elements in the list, then use the count to control the statements downstream of SQL. This is demonstrated below.

Example 5 (Revised)

```
%macro temp;
proc sql noprint;
  select count(*), trim(name)
    into :nVars,   :vars separated by ' '
    from dictionary.columns
    where libname='INDATA' and
          memname='AE' and
          substr(NAME,1,2)='AE' and
          type = 'char';
quit;

%if &nVars. > 0 %then %do;
    data test;
         set indata.ae;
         array ae(*) &vars.;
         do i = 1 to dim(ae);
            if ae(i) ^= ' ' then do;
               NonMissingAE = 1;
               leave;
               end;
         end;
         if NonMissingAE then output;
    run;
    %end;
%mend;
```

Example 6: Identify Features Incompatible with Transport Format

Remember the days of limits such as eight-character variable names, 200 byte character variables, and the like? They live on in reality as well as our memory, namely in SAS transport files. Rather than let SAS issue warnings and errors when it encounters attributes incompatible with the transport standard, we can use the COLUMNS table to identify problems. In the example, data set V8 has a number of features that are problematic: variable HAS GAP has an embedded blank; TOOLONG, not surprisingly, exceeds the length limit; and LONGLBL has a label that exceeds the transport file maximum.

The SQL code selects columns from the COLUMNS table, filtering on characteristics that will be a problem with transport format. The result set is stored in data set WHOOPS.

Example 6 (Part 1)

```
data v8;
length 'has gap'n LongVarName clean $1 longlbl 3
       TooLong $1000;
label longlbl = 'This label exceeds the 40 char
                 limit!!!!!!!!';
run;

proc sql noprint;
     create table whoops as
     select name, label, length
     from dictionary.columns
     where libname = 'WORK' and memname = 'V8' &
           (index(trim(name), ' ') > 0 |
            length(name)     > 8 |
            length(label)    > 40 |
            length           > 200
           ) ;
```

PRINT procedure output of data set WHOOPS is shown below. Note that this use of the dictionary data is very basic. It's not hard to think of extensions to the program. We could, for example, make it into a macro, passing it a one or two-level data set name.

Example 6 (Part 2)

```
name                                label
has gap
LongVarName
longlbl         This label exceeds the 40 char
                limit!!!!!!!!
TooLong

name          length
has gap            1
LongVarName        1
longlbl            3
TooLong         1000
```

Example 7: Generate Code from SQL

And then there are times when even the metadata isn't enough. Such is the case with transport files, where data sets have to be referenced individually, rather than collectively (a LIBNAME must point to a specific dataset, rather than a directory). You could manually create the requisite LIBNAME and CONTENTS procedure statements to get what you need, maybe even making these into a macro that you could invoke for each data set.

A non-obvious but effective approach is shown in Example 7. We identify the transport files programmatically, via a DOS command (DIR). We then process the command's output, creating data set XPT_FILES, with variable MEMNAME. Once this is done, we can create the LIBNAME and other statements to process each data set using SQL. The SELECT statement concatenates literals containing SAS statements and variables containing data set names. The result is stored in macro variable CONTENTS.

Example 7 (Part 1)

```
filename xDir pipe 'dir /b /on "c:\tmp\*.xpt" ';
data XPT_files;
     infile xDir;
     input;
     if index(_infile_, '.') then do;
        memname = scan(_infile_, 1, '.');
        output;
        end;
```

```
run;

proc sql noprint;
     select   "libname TRANS sasv5xpt 'c:\temp\"
           || trim(memname)
           || ".xpt'; proc contents data=trans."
           || trim(memname)
           || ";run;"
          into :contents separated by ' '
     from XPT_files
     ;
quit;
&contents.;
```

If data sets AE and DEMOG were in the directory, the following would be the value of macro variable CONTENTS.

Example 7 (Part 2)

```
libname TRANS sasv5xpt 'c:\temp\ae.xpt'; proc
contents data=trans.ae;run; libname TRANS
sasv5xpt 'c:\temp\dmg.xpt'; proc contents
data=trans.dmg;run;
```

The Example doesn't use dictionary tables, but is useful for demonstrating the power of the SELECT statement.

Example 8: Print from Every Data Set in a Library

Just as we went to some length in Example 7 to avoid manually writing statements for each data set in a library, here we also automatically generate code. This time, we use the TABLES table, identifying data sets in a library and printing up to 10 observations from each data set. *A priori* knowledge of the data set names is not needed. The macro gets this information from the metadata.

Example 8

```
libname inmeta 'J:\Client\Project\DATA\DERIVE';

%Macro Printit ;
  proc sql noprint;
      select memname into :dsns separated by ' '
      from dictionary.tables
      where libname="INMETA";
  quit;

  %put &sqlobs. data sets in INMETA are: &dsns ;

  %if &sqlobs. = 0 %then %do;
      %put Nothing to do!;
      %goto bottom;
      %end;

  %let i =1 ;
  %do %until (%scan(&dsns,&i)= );
      %let dsn = %scan(&dsns,&i);

      proc print data= inmeta.&dsn(obs=10);
         title2 "First 10 records from &DSN" ;
      Run;
      %let i = %eval(&i + 1);
  %end;

  %bottom: ;
%Mend;
```

As in previous Examples, it's interesting to consider improvements that we could make. An obvious tweak is controlling the number of observations to print. This could be done via a parameter defaulting to, say, 10 observations. A more significant improvement would be changes to the handling of empty (0-observation) data sets. Currently, if a data set is present but empty, nothing will be printed. We could count the observations in each data set (macro variable DSN) and print the data or write a message to the output file saying the data set was empty.

Example 9: Enhance Example 8

The previous Example was helpful, since it quickly listed observations from a data set, giving a feel for its contents. A more practical, operational example follows. It creates two lists, one from the TABLES table, as before, and one from a data set with cardiovascular disease (CVD) indicators. Macro variable IDS is a comma-separated list of patients with a CVD history. This list is used as a filter for the PRINT procedure executed within the macro %DO loop. Notice that we code multiple SELECT statements in SQL; it is not necessary to invoke SQL separately.

Example 9

```
libname VACC 'S:\RHO\VASOCOR\VVS_Acc\data\crf' ;

proc sql noprint;
     select quote(trim(id))
       into :ids separated by ','
       from vacc.riskmstr
      where CVDYN='Y'
      ;
      select memname into :dsns separated by ' '
        from dictionary.tables
       where libname="VACC"
      ;
quit;

%put ids with CVD are: &ids ;
%put data set in VACC are: &dsns ;

%Macro printit ;
       %let i =1 ;
       %do %until (%scan(&dsns,&i)= ) ;
           %let dsn = %scan(&dsns,&i);
           proc print data= Vacc.&dsn ;
           where id in(&ids) ;
           title2 "Records with CVD from &DSN.";
           run;
           %let i = %eval(&i + 1);
       %end;
%mend;
```

Example 10: Calculation As Part of an Expression

We have already seen a use of the SELECT statement for creating groups of executable statements (Example 7). To emphasize the statement's power, and its importance in reducing the amount of code that's written and maintained, we present another example. Using the TABLES table and a knowledge of our system's disk sector size, we compute the size of each member in a library. Variable SIZEMB is created within SQL, ready to use from data set SIZES. Since we create the variable straight from the table, no post-processing DATA step is required. You could, of course, process SASHELP.VTABLE in a DATA step to achieve the same result, but recall earlier comments about the typically less-efficient view processing.

Example 10

```
proc sql noprint;
     create table sizes as
     select memname,
            (npage * 16384 ) / 1048576 as sizeMB
            /* divide by 1,024 for KB */
            /* divide by 1,048,576 for MB */
     from dictionary.tables
     where libname = 'WORK';
quit;
```

Example 11: Count Observations in a Data Set

A data set's "n" is one of its most fundamental characteristics, and one that is often needed for decision-making in utility programs. This Example presents one of *many* possible approaches to identifying the count. The user passes the one or two-level name of the data set and, optionally, the name of the global macro variable that will hold the count. The variable is populated using the TABLES table, and is reset to -1 if the data set was not found. The COUNT variable, therefore, communicates whether the data set exists (value >= 0) and if it is populated (> 0).

Example 11 (Part 1)

```
%macro countobs(datastor=, count=_count_);
       %local dataok;
       %global &count.;

       %let datastor = %upcase(&datastor);
       %if %index(&datastor, .) > 0 %then %do;
           %let libname = %scan(&datastor,1,.);
           %let memname = %scan(&datastor,2,.);
           %end;
           %else %do;
                 %let libname = WORK;
                 %let memname = &datastor;
                 %end;
       proc sql noprint;
            select nobs into :&count
            from dictionary.tables
            where libname="&libname." &
                  memname="&memname." ;
       quit;
       %if &sqlobs. = 0 %then %let &count. = -1;
       %put COUNTOBS: Count variable
            %upcase(&count)=%left(&&&count);
%mend;
```

The Example below uses COUNTOBS, and takes full advantage of the information contained in the macro variable. If it has a value greater than 0, we print the data set. If the value is 0, we write a message to the print file. Finally, if the value is -1, we write to the print file, saying it could not be located. Rather than just not print empty or missing data sets, we take the time to say *why* we didn't print anything. The cost of extra macro language coding is far outweighed by the benefits of improved user understanding of the data.

Example 11 (Part 2)

```
%macro report(data=);
       %countobs(datastor=&data., count=n);
       %if &n. > 0 %then %do;
           proc print data=&data.;
                title "Data Set &data.";
           run;
           %end;
           %else %if &n. = 0 %then %do;
                 data _null_;
                      file print;
                      put "&data. was empty!";
                 run;
                 %end;
           %else %if &n. = -1 %then %do;
                 data _null_;
                      file print;
                      put "&data. could not be "
                          "located!";
                 run;
           %end;
%mend report;
```

Example 12: Option Capture and Reset

In the "Table and View Organization" section, above, we noted the power of the GETOPTION function's ability to store keyword options (those of the form *option=value*). The following Example takes advantage of both GETOPTION and the OPTIONS table. It is a pared-down version of a larger environment-resetting utility.

The macro is recognition of the need for utility programs to be good "guests" in the programs that call them. Rather than adjust centering, page dimensions, error-handling, and other settings within a macro and then return to the calling program, it would be desirable to make the changes as needed, *and then restore them to their original settings*.

This is accomplished by calling the macro twice: at the start of a utility / macro and at its termination. When called at the start (ACTION=START), it creates data set __OPT_START__, which is essentially a snapshot of all current options, stored in a form suitable for inclusion in an OPTIONS statement later on. If RESET is called at the end of a utility (ACTION=END), we use SQL to compare the current option settings to those at the beginning, as stored in __OPT_START__. The differences are stored in macro variable __SETTINGS. If the count of observations with differences (_NOPTS) is non-zero, we insert an OPTIONS statement with __SETTINGS.

Example 12 (Part 1)

```
%Macro Reset(action=);
    %let action = %upcase(&action.);

    %if &action. = START %then %do;
        proc sql noprint;
        create table __opt_start__ as
        select optname,
               setting as original_setting,
               getoption(optname, 'keyword')
                   as original_keyword
        from dictionary.options
        ;
        %end;

    %if &action. = END %then %do;
        proc sql noprint;
        select count(*),
               e.optname,
               trim(e.optname) as trimname,
               getoption(e.optname, 'keyword')
                   as setting
               into :_Nopts,
                    :__temp,
                    :__names separated by ' ',
                    :__settings separated by ' '
        from dictionary.options as e,
             __opt_start__       as s
        where s.original_setting ^= e.setting &
              s.optname           = e.optname
        ;
        quit;

        %if &_Nopts. > 0 %then %do;
            %put Restore values of [&__names.];
            options &__settings.;
            %end;
        %end;

%mend;
```

A representative calling sequence for the macro is shown below.

Example 12 (Part 2)

```
%macro PrettyPrint(data=);
       %reset(action=start);

       options leftmargin=1in rightmargin=1in
               nodate nonumber
               dkricond=nowarn ;
       … other statements …

       %reset(action=end);
%mend;
```

CONCLUSION

The dictionary tables provide straightforward access to a wealth of information about the SAS environment. They are best utilized when you take the time to understand their contents and attributes, develop expertise with SQL and the macro language, and have a generalized application to work on. The up-front investment of your time is well worth the payoff of having solid, elegant, and generalized applications.

AUTHOR CONTACT

Your comments and questions are valued and welcome. Address correspondence to:

Frank DiIorio: frank@CodeCraftersInc.com
Jeff Abolafia: jabolafi@rhoworld.com

APPENDIX A: OTHER TABLES AND VIEWS

dictionary.catalogs
sashelp.vcatalg

➲ libname	$8	Library name [upper case]
➲ memname	$32	Member name [upper case]
➲ memtype	$8	Member type [CATALOG]
➲ objname	$32	Object name [upper case]
➲ objtype	$8	Object type [SCL|FRAME|FORMAT|…]
objdesc	$256	Object description
created	num	Date created [DATETIME informat, format]
modified	num	Date modified [DATETIME informat, format]
alias	$8	Object alias [upper case]
level	num	Library concatenation level [0, 1, 2, …]

dictionary.dictionaries
sashelp.vdctnry

➲ memname	$32	Member name
memlabel	$256	Member label
➲ name	$32	Column name
type	$4	Column type
length	num	Column length
npos	num	Column position
varnum	num	Column number in table
format	$49	Column format
informat	$49	Column informat

dictionary.engines
sashelp.vengine

➲ engine	$8	Engine name [upper case]
alias	$8	Alias [upper case]
description	$40	Description
preferred	$3	Preferred? [yes|no]
properties	$1024	Engine dialog properties

dictionary.styles
sashelp.vstyle

➲ libname	$8	Library name
➲ memname	$32	Member name
➲ style	$32	Style name
crdate	num	Date created [DATETIME informat, format]

dictionary.formats
sashelp.vformat

➲ libname	$8	Library name [upper case] [source='C']
➲ memname	$32	Member name [upper case] [source='C']
➲ path	$1024	Path name [case preserved] [source='U']
➲ objname	$32	Object name [upper case] [e.g. GROUP, TYPE] [source='U']
➲ fmtname	$32	Format name [upper case] [e.g., GROUP, $TYPE] [no decimal point]
➲ fmttype	$1	Format type [F (format) I (informat)]
source	$1	Format source [U (user) B (built-in) C (from a catalog, i.e., created by PROC FORMAT)]
minw	num	Minimum width
mind	num	Minimum decimal width
maxw	num	Maximum width
maxd	num	Maximum decimal width
defw	num	Default width
defd	num	Default decimal width

dictionary.indexes
sashelp.vindex

➲ libname	$8	Library name [upper case]
➲ memname	$32	Member name [upper case]
➲ memtype	$8	Member type [DATA]
➲ name	$32	Column name [case varies]
idxusage	$9	Column index type [COMPOSITE|SIMPLE]
➲ indxname	$32	Index name [upper case]
indxpos	num	[Position of column in concatenated key | missing if not concatenated key]
nomiss	$3	NOMISS option [yes|*blank*]
unique	$3	UNIQUE option [yes|*blank*]

dictionary.libnames
sashelp.vlibnam

➲ libname	$8	Library name [upper case]
engine	$8	Engine name
➲ path	$1024	Path name
➲ level	num	Library concatenation level [0, 1, 2, …]
fileformat	$8	Default file format
readonly	$3	Read only? [no|yes]
sequential	$3	Sequential? [no|yes]
sysdesc	$1024	System information description
sysname	$1024	System information name
sysvalue	$1024	System information value

dictionary.views
sashelp.vview

➲ libname	$8	Library name [upper case]
➲ memname	$32	Member name [upper case]
memtype	$8	Member type [VIEW]
engine	$8	Engine name [SASESQL|SASDSV| …]