Paper 121-2022

Using Dictionary Tables to Create Dynamic Programs

Jake Reeser, NORC at the University of Chicago

ABSTRACT

This paper discusses the Dictionary library, and how to utilize some of the tables in this library to write dynamic programs. I will cover what the dictionary library is, how to access this library, useful tables in it, and how to use these tables to write data driven code. I will specifically look at the two tables Dictionary.COLUMNS and Dictionary.TABLES, and how to utilize these tables to pass in parameters to macros. This technique allows users to easily call a repeated macro, while also limiting user input errors.

INTRODUCTION

SAS maintains information (also referred to as metadata) about your current SAS session. This ranges from information regarding SAS libraries, datasets, macros, formats, macros, indexes and system options. This information is stored in the dictionary library and can be accessed in your SAS session. These tables are dynamic, and update as you make changes in your SAS session. The metadata in these tables is useful for writing programs that are data driven. Data driven code is more efficient, requires less maintenance over time, and is also easier to read.

WHAT IS THE DICTIONARY LIBRARY?

The Dictionary library is a read-only SAS library that provides dynamic information about the user's current SAS session. As you make changes to your SAS session, the tables in the Dictionary library will be updated to reflect those changes. This library is reserved for PROC SQL and cannot be accessed in the DATA step. It is possible to copy these tables into the work library, but this would cause these tables to no longer dynamically update.

The name of each Dictionary table along with its label is presented below.

Figure 1 : Dictionary Tables and descriptions

Member Name	Data Set Label
CATALOGS	Catalogs and catalog-specific information
CHECK_CONSTRAINTS	Check constraints
COLUMNS	Columns from every table
CONSTRAINT_COLUMN_USAGE	Constraint column usage
CONSTRAINT_TABLE_USAGE	Constraint table usage
DATAITEMS	Information Map Data Items
DESTINATIONS	Open ODS Destinations
DICTIONARIES	DICTIONARY tables and their columns
ENGINES	Available engines
EXTFILES	Files defined in FILENAME statements, or implicitly
FILTERS	Information Map Filters
FORMATS	Available formats

FUNCTIONS	Available functions
GOPTIONS	SAS/GRAPH options
INDEXES	Indexes
INFOMAPS	Information Maps
LIBNAMES	LIBNAME information
LOCALES	Available Locales
MACROS	Defined macros
MEMBERS	Tables, catalogs, and views
OPTIONS	SAS options
PROMPTS	Information Map Prompts
PROMPTSXML	Information Map Prompts XML
REFERENTIAL_CONSTRAINTS	Referential constraints
REMEMBER	Remembered information?
STYLES	Styles?
TABLES	Tables and table-specific information
TABLE_CONSTRAINTS	Table constraints
TITLES	TITLE statements
VIEWS	Views and view-specific information
VIEW_SOURCES	Sources Referenced by View
XATTRS	Extended Attributes

ACCESSING THE DICTIONARY LIBRARY AND DISPLAYING TABLE DEFINITIONS

The Dictionary library can be accessed only through PROC SQL. To see how a specific table in the dictionary library can provide, the DESCRIBE TABLE statement can be used. Specifically, the DESCRIBE TABLE statement writes a CREATE TABLE statement of the specified table to the SAS log. Using the describe table statement on the MACROS can be done like so:

```
proc sql;
   describe table dictionary.macros;
quit;
```

The resulting SAS log:

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

 create table DICTIONARY.MACROS
   (
    scope char(32) label='Macro Scope',
    name char(32) label='Macro Variable Name',
    offset num label='Offset into Macro Variable',
    value char(200) label='Macro Variable Value'
   );

 76         quit;
```

NOTE: The same information in dictionary tables can also be accessed in the DATA and PROC steps. This information is in the SASHELP library and are stored as views.

DICTIONARY.TABLES

The TABLES Dictionary table contains information about all tables defined in the current SAS session. It is useful to create lists of all tables in a library to allow for easier data processing. It is also useful for quick spot checks of certain table properties, such as if the number of observations in a table is as expected. The definition of the TABLES Dictionary table can be determined using the below code:

```
proc sql;
   describe table dictionary.tables;
quit;
```

The resulting SAS log:

```
NOTE: SQL table DICTIONARY.TABLES was created like:
 create table DICTIONARY.TABLES
   (
    libname char(8) label='Library Name',
    memname char(32) label='Member Name',
    memtype char(8) label='Member Type',
    dbms_memtype char(32) label='DBMS Member Type',
    memlabel char(256) label='Data Set Label',
    typemem char(8) label='Data Set Type',
    crdate num format=DATETIME informat=DATETIME label='Date Created',
    modate num format=DATETIME informat=DATETIME label='Date Modified',
    nobs num label='Number of Physical Observations',
    obslen num label='Observation Length',
    nvar num label='Number of Variables',
    protect char(3) label='Type of Password Protection',
    compress char(8) label='Compression Routine',
    encrypt char(8) label='Encryption',
    npage num label='Number of Pages',
    filesize num label='Size of File',
    pcompress num label='Percent Compression',
    reuse char(3) label='Reuse Space',
    bufsize num label='Bufsize',
    delobs num label='Number of Deleted Observations',
    nlobs num label='Number of Logical Observations',
    maxvar num label='Longest variable name',
    maxlabel num label='Longest label',
    maxgen num label='Maximum number of generations',
    gen num label='Generation number',
    attr char(3) label='Data Set Attributes',
    indxtype char(9) label='Type of Indexes',
    datarep char(32) label='Data Representation',
    sortname char(8) label='Name of Collating Sequence',
    sorttype char(4) label='Sorting Type',
    sortchar char(8) label='Charset Sorted By',
    reqvector char(24) format=$HEX48 informat=$HEX48 label='Requirements
Vector',
    datarepname char(170) label='Data Representation Name',
    encoding char(256) label='Data Encoding',
    audit char(3) label='Audit Trail Active?',
    audit_before char(3) label='Audit Before Image?',
```

```
    audit_admin char(3) label='Audit Admin Image?',
    audit_error char(3) label='Audit Error Image?',
    audit_data char(3) label='Audit Data Image?',
    num_character num label='Number of Character Variables',
    num_numeric num label='Number of Numeric Variables',
    diagnostic char(256) label='Diagnostic Message from File Open Attempt'
   );
75         quit;
```

Particularly useful variables in this table are NOBS, OBSLEN, and NVARS. These provide information about your table that can be used for quality checks. The LIBNAME variable is also useful when you want to filter to tables in a specific library.

Using the Baseball table in the SASHELP library, we can see what the TABLES dictionary table looks like:

```
proc sql;
   select *
   from dictionary.tables
   where memname='BASEBALL';
quit;
```

This results in the following output:

Figure 2 : Result window of Dicitionary.TABLES

Library Name	Member Name	Member Type	DBMS Member Type	Data Set Label	Data Set Type	Date Created	Date Modified	Number of Physical Observations	Observation Length	Number of Variables	Type of Password Protection	Compression Routine
SASHELP	BASEBALL	DATA		1986 Baseball Data	DATA	05AUG20:20:29:47	05AUG20:20:29:47	322	216	24	---	NO

Encryption	Number of Pages	Size of File	Percent Compression	Reuse Space	Bufsize	Number of Deleted Observations	Number of Logical Observations	Longest variable name	Longest label	Maximum number of generations	Generation number	Data Set Attributes	Type of Indexes
NO	2	196608	0	no	65536	0	322	9	27	0	.	ON	

Type of Indexes	Data Representation	Name of Collating Sequence	Sorting Type	Charset Sorted By	Requirements Vector	Data Representation Name	Data Encoding	Audit Trail Active?	Audit Before Image?	Audit Admin Image?	Audit Error Image?	Audit Data Image?	Number of Character Variables	Number of Numeric Variables
	NATIVE				181F10112222003333010232043301233300 1C0000100301	WINDOWS_64	us-ascii ASCII (ANSI)	no	no	no	no	no	6	18

DICTIONARY.COLUMNS

The COLUMNS Dictionary table contains information about all variables on currently defined tables in a SAS session. It displays similar information to what the PROC CONTENTS procedure produces. The COLUMNS table is useful in doing quality checks on the metadata of variables. It can also be used to write dynamic programs involving looping through all variables on a table. The definition of the columns dictionary table can be determined using the below code:

```
proc sql;
   describe table dictionary.columns;
quit;
```

The resulting SAS log:

```
NOTE: SQL table DICTIONARY.COLUMNS was created like:
 create table DICTIONARY.COLUMNS
   (
    libname char(8) label='Library Name',
    memname char(32) label='Member Name',
    memtype char(8) label='Member Type',
    name char(32) label='Column Name',
    type char(4) label='Column Type',
    length num label='Column Length',
    npos num label='Column Position',
    varnum num label='Column Number in Table',
    label char(256) label='Column Label',
    format char(49) label='Column Format',
    informat char(49) label='Column Informat',
    idxusage char(9) label='Column Index Type',
    sortedby num label='Order in Key Sequence',
    xtype char(12) label='Extended Type',
    notnull char(3) label='Not NULL?',
    precision num label='Precision',
    scale num label='Scale',
    transcode char(3) label='Transcoded?',
    diagnostic char(256) label='Diagnostic Message from File Open Attempt'
   );
 76         quit;
```

Useful variables in the COLUMNS table include TYPE, LENGTH, LABEL, and FORMAT. These variables include useful information about the columns in your dataset and can be used in programs to check the quality of your data. The NAME variable is also useful, as you can use it to create macro lists of all the variables in a particular dataset.

Using the Baseball table in the SASHELP library, we can see what the COLUMNS dictionary table looks like:

```
proc sql;
   select *
   from dictionary.columns
   where memname='BASEBALL';
quit;
```

This results in the following output:

Figure 3 : Result window of Dicitionary.COLUMNS

Library Name	Member Name	Member Type	Column Name	Column Type	Column Length	Column Position	Column Number in Table	Column Label
SASHELP	BASEBALL	DATA	Name	char	18	144	1	Player's Name
SASHELP	BASEBALL	DATA	Team	char	14	162	2	Team at the End of 1986
SASHELP	BASEBALL	DATA	nAtBat	num	8	0	3	Times at Bat in 1986
SASHELP	BASEBALL	DATA	nHits	num	8	8	4	Hits in 1986
SASHELP	BASEBALL	DATA	nHome	num	8	16	5	Home Runs in 1986
SASHELP	BASEBALL	DATA	nRuns	num	8	24	6	Runs in 1986
SASHELP	BASEBALL	DATA	nRBI	num	8	32	7	RBIs in 1986
SASHELP	BASEBALL	DATA	nBB	num	8	40	8	Walks in 1986
SASHELP	BASEBALL	DATA	YrMajor	num	8	48	9	Years in the Major Leagues
SASHELP	BASEBALL	DATA	CrAtBat	num	8	56	10	Career Times at Bat
SASHELP	BASEBALL	DATA	CrHits	num	8	64	11	Career Hits
SASHELP	BASEBALL	DATA	CrHome	num	8	72	12	Career Home Runs
SASHELP	BASEBALL	DATA	CrRuns	num	8	80	13	Career Runs
SASHELP	BASEBALL	DATA	CrRbi	num	8	88	14	Career RBIs
SASHELP	BASEBALL	DATA	CrBB	num	8	96	15	Career Walks
SASHELP	BASEBALL	DATA	League	char	8	176	16	League at the End of 1986
SASHELP	BASEBALL	DATA	Division	char	8	184	17	Division at the End of 1986
SASHELP	BASEBALL	DATA	Position	char	8	192	18	Position(s) in 1986
SASHELP	BASEBALL	DATA	nOuts	num	8	104	19	Put Outs in 1986
SASHELP	BASEBALL	DATA	nAssts	num	8	112	20	Assists in 1986
SASHELP	BASEBALL	DATA	nError	num	8	120	21	Errors in 1986
SASHELP	BASEBALL	DATA	Salary	num	8	128	22	1987 Salary in $ Thousands
SASHELP	BASEBALL	DATA	Div	char	16	200	23	League and Division
SASHELP	BASEBALL	DATA	logSalary	num	8	136	24	Log Salary

Column Format	Column Informat	Column Index Type	Order in Key Sequence	Extended Type	Not NULL?	Precision	Scale	Transcoded?
			0	char	no	0	.	yes
			0	char	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	char	no	0	.	yes
			0	char	no	0	.	yes
			0	char	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	num	no	0	.	yes
			0	char	no	0	.	yes
			0	num	no	0	.	yes

USING DICTIONARY TABLES TO DRIVE DATA PROCESSING

Writing data driven code generally involves using your data or its metadata to drive data processing. Or simply, it is using our data to write our code. This allows for several advantages, as data driven code is easier to read, takes less time to write, requires fewer overall lines of code, and requires less updates/modifications to your program over time.

CREATING MACRO VARIABLES IN PROC SQL

There are multiple ways to write data driven code, but in this paper, I will focus on doing this via passing in our metadata to macro functions in the SQL procedure. The INTO statement creates macro variables in the SQL procedure. This allows us to store a singular value of a column from a dataset.

Using the SASHELP.BASEBALL table, we can store the number of observations in the table in a macro variable:

```
proc sql;
   select count(*)
   into :nobs
   from sashelp.baseball;
quit;
%put &=NOBS;
```

Looking at the SAS log, we can tell how many observations are stored in our macro variable, `nobs`:

```
78          %put &=NOBS;
 NOBS=      322
```

CREATING MACRO LISTS

It is also possible to return multiple values from the INTO statement and create a macro list by using the SEPARATED BY statant. When combined with the COLUMNS Dictionary table, macro lists are useful for iteratively looping through variables. Say you wanted to apply a certain function or logic to either all variables, only character variables, or only numeric variables. Instead of manually typing in all the variables that you want to apply your macro function to, you can instead pass these variables into a macro list, and then literately loop through that list in the data step.

For this example, we want to apply the UPCASE function to all character variables in the SASHELP.BASEBALL dataset. First, we can utilize the COLUMNS table to create a list of our character variables, as well as the number of character variables:

```
proc sql;
   select distinct name, count(*)
   into :varlist separated by '~',
          :nvars
   from dictionary.columns
   where libname='SASHELP' and memname ='BASEBALL' and type='char';
quit;
%put &=varlist;
```

You can see what the `varlist` variable looks like in the log by calling the %PUT statement:

```
  73         %put &=varlist;
VARLIST=Div~Division~League~Name~Position~Team
```

Now that the `varlist` and `nvars` macros are created, we can iteratively loop through the character variables and apply the upcase function:

```
%macro upcase_chars();
  data work.upcase_baseball;
    set sashelp.baseball;
    %do i=1 %to &nvars;
         %scan(&varlist,&i.,'~') = upcase(%scan(&varlist,&i.,'~')) ;
    %end;
%mend;
%upcase_chars
```

PASSING METADATA INTO MACRO FUNCTIONS

When working with a large amount of data or datasets, sometimes you want to apply the same macro to multiple different variables or tables. Instead of writing out each macro call individually, we can pass the relevant metadata directly into macro functions. This is done by embedding your macro call into the SELECT statement and putting this into its own macro variable. For example, say we want to apply a MACRO function to every table in the SASHELP library. For simplicity's sake, the below READIN macro will just read in the input library, in this case SASHELP, into the WORK library:

```
%macro readin(tbl=, libin=, libout=work);
   data work.&tbl._temp ;
         set &libin..&tbl;
   run;
%mend readin;
```

There are 194 tables in the SASHELP library, so it would take some time to write out each table name individually. Using the TABLES dictionary table, we can pass all the table names from this library into out macro function, which then will get stored in a macro variable:

```
proc sql;
   select distinct
   cats('%readin(tbl=',memname,',libin=',libname,',libout=WORK)')
   into :tbllist separated by ' '
   from dictionary.tables
   where libname='SASHELP' and typemem='DATA';
quit;
```

By calling the created `tbllist` macro variable, the READIN macro function would then be executed once per table:

```
&tbllist.
```

The resulting SAS Log:

```
SYMBOLGEN:  Macro variable TBLLIST resolves to
%readin(tbl=AACOMP,libin=SASHELP,libout=WORK)
             %readin(tbl=AARFM,libin=SASHELP,libout=WORK)
%readin(tbl=AIR,libin=SASHELP,libout=WORK)
             %readin(tbl=APPLIANC,libin=SASHELP,libout=WORK)
%readin(tbl=ASSCMGR,libin=SASHELP,libout=WORK)
             %readin(tbl=ATTRLOOKUP,libin=SASHELP,libout=WORK)
%readin(tbl=BASEBALL,libin=SASHELP,libout=WORK)
             %readin(tbl=BEI,libin=SASHELP,libout=WORK)
%readin(tbl=BIRD,libin=SASHELP,libout=WORK)
             %readin(tbl=BIRTHWGT,libin=SASHELP,libout=WORK)
%readin(tbl=BMIMEN,libin=SASHELP,libout=WORK)
             %readin(tbl=BMT,libin=SASHELP,libout=WORK)
%readin(tbl=BRM,libin=SASHELP,libout=WORK)
```

CONCLUSION

The dictionary table provides the user a ton of great information about their current SAS session. The variables in these tables can be used programmatically to quality check data, and, when used along with PROC SQL, allow for easier data processing. These tables can also be used in data driven programming, which allows for less manual coding, easier understanding of code, and causes updating programs to be less labor intensive.

REFERENCES

Fehd, Ronald and Art Carpenter, 2007. “List Processing Basics: Creating and Using Lists of Macro Variables”. Proceedings of SAS Global Forum 2007. Cary, North Carolina; SAS Institute, Inc. Available at http://www2.sas.com/proceedings/forum2007/113-2007.pdf

Lafler, Kirk Paul, 2005. “Exploring DICTIONARY Tables and Views”. Proceedings of SUGI 30. Cary, North Carolina; SAS Institute, Inc. Available at https://support.sas.com/resources/papers/proceedings/proceedings/sugi30/070-30.pdf

CONTACT INFORMATION

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

Jake Reeser

NORC at the University of Chicago

55 E Monroe

Chicago, IL 60603

Reeser-jake@norc.org