Paper 3607-2019

FETCH()ing Use Cases for the Data Access Functions

Chad Mukherjee, BMO Financial Group

ABSTRACT

The SAS® data access functions [OPEN(), FETCH(), FETCHOBS(), GETVARN(), GETVARC(), ATTRN(), ATTRNC(), VARNAME(), and CLOSE()] are powerful and under-used elements of the SAS programming language, with broad applications in both macro and DATA step contexts. This paper demonstrates the mechanics and several common use cases of these functions in order to accomplish programming feats that would otherwise be a bit awkward or inefficient to attempt in SAS. In order to capture the most value from this paper, you should have a solid understanding of the DATA step program data vector (PDV) and typical DATA step behavior, as well as an established comfort level with the SAS macro language and the %SYSFUNC() macro function.

INTRODUCTION

Most SAS Programmers are introduced to SAS with a DATA step—the most fundamental and flexible data manipulation tool available to the SAS language. A typical DATA step might look like this trivial example:

```
data work.conference_demo;
    set sashelp.cars;
    row_number = _n_;
run;
```

The above DATA step creates a new dataset *WORK.CONFERENCE_DEMO* by doing the following:

- Iterate over each record in *SASHELP.CARS*
    - Load each observation's variable values into the PDV
    - Write the current DATA step iteration *_N_* to the *ROW_NUMBER* variable in the PDV
    - Write the PDV out as a new record in *WORK.CONFERENCE_DEMO*

The vast majority of DATA steps in the wild follow some variation of the above pattern, and for good reason—most of our data manipulation needs can be met by iterating over each row in a dataset, populating new variables, and writing the contents of the PDV out into a new dataset.

Yet sometimes, unique problems require unique solutions. The SAS data access functions give us more flexibility in our interactions with SAS data sets, so when we encounter unusual challenges we can resolve them with elegant solutions.

THE BASICS

The OPEN() function creates a pointer (a unique numeric identifier) in your environment (Macro or DATA step) which may be later used to access the dataset in question. Each time the OPEN() function successfully opens a dataset, it generates a new identifier (even if the dataset being opened already has a pointer assigned to it).

The FETCH() and FETCHOBS() functions read an observation from a dataset previously opened using the OPEN() function, and load it into the Data Set Data Vector (DDV). A DDV

is similar to the PDV—it is an in-memory vector containing variables and their associated values, and it can be interacted with by using the data access functions referenced in this paper. Unlike the PDV, there is a separate DDV for each dataset identifier generated by the OPEN() function.

The ATTRN() and ATTRC() functions return attributes of a table opened by the OPEN() function. ATTRN() is used to return "numeric" attributes, and ATTRC is used to return "character" attributes, but both take a *data-set-id* and an *attribute-name* as arguments.

The GETVARN() and GETVARC() functions return values from rows loaded into the DDV by the FETCH() or FETCHOBS() functions. These functions require a variable ***number*** rather than a variable name as input, which can be obtained using the VARNUM function.

The CLOSE() function closes an open dataset—this automatically happens at the end of a DATA step, but is very important to remember in macro contexts.

MACRO USE CASES

The following use cases and techniques will demonstrate relatively common scenarios that can benefit greatly from the powerful and flexible data access functions in a macro language context.

MACRO USE CASE 1: RETRIEVING ALL VARIABLE NAMES FROM A DATASET

Sometimes it may be beneficial to explicitly reference the variables in a dataset—this can be accomplished by hard-coding the variable names, but such a solution will be insufficient when the variables are changing or if the number of variables makes it impractical. A dynamic solution is often more ideal.

Such an occasion might arise if you need to append one dataset to another when the base table has fewer variables than the dataset being appended. You can use the FORCE option, but this produces a warning in the log—we can get a clean log if we explicitly specify the columns in a KEEP= dataset option.

One technique might be to use dictionary tables:

```
proc sql noprint;
/*--- BEGIN SETUP ---*/
   create table work.car_name_list as
   select make, model
      from sashelp.cars
         where upper(make) eq 'NISSAN';
/*--- END SETUP ---*/
   select upper(name) into :dataset_column_names separated by ' '
         from dictionary.columns
               where libname eq 'WORK' and
                           memname eq 'CAR_NAME_LIST';
quit;

proc datasets lib = work nolist nodetails;
   append base = work.car_name_list
          data = sashelp.cars
                  (where = (upcase(make) eq 'HONDA')
                   keep = &dataset_column_names);
quit;
```

This solution is dynamic and works, but it added an entire extra step just to obtain the variable names we needed. By adding the ***%dsvarlist()*** macro (included in some SAS

solutions) to our SASAUTOS path, we can achieve the same end without the intermediate SQL procedure step:

```
/*--- BEGIN SETUP ---*/
proc sql noprint;
   create table work.car_name_list as
   select make, model
      from sashelp.cars
         where upper(make) eq 'NISSAN';
quit;
/*--- END SETUP ---*/

proc datasets lib = work nolist nodetails;
   append base = work.arbitrary_data
          data = work.more_detailed_arbitrary_data
                  (keep = %dsvarlist(work.arbitrary_data));
quit;
```

MACRO USE CASE 2: RETRIEVING VARIABLE ATTRIBUTES FROM A DATASET

Dataset metadata is another example of information available to you as a SAS programmer, but potentially awkward to obtain without using the data access functions.

Again, we find that while gathering variable labels is very much possible using dictionary tables, it introduces an intermediate step which may obscure the true purpose of our code. Consider the trivial situation where we are printing each variable in a dataset by itself with an accompanying title leveraging each variable's label (if applicable):

```
%macro print_vars(in_ds = );

/* Intermediate step to parse library and dataset name from &in_ds  */
%let dataset_name = %scan(&in_ds, -1, .);
%let libname = %sysfunc(coalescec(%scan(&in_ds, -2, .),WORK));

/* Inventory variables and corresponding labels */
proc sql noprint;
   select name,
          coalescec(label, name)
            into :cars_vars separated by ' ',
                 :varlabels separated by '|'
      from dictionary.columns
         where libname eq "%upcase(&libname)" and
               memname eq "%upcase(&dataset_name)";
quit;

/* Loop through variables to be printed */
%do i = 1 %to %sysfunc(countw(&cars_vars, %str( )));
   %let variable = %scan(&cars_vars, &i, %str( ));
   %let label = %scan(&varlabels, &i, |);

   title "All Values of &label";

   proc print data = &in_ds;
      var &variable;
   run;

%end;
```

```
%mend print_vars;
%print_vars(in_ds = sashelp.cars);
```

We've run into a situation similar to the earlier example of retrieving variable names from a dataset—additional intermediate steps which make it less clear what the purpose of the code is. Alternatively, we could add a new macro ***%varlabel()*** to our autocall library that will smooth the process significantly—with the help of ***%varlabel()*** as well as our now-familiar friend ***%dsvarlist()***, we can accomplish this same task in an exceedingly straightforward manner:

```
%macro print_vars(in_ds = );

   /* Inventory variables */
   %let cars_vars = %dsvarlist(sashelp.cars);

   /* Loop through variables to be printed */
   %do i = 1 %to %sysfunc(countw(&cars_vars), %str( ));
      %let variable = %scan(&cars_vars, &i, %str( ));
         /* Get label for use in title */
      %let label = %sysfunc(coalescec(%varlabel(&in_ds,
&variable),&variable));

      title "All values of &label";

      proc print data = &in_ds;
         var &variable;
      run;
   %end;

%mend print_vars;
```

MACRO USE CASE 3: RETRIEVING DATASET CONTENTS ON-THE-FLY

Most good software or code design attempts to separate business logic from implementation of the logic—this ensures that business logic can be added or removed without fundamentally changing the mechanics of the logic implementation. In a SAS programming context, this concept frequently manifests in the form of "control tables", which help guide the execution of a process flow.

For the sake of this paper, we will imagine that we have a control table that contains the names of datasets that we would like to delete from the WORK library at the end of a program.

```
/*--- BEGIN SETUP ---*/
/* Assign dummy libname */

libname ctl_tbls "%sysfunc(pathname(work))";

/* Copy contents of SASHELP into CTL_TBLS */
proc datasets lib = sashelp nolist nodetails;
   copy out = work memtype = data;
quit;

/* Create ds_delete_list control tables */
data ctl_tbls.ds_delete_list(keep = dataset_name);
   length dataset_name $32;
   tables_to_delete = 'CARS COMPANY CREDIT';
```

```
   do i = 1 to countw(tables_to_delete, ' ');
      dataset_name = scan(tables_to_delete, i, ' ');
      output;
   end;
run;
/*--- END SETUP ---*/

/* Inventory tables to delete */
proc sql noprint;
   select dataset_name into :datasets_to_delete separated by ' '
      from ctl_tbls.ds_delete_list;
quit;

proc datasets lib = work nolist nodetails;
   delete &datasets_to_delete;
quit;
```

The above example is simple enough, but we are again required to perform an intermediate step (gathering the list of datasets to be deleted) before actually executing our delete step. Adding another macro ***%get_column_values()*** to our autocall library lets us pull the dataset names from the control table on-the-fly, so we don't have to write a whole extra step to load them into a macro variable. The resulting code is succinct and quite readable[1]:

```
proc dataset lib = work nolist nodetails;
   delete %get_column_values(ctl_tbls.ds_delete_list.dataset_name);
quit;
```

DATA STEP USE CASES

The Data Access functions aren't limited to use in the macro language—they are DATA step functions, after all! These functions give us paradigm-shifting flexibility when it comes to accessing and manipulating data inside a DATA step.

DATA STEP USE CASE 1: ON-DEMAND LAGS

The LAG() function is well-documented and has been a part of the SAS language for years, but the notion of the *queue* associated with the function can take some time to fully understand, and can still be frustrating if you aren't paying close attention.

Consider a dataset containing multiple arbitrary time series—we could create an example with the following code:

```
data series(keep = date series_id series_value);
   format date date9.;
   series_ids = 'a b c';
   do j = 1 to countw(series_ids,' ');
      series_id = scan(series_ids, j, ' ');
      do i = 1 to 12;
         date = intnx('month','30jan2018'd,i,'e');
         series_value = i;
         output;
```

[1] It should be mentioned that retrieving dataset values in this way is ***significantly slower*** than using traditional means. The difference will usually be worth the improvement in readability when the dataset being queried contains fewer than 100 rows and/or the performance difference is negligible in the context of the entire process' execution (i.e. value retrieval takes 3 seconds instead of .01 seconds, but the entire process takes 1 hour).

```
        end;
    end;
run;
```

Then, consider calculating a corresponding lagged series for each series in the dataset. If we don't know how the LAG() function works, we start with something like this:

```
data work.lagged_series;
   set work.series;
   lagged_series_value = lag(series_value);
run;
```

This *almost* works, but the first lagged value in each series after the first will be the last value of the previous series, instead of the missing value we might expect. Our logical next step might be to sort the dataset, then use by-group processing to conditionally calculate our lags, like this:

```
proc sort data = work.series;
   by series_id date;
run;

data work.lagged_series;
   set work.series;
   by series_id date;
   if not first.series_id then lagged_series_value = lag(series_value);
run;
```

As it turns out, this is actually ***even worse*** than our first attempt, because we're not feeding the queue properly. A typical final response is to use the lag function in an intermediate variable assignment, then conditionally assign the final variable to the intermediate value:

```
proc sort data = work.series;
   by series_id date;
run;

data work.lagged_series;
   set work.series;
   by series_id date;
   lagvalue = lag(series_value);
   if not first.series_id then lagged_series_value = lagvalue;
run;
```

This is a completely acceptable solution, but can be a pain to manage if you are having to calculate lags of thirty different distances, rather than just one (in which case you must create 30 intermediate variables). The data access functions provide a solution which allows you to sidestep the lag function queue and the intermediate variable assignments that sometimes come with it:

```
proc sort data = work.series;
   by series_id date;
run;

data work.lagged_series(drop = dsid rc);
   set work.series;
   by series_id date;
   retain dsid;
   if _n_ eq 1 then dsid = open('work.series');
   if not first.series_id then do;
      rc = fetchobs(dsid,_n_ -1);
```

```
        lagged_series_value = getvarn(dsid, varnum(dsid, 'series_value'));
    end;
run;
```

DATA STEP USE CASE 2: LOOKING AHEAD IN A DATA STEP

While the LAG() function provides a mechanism to retain values without using the retain statement or altering the contents of the PDV,there is no corresponding function which looks to the *next* value of a variable. There are a few ways of doing this without using the data access functions, but none are ideal:

1. PROC EXPAND offers a non-DATA step method of looking forward (the *lead* transformation), but it is only available with a SAS/ETS license.
2. You could use a tedious process of reverse sorting, lagging, and reverse sorting again to find leads
3. One could use the dataset on which the lookahead should be performed as a "second" dataset in the SET statement, offset by a row.

Option 3 is probably our best bet, but still not ideal:

```
data work.lead_coal;
   set sashelp.usecon;
   set sashelp.usecon
         (firstobs = 2
         rename = (coal = lead_coal)
         keep = coal);
run;
```

The above works, but it's not immediately clear what we're doing without introducing a very thorough supporting comment. Additionally, the PDV gymnastics we're performing make it clear that this is not an elegant solution.

Instead, we can clearly accomplish what we're trying do with the data access functions:

```
data work.lead_coal(drop = fetchrc dsid);
   set sashelp.usecon;
   retain dsid;
   if _n_ eq 1 then dsid = open('sashelp.usecon');
   fetchrc = fetchobs(dsid, _n_ + 1);
   lead_coal = getvarn(dsid, varnum(dsid, 'coal'));
run;
```

The above *does* require more lines of code, but it can be easily understood even without comments, if the reader is familiar with the functions. The DDV with the dsid value *1* protects us from the potential PDV collisions in the previous example. The code follows a very simple process:

1. Load SASHELP.USECON into the PDV with the SET statement
2. During the first DATA step iteration, open the SASHELP.USECON dataset for reading and load the dataset identifier into the retained variable DSID
3. For all DATA step iterations:
    a. Load the row *ahead of* the row currently loaded in the PDV into the DDV
    b. Retrieve the value of COAL from the DDV and assign it to the LEAD_COAL variable.

This process is particularly helpful if you have many look-ahead operations to perform—instead of *n* SET statements, each with a different FIRSTOBS= value and just as many RENAME= options, you may simply open the dataset once, then fetch the appropriate row for each lead and retrieve the value in the DDV.

DATA STEP USE CASE 3: MEMORY-INEXPENSIVE LOOKUPS

There are many well-known methods to join reference or lookup datasets to data containing the lookup key in SAS:

- Use PROC SQL to join the tables
- Use a DATA step merge
- Use a hash object lookup

The hash object lookup in particular is often an excellent choice, especially when joining a small lookup to a large dataset containing the key. By loading the entire lookup dataset into memory, performing lookups can be incredibly fast. Consider the following arbitrary example:

```
/*--- BEGIN SETUP ---*/
%let num_loans = 1000;
%let num_months = 120;

/* Create lookup table containing historical risk ratings */
data work.historical_loan_ratings(keep = date loan_number risk_rating);
   format date date9.;
   possible_risk_ratings = 'A B C D';
   do loan_number = 1 to &num_loans;
      do date_index = 1 to &num_months;
         date = intnx('month', '31dec1999'd,date_index,'e');
         rating_num = ceil(4*ranuni(123));
         risk_rating = scan(possible_risk_ratings,rating_num,' ');
         output;
      end;
   end;
run;
/*--- END SETUP ---*/

/* Look up risk ratings as we iterate through loan numbers and dates */
data work.loans(keep = date loan_number risk_rating);
   format date date9.;
   if 0 then set work.reference_table (keep = loan_number date risk_rating);
   declare hash historical_loan_ratings(dataset:"work.reference_table");
      historical_loan_ratings.definekey( "loan_number" , "date");
      historical_loan_ratings.definedata( "risk_rating");
      historical_loan_ratings.definedone();
   do loan_number = 1 to &num_loans;
      do date_index = 1 to &num_months;
         date = intnx('month', '31dec1999'd,date_index,'e');
         rc = historical_loan_ratings.find();
         output;
      end;
   end;
run;
```

In this example, we first emulate an ETL process where historical risk ratings are associated with loans over a ten-year period. Then, we iterate through the loans and dates in a separate step, retrieving the risk ratings from the hash object based on the loan number and date. The above is a good solution when there are only 1000 loans, but should we do if we have 1 million loans? Trying to load the reference table into a hash lookup has the potential for disaster—we have a good chance of running out of memory. If we get some help from our upstream ETL process to produce an intermediate lookup dataset, we can use a combination of hash object and data access functions to perform the same lookup on a much larger scale without exceeding our memory limitations:

```
   /*--- BEGIN SETUP ---*/
   %let num_loans = 1000000;
   %let num_months = 120;

   /* Create lookup table containing historical risk ratings */
   data work.historical_loan_ratings(keep = date loan_number risk_rating)
        work.hist_loan_index(keep = loan_number first_record);
      format date date9.;
      possible_risk_ratings = 'A B C D';
      first_record = 1;
      do loan_number = 1 to &num_loans;
         /* Generate secondary/intermediate lookup table to give us info about the
real lookup */
         output work.hist_loan_index;
         do date_index = 1 to &num_months;
            date = intnx('month', '31dec1999'd,date_index,'e');
            rating_num = ceil(4*ranuni(123));
            risk_rating = scan(possible_risk_ratings,rating_num,' ');
            output work.historical_loan_ratings;
            /* Make sure to iterate our record counter */
            first_record + 1;
         end;
      end;
   run;
   /*--- END SETUP ---*/

   /* Look up risk ratings as we iterate through loan numbers and dates */
   data work.loans(keep = date loan_number risk_rating);
      format date date9.;
      dsid = open('work.historical_loan_ratings');
      /* Initialize the variables found in work.reference_table */
      if 0 then set work.hist_loan_index (keep = loan_number first_record);
      /* Declare our hash object */
      declare hash first_loan_record(dataset:"work.hist_loan_index");
         first_loan_record.definekey( "loan_number");
         first_loan_record.definedata( "first_record");
         first_loan_record.definedone();

      do loan_number = 1 to &num_loans;
         /* Figure out what the first record in the lookup table is for the given loan
*/
         rc = first_loan_record.find();
         do date_index = 1 to &num_months;
            date = intnx('month', '31dec1999'd,date_index,'e');
            /* Iterate through the lookup table as we iterate through dates */
            fetchrc = fetchobs(dsid, first_record + date_index - 1);
            risk_rating = getvarc(dsid,varnum(dsid, 'risk_rating'));
            output;
         end;
      end;
   run;
```

In the process above, we:

1. Generate an additional dataset (we'll call it our *index* dataset) in our ETL step to tell us which row each loan starts on in the reference dataset.
2. Create a hash object containing our index dataset, rather than the historical loan ratings dataset.

3. Look up the first record for each loan in the reference dataset using the index hash object.
4. Use the FETCHOBS() and GETVARC() functions to pull the appropriate record directly from disk and populate our risk rating from the value held in the DDV.

This method is not as fast as the hash object-only method if the lookup data is small enough to fit in memory, but scales very nicely as we move into larger lookup data territory.

CONCLUSION

The data access functions give you a veritable swiss-army knife to work with as a programmer—these use cases are truly just the beginning. It is my hope that you are able to able to eliminate unnecessary steps using these tools, and solve problems that may have seemed insurmountable before.

APPENDIX

%DSVARLIST()

```
%macro dsvarlist(ds);

   %local dsid rc nvars i;

   %let dsid = %sysfunc(open(&ds));

   %do i = 1 %to %sysfunc(attrn(&dsid,nvars));
      %sysfunc(varname(&dsid,&i))
   %end;

   %let rc=%sysfunc(close(&dsid));

%mend dsvarlist;
```

%VARLABEL()

```
%macro varlabel(ds, var);

   %local dsid out;

   %let dsid = %sysfunc(open(&ds,is));

   %let out = %sysfunc(varlabel(&dsid,%sysfunc(varnum(&dsid,&var))));

   %let dsid = %sysfunc(close(&dsid));

   &out

%mend varlabel;
```

%GET_COLUMN_VALUES()

```
%macro get_column_values(target_column);

   %local dsid table_name column_name tablevars;

       /* Conditionally inject WORK as the libname if necessary */
   %if %sysfunc(countw(&target_column,.)) eq 2 %then %let target_column =
work.&target_column;
       /* Parse &target_column to get the fully-qualified table name */
```

```
    %let table_name = %scan(&target_column,1,.).%scan(&target_column,2,.);
        /* Parse &column_name to retrieve the actual column name */
    %let column_name = %scan(&target_column,-1,.);

        /* Always use %local to minimize side-effects */
    %let tablevars = %dsvarlist(&table_name);
    %local &tablevars;

    %let dsid = %sysfunc(open(&table_name));
        /* %syscall set lets us grab values without figuring */
        /* out what the variable type is, but we end up assigning */
        /* values for each of the dataset variables */
    %syscall set(dsid);
        /* Iterate over the column values for each row in the table */
    %do %while(%sysfunc(fetch(&dsid)) eq 0);
          %sysfunc(strip(&&&column_name));
    %end;

    %let dsid = %sysfunc(close(&dsid));

%mend get_column_values;
```

CONTACT INFORMATION

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

Chad Mukherjee
chad.d.mukherjee@gmail.com