Efficient Processing of Long Lists of Variable Names

Paulette W. Staum, Paul Waldron Consulting, West Nyack, NY

ABSTRACT

Many programmers use SAS® macro language to manipulate lists of variable names. They add prefixes or suffixes, insert delimiters, sort lists, identify matching words, etc. When these tools are used for lists with hundreds of entries, they can be surprisingly slow, taking minutes for simple string manipulation. Other methods of implementing list manipulations can significantly speed the processing of long lists. This paper will use DATA step techniques (including hash tables) and PROC SQL to dramatically reduce the time required for processing long lists.

INTRODUCTION

If generic programs are used for many different input data sets, they usually need to process lists of variables. These lists can come from input data sets or they can be specified by the people who use the programs. Much of the processing of these lists is standard: adding a prefix or suffix to each variable name, inserting delimiters between variable names, sorting the list of variables, etc. These tasks are often handled by utility macros.

Usually the lists are stored in macro variables and are manipulated by macro code. Since macro code generally runs in memory without disk access, this seems likely to be efficient. If your data sets have many observations and variables, you expect that most of the processing time will be spent on data handling. However, if your data sets contain hundreds of variables, the processing of variable lists can be the slowest part of the programs.

There are non-macro methods for processing these lists. Some of them are:

- PROC SORT
- PROC SQL
- DATA step with scan function
- DATA step with hash object

We will consider two examples of standard list processing tasks:

- Identifying duplicate items in a list
- Sorting a list

Simple macro language methods for this type of task require comparing each pair of items in a list. The execution times for these examples increase dramatically when there are large numbers of items in the lists. Simpler tasks, such as adding prefixes or inserting delimiters, is not increased as much by long list lengths.

BASIC TOOLS

Some lists are naturally created in data sets. Others are naturally stored as macro variables. Macros can convert lists between the two forms.

These macros will:

- Convert lists from data sets to macro variables
- Convert lists from macro variables to data sets
- Count the number of items in a macro variable list or data set

Note that these sample macros are examples, not production quality macros. Production macros would allow the user more control over the returned macro variable and the list delimiter. They would check for errors, use more macro quoting functions, clean up any environmental changes, provide more complete documentation, etc.

CONVERTING DATA SET LISTS TO MACRO VARIABLE LISTS

The first macro converts a data set list to a macro variable list. It uses PROC SQL because it is faster than using a DATA step with CALL SYMPUT.

```
%macro GetList(InDS=, Var=, OutStr=);
  %* InDS is the input data set.
     Var  is the variable holding the item.
     OutStr is the macro variable name.;

  %* Create output macro variable if needed;
  %if %symexist(&OutStr)=0 %then %do;
    %global &OutStr;
  %end;

  proc sql noprint;
    select &var into :&OutStr separated by ' '
     from &InDS;
  quit;
%mend GetList;
```

The output macro variable may already exist as a local variable in a calling macro environment. If the output macro variable does not exist, it is declared as a global macro variable to make sure that it will be available outside of the GetList macro.

Here is a sample call.

```
%getlist(InDS=sashelp.class, var=name, OutStr=FromGetList);
%put FromGetList = &FromGetList;
```

The next macro is a special case of the GetList macro. It extracts a list of the variables in a data set from a PROC SQL dictionary table.

```
%macro GetVars(InDS=, OutStr=);
  %if %symexist(&OutStr)=0 %then %do;
    %global &OutStr;
  %end;
  %let InDS=%upcase(&InDS);
  %if %index(&InDS,.)=0 %then %let InDS=work.&InDS;

  proc sql noprint;
    select name
       into :&OutStr separated by ' '
       from dictionary.columns
       where libname=”%scan(&InDS,1,.)”
         and memname=”%scan(&InDS,2,.)”;
  quit;
%mend GetVars;
```

The libname and memname are automatically returned from the dictionary table in upper case. It is important to upper case the input parameter for the data set name. Here is a sample call.

```
%getvars(inds=sashelp.class, outstr=FromGetVars);
%put FromGetVars = &FromGetVars;
```

COUNTING ITEMS IN A MACRO VARIABLE LIST

Counting items in a macro variable list is a basic function that can be done in many different ways. Here is one example. This macro can be used like a function call, returning the number of words or items.

```
%macro NWords(InStr=);
  %local textstr count;
  %let textstr = %cmpres(%nrbquote(&InStr));
  %let count = %sysfunc(countc(%nrbquote(&textstr), %str( )));
  %eval(&count+1)
%mend NWords;
```

Here is a sample call.

```
%put NItems = %NWords(InStr=abc xy_z);
```

CONVERTING MACRO VARIABLE LISTS TO DATA SET LISTS

Another macro converts a macro variable list to a data set list.

```
%macro LoadWords(InStr=, OutDS=work.words, Var=word, MaxLen=200);
%* InStr is a macro variable with a list.
   OutDS is the output data set.
   Var is the name of the created variable.
   MaxLen is the maximum length of a word.;

  %local I NWords;
  %let NWords = %NWords(instr=&instr);
  data &OutDS;
    length &var $&MaxLen;
    %do i = 1 %to &NWords;
      &var = "%scan(&InStr,&i)";
      output;
    %end;
  run;
%mend LoadWords;
```

This approach requires making an assumption about the maximum length of an item in the list. The MaxLen parameter allows control over this assumption.

If the length of the quoted string exceeds 200 characters, there will be a message in the log about possible missing closing quotes. This message can be suppressed by setting the option NOQUOTELENMAX. Here is a sample call.

```
%LoadWords(InStr=a b c d e f XYZ, OutDS=work.words);
```

COUNTING ITEMS IN A DATA SET LIST

Obviously, to count the number of items in a data set list, just count the observations in the data set. This is not necessary for the methods described below, but it is included for completeness.

```
%macro NObs(InDS=);
  %let dsid=%sysfunc(open(&inds));
  %if &dsid %then %do;
    %let nobs= %sysfunc(attrn(&dsid,NLOBSF));
    %let dsid=%sysfunc(close(&dsid));
  %end;
  %else %let nobs=-1;
  &nobs
%mend NObs;
```

Using the NLOBSF attribute with the ATTRN function gets the effective number of observations from the data set.

Here is a sample call.

```
%put NumObs = %NObs(InDS=sashelp.class);
```

Now we have some basic building blocks to use when testing different ways of processing lists. Next let's discuss two standard list processing tasks. The first example will introduce different ways of identifying duplicates in a list of items. The second example will discuss different ways of sorting a list of items.

EXAMPLE 1: IDENTIFYING DUPLICATE ITEMS

One type of list processing macro searches a list of variables for duplicate items.

- It determines the number of words in a list.
- It compares each word to every other word. (It checks each pair of words only once, not twice.)
- It outputs a list of the words which were duplicated.

There are several design decisions to be made.

1. Are case differences important or unimportant when comparing words?
2. Should each duplicate be in a separate macro variable or should all duplicates be in one list? If there are embedded spaces in the list values, separate macro variables will be more useful. In the context of lists of variables, one combined list is likely to be more useful.
3. What order should be used for the output list of duplicates? Should it be the input order or an alphabetic order?
4. If a word appears more than two times in the list, should it be included in the output once or multiple times?

In our examples, 1) the results are case sensitive and 2) a combined macro variable list is produced. It is easy to switch to case insensitivity and to producing many macro variables as output. The 3) order and 4) redundancy of the output are more intrinsically tied to the method used to identify duplicates, as we will see below.

A sample call for this type of macro is:

```
%DupWords(InStr=Black Berry Blue Berry Cherry Pine Apple Straw Berry Apple,
          OutStr=dups)
%put Duplicates = &dups;
```

The InStr parameter is the list of items to check. The OutStr parameter is the name of the output macro variable.

MACRO LOOP METHOD

The essential functionality in a simple macro method consists of two loops. The first loop isolates each item in the list. The second loop compares the items. This structure is efficient because it minimizes the number of calls to the scan function.

```
%macro DupWords(InStr=,OutStr=);
  %local i NWords;
  %if %symexist(&OutStr)=0 %then %do;
    %global &OutStr;
  %end;
  %else %let &OutStr=;
```

```
  %let NWords = %NWords(InStr=&InStr);
  %do i = 1 %to &NWords;
    %local word&i;
    %let word&i = %scan(&InStr,&i,%str( ));
  %end;

  %do i = 1 %to &NWords;
    %do j = &i + 1 %to &NWords;
      %* Are two words the same?;
      %if &&word&i = &&word&j %then %do;
        %let &OutStr=&&&OutStr &&word&i;
        %goto found;
      %end;
    %end;
  %found:
  %end;
%mend DupWords;
```

This macro loop example keeps the output list of duplicates in input order. It also retains redundant duplicates. There is a maximum size for each macro variable of 64K. It is possible to exceed this limit for a list of variables in a data set with thousands of variables. In this case, methods without macro variables must be used.

The tests were done on an IBM ThinkPad PC in a Windows XP environment. Comparable results were obtained in a Windows server environment, so the results are not specific to any one environment. All times are wall clock time in seconds. This type of macro runs quickly for up to about 100 items. Above 100 items, the required time increases rapidly. This is a problem when long lists (for example for data sets with more than 500 variables) are processed.

Identifying Duplicates in a List

Number of Words	Macro Execution Time in Seconds
40	.12
88	.40
176	1.41
220	2.23
440	8.99
506	11.58
1012	47.62

The SAS system option MSYMTABMAX controls the amount of memory available to the macro symbol table. The names and values of macro variables are stored in the macro symbol table. If the table's size exceeds the value of MSYMTABMAX, additional macro variables are stored on disk and macro variable processing becomes very slow. The default setting of MSYMTABMAX is 4,194,304. The test results were similar with MSYMTABMAX set to larger or smaller values (8,388,728 and 2,097,512).

Let's look at other solutions. For some of these methods, you need to convert lists between a macro variable and a data set. The macro %LoadWord (described above) is used to convert a list from a macro variable to a data set. The macro %GetList is used to convert a list from a data set to a macro variable.

PROC SORT METHOD

One alternative is to load the list into a data set and sort the data set using the NODUPKEY and DUPOUT options of PROC SORT. Using NODUPKEY limits the primary output to one observation per key value. (The key value is a word in the list.) The DUPOUT option names a data set which will hold any observations with duplicate words. This PROC SORT example produces the output list in alphabetic order. It also retains redundant duplicates.

```
%macro DupWords(InStr=,OutStr=);
  %LoadWords(InStr = &InStr);
  proc sort data=work.words dupout=work.dups nodupkey;
    by word;
  run;
  %GetList(InDS=work.dups,Var=Word, OutStr=&OutStr);
%mend DupWords;
```

PROC SQL METHOD

Another alternative is to load the list into a data set and use PROC SQL to extract the duplicate items from the list. This PROC SQL example produces the output list in alphabetic order and eliminates redundant duplicates.

```
%macro DupWords(InStr=,OutStr=);
  %LoadWords(InStr = &InStr);
  proc sql noprint;
    create table work.dups(drop=num) as
    select word, count(*) as num
      from work.words
      group by word
      having num > 1;
  quit;
  %GetList(InDS=work.dups,Var=Word, OutStr=&OutStr);
%mend DupWords;
```

DATA STEP SCAN METHOD

Another alternative is to use a DATA step with a scan function to identify duplicates. The logic is identical to the macro loop method logic.

```
%macro DupWords(InStr=,OutStr=, MaxLen=200);
  %local NWords;
  %if %symexist(&OutStr)=0 %then %do;
    %global &OutStr;
  %end;
  %else %let &OutStr=;
  %let NWords = %NWords(instr=&instr);
  data work.dups(keep=word);
    length word iword1-iword&NWords $&maxlen;
    array words {*} iword1-iword&NWords;
    do i = 1 to &NWords;
      words{i}=scan("&InStr",i);
    end;
   do i = 1 to &NWords;
     do j = i + 1 to &NWords;
       if words{i} = words{j} then do;
         word=words{i};
         output;
         goto found;
       end;
     end;
    found:
    end;
    stop;
  run;
  %getlist(InDS=work.dups, var=word, OutStr=&OutStr);
%mend DupWords;
```

DATA STEP HASH OBJECT METHOD

You can also use a DATA step with a hash object to identify duplicates. Quoting the online documentation: "The hash object provides an efficient, convenient mechanism for quick data storage and retrieval. The hash object stores and retrieves data based on lookup keys."

```
%macro DupWords(InStr=,OutStr=, MaxLen=200);
  %local NWords NDups;
  %if %symexist(&OutStr)=0 %then %do;
    %global &OutStr;
  %end;
  %else %let &OutStr=;

  data work.dups(keep=word);
    length word $&maxlen;
    declare hash words(ordered:'yes');
    rc=words.defineKey('word');
    rc=words.defineDone();
    NWords=count(trim("&InStr"),' ') + 1;
    do i = 1 to NWords;
      word = scan("&InStr",i," ");
      rc=words.add();
     if rc ^= 0 then output;
    end;
    stop;
  run;
  %getlist(InDS=work.dups, var=word, OutStr=&OutStr);

%mend DupWords;
```

The hash object does not permit duplicate keys, so any attempt to add a duplicate item will cause an error code to be returned. Essentially this method attempts to insert all the list items into a hash object and returns a list of the unsuccessful attempts.

This hash table example produces the output list in input order. It also retains redundant duplicates.

The hash table results are not significantly affected by varying the hash table size using "hashexp".

COMPARISONS

The results of comparing the different methods to identifying duplicate items in a list are:

Identifying Duplicates in a List

Number of Words	Macro	PROC SORT	PROC SQL	DATA Step Scan	DATA Step Hash Object
40	.12	.04	.03	.03	.04
88	.40	.07	.05	.05	.05
176	1.41	.12	.11	.11	.03
220	2.23	.23	.21	.20	.03
440	8.99	.66	.63	.65	.05
506	11.58	.68	.65	.67	.07
1012	49.05	2.46	2.39	2.37	.09

All the DATA step / procedure methods are more efficient than the simple macro code method. For short lists, the timing differences are not important. For long lists, non-macro methods are much faster. The hash object is an excellent choice.

When the output order or the handling of redundant duplicates is critical, the differences in the output of the methods may limit your range of choice. This table summarizes the output characteristics of the different methods.

Comparing Methods of Identifying Duplicates in a List

Method	Sample Result	Order	Redundancy
Macro	Berry Berry Apple	Original	Kept
Sort	Apple Berry Berry	Alphabetic	Kept
SQL	Apple Berry	Alphabetic	Omitted
Data Step / Scan	Berry Berry Apple	Original	Kept
Data Step / Hash	Berry Berry Apple	Original	Kept

EXAMPLE 2: SORTING LISTS

Let's take a quick look at another example of a standard list processing task. Suppose you need to sort a list of items that is stored in a macro variable. You start with an unordered list ("go forth and multiply") and finish with an ordered list ("and forth go multiply"). You can do this with simple macro loops or you can use PROC SORT or PROC SQL. The hash table method is of limited use for this sorting task because its output will not include duplicate items. Here is the PROC SQL method.

```
%macro SortWords(InStr=,OutStr=);
  %if %symexist(&OutStr)=0 %then %do;
    %global &OutStr;
  %end;
  %LoadWords(InStr = &InStr);

  proc sql noprint;
   select word into :&OutStr separated by ' '
     from work.words
     order by word;
  quit;
%mend SortWords;
```

Here is a sample call.

```
%SortWords(InStr= go forth and multiply, OutStr=OrderedList);
%put OrderedList = &OrderedList;
```

The results of comparing different methods for various lengths of lists are:

Sorting a List

Number of Words	Macro	PROC SORT	PROC SQL
88	1.10	.04	.03
176	6.53	.09	.08
220	12.3	.14	.11
440	91.6	.58	.38
506	136.4	1.07	.48

Again, a simple macro method is significantly slower than using a procedure. The fastest method is to use PROC SQL to create a sorted list.

CONCLUSIONS

Macro code does complex processes of long lists of items very slowly. If your program requires examining every pair of items in long lists, you can improve preformance by replacing macro loops with a DATA step or a PROC SQL step.

REFERENCES

Arthur L. Carpenter, California Occidental Consultants, “Storing and Using a List of Values in a Macro Variable”, SUGI 30

Robert J. Morris, RTI International, “Text Utility Macros for Manipulating Lists of Variable Names”, SUGI 30

Ian Whitlock, “Macro Bugs – How to Create, Avoid and Destroy Them”, SUGI 30

ACKNOWLEDGMENTS

Thanks to Randy Poindexter of SAS® Technical Support for improving the macro method. Thanks to Tom Hoffman of Hoffman Consulting for the code to count items in a list. Any remaining inefficiencies are my own.

CONTACT INFORMATION

Suggestions, questions and comments are encouraged. Contact the author at:

Paulette Staum

Paul Waldron Consulting, Inc.

2 Tupper Lane

West Nyack, NY 10994

Phone: (845) 358-9251

Email: staump@optonline.net