How To Use Proc SQL select into for List Processing
Ronald J. Fehd, Centers for Disease Control and Prevention, Atlanta, GA, USA

ABSTRACT
The SAS® macro language is simple, yet powerful. List Processing with Proc SQL is also simple, yet powerful. This Hands On Workshop paper provides programmers with knowledge to use the Proc SQL select into clause with the various SQL dictionaries to replace macro arrays and %do loops.
Expected audience is intermediate to advanced users, and macro programmers.

Keywords: dynamic programming, list processing, macro, SQL.

INTRODUCTION
1. How do I process every column in a dataset?
2. How do I process every file in a folder?
3. How do I process every member in a libref?
   or
4. How do I process every item in a list?

In this paper I review the theory of programming, how to process every item in a list using the utter simplicity of Proc SQL select into :list processing with SQL's dictionary tables.

PREREQUISITES
Students are expected to have the following minimum background:

- programming experience: three to seven years
- data step: data structure allocation with attribute or length statements
- macro language: allocate macro variables write macros with one or more steps
- procedures: Contents, Print

TOPICS
- programming theory: vocabulary
- Proc SQL syntax
- list processing (dynamic programming) with dictionaries:
  - columns
  - dictionaries, v9
  - filenames, not an sql dictionary, read with scl functions
  - macros
  - options, v9: group
  - tables
PROGRAMMING THEORY

We communicate in a natural language English (or Chinese, Dutch, French, or German) about the artificial language SAS. I use these computer science terms and concepts throughout this paper.

<table>
<thead>
<tr>
<th>program</th>
<th>data structure</th>
<th>algorithm</th>
<th>(also: metadata)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>attributes</td>
<td>declarative, information</td>
<td>compile-time</td>
</tr>
<tr>
<td></td>
<td>name : variable or column</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>type : character or numeric</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>length: in bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>character: 1–32,767</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>numeric : 1– 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>format</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>label</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>see Online Help: Index:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>declarative DATA step statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>organization</td>
<td>executable: action or control statements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>array: has numbered elements</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>do I = 1 to dim(array-name);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>put array-name(I);</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>list : has unnumbered items</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>do over array-name;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>put array-name;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>drop, keep</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>retain</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>where</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>input:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libref, data, variable</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>process</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>perform actions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>output:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>libref, data, to log or list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SQL SYNTAX

There are five SQL statements:

1. proc
2. create, closure (;): line 6
3. describe
4. select, closure (;): line 22
5. quit

The keyword select has one required clause, from, and five optional clauses: into, where, group by, having, and order by, which might be viewed more clearly conceptually in this hierarchy:

```
select
  into
from
where
  group by
  having
order by
```

```sas
PROC SQL;
PROC SQL noprint;
create table table-name as
  query-expression
  <order by order-by-item;
  <, ... order-by-item>>;
describe table table-name <, ... table-name>;
select <distinct> object-item
  <function>(object-item)
  <, ... object-item>
  into :macro-variable <separated by ' '>
  :macro-variable-A, :macro-variable-B
  :macro-variable1 - :macro-variable9999
  from Libref.Data
  where ColumnChar eq 'value'
  ColumnNum eq <num-value>
  and ColumnChar2 eq 'value2'
  group by group-by-item
  <, ... group-by-item>
  having sql-expression
  order by order-by-item
  <, ... order-by-item>>;
quit;
```
SQL BASIC PROCESSES

Proc SQL can be used to do each of the basic processes:

1. list data structure
2. list data
3. list only subset
4. create data
5. unique

LISTING DATA STRUCTURE

Proc SQL works like Proc Contents. Instead of the `data =` option, SQL has the `describe table` statement.

```sas
PROC Contents data = SAShelp.Class;
PROC SQL; describe table SAShelp.Class;
quit;
```

LISTING DATA

Proc SQL works like Proc Print. In the Print method the object is referred to with the `data =` option. In SQL the object reference is the `from` clause. Star (asterisk: `*`) means all variables.

```sas
PROC Print data = SAShelp.Class;
var _all_
PROC SQL; select * /* _all_ */
from SAShelp.Class;
quit;
```

LISTING SUBSETS

The `where` statement is available in all procedures. I illustrate it here as a data step option. The SQL `select ... from` statement has a `where` clause.

```sas
PROC Print data = SAShelp.Class
(where = ( Sex eq 'F'
and Age ge 14));
var Name Age;
PROC SQL; select Name, Age
from SAShelp.Class
where Sex eq 'F'
and Age ge 14;
quit;
```

CREATING DATA

A common task is to copy a permanent data set from a permanent storage library to the work library. The SQL statement `create table` provides a similar data manipulation environment.

```sas
DATA Work.Class;
set SAShelp.Class;
PROC SQL; create table Work.Class as
select *
from SAShelp.Class;
quit;
```

MAKING UNIQUE LIST

The SQL `select` statement has a `distinct` function, which can be used to collapse many instances of variable values into a unique list.

```sas
PROC Sort data = SAShelp.Class nodupkey
out = UniqueAge
(keep = Age);
by Age ;
PROC SQL; create table UniqueAge as
select distinct Age
from SAShelp.Class
quit;
```
SUMMARY OF BASIC SQL

There are several differences between the syntax of proc SQL and other procedures.

The two most important to note are that column (variable) names are separated by commas, and dictionary tables’ values are upper case.

**select:** use comma as delimiter between column names

**wrong:** select Column1 Column2 Column3

**right:** select Column1, Column2, Column3

**where:** values in dictionary tables are upper case

**wrong:** where Libname eq ‘SAShelp’

**right:** where Libname eq ‘SASHHELP’


LIST PROCESSING: WRITING CONSTANT TEXT

The select statement accepts strings as one of its arguments; each string can be either single- or double-quoted, which allows the use of macro variables.

Note: the length of column MemName is 32, which accounts for the wide space between the words Class and has.

The above example selects four objects. Now let us concatenate text and variable value, using double bang (!!, two exclamation points) as the join operator and put that text into a macro variable.

Note: line 9; The `like` operator chooses only names beginning with 'V'.

Note: line 12: the statements in the macro variable are procedure statements, therefore they must be executed after the `quit;` statement in line 11.

Statements in the macro variable List.

Dilorio and Abolafia [14, sugi29.237] discuss the SAShelp views associated with SQL dictionaries.

Compare with program `list-describe-table-dictionaries` below, which lists SQL dictionaries.

Output from statements in the macro variable List.
LIST PROCESSING SUMMARY

The are several steps in writing statements to a macro variable and executing them:

1. input: data structure
   (a) identify the input table
   (b) examine its data structure

2. process:
   (a) concatenation of text and values into macro variable
      i. identify subset, if any; note: values in ALL CAPS?
      ii. clarify the text surrounding the variable value(s):
           prefix, infix(es), suffix
   (b) remember closure or delimiter:
      clause(s): comma, space, other?
      statement(s): semicolon (;)
      step boundary: run;
   (c) execute the statements: SQL: before quit;
      SAS: procedures, macros: after quit;

3. output: consider ODS

In the next sections I use this list processing check list to examine several of the more commonly used dictionaries.

The first examples — dictionaries, macros and options — illustrate writing constant text. In the second section, I show how to write macro calls to generate more complicated amounts of text when reading columns (variables), filenames, and tables.

DICTIONARY DICTIONARIES

Now let's look at everything you ever wanted to know about all those SQL dictionaries.

Statements to list the data structure:
Note: V9 only.

```
PROC SQL; describe table Dictionary.Dictionaries;
quit;
run;
```

Log with the data structure:

```
create table DICTIONARY.DICTIONARIES
  (memname char(32) label='Member Name',
   memlabel char(256) label='Dataset Label',
   name char(32) label='Column Name',
   type char(4) label='Column Type',
```

Dictionary.Dictionaries (new in V9) is unique on MemName + Name (Column Name). We need a data set (list) unique on MemName in order to write these statements:

This program reads the SQL table Dictionary.Dictionaries, makes a data set with the names of all the SQL dictionaries, writes describe table statements for each, then executes those statements on line 10. The log contains the data structure of each SQL dictionary.

```
%note: need v9 for D.Dictionaries;
Proc SQL; create table List as
   select distinct MemName
   from Dictionary.Dictionaries;
select 'describe table Dictionary.' !! trim(MemName) !! ';' into :List separated by ' ' from List;
&List.;
quilt;
run;
```

Note line 10: the statements in the macro variable are SQL describe statements, therefore they must be executed before the quit; statement.

Compare with program `select-text-into-list` above, which lists SASHelp views.

This program illustrates a two-statement solution to this problem. See program `ProcSQL-list-groups` below for a single-statement solution.

**DICTIONARY MACROS**

The statement `%Put _user_;` writes an unsorted list of macro variable names and values to the log.

Here is the data structure of Dictionary.Macros:

```
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'
)
```

The task is to write a `%put` statement for each macro variable. Hard-coded this would be:

```
%put a: &a.;
%put b: &b.;
%put z: &z.;
```

This program writes the put statements into the macro variable named list and executes them, line 13.

```
%let Z = the last one;
%let A = 1st item;
%let M = middle;
Proc SQL; select '%Put ' !! trim(Name) !! ': ' !! trim(Value) !! ';' into :List separated by ' ' from Dictionary.Macros where Scope eq 'GLOBAL' and not(Name like 'SQL%') order by Name;
quilt;
&List.;
run;
```

Note: line 13; the statements in the macro variable are SAS statements, therefore they must be executed after the quit; statement.
The statements written to the log are a sorted list of global macro variables.

The statements in macro variable List:

```
A: 1st item
M: middle
Z: the last one
```

**DICTIONARY OPTIONS**

Let's take a look at Dictionary.Options. These next programs show how to find out:

- definitions and values of each option
- what options are in each group

What is the data structure?

```
create table DICTIONARY.OPTIONS
(
    optname char(32) label='Option Name',
    opttype char(8) label='Option type',
    setting char(1024) label='Option Setting',
    optdesc char(160) label='Option Description',
    level char(8) label='Option Location',
    group char(32) label='Option Group'
)
```

Note: group is available in v9.

Use list processing technique to write text:

```
options linesize = max;
PROC SQL; select 'Proc Options define value option = '||trim(OptName)||';'
    into :List separated by ' ' from Dictionary.Options;
quit;
&List.;
run;
```

Here's the list of values and definitions of all options. The log is over 4600 lines long.

In program `list-describe-table-dictionaries` above, using the distinct function, we created a table of the unique dictionaries before we could write the text of the describe table statements. We can reduce two statements to one by creating a named column with the distinct values and then refer to the new column in our text concatenation. This trick requires that we make two macro variables, Item and List, for each of the columns: Group as Item, and text.
This example shows that the variable value can be used more than once, and there is no limit to how much text can be concatenated, either before or after use of the variable value. The calculated keyword indicates that the column Item has been created — distinct Group as Item — and is not in the the table being read: from Dictionary.Options.

```
%Let List = *no rows selected;
Proc SQL; select distinct Group as Item,
  %Put Group: ' !! calculated Item
  %Put Group: ' !! calculated Item
  ;run;
  into :Item, :List separated by ',
  from Dictionary.Options;
quit;
&List.;
run;
```

Note that the column Item retains the label of Group.

The run; statement ensures that the the %Put Group: statement is written before each group.

**LIST PROCESSING: WRITING MACRO CALLS**

In the following sections I provide program templates for processing the three most commonly used lists: columns (variables), tables (data set names), and files.

**DICTIONARY.COLUMNS**

The task is to process every column in a data set.

```
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',
  label char(256) label='Column Label',
  format char(49) label='Column Format',
)
The primary parameter is the libref, line 1; see references in lines 22 and 27.
The secondary parameter is the table name in the libref, lines 3–6, referred to in lines 23 and 28.

A tertiary parameter, provided for clarity, is the member type, in (data, view), lines 8–9, see line 24.
Do you want to review the statements generated? or is this program running in production? Choose to enable either of lines 11 or 12.

The macro ProcVar is not yet defined at this time; this does not matter as we are writing the statements but have not yet executed them, line 44.

Both global macro variables LibName and MemName are indirectly referenced via macro parameters libref and data as their defaults.

This is a simple processing example. Your own code goes here.

Last, but not least, execute.

**DICTIONARY.TABLES**

In this example I show a production example of two programs where the first program containing the processing macro ProcDsn, contains a call (using the %include statement) of the second list processing program. This illustrates the concept of having one program define the parameters of another.

The task is to process every data set in a library. What are the columns whose values are to be passed to the processing macro?
The primary parameter is the libref, line 1.

Do you want to subset the processing by choosing the member type before writing the statement? Enable either line 6 or 7.

Do you want to review the statements generated? or is this program running in production? Choose to enable any or all of the testing statements in lines 11–13.

As in the processing of Dictionary.Columns program above, the statements written include most of the list attributes to be passed to the macro.

This is a simple processing example. Your own code goes here.

Call the list processing program that will read all tables in the library defined here and execute the macro ProcDsn defined in this program. Note that you could have other programs defining the macro ProcDsn and relying on the subroutine select-from-D-Tables.

Compare with program select-from-D-Columns above.
FILENAMES

Processing a list of filenames is different from the previous examples as there is no SQL dictionary of folders and filenames. Reading the list of files using SQL requires some interesting tricks. In this example, I have polished a program written by Hamilton [20, sugi31.046].

The primary parameter is the folder name.

The Digits data set is used to make a larger data set, FileNmbrs, containing the file-numbers used by the dread function.

Omitted lines 53–64 contain allocations of macro variables E3 and E4 which are necessary for processing lists of 999 and 9999 files.

Two select statements are required: the first to assign a fileref, open the folder and read the number of files present.

The second statement writes the macro calls.

Line 76 generates the list of file numbers.

Your filename-processing statements go here.

After executing the macro calls then do housecleaning: symbol-delete all the macro variables used.

CONCLUSION

Any data set is a candidate for use by list processing. To produce dynamic programs follow these simple steps:

- identify the data set (table)
- examine its data structure
- identify the variables (columns) that contain parameter values
- develop a program with example code
- use proc sql to write that code as text or macro call, substituting variable names for values
- sit back and watch the log zoom by
ACKNOWLEDGEMENTS
Ian Whitlock and Sig Hermansen have piqued my interest in SQL over the years with their contributions to SAS-L. Toby Dunn provided commentary on numerous examples.

SUGGESTED READINGS

bookshelf
Carpenter [6, saspress.59224], Celko [8, Celko.2000], and Celko [9, Celko.2005]

basics
Dickstein and Pass [12, sugi29.269], Hu [22, sugi29.042], Lund [25, sugi30.257], Ronk [27, sugi29.268], Wells [31, sugi26.105], Winn, Jr. [34, sugi22.067]

intermediate
and advanced concepts: Barber [4, sugi22.198], Hamilton [20, sugi31.046], Hermansen [21, sugi22.035], Lafler [23, sugi28.019], Loren and Nelson [24, sugi23.031], Winn, Jr. [35, sugi23.035]

list processing
Abolafia [1, sugi30.031], Andrews [3, sugi31.039], Beakley and McCoy [5, sugi29.078], ch. 9, dynamic programming, Carpenter [6, saspress.59224], Carpenter [7, sugi30.028], Fehd [17, sgf2007.028], Fehd and Carpenter [18, sgf2007.113], Pollack [26, sugi30.057], Varney [30, sugi31.045]

macros and SQL

BIBLIOGRAPHY


To get the code examples in this paper search [http://www.sascommunity.org](http://www.sascommunity.org) for the HOW SQL for List Processing zip.

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

About the author: Ronald Fehd  
Centers for Disease Control  
4770 Buford Hwy NE  
Atlanta GA 30341-3724  
mailto:RJF2@cdc.gov

- **education:** B.S. Computer Science, U/Hawaii, 1986  
- **SUGI attendee** since 1989  
- **SAS-L reader** since 1994  
- **experience:** programmer: 20+ years  
- **data manager at CDC, using SAS:** 18+ years  
- **author:** 12+ SUG papers  
- **SAS-L:** author: 4,000+ messages to SAS-L since 1997  
- **Most Valuable SAS-L contributor:** 2001, 2003

**Document Production:** This paper was typeset in [LaTeX](https://www.latextemplates.com). For further information about using LaTeX to write your SUG paper, consult the SAS-L archives: [http://www.listserv.uga.edu/cgi-bin/wa?S1=sas-l](http://www.listserv.uga.edu/cgi-bin/wa?S1=sas-l)

Search for:  
The subject is or contains: LaTeX  
The author’s address: RJF2  
Since: 01 June 2003