

Paper SAS4091-2020

Common Tasks Done with CASL

Jane Eslinger, SAS Institute Inc.

ABSTRACT

Some tasks are so common they are used in almost every program; tasks such as loading data tables and then either deleting or saving those tables. Other tasks include looping through the data, summarizing the data, and sorting the data. As a SAS® programmer, you know the code that you need to accomplish these common tasks. CASL, the language created to communicate with the SAS® Cloud Analytic Services (CAS) server, might be new to you. This paper demonstrates how to perform those common tasks with CASL. Understanding how to perform common tasks and having the code to do them empowers you to develop CASL programs and take full advantage of the CAS server.

INTRODUCTION

As you transition to using the SAS® Viya environment with the SAS® Cloud Analytic Services (CAS) server, there are several reasons to program in CASL. Using CASL ensures that all the code is processed by the CAS server. CAS actions are needed for tasks that do not have an equivalent procedure. In addition, CAS actions allow you to specify more parameters than a CAS-enabled procedure.

Some tasks are needed in multiple programs and can be used for any number of use cases. Developing code for these tasks helps you learn CASL and, just as importantly, helps you complete the creation of programs quickly. For example, loading data into in-memory CAS tables and then managing those tables is needed in every program. You might also need to loop through data, summarize it, or sort it.

This paper provides CASL code examples for completing these common tasks. Where appropriate it also provides a comparison to code you might have used in SAS®9. Please note the code examples in this paper were developed in SAS® Studio, but the code can also be submitted from the SAS windowing environment and SAS® Enterprise Guide®, if they have been configured to connect to the CAS server. The actions can also be used from Python, R, or Lua, but each language uses a slightly different syntax than shown here.

SYNTAX

CAS actions perform a single task. A CAS action is one statement. The syntax is:

```
ActionSet.Action / Parameters;
```

Action names must be unique within an action set; however, they are not required to be unique across action sets. Therefore, the action set name is required only for actions that are not unique across action sets.

Also, from the SAS programming environments, action statements are not stand-alone statements. They must be placed inside a CAS procedure step.

The examples in this paper demonstrate both using an action set name and not using an action set name. For some examples the action set name is included in order to demonstrate valid syntax, though it is not strictly required for the code to execute properly. Each example sits within its own PROC CAS step, but that is not a requirement. You can

place as many action statements within a PROC CAS step as you need to accomplish your goal.

EXAMPLE DATA

This paper assumes that you have already made a connection to the CAS server. Some of the examples in this paper assume that you have various tables in one or more caslibs. Those examples use generic caslib and table names.

For the examples that need to generate specific output in order to demonstrate a technique, the MyData table is used. The following step generates MyData and stores it in the CASUSER caslib.

```
libname myuser cas caslib='casuser';
data myuser.mydata;
  length color shape $8;
  array X{100};
  do k=1 to 9000;
  do i=1 to 50;
    X{i} = rand('Normal',0, 4000);
  end;
  do i=51 to 100;
    X{i} = rand('Normal', 100000, 1000000);
  end;
  if x1 < 0 then color='red';
  else if x1 < 3000 then color='blue';
  else color='green';
  if x2 < 0 then shape='square';
  else if x2 < 3000 then shape='circle';
  else shape='diamond';
  output;
end;
run;
```

HOUSEKEEPING

This section is labeled “housekeeping,” but it could just as easily be called “table management.” As you transition to using SAS Viya you need to learn the following common tasks: loading data to the CAS server, obtaining the contents of caslibs and in-memory tables, saving an in-memory table to a caslib's data source, and dropping tables from memory. The CASL code examples to accomplish those tasks are described in this section.

LOADING

The first step to doing anything in a SAS Viya environment is loading data into memory. In-memory data is the data that you summarize, analyze, or use for scoring models. In short, it is the data that you need to do your job.

The data that you need to work on might originally be stored in a CSV file, an Excel file, an external database, such as Hadoop, or a SAS 9 data set. You can create an in-memory table through a DATA step, like in the “Example Data” section of this paper.

The code in Example 1 loads a SAS 9 data set, called Sample, that is stored in the ProjectA caslib. The in-memory table is loaded to the CASUSER caslib and is named Example1.

For this example, notice the CASLIB= parameter explicitly states where the data set is stored. The PATH= parameter must be a path that is relative to the active caslib. If the file that you are trying to load is in the active caslib, the CASLIB= parameter is not necessary. Otherwise, if the file you are trying to load is not in your active caslib, you need to specify

CASLIB=. In addition, the file might be in a folder within the caslib. For those circumstances, include the folder name in the PATH= value and follow it with a slash (/).

Example 1:

```
proc cas;
  table.loadtable /
    path="sample.sas7bdat"
    caslib="projecta"
    casout={caslib="casuser" name="example1" replace=true};
quit;
```

Alternatively, you do not have to specify the extension of the file, like SAS7BDAT, if you provide the FILETYPE= parameter, as in Example 2. For SAS data sets the value is BASESAS. Be sure to check the documentation for other appropriate values.

Example 2:

```
proc cas;
  table.loadtable /
    path="sample"
    caslib='projecta'
    casout={caslib="casuser" name="example2" replace=true}
    importoptions={filetype='basesas'};
quit;
```

Examples 1 and 2 load a data set that is already in a caslib. With CASL, to load a data set that is not in a caslib, you must use the UPLOAD statement. You must know the path where the data set is located, and that path must be visible to the CAS server. The path can be hardcoded, as it is in Example 3, or obtained through other coding means. Note that the path includes the name of the data set and the file extension of SAS7BDAT.

Example 3:

```
proc cas;
  upload path="/tmp/sample.sas7bdat"
    casout={name='example3' caslib='casuser'};
quit;
```

The code to load a CSV or Excel file is very similar to loading a data set. You specify the appropriate file name with extension, as in Example 4.

Example 4:

```
proc cas;
  table.loadtable /
    path="stores.csv"
    casout={caslib="projecta" name="example4" replace=true};
quit;
```

Example 5 loads the data from an Excel file into an in-memory table. Please be aware that on a Linux system the name of the file is case sensitive. An error is generated if you do not use the correct capitalization. Notice this example does not include the action set name, TABLE.

Example 5:

```
proc cas;
  loadtable /
    path='SAMPLE1.xlsx'
```

```

importoptions={filetype="excel"}
casout={caslib="projecta" name="example5" replace=true};
quit;

```

From the previous examples, you can see that the LOADTABLE action is the predominant method for loading data into memory. Remember, LOADTABLE is used when the data file already resides in a caslib.

OBTAINING CONTENTS

Caslibs are designed to contain source files, like CSV and SASHDAT, as well as in-memory tables. For various reasons, you might need to determine the names of those files and in-memory tables. There is one action to list the files in a caslib and another action to list the in-memory tables in a caslib. A third action provides the contents of a specific in-memory table.

Example 6 lists the files in the ProjectA caslib. Included in the TABLE.FILEINFO statement is the ALLFILES=TRUE parameter, which ensures SAS and SAS7BCAT files are included in the results along with all other file types. Output 1 shows an example of what the results might look like.

Example 6:

```

proc cas;
  table.fileinfo /
    caslib='projecta'
    allfiles=true;
quit;

```

Results from table.fileinfo

File Information for root of caslib projecta.					
Permission	Owner	Group	Name	Size of File in Bytes	Time
rw-r--r--	user1	sasusers	sales.csv	622	2019-10-14T17:32:36-05:00
-rw-r--r--	user1	sasusers	final1.sas7bdat	458752	2019-12-10T14:30:48-05:00
-rw-r--r--	user1	sasusers	final2.sas7bdat	24510464	2019-12-01T15:58:20-05:00
-rwxr-xr-x	cas	cas	sales.sashdat	8664	2019-11-10T15:47:54-05:00
-rw-r--r--	user1	sasusers	test.csv	551	2019-11-18T13:23:23-05:00
-rw-r--r--	user1	sasusers	createds.sas	1560	2019-10-20T16:40:12-05:00
-rw-r--r--	user1	sasusers	means.sas	1092	2019-10-21T17:12:41-05:00
rw-r--r--	user1	sasusers	hash.txt	2200	2019-09-09T13:37:33-05:00
drwxr-xr-x	user1	sasusers	PDFs	132	2019-09-22T15:19:12-05:00
rw-r--r--	user2	sasusers	DataForCAS.xlsx	12787708	2019-09-18T12:00:29-05:00

Output 1: TABLE.FILEINFO Results

Example 7 lists the in-memory tables in the ProjectA caslib. This code is very useful to see which of the tables have a session scope and which have a global scope. Remember, session-scope tables are available only to the session in which they were created. If you want other applications or other users to have access to the table, it needs to have global scope. Output 2 shows an example of what the results might look like.

Example 7:

```
proc cas;
  tableinfo /
    caslib='projecta';
quit;
```

Results from table.tableinfo

Table Information for Caslib PROJECTA(user1)												
Table Name	Number of Rows	Number of Columns	Number of Indexed Columns	NLS encoding	Created	Last Modified	Promoted Table	Duplicated Rows	View	Source Name	Source Caslib	Compressed
VS2	8207	24	0	utf-8	2019-11-06T09:19:29-05:00	2019-11-06T09:19:29-05:00	Yes	No	No	VS.sashdat	PROJECTA(user1)	No

Output 2: TABLEINFO Results

The TABLE.COLUMNINFO action provides information about the contents of an in-memory table. It provides a column list much like PROC CONTENTS provides a variable list for SAS 9 data sets. Example 8 contains the syntax. Output 3 shows part of the results.

Example 8:

```
proc cas;
  columninfo /
    table={caslib="casuser" name='mydata'};
quit;
```

Results from table.columninfo

Column Information for MYDATA in Caslib CASUSER(user1)						
Column	Id	Type	Length	Formatted Length	Format Width	Format Decimal
color	1	char	8	8	0	0
X1	2	double	8	12	0	0
X2	3	double	8	12	0	0
X3	4	double	8	12	0	0
X4	5	double	8	12	0	0
X5	6	double	8	12	0	0

Output 3: COLUMNINFO Results (Partial)

The examples in this section generated printed results. However, you can use the RESULT= parameter in each action to route the information to a result dictionary that can then be used in further actions. RESULT= is shown in examples later in the paper. Be aware that each action accepts more parameters than shown here.

SAVING

In a SAS Viya environment, the term “saving” might have two different meanings. The first meaning is to save an in-memory table to a caslib as a data source. This creates a SASHDAT file that is stored on disk and can survive losing a session or having to restart the CAS server. The SASHDAT format was created specifically for the CAS server, and therefore loading the data from a SASHDAT file is faster than loading from other file types.

The TABLE.SAVE action, shown in Example 9, saves the in-memory table called Example3 to the SASHDAT file called Sales_final. Both the in-memory table and the SASHDAT file are in the ProjectA caslib.

Example 9:

```
proc cas;
  table.save /
    table={name="example3" caslib="projecta"}
    name="sales_final"
    caslib="projecta";
quit;
```

The second meaning is to save to external databases, CSV files, Excel files, and SAS 9 data sets, which the SAVE action also enables you to do. Example 10 uses the SAVE action to create a data set. The data set has a UTF-8 encoding, which is important to know if the data set is shared with other SAS 9 users. Again, the name of the in-memory table is Example3, currently in the ProjectA caslib. The data set is saved in the same caslib, with the name Sales_final. The VARS= parameter subsets the variables from the in-memory table that are included in the data set; only DATE, REVENUE, and PROFIT are kept.

Example 10:

```
proc cas;
  table.save /
    caslib='projecta'
    name='sales_final'
    exportoptions={filetype='basesas' }
    table={caslib='projecta' name='example3'
      vars={{name='date' } {name='revenue' } {name='profit'}}};
quit;
```

You can use the FILEINFO action from Example 6 to check that the data set is successfully saved.

Though it does not save permanent versions of your table, the SAVERESULT statement is also handy to know. You must use the SAVERESULT statement to save the dictionary from the RESULT= parameter into an in-memory table.

In the code below, Example 11, two dictionaries are created. The first holds the name of the in-memory table MyData (created in the "Example Data" section). The second dictionary holds the name of the caslib where MyData resides, the CASUSER caslib. The WHERE operator is used to subset to green records within MyData. The FETCH action places those records into a dictionary, which the SAVERESULT action saves to an in-memory table.

Example 11:

```
proc cas;
  mytbl.name = "mydata"; /*create a dictionary with the table name*/
  mytbl.caslib="casuser"; /*create a dictionary with the caslib name*/
  mytbl.where = "'green' = color"; /*subset the data*/
  table.fetch result=r /table=mytbl;
  saveresult r caslib=casuser casout='green' replace;
run;
quit;
```

DROPPING

Though CAS is designed to hold a lot of information, in-memory space is not infinite. When free memory is extremely low the server writes in-memory blocks to the cache. This prevents the server from running out of memory, but the cache space is not infinite either.

Global-scope in-memory tables remain in memory until they are dropped. Session-scope tables are dropped from memory when the session ends. It is highly recommended that you drop any in-memory tables, either global or session scope, when they are no longer needed. This prevents tables from unnecessarily consuming in-memory space.

The TABLE.DROPTABLE action drops in-memory tables. There is no “undo” so be sure you are ready to drop a table before you do so. Example 12 demonstrates dropping a specific table from the ProjectA caslib. The QUIET=TRUE parameter prevents an error from being written to the log if the table does not exist in-memory. Remember, in-memory tables are the ones used for all your analytics work.

Example 12:

```
proc cas;
  table.droptable /
    caslib='projecta'
    name='example5'
    quiet=true;
quit;
```

CAS enables you to use the same name for session and global tables, but this is not recommended. A session-scope table is very good for temporary tables or ones that you need to make changes to and then promote. Name session-scope tables something that you would not use for a global table. A global-scope table is best for tables that multiple users need access to and tables that are accessed through multiple SAS Viya applications, such as SAS Studio and SAS Visual Analytics.

When coding, always check for a global-scope table and drop it before trying to promote a new version. CAS cannot promote a table and replace it at the same time. The table must be dropped before it can be promoted.

The TABLE.TABLEEXISTS action checks for the existence of a table and returns an integer value indicating whether the table has a session scope or a global scope. TABLEEXISTS finds a session-scope table first. A return value of 1 indicates a session-scope table, and a 2 indicates a global-scope table. The DROPTABLE action always looks for and finds a session-scope table first as well. Therefore, if you suspect that you have the same-named table with both scopes, you can check for the existence of both and then drop that version if it is found. This technique is demonstrated in Example 13.

Example 13:

```
proc cas;
  /*check for session scope table*/
  table.tableexists result = r1 /
    caslib="projecta"
    name="final";
  /*drop session scope table*/
  if r1.exists = 1 then do;
    action table.droptable /
      caslib="projecta"
      name="final";
  end;

  /*check for global scope table*/
```

```

table.tableexists result = r2 /
  caslib="projecta"
  name="final";
/*drop global scope table*/
if r2.exists = 2 then do;
  action table.droptable /
    caslib="projecta"
    name="final";
end;
quit;

```

LOOPING

The previous section provided code for housekeeping tasks like loading and saving data. The examples concentrated on one table at a time. Those same tasks might need to be accomplished for all tables or all caslibs. Code that loops through all entities is the next step in automating your code.

Looping with CASL works much the same way as it does in SAS 9 and contains the same statements: DO, DO OVER, DO UNTIL, and DO WHILE. With these statements, you need to know where to start, where to end, and how to reference the objects that you are trying to manipulate.

To understand the examples in this section, you first need to understand a little bit about dictionaries and result tables. Results from actions are represented as dictionaries. If you use the RESULT= parameter in an action statement, you are saving the dictionary as a variable that you can perform further processing on. Depending on the action, the dictionary that is created contains arrays, directories, result tables, or other dictionaries.

The examples in this section use actions that create result tables inside of the result dictionary. The DESCRIBE statement provides information in the log about the structure of the result dictionary, including the names of any result tables. It is a very useful learning and debugging tool, no matter whether you are just getting started with CASL or have been using it a while. The syntax is DESCRIBE and the name of the variable:

```

proc cas;
  <action statement> result=ti;
  describe ti;
quit;

```

The syntax for referencing a result table inside of a dictionary is the name of the dictionary, a dot, and the name of the table. For example, if the name of the variable containing the dictionary is TI and the name of the result table is TABLEINFO, you use TI.TABLEINFO to access the table.

Drop All Tables in a Single Caslib

The goal of Example 14 is to drop all tables in a specific caslib, CASUSER. The TABLE.TABLEINFO action provides information about each table in a caslib. The result table contains one row for each in-memory table in the caslib, with a column called NAME that contains the name of an in-memory table.

The DO OVER statement iterates over every row in the result table (TI.TABLEINFO).

The TABLE.DROPTABLE action drops in-memory tables. The key part of this statement is the NAME= parameter, which contains the name of the table to drop. This parameter takes a string, but it also accepts a reference to the in-memory table. The first part of the reference, i, is the name of the iterator from the DO OVER statement. The second part of the reference, NAME, is the column from the result table that contains the name of the in-memory table.

Example 14:

```
proc cas;
  table.tableinfo result=ti /
  caslib='casuser';
  do i over ti.tableinfo;
    table.droptable /
      caslib="casuser"
      name=i.name;
  end;
quit;
```

Please note that the iterator of *i* is used because SAS 9 programmers often use *i* with loops. However, that is a variable like any other and can be given any name that is not a keyword.

You might be interested in the number of tables that are in a specific caslib. Using the EXISTS and DIM functions, you can place the number of tables in a variable that can be used as your stopping value, demonstrated in Example 15.

Example 15:

```
proc cas;
  tableinfo result=r /caslib='casuser';
  if exists(r, 'tableinfo') then numtable = dim(r.tableinfo);
  else numtable = 0;
quit;
```

Drop All Tables in All Caslibs

You can take the previous example further and drop all in-memory tables from all caslibs. This task requires nested loops. You need one loop for determining the name of all caslibs and iterating over those names. A second loop is needed for iterating over the tables within each caslib.

The code in Example 16 uses the CASLIBINFO action to obtain a dictionary containing the names of all caslibs. The first loop iterates over the variable *C*, which contains the result dictionary. Nested within the first loop is the code from Example 14 that loops over the names of the in-memory tables in a given caslib.

The second loop begins with an IF condition checking the existence of the tableinfo results table. It is possible that not all caslibs contain in-memory tables. The TABLEINFO action creates a dictionary with zero entries when this happens. To avoid warnings in the log, check for the existence of the results table before trying to iterate over it.

Example 16:

```
proc cas;
  table.caslibinfo result=c;
  do j over c.caslibinfo;
    table.tableinfo result=ti /
      caslib=j.name;
    if exists(ti, 'tableInfo') then do;
      do i over ti.tableinfo;
        table.droptable /
          caslib=j.name
          name=i.name;
      end;
    end;
  end;
quit;
```

Note: The above code should be used with caution. You might delete more tables than you intend, depending on your permission level. Conditional statements can be added to the code to prevent deletion from specific caslibs.

SUMMARIZING

Summarization is another common task. Means, sums, and percentiles are common statistics used to describe data. Just as with SAS 9, CASL has multiple ways of summarizing data. In the case of CASL, this means multiple actions have summarizing functionality.

This section looks at three actions: SIMPLE.SUMMARY, AGGREGATION.AGGREGATE, and DATAPREPROCESS.RUSTATS. Each of these actions generates summary statistics. Though there might be more actions that generate the same statistics, these three are a good place to start as you learn CASL.

SIMPLE.SUMMARY

The purpose of the Simple Analytics action set is to perform basic analytical functions. One of the actions in the action set is the SUMMARY action, used for generating descriptive statistics like the minimum, maximum, mean, and sum.

This example, Example 17, demonstrates obtaining the sum, mean, and n statistics for five variables (x1–x5) and grouping the results by COLOR, using the MyData table created previously. The numeric input variables are specified in the INPUTS= parameter. The desired statistics are specified in the SUBSET= parameter.

Example 17:

```
proc cas;
  simple.summary /
    inputs={"x1" "x2" "x3" "x4" "x5"}
    subset={"sum" "mean" "n"}
    table={caslib="casuser" name="mydata" groupBy={"color"}}
    casout={caslib="casuser" name="mydata_summary" replace=true};
run;
  table.fetch /
    table={caslib="casuser" name="mydata_summary"};
run;
quit;
```

The SUMMARY action creates a table that is named Mydata_summary. The TABLE.FETCH action is included to show the contents of the table.

The Mydata_summary table can be used as input for other actions, its variable names can be changed, or it can be transposed. Now that you have the summary statistics, you can use them however you need to.

AGGREGATION.AGGREGATE

Many SAS procedures have been CAS-enabled, which means you can use a CAS table as input. However, specifying a CAS table does not mean that all processing takes place on the CAS server. Not every statement, option, or statistic is supported on the CAS server for every procedure, which is one reason to develop code that uses CAS actions instead of procedures. You need to be aware of what is not supported so that you do not run into issues if you choose to use a CAS-enabled procedure. In the documentation for each procedure, refer to the CAS processing section to find the relevant details.

When a procedure is CAS-enabled, it means that, behind the scenes, it is submitting an action. The MEANS and SUMMARY procedures submit the AGGREGATION.AGGREGATE action.

With PROC MEANS it is common to use a BY or CLASS statement and ask for multiple statistics for each analysis variable, even different statistics for different variables. Here is an example:

```
proc means sum data=myuser.mydata noprint;
  by color;
  var x1 x2 x3;
  output out=myuser.test(drop=_type_ _freq_) sum(x1 x3)=x1_sum x3_sum
    max(x2)=x2_max std(x3)=x3_std;
run;
```

The AGGREGATE action produces the same statistics and the same structured output table as PROC MEANS as shown by Example 18.

Example 18:

```
proc cas;
  aggregation.aggregate /
    table={name="mydata" caslib="casuser" groupby={"color"}}
    casout={name="mydata_aggregate" caslib='casuser' replace=true}
    varspecs={{name='x1' summarysubset='sum' columnnames={'x1_sum'}}
              {name='x2' agg='max' columnnames={'x2_max'}}
              {name='x3' summarysubset={'sum', 'std'}
                columnnames={'x3_sum' 'x3_std'}}}}
    savegroupbyraw=true
    savegroupbyformat=false
    raw=true;
quit;
```

The VARSPECS= parameter might be confusing. It is where you specify the variables that you want to generate statistics for, which statistics to generate, and what the resulting column should be called. Check the documentation: depending on the desired statistic, you need to use either SUMMARYSUBSET or AGG arguments.

If you are using the GROUPBY= parameter, you most likely want to use the SAVEGROUPBYRAW=TRUE parameter. Otherwise, you must list every grouping variable in the VARSPECS= parameter. Also, the SAVEGROUPBYFORMAT=FALSE parameter prevents the output from containing _f versions (formatted versions) of all of the grouping variables.

DATAPREPROCESS.RUSTATS

The RUSTATS action, in the Data Preprocess action set, computes univariate statistics, centralized moments, quantiles, and frequency distribution statistics. This action is extremely useful when you need to calculate percentiles. If you ask for percentiles from a procedure, all the data is moved to the compute server and processed there, not on the CAS server.

Example 19 has an extra step. Actions require a list of variables, which can be cumbersome when you want to generate summary statistics for more than a handful of variables. Macro variables are a handy way to insert a list of strings (variable names in this case) without having to enter all the names yourself. The SQL procedure step generates a macro variable containing the names of all of the numeric variables. The macro variable is referenced in the INPUTS parameter.

The RUSTATS action has TABLE= and INPUTS= parameters like the previous actions. The REQUESTPACKAGES= parameter is the parameter that allows for a request for percentiles.

The example also contains a bonus action, TRANSPOSE. The goal is to have a final table, Mydata_rustats2, with a structure like PROC MEANS would generate. The tricky part is the COMPUTEDVARSPROGRAM= parameter.

The table generated by the RUSTATS action has a column called `_Statistic_` that contains **the name of the statistic. However, it contains "Percentile" multiple times. A different** variable, `_Arg1_`, contains the value of the percentiles (1, 10, 20, and so on). The values of `_Statistic_` and `_Arg1_` need to be combined, and that new combined value generates the new variable names in the final table.

The COMPUTEDVARS= parameter specifies the name of the new variable that holds the concatenation of `_Statistic_` and `_Arg1_`. The COMPUTEDVARSPROGRAM= parameter tells CAS how to create the values for NewID. The NewID value is then used in the ID= parameter to make the new variable names.

Example 19:

```
proc sql noprint;
  select quote(strip(name)) into: numvars separated by ','
  from dictionary.columns
  where libname='MYUSER' and memname='MYDATA' and type='num';
quit;

proc cas;
  dataPreprocess.rustats /
    table={name="mydata" caslib="casuser"}
    inputs={&numvars}
    requestpackages={{percentiles={1 10 20 30 40 50 60 70 80 90 95 99}
      scales={"std"}}}
    casoutstats={name="mydata_rustats" caslib="casuser"};
run;

transpose.transpose /
  table={caslib='casuser' name="mydata_rustats" groupby={"_variable_"}
  computedvars={{name="newid" format="$20."}}
  computedvarsprogram="newid=strip(_statistic_) ||
    compress(strip(_arg1_),'.-');"}
  transpose={"_value_"}
  id={"newid"}
  casout={caslib='casuser', name="mydata_rustats2", replace=true};
run;
quit;
```

ORDERING

In SAS 9 the order of observations in a data set is integral to how processing works because observations are processed sequentially. In addition, any use of BY statements requires data sets to be sorted in a specific order.

The CAS server is a distributed computing environment. Distributed computing environments do not have a concept of ordering. The order of rows in a CAS table is not predictable. That is not a bad thing! CAS returns rows as the processing completes, and that is what helps it run so quickly. Also, CAS groups or orders, where necessary, on the fly, meaning you do not need the extra step of sorting the table.

Various actions accept either the SORTBY= or GROUBY= parameter, or both. Example 20 demonstrates using the SORTBY= parameter with the FETCH action to return results from the MyData table in the order of COLOR and SHAPE.

Example 20:

```
proc cas;
  table.fetch /
    table={caslib="casuser" name="mydata"}
    sortby={{name="color" order="descending"}
            {name="shape" order="descending"}};
quit;
```

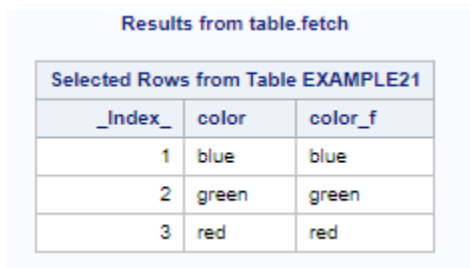
If the order of rows is important for the task that you are trying to accomplish, that task must be run on a single node or processed by the SAS run-time environment. However, once the data is pushed back to multiple nodes, the distributed environment, you lose the order enforced on the single node.

Nevertheless, there are a couple of actions that perform similar tasks to PROC SORT. The GROUPBY action creates a table with unique values of a variable or the unique combinations of multiple variables.

Example 21 creates a table that contains the unique color values from the Mydata table. The INPUTS parameter takes the variable(s) that you want the unique values for. A FETCH action is included to see the results of the GROUBY action, shown in Output 4.

Example 21:

```
proc cas;
  simple.groupby result=r /
    inputs={"color"}
    table={name="mydata" caslib='casuser'}
    casout={name="example21" caslib='casuser'};
run;
fetch / table={caslib="casuser" name="example21"};
run;
quit;
```



The screenshot shows a SAS output window titled "Results from table.fetch". Inside the window, there is a sub-header "Selected Rows from Table EXAMPLE21" above a table with three columns: "_Index_", "color", and "color_f". The table contains three rows of data.

Index	color	color_f
1	blue	blue
2	green	green
3	red	red

Output 4: GROUPBY Results

Beginning in SAS Viya 3.5 there is a DEDUPLICATE action that eliminates rows that have duplicate or unique group-by variable values. Example 22 generates a table that has one row for each combination of the color and shape variables. Also, the VARS= parameter lists only the columns that should be present in the resulting table, shown in Output 5.

Example 22:

```
proc cas;
  deduplication.deduplicate/
    table={name="mydata" caslib='casuser' groupby={"color" "shape"}
    vars={{name="color"} {name="shape"}}}
    casout={name="example22" caslib='casuser'}
    noduplicatekeys=true;
run;
```

```

fetch / table={caslib="casuser" name="example22"};
run;
quit;

```

Results from table.fetch

Selected Rows from Table EXAMPLE22		
Index	color	shape
1	green	circle
2	red	diamon
3	blue	circle
4	red	square
5	red	circle
6	green	diamon
7	blue	diamon
8	green	square
9	blue	square

Output 5: DEDUPLICATE Results

If you look at the Example22 table, you might notice that, though it contains the unique combinations, the rows are not in the order PROC SORT would have given you. Again, this is because the CAS server does not have the same concept of order as SAS 9. You can print them in the desired order by using the syntax from Example 20.

CONCLUSION

Just like your SAS 9 programming, you perform some of the same tasks in every program with CASL. Using CASL ensures the steps are run on the CAS server and give you more power and flexibility than CAS-enabled procedures.

The examples in this paper covered a range of tasks, like managing your in-memory tables, looping, summarizing, and ordering data. You can take these examples and build your own set of CASL common tasks.

REFERENCES

SAS Institute Inc. 2019. SAS® Viya® 3.5 Actions and Action Sets by Name and Product. Cary, NC: SAS Institute Inc. Available at https://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=allprodsactions&docsetTarget=titlepage.htm.

SAS Institute Inc. 2019. *Getting Started with CASL Programming 3.5*. Cary, NC: SAS Institute Inc. Available at https://documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.5&docsetId=casl&docsetTarget=titlepage.htm.

Sober, Steven. 2019. "What is CASL?" In *Proceedings of the SAS Global Forum 2019 Conference*. Cary, NC: SAS Institute Inc. Available at <https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3040-2019.pdf>.

Pendergrass, Jerry. 2019. "Got Results? Let CASL Help Turn Them into Superior Reports." In *Proceedings of the SAS Global Forum 2019 Conference*. Cary, NC: SAS Institute Inc. Available at <https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3177-2019.pdf>.

Eslinger, Jane. 2019. "Summarization in CASL." Available at <https://blogs.sas.com/content/sgf/2019/08/08/summarization-in-casl/>.

ACKNOWLEDGMENTS

The author is immensely grateful to Marcia Surratt and Brian Kinnebrew for contributing to this paper.

RECOMMENDED READING

- SAS blog posts that have the tag "CASL." Available at <https://blogs.sas.com/content/tag/casl/>.

CONTACT INFORMATION

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

Jane Eslinger
SAS Institute Inc.
919.531.2926
Jane.Eslinger@sas.com
www.sas.com

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.

Other brand and product names are trademarks of their respective companies.