SAS® Cloud Analytic Services 3.1: Accessing and Manipulating Data

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® Cloud Analytic Services 3.1: Accessing and Manipulating Data*. Cary, NC: SAS Institute Inc.

SAS® Cloud Analytic Services 3.1: Accessing and Manipulating Data

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

September 2016

3.1-P1:casdataam

Contents

Chapter 1

Accessing Data

Introduction

SAS Viya analytical procedures and some DATA step features use input data from in-memory tables on SAS Cloud Analytic Services only. This document takes you through common tasks for loading and accessing your input data with SAS Cloud Analytic Services. The methods that these tasks use for loading and accessing are:

- caslibs
- CASUTIL procedure
- DATA step

You can use SAS/CONNECT to transfer remote data sources directly into in-memory tables. However, SAS/CONNECT is a separately licensed product. For more information about SAS/CONNECT, see *SAS/CONNECT for SAS Viya: User's Guide*.

Terms to Be Familiar With

Before we begin, here are a few important terms:

active caslib
: your session must have a default location for server-side data access. This is the active caslib. The term "active caslib" is used rather than default caslib because the caslib that your session uses is modified as caslibs are added and dropped.

caslib
: the mechanism for accessing data with SAS Cloud Analytic Services. At it's simplest, a caslib provides access to files in a data source, such as a database or file system directory, and to in-memory tables.

data connector
: a data connector is the software that is used with a caslib to read server-based data sources like databases and Hive. There are a few data connectors for file-based caslibs. These data connectors are used to control reading data files such as setting the file encoding.

file
: the source data that is in a caslib's data source. For a caslib that uses a path-based data source, this is natural. For a caslib that uses a database as a data source, the tables in the database are referred to as files.

session
: when you initially connect to SAS Cloud Analytic Services, your session is started on the server. Data access and communication is performed through the session. Your programs communicate with the session to request actions. Many sessions can operate concurrently, actions execute serially within a session. In most cases, programmers start and use one session only.

table
: is used to refer to in-memory data. After a file (using the preceding definition) is loaded into the server, it is referred to as a table.

Common Tasks for Accessing and Manipulating Data

Table 1.1 *Common Tasks for Accessing Data*

Task	Sample Syntax

Load a SAS data set.
Tip: This is a good learning step if you are familiar with SAS and want to learn how SAS libraries, data sets, and SAS Cloud Analytic Services work together.
	`proc casutil;`
	`  load data=libref.member-name`
	`  casout="table-name";`
	`run;`

Load a client-side data file.	`proc casutil;`
	`  load file="/path/to/file.suffix"`
	`  casout="table-name";`
	`run;`

List caslibs. This shows you the server-side data sources that SAS Cloud Analytic Services can access.	`caslib _all_ list;`
Add a file-based caslib. *Tip:* Remember that the specified PATH= must be accessible from the host for the SAS Cloud Analytic Services controller.	`caslib data datasource=(srctype="path")` `  path="/data01";`

Determine the data files in a caslib that the server can access.	`proc casutil;`
	`  list files incaslib="name";`
	`run;`

Load a server-side data file.	`proc casutil;`
	`  load casdata="file-name.suffix"`
	`  casout="table-name";`
	`run;`

Load a table from database.	`proc casutil;`
	`  load casdata="database-table-name"`
	`  casout="table-name";`
	`run;`

List in-memory tables.	`proc casutil;`
	`  list tables incaslib="name";`
	`run;`

Assign a CAS engine libref and bind it to a caslib. This is how you access tables with a SAS procedure or the DATA step.	`libname mycas cas caslib="name";`

Load a Client-Side File

Prerequisites

The following example assumes that you have a Microsoft Excel file. The sample code assumes that a file named **/data/WorldData.xlsx** is available in the **Server Files and Folders** section of SAS Studio.

Example

This example shows how to load data from a SAS library and how to load data from a Microsoft Excel file. This approach is appropriate for smaller tables and ad hoc data analysis.

```
caslib hps datasource=(srctype="hdfs")                                  /* 1 */
       path="/hps";

proc casutil incaslib="hps" outcaslib="hps";                            /* 2 */

   load data=sashelp.iris promote;                                      /* 3 */

   contents casdata="iris";

   load file="/data/WorldData.xslx" casout="worlddata";                 /* 4 */

   contents casdata="worlddata";
run;
```

1. Add a caslib to access the **/hps** directory in HDFS. By default, adding a caslib sets it as the active caslib.
2. The CASUTIL procedure statement includes the INCASLIB= and OUTCASLIB= options. This is a best practice to ensure that tables are read from the caslib that you expect and are saved to the caslib that you expect.
3. The DATA= argument indicates that the table is transferred from the SAS client host to SAS Cloud Analytic Services. Replace the Sashelp.Iris value with a libref and table that you want to use. The PROMOTE option makes the Iris table a global-scope table and available to other sessions that use the Hps caslib. Use the CONTENTS statement to ensure that the table includes the column names and data types that you expect.
4. The FILE= argument indicates that the file is a client-side file that is accessible to SAS and not to SAS Cloud Analytic Services. The file, WorldData.xlsx, is transferred to the server and then imported with a table name of Worlddata. This LOAD statement does not include the PROMOTE option, so the in-memory Worlddata table can be accessed only from the same session.

Key Ideas

- The LOAD DATA= and LOAD FILE= statements in the CASUTIL procedure are used for accessing client-side data.
- The CONTENTS statement is used to display information such as column names and data types.
- By default, when you add a caslib, that caslib becomes the active caslib. Use the NOTACTIVE option to add a caslib without making it active.

See Also

- "CASUTIL" in *SAS Cloud Analytic Services: Language Reference*
- "CASLIB Statement" in *SAS Cloud Analytic Services: Language Reference*

Load a Server-Side File

Prerequisites

The following example assumes the following:

- You can create a small CSV file in the file system that is associated with the directory for your personal caslib.
- The Microsoft Excel file used in this example has historical consumer price index (CPI) data for food from http://www.ers.usda.gov/data-products/food-price-outlook.aspx.

Example

This example shows how to access two server-side files and load the data into CAS:

- a CSV file that describes the performance of a toy catapult. The first line of the file does not contain column names. The example shows how to specify names.
- a Microsoft Excel file.

For the CSV file, a description for the data is shown in the example. The values are as follows:

```
5,10,10,11,10,11,3
5.5,16,3,16,1,15,6
6,23,0,18,7,20,5
6.5,23,3,28,6,26,0
7,27,3,25,10,23,1
```

The program is as follows:

```
cas casauto sessopts=(caslib="casuser");                    /* 1 */
```

```
libname mycas cas caslib="casuser";                                    /* 2 */

/* first, load the data from the CSV file */
proc casutil incaslib="casuser" outcaslib="casuser";

  contents casdata="catapult.csv";                                     /* 3 */

  load casdata="catapult.csv" casout="catapultraw"
       importoptions=(filetype="csv"                                   /* 4 */
          encoding="latin1"
          getnames="false"
          vars=(
            (name="turns",     label="Number of turns",       type="double"),
            (name="first_ft",  label="Feet for first try",    type="double"),
            (name="first_in",  label="Inches for first try",  type="double"),
            (name="second_ft", label="Feet for second try",   type="double"),
            (name="second_in", label="Inches for second try", type="double"),
            (name="third_ft",  label="Feet for third try",    type="double"),
            (name="third_in",  label="Inches for third try",  type="double")
          )
      ) replace;

    save casdata="catapultraw" replace;                                /* 5 */
    contents casdata="catapultraw.sashdat";
quit;

data mycas.catapult (promote=yes) / sessref=casauto;                   /* 6 */
   set mycas.catapultraw;
   first  = 12 * first_ft  + first_in;
   second = 12 * second_ft + second_in;
   third  = 12 * third_ft  + third_in;
run;

proc casutil incaslib="casuser";
   contents casdata="catapult";                                        /* 7 */
   droptable casdata="catapultraw";                                    /* 8 */
quit;

/* simple scatter plot */
proc sgplot data=mycas.catapult;                                       /* 9 */
     scatter x=turns y=first;
     scatter x=turns y=second;
     scatter x=turns y=third;

     /*--X Axis--*/
     xaxis grid label="Number of turns";

     /*--Y Axis--*/
     yaxis grid label="Distance, in inches";;
run;

/* second, load the Excel file */
proc casutil incaslib="casuser" outcaslib="casuser";
   list files;                                                         /* 10 */
   contents casdata="historicalcpi.xls";
```

```
   load casdata="historicalcpi.xls" casout="historicalcpi"            /* 11 */
        label="Historical CPI data, 1974 through 2014; updated 3/3/2015";

   save casdata="historicalcpi" replace;
quit;
```

1 The SESSOPTS= option is used with the CASLIB= session option to ensure that the Casuser personal caslib is set as the active caslib.

2 The CAS engine LIBNAME statement assigns the Mycas libref and binds it to the Casuser caslib.

3 The CONTENTS statement shows the file information and column information for the CSV file. See Figure 1.1 on page 8.

4 The CASDATA= argument indicates that the file is read from the caslib's data source. The IMPORTOPTIONS= specify how to read the file.

5 The SAVE statement makes a copy of the imported data as a SASHDAT file. This is part of the data life cycle. If the file is imported correctly, then subsequent analyses of the data can begin from the SASHDAT file. The CONTENTS statement shows that the column names and labels are applied. See Figure 1.2 on page 8.

6 The DATA step is used to combine each set of foot and inch measures into a single column. The PROMOTE= option is used to make the table available to other sessions that you start. The SESSREF= option is used to ensure that the DATA step runs in CAS.

7 The last CONTENTS statement is used to display the table information, table details, and column information for the in-memory table. See Figure 1.3 on page 9.

8 The DROPTABLE statement is used to free the memory resources that are used for the data from the CSV file. The copy of the data that was made with the SAVE statement is not deleted, only the in-memory resources are freed.

9 The SGPLOT procedure uses the Mycas CAS engine libref. When Mycas was assigned at the start of the program, the CASLIB= option bound it to the Casuser caslib. This ensures that the libref always accesses tables in that caslib.

10 The LIST FILES statement is used to list the files in the caslib's data source. In this case, the personal caslib, Casuser, uses the OS file system. See Figure 1.4 on page 9.

11 The LOAD CASDATA= argument specifies the Historicalcpi.xls file. The LABEL= option is used to specify a description of the data.

Results

The following display shows the results of the CONTENTS statement. Notice that the anticipated column names match the first line of the CSV file. This is corrected in the subsequent LOAD CASDATA= statement when the GETNAMES= option is set to false.

Figure 1.1 *CONTENTS Statement Results for the CSV File*

The CASUTIL Procedure

File Information for catapult.csv in caslib CASUSER( ).					
Name	Permission	Owner	Group	File Size	Last Modified
catapult.csv	-rw-r--r--			0.1KB	01Feb2016:14:50:35

Column Information for catapult.csv in Caslib CASUSER( )			
Column	Type	Length	Formatted Length
5	double	8	12
10	double	8	12
10	double	8	12
11	double	8	12
10	double	8	12
11	double	8	12
3	double	8	12

The following display applies to the second CONTENTS statement, after the LOAD CASDATA= statement that include the IMPORTOPTIONS= settings. Notice that the column names and labels are applied.

Figure 1.2 *CONTENTS Statement Results for the SASHDAT File*

File Information for catapultraw.sashdat in caslib CASUSER( ).					
Name	Permission	Owner	Group	File Size	Last Modified
catapultraw.sashdat	-rwxr-xr-x			24.2KB	01Feb2016:19:27:28

Column Information for catapultraw.sashdat in Caslib CASUSER( )				
Column	Label	Type	Length	Formatted Length
turns	Number of turns	double	8	12
first_ft	Feet for first try	double	8	12
first_in	Inches for first try	double	8	12
second_ft	Feet for second try	double	8	12
second_in	Inches for second try	double	8	12
third_ft	Feet for third try	double	8	12
third_in	Inches for third try	double	8	12

Figure 1.3 *CONTENTS Statement Results for the In-Memory Table*

Table Information for Caslib CASUSER( )										
Table Name	Number of Rows	Number of Columns	NLS encoding	Created	Last Modified	Promoted Table	Duplicated Rows	View	Source Name	Compressed
CATAPULT	5	10	utf-8	01Feb2016:19:28:25	01Feb2016:19:27:56	Yes	No	No	_T_1AAD1589_7FF56342BBA8	No

Detail Information for catapult in Caslib CASUSER( ).											
Node	Number of Blocks	Active Blocks	Rows	Fixed Data size	Variable Data size	Blocks Mapped	Memory Mapped	Blocks Unmapped	Memory Unmapped	Blocks Allocated	Memory Allocated
ALL	5	5	5	400	0	0	0	5	400	0	0

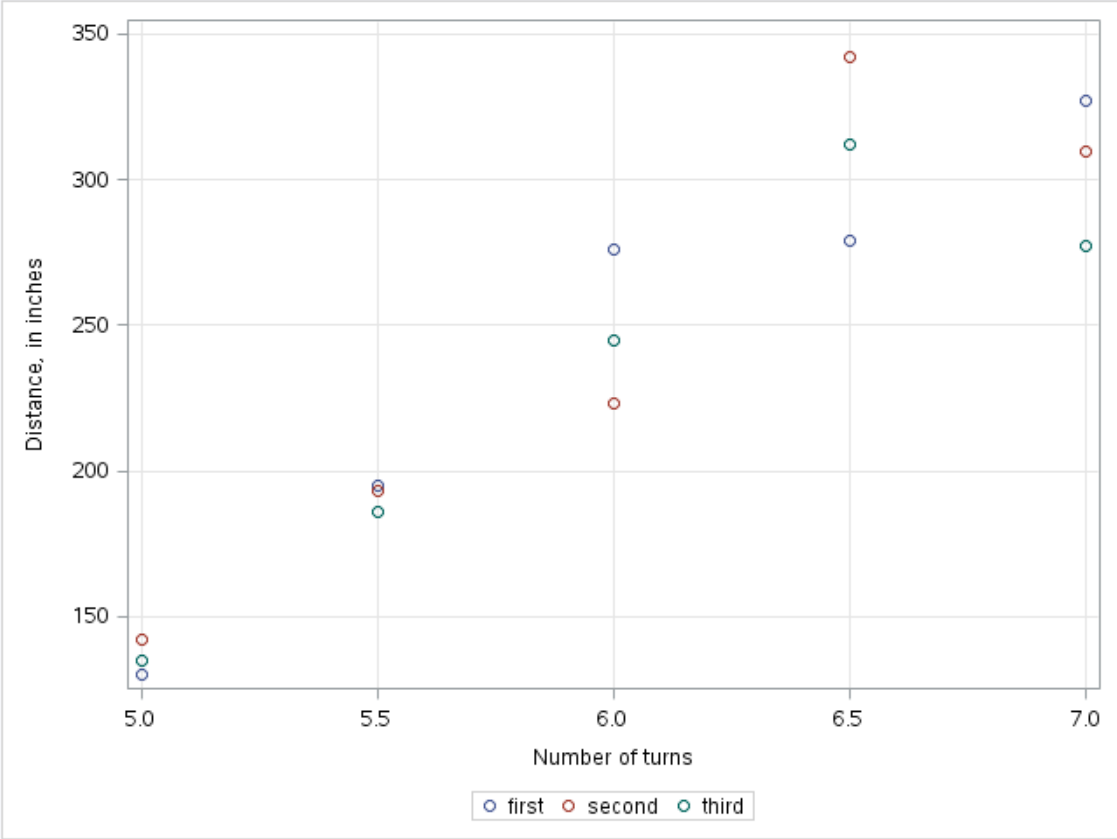
Column Information for CATAPULT in Caslib CASUSER( )				
Column	Label	Type	Length	Formatted Length
turns	Number of turns	double	8	12
first_ft	Feet for first try	double	8	12
first_in	Inches for first try	double	8	12
second_ft	Feet for second try	double	8	12
second_in	Inches for second try	double	8	12
third_ft	Feet for third try	double	8	12
third_in	Inches for third try	double	8	12
first		double	8	12
second		double	8	12
third		double	8	12

Figure 1.4 *LIST FILES Statement Results for a Path-Based Caslib*

The CASUTIL Procedure

File Information for root of caslib CASUSER( ).					
Name	Permission	Owner	Group	File Size	Last Modified
catapult.csv	-rw-r--r--			0.1KB	01Feb2016:14:50:35
historicalcpi.xls	-rwxr-xr-x			56.5KB	02Feb2016:19:08:19
mycas.distinct.sashdat	-rwxr-xr-x			7.2KB	12Nov2015:16:48:12
cars_large_part.sashdat	-rwxr-xr-x			20.4MB	26Jan2016:12:44:45

Figure 1.5 *Scatter Plot Results for the Catapult Table*



The following display shows the results of the CONTENTS statement for the Historicalcpi.xls file. By default, the column names are read from a file.

Figure 1.6 *CONTENTS Statement Results for the XLS File*

The CASUTIL Procedure

File Information for historicalcpi.xls in caslib CASUSER( ).					
Name	**Permission**	**Owner**	**Group**	**File Size**	**Last Modified**
historicalcpi.xls	-rwxr-xr-x			56.5KB	02Feb2016:19:08:19

Column Information for historicalcpi.xls in Caslib CASUSER( )				
Column	**Type**	**Length**	**Formatted Length**	**Format**
Consumer price indexes historical data, 1974 through 2014	varchar	0	0	$
B	varchar	0	0	$
C	varchar	0	0	$
D	varchar	0	0	$
E	varchar	0	0	$
F				
			0	$
	varchar	0	0	$
AO	varchar	0	0	$
AP	varchar	0	0	$
AQ	varchar	0	0	$
AR	varchar	0	0	$
AS	varchar	0	0	$
AT	varchar	0	0	$

Key Ideas

- Reading files from a caslib's data source is the most efficient way to access data. One key to recognizing that data is read from a caslib's data source is the presence of the CASDATA= argument.
- The CONTENTS statement in the CASUTIL procedure can display information for files, tables, and columns. To view information about a file, specify the filename, including the suffix in the CASDATA= option. After the table is loaded into memory, you drop the suffix or use the table name that you specified in the CASOUT= option.
- You can specify IMPORTOPTIONS= to describe how to load the data. For delimited files, the common options are to specify the file encoding and whether to get column names from the first line of the file.
- After the data for a file is imported, save a copy as a SASHDAT file.

See Also

- "CASUTIL" in *SAS Cloud Analytic Services: Language Reference*
- "CASLIB Statement" in *SAS Cloud Analytic Services: Language Reference*
- Chapter 2, "DATA Step Programming," on page 19

Load a Database Table

Prerequisites

The following example assumes the following:

- You are granted access to data in the data source.
- You know the connection information such as host, port, and so on.
- Your SAS Cloud Analytic Services installation is licensed and configured to use the client software for the data source vendor that you want to access. For installation-time configuration information, see *SAS Viya: Deployment Guide*.

Example

Loading tables into the server from a caslib's data source is the most efficient way to load data. In this example, a table is read from Oracle and the in-memory table is kept in the same caslib.

```
caslib oralib datasource=(                                          /* 1 */
    srctype="oracle",
    uid="DBUSER",
    pwd="secret",
    path="//dbserver.example.com:1521/dbname",
    schema="DBUSER"
    );


proc casutil;
   list files;                                                      /* 2 */

   droptable casdata="sales" quiet;                                 /* 3 */
   contents casdata="sales";                                        /* 4 */

   load casdata="sales" casout="sales" promote                      /* 5 */
       label="Fact table for User-to-Item Analysis"
       varlist=(
            (name="USERID"  label="User ID"),
            (name="ITEMID"  label="Item ID")
       );

   contents casdata="sales";                                        /* 6 */
quit;
```

```
run;
```

1 Add a caslib that uses Oracle as the data source. Oralib becomes the active caslib for the session and the subsequent programming statements use it for input and output.

2 The LIST FILES statement displays the tables that are available in the Oracle database.

3 The DROPTABLE statement includes the QUIET option. Running this statement is useful on repeated runs because it ensures that no table named Sales can be in-memory to interfere with the subsequent LOAD CASDATA= statement.

4 Because the first CONTENTS statement follows the DROPTABLE statement, this ensures that the table information and column information from Oracle are read.

5 The CASDATA= argument in the LOAD statement indicates that the Sales table is read from the caslib's data source (Oracle) into SAS Cloud Analytic Services. Options are specified to add labels to the table and columns.

6 Because the last CONTENTS statement follows the LOAD statement, table information and column information is displayed for the in-memory copy of the Sales table that was read from Oracle.

Key Ideas

- The CASLIB statement adds a server-side data source to SAS Cloud Analytic Services.
- In this example, the active caslib is Oralib. Remember that when you add a caslib, by default, it becomes the active caslib.
- For information about data source connection parameters, see "Data Connectors" in *SAS Cloud Analytic Services: Language Reference*.

Save an In-Memory Table

Example

This example demonstrates the following:

- saving a table to caslib named Hps that uses HDFS as a data source. The table is saved as a SASHDAT file.
- saving a table from a caslib that uses Oracle to a caslib named Hps that uses HDFS as a data source. The table is saved as a SASHDAT file.

```
proc casutil incaslib="hps" outcaslib="hps";
   load casdata="customers.sashdat" casout="customers";
run;

/* From some other application, or a DATA step, the */
/* Customers table is modified with a change that   */
/* is important to save.                            */
proc casutil  incaslib="hps" outcaslib="hps";
   save casdata="customers" replace;                                /* 1 */
```

```
run;

caslib orsales datasource=(
   srctype="oracle"
   username="DBUSER"
   password="secret"
   path="//dbserver.example.com:1521/sales"
   schema="DBUSER"
   );

proc casutil incaslib="orsales";
   load casdata="sales" casout="sales";                          /* 2 */

   save casdata="sales" outcaslib="hps";
run;
```

1 The SAVE statement uses the Hps caslib from the OUTCASLIB= procedure statement option. The next time a LOAD statement is used with the Customers table, the table includes the changes. You can use the CASOUT= option to specify an alternative filename other than the default, customers.sashdat.

2 The LOAD statement reads the table named Sales from the Orsales caslib that uses Oracle Database as a data source. To save a copy of the table in HDFS, the SAVE statement uses the OUTCASLIB= option to specify the Hps caslib that uses HDFS as a data source.

Key Ideas

- You can save in-memory tables as SASHDAT files or CSV files in path-based caslibs only. If you need to save a table from a caslib that is not path-based, then you can specify the OUTCASLIB= option.
- For information about data source connection parameters to use in the CASLIB statement, see "Data Connectors" in *SAS Cloud Analytic Services: Language Reference*.

Drop an In-Memory Table

Example

When you drop an in-memory table, only the in-memory table is affected. If the table was loaded from a caslib's data source, the table in the data source is unaffected.

```
proc casutil incaslib="hps" outcaslib="hps";
   load casdata="sales.sashdat" casout="sales";                  /* 1 */
run;

proc casutil incaslib="hps";
   droptable casdata="sales";                                    /* 2 */
run
```

1 The LOAD statement loads a file named Sales.sashdat from the Hps caslib.
2 The DROPTABLE statement drops the in-memory table.

Key Ideas

- If you drop a session-scope table, then only the session that loaded the table is affected.
- If you drop a global-scope table, then the table might be accessed from multiple sessions. The table is dropped after any actions that access the table are complete.
- Be aware that dropping a global-scope table can affect other sessions if the actions that are run by other sessions expect the table to be in memory.

Delete a File from a Caslib's Data Source

Example

When you delete a file from a data source, it does not affect an in-memory copy. The term "file" refers to a file in a caslib with a path-based data source or a table in a caslib with a server-based data source.

The following example demonstrates deleting a file named Sales from the data source associated with the Hps caslib.

```
proc casutil;
   deletesource casdata="sales.sashdat" incaslib="hps";
run;
```

Key Ideas

- You can delete files from path-based caslibs only.
- Include the filename suffix in the CASDATA= argument.

Data Compression

Overview of Data Compression

SAS Cloud Analytic Services supports and performs all compression for in-memory tables. When you transfer a table to the server in a DATA step and specify the COMPRESS= data set option, rows are sent to the server as is and the server compresses them.

All data in a row, both character and numeric variables, are compressed. Every row in a table is compressed. The server does not support some rows in compressed form and

others as uncompressed. The server can report the uncompressed size of the table, the compressed size, and the compression ratio.

For matrices of computed doubles (those with many decimal places), compression might not reduce the storage requirements at all. For rows with many long character variables that consist mostly of blanks, the compression ratio can be very high. For rows with mixed variables, where most doubles do not have fractional parts and most character variables have a small amount of blank padding, the compression ratio is typically moderate. As with most cases of using compression, character variables tend to compress the most and the ratio depends on your data.

Compressed Tables and the DATA Step

This example shows how to use the COMPRESS= data set option for SAS Cloud Analytic Services.

Example Code 1.1 *Creating a Compressed Table with a DATA Step*

```
libname mycas cas host="cloud.example.com" port=5570;

data mycas.prdsale (compress=yes);
    set sashelp.prdsale;
run;
```

After the table is loaded into memory, you can access the compressed table with the mycas.prdsale table reference.

SAS Cloud Analytic Services supports the APPEND= data set option for compressed tables. This example shows how to add new (uncompressed) rows to the compressed table.

Example Code 1.2 *Appending Rows to a Compressed Table*

```
data mycas.prdsale(append=yes);
    somelib.newrows;
run;
```

Because the mycas.prdsale table is already compressed, the new rows are automatically compressed as they are appended to the table. Specifying COMPRESS= with APPEND= has no effect. If the table is compressed, the server compresses the new rows. If the table is not compressed, then the server does not compress the new rows even if you specify COMPRESS=YES. The compressed or uncompressed state of the table determines how rows are appended.

Partitioning and compression are supported together. This example creates a new in-memory table that is partitioned and compressed.

Example Code 1.3 *Creating a Partitioned and Compressed Table*

```
data mycas.iris (partition=(species) compress=yes);
    set sashelp.iris;
run;

data mycas.iris (append=yes);
    set somelib.moreirises;
run;
```

In the first DATA step statement, the Iris data set is loaded into memory on the server. The table is partitioned by the formatted values of the Species variable. The table is also compressed. In the second DATA step statement, the table is appended to with more

rows. Because the in-memory table is already partitioned and compressed, the new rows are automatically partitioned and compressed when they are appended.

Compressed Tables and the CAS Procedure

You can use the CAS procedure to load data in memory on distributed SAS Cloud Analytic Services.

```
proc cas add data=sashelp.prdsale port=19999 compress;
    performance host="my";
run;
```

This example uses the COMPRESS= data set option to read the prdsale data set from the sashelp library and compress it in-memory on SAS Cloud Analytic Services. Be aware that you must specify the COMPRESS= for each table that you want to load in compressed form. You cannot specify COMPRESS= with the CREATE= option when you start SAS Cloud Analytic Services and have it automatically compress all tables.

When you read SASHDAT tables into memory, compression depends on these factors for the resulting in-memory tables:

- whether a WHERE clause is used
- whether the SASHDAT table is compressed on disk

If you specify a WHERE clause and the COMPRESS= option, SAS Cloud Analytic Services evaluates the WHERE clause as it reads data from HDFS and compresses the rows that meet the WHERE clause criteria. The memory efficiencies of the SASHDAT table format are forfeited in this scenario because SAS Cloud Analytic Services had to apply the WHERE clause.

If you do not specify a WHERE clause, SAS Cloud Analytic Services ignores the COMPRESS= option and relies on whether the SASHDAT table is compressed. If the SASHDAT table is compressed, the in-memory representation of the table is also compressed. If the SASHDAT table is not compressed, then neither is the in-memory representation. SAS Cloud Analytic Services ignores the option so that it can keep the memory efficiencies of the SASHDAT table format: When a SASHDAT table is loaded in memory, the in-memory representation is identical to the on-disk representation.

Performance Considerations

Compression exchanges less memory use for more CPU use. It slows down any request that processes the data. An in-memory table consists of blocks of rows. When the server works with a compressed table, the blocks of rows must be uncompressed before the server can work with the variables. In some cases, a request can require five times longer to run with a compressed table rather than an uncompressed table.

For example, if you want to summarize two variables in a table that has 100 variables, all 100 columns must be uncompressed in order to locate the data for the two variables of interest. If you specify a WHERE clause, then the server must uncompress the data before the WHERE clause can be applied. Like the example where only two of 100 variables are used, if the WHERE clause is very restrictive, then there is a substantial performance penalty to filter out most of the rows.

Working with SASHDAT tables that are loaded from HDFS is the most memory-efficient way to use the server. Using compressed SASHDAT tables preserves the memory efficiencies, but still incurs the performance penalty of uncompressing the rows as the server operates on each row.

Interactions

Here are the interactions for compressed tables and SAS programs.

- You can use a compressed table in programs like any other table.
- You can define calculated columns for compressed tables with the TEMPNAMES= data set option.
- You can append to compressed tables with the APPEND= data set option. This is also supported for compressed tables that have partitioning. However, you cannot append to a compressed table that is partitioned and has an ORDERBY= data set option specification.
- You can specify a sort order for a compressed table. Applying the sort has a performance penalty.
- You can use the SET statement to append a compressed table to an uncompressed table. New rows are uncompressed before being added to the target table.
- You can use the UNCOMPRESS statement to create an uncompressed temporary table. The uncompressed table provides better performance.

Limitations

These limitations exist when you work with compressed tables.

- You cannot apply compression to views that are created with the SCHEMA statement.
- You cannot use compressed tables with the SCHEMA statement, even if you specify MODE=TABLE.
- You cannot append to compressed tables that have an ORDERBY= data set option specification within the partitions.
- The SAS Cloud Analytic Services and the CAS LIBNAME engine do not support appending to compressed tables when you use the ORDERBY= data set option. However, you can load the table in memory in uncompressed form with partitioning and ordering and then use the COMPRESS= data set option. This creates a compressed and partitioned temporary table in which the rows are ordered.

See Also

- "APPEND= Data Set Option" in *SAS Cloud Analytic Services: Language Reference*
- "COMPRESS= Data Set Option" in *SAS Cloud Analytic Services: Language Reference*
- "ORDERBY= Data Set Option" in *SAS Cloud Analytic Services: Language Reference*
- "TEMPNAMES= Data Set Option" in *SAS Cloud Analytic Services: Language Reference*

Chapter 2

DATA Step Programming

Common Tasks for Running the DATA Step

Run the DATA step in CAS	`libname mycas cas;`
	`data mycas.hello;`
	`  put 'Hello from ' _hostname_`
	`  'thread # '  _threadid_;`
	`  x=1;`
	`run;`

Run the DATA step on a table in CAS	`libname mycas cas;`
	`proc casutil;`
	`  load data=sashelp.cars;`
	`run;`
	`data mycas.cars2;`
	`  set mycas.cars;`
	`  KPL_City = 0.425 * MPG_City;`
	`  KPL_Highway = 0.425 * MPG_Highway;`
	`run;`

Create CHAR and VARCHAR variables	`libname mycas cas;`
	`data mycas.string;`
	`  length x varchar(30);`
	`  length y $10;`
	`  x = 'abc'; y = 'def';`
	`run;`

Add and promote a table	`data mycas.cars_large(copies=1 promote=yes);`
	`  set sashelp.cars;`
	`  do i = 1 to 200; output; end;`
	`run;`

Merge CAS tables	`libname mycas cas;`
	`data mycas.table3;`
	`  merge mycas.table1 mycas.table2;`
	`  by var1 var2;`
	`run;`

For a complete list of examples, see "Examples" on page 46.

DATA Step Basics

Running the DATA Step

The DATA Step in SAS Viya

The SAS DATA step can be used to manipulate and prepare data for analysis and predictive modeling. In SAS Viya, the DATA step runs in both a traditional SAS client session as well as in SAS Cloud Analytic Services (CAS). CAS provides the high-performance environment for running the DATA step in parallel and in multiple threads on distributed, in-memory data. For a basic overview of SAS DATA Step processing, see DATA Step Processing at https://support.sas.com/documentation/cdl/en/lrcon/68089/HTML/default/viewer.htm.

In SAS Viya, the DATA step runs in one of two environments:

- In a **SAS client session**, where it runs in a single thread using the SAS V9 engine.
- In a **CAS server session**, where it runs in a single thread or in multiple threads using the CAS engine.

For more information about these processing environments, see "DATA Step Processing Modes" on page 29.

DATA Step Processing in CAS

When the DATA step runs in CAS, it runs either in a single thread or in multiple threads:

- Single thread – the DATA step runs as one process on an entire, unpartitioned table. Running the DATA step in a single thread is not recommended for very large tables.
- Multiple threads – the DATA step runs in multiple threads, in parallel. Each thread operates on a different part of the data. The DATA step can run in multiple threads in a single-machine CAS server or in a distributed CAS server.

See "Controlling DATA Step Processing" on page 31 for information.

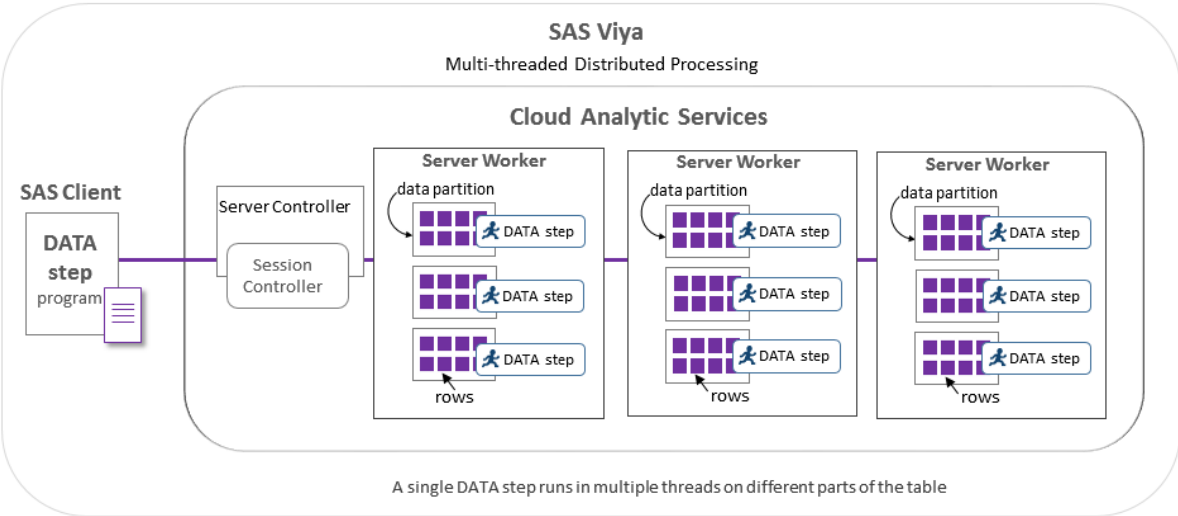
Multithreaded DATA Step Processing in CAS

- Distributed CAS server – a multi-node server environment in which CAS partitions and distributes data across multiple physical machines. The DATA step runs by default in multiple threads on every available node. When the DATA step runs in multiple threads in a distributed CAS server, CAS partitions and distributes the table

data across the cluster. Then, the DATA step runs in multiple threads on each node, allocating one DATA step thread per partition. Thus, each thread processes only a portion of the table. If your data is big, your big data is distributed and simultaneously processed. The DATA step can also run in a single thread in a distributed CAS server. If the DATA step runs in a single thread, all table data is processed in one thread on a single machine regardless of the number of available worker nodes. This approach is not recommended for big data.

- Single-Machine CAS server – a single computer is configured to run as the CAS server. CAS does not distribute table data. The DATA step can run either in a single thread or in multiple threads in a single-machine server. When you run the DATA step in multiple threads in a single-machine CAS server, the processing is performed on each CPU core so that threaded processing can take place.

Note: By default, the DATA step runs in multiple threads on all available nodes. Each thread operates on a different part of the data.



For information about CAS architecture and CAS Data, see "Architecture" in *SAS Cloud Analytic Services: Fundamentals* and "Data" in *SAS Cloud Analytic Services: Fundamentals*.

How to Run the DATA Step in CAS

To run the DATA step in CAS, you first specify the LIBNAME statement with the CAS option to create a CAS engine libref. Then, you specify the libref on the output and input data sets as follows:

```
libname mycas cas;
data mycas.bigcars;
   set mycas.cars(where=(weight>5500));
run;
```

The DATA step runs automatically in CAS as long as you use the CAS engine (libref) on the input and output table names. You can also use the DATA step as a data loader to load SAS data sets into CAS, as shown here:

```
libname mycas cas;
data mycas.cars(where=(weight>6000));
     set sashelp.cars;
     keep make model type weight MPG_City;
  run;
```

To see a more detailed example of running the DATA step in CAS, see Example: "DATA Step Program Walk-through" on page 86. For more information about other types of data you can access and load to CAS, see Chapter 1, "Accessing Data," on page 4.

Inter-row Dependencies and Multithreaded DATA Step Execution

A multithreaded DATA step can reduce run time and improve performance. But it can also pose challenges when you want to perform operations that have inter-row dependencies.

Functions or statements that rely on information from one row to another might return unexpected results when they are used in a multithreaded DATA step. For example, the RETAIN statement retains a value from one row to the next. In a multithreaded DATA step, the rows in a table are not processed together in the same DATA step thread, nor is their order guaranteed. Therefore, the RETAIN statement can no longer operate on the entire table, which is needed for this type of operation.

Here are some SAS language elements that have similar inter-row dependent operations:

- RETAIN statement
- DIF and LAG functions
- .FIRST variable and .LAST variable
- Temporary arrays

When using these statements, you might need to run a second DATA step that runs in one thread. You do this by specifying the SINGLE = YES option in the DATA statement. This example uses the mycas.example data set in the example "Impute Missing Values Using the LAG Function" on page 64.

```
data mycas.example2 / single=yes;
   set mycas.example;
   lag_price = lag(price);
   put "The previous price was " lag_price;
run;
```

For more information about temporary arrays, see ARRAY Statement and Example 3: Creating an Array for Temporary Use in the Current DATA Step.

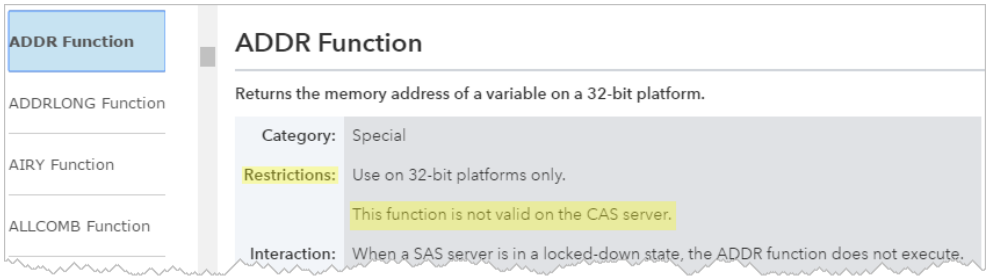
For more information about the FIRST. and LAST. variables, see How SAS Determines FIRST.variable and LAST.variable.

Language Elements Support and Limitations

Not all SAS language elements are supported in CAS. Language elements that are supported in CAS are a subset of the SAS language. Below is a list of links to the reference documentation for the SAS language elements that are supported in the DATA step in SAS Viya.

- Dictionary of Statements for SAS Viya
- Dictionary of Functions and CALL Routines for SAS Viya
- Dictionary of Formats and Informats for SAS Viya
- Dictionary of Data Set Options for SAS Viya
- Dictionary of Component Objects for SAS Viya

Language elements that are supported in SAS Viya but not in CAS are marked in the documentation with a "Restriction" as shown in the following image:



CAS-specific DATA Step Language Elements:

- "SAS Cloud Analytic Services: Language Reference" in *SAS Cloud Analytic Services: Language Reference*
- "Functions" in *SAS Cloud Analytic Services: Language Reference*
- CAS Data Set and LIBNAME Statement Options

Other Documentation Useful in DATA Step Programming:

- "SAS Viya Data Management and Utility Procedures Guide" in *SAS Viya Data Management and Utility Procedures Guide*
- "SAS Viya Macro Language: Reference" in *SAS Viya Macro Language: Reference*
- "SAS Viya National Language Support: Reference Guide" in *SAS Viya National Language Support: Reference Guide*
- "Dictionary of SAS System Options" in *SAS Viya System Options: Reference*

Restrictions for DATA Step Processing in CAS

The following table contains restrictions on the use of some DATA step language elements in CAS.

Table 2.1 *Restrictions and Notable Behaviors for DATA Step Processing*

Feature	Description
BY variables	BY variables are supported in the DATA step in CAS with the following differences: • CAS partitions tables based on the first BY variable and then orders the rows within each partition based on the BY variables. • If there is a format applied to the first BY variable, CAS uses the formatted values, rather than the raw values to partition the data. • Sorting is not required for BY variables as it is in a SAS DATA step. CAS implicitly orders the data. For more information, see "How CAS Groups Data with BY Variables" on page 33, "How BY Variables Affect Multithreaded DATA Step Execution" on page 34, and the example "Group Variables in Descending Order Using the DATA Step BY Statement" on page 75.

BY statement, DESCENDING option	The DESCENDING option in the BY statement is not supported in a DATA step that is running in CAS. See "Group Variables in Descending Order Using the DATA Step BY Statement" on page 75 for an example that shows how to group in descending order.
FORMAT procedure	User-defined formats created with PROC FORMAT must exist both in SAS and in CAS when using them in a DATA step that is running in CAS. If the format is not available to SAS, the SAS DATA step will fail before it is sent to run in CAS. For more information, see "FORMAT" in *SAS Viya Data Management and Utility Procedures Guide*.
Formats	These formats are not supported in a DATA step that is running in CAS: • WORDS • WORDSF • WORDDATE (use NLDATES instead) • WEEKDATE (use NLDATEW instead) The NLDATEW format and the NLDATE format automatically translate the date values to the language specified in your LOCALE= setting.
Functions	The default encoding scheme for sessions in SAS Viya is UTF-8. In a UTF–8 environment, data can contain multi-byte characters that are equivalent to more than 1 byte. Therefore, functions that parse character strings to calculate length and positional values might return unexpected results. For example, the LENGTH function returns the number of bytes rather than the number of characters. See "Index CHAR and VARCHAR Character Strings" on page 65 for an example of this behavior in functions. For more information about UTF-8 session encoding, see"UTF-8 Session Encoding and Migrating SAS Data Sets" on page 45.

INFILE, INPUT, and DATALINES statements	These language elements are supported only in a DATA step that is running in the SAS client session. This includes the DATA step that is used to load data into CAS. For example, the following DATA step is valid because the processing is being done in the local SAS session and not in the CAS server session:
	`data mycas.weight;`
	`  input PatientID $ Week1 Week8 Week16;`
	`  loss=Week1-Week16;`
	`  datalines;`
	`2477 195 177 163`
	`2431 220 213 198`
	`2456 173 166 155`
	`2412 135 125 116`
	`;`
	Even though the DATA step is using the CAS engine to write output to CAS, the DATA step and the statements within it are not processing in CAS. They are processing in the SAS client session. To run the DATA step in CAS, both the input and output table names must contain CAS engine librefs.

Macros

The SAS macro language is supported in SAS Viya. You can create macros that generate SAS code that executes in CAS, but the macro code itself executes in the SAS Studio client session and not in CAS. Because all macro code is pre-compiled before it is sent to CAS for processing, you can use SAS macros in a variety of ways in SAS Viya. For example, you can use macros to execute PROC CAS, PROC CASUTIL, and CAS DATA step code. Here are a few examples:

```
 /* Create a macro for getting server information */
%macro aboutServer;
   serverStatus;
%mend;
proc cas;
   %aboutServer;
run;
```

```
 /* Create a macro for loading a SASHELP data set into CAS */
%macro loadit(name=);
proc casutil;
   load data=sashelp.&name
   outcaslib='casuserhdfs'
   replace;
run; quit;
%mend loadit;
%loadit(name=cars)
```

```
/* Create a macro for saving a SASHELP data set in CAS */
%macro loadsaveit(name=);
proc casutil;
   load data=sashelp.&name
   outcaslib='casuserhdfs'
   replace;
   save casdata="&name" replace;
run; quit;
%mend loadsaveit;
%loadsaveit(name=cars);
```

```
/* Create a macro for dropping a table from memory */
%macro dropit(name=);
proc casutil incaslib="casuserhdfs";
  droptable casdata="&name";
  list files;
run; quit;
%mend dropit;
%dropit(name=cars)
```

Restrictions in macro processing include the use of the SYMGET function and the CALL SYMPUT routine in the CAS DATA step. In other words, these functions are supported in a DATA step that is processing in SAS but they are not supported in a CAS DATA step. For information about where the DATA step runs in SAS Viya, see "The DATA Step in SAS Viya" on page 22.

SYMGET is supported in SAS Studio client session processing:

```
data test;
  x = SYMGET('SYSDATE9');
  put x;
run;
```

But, it is not supported in a DATA step that is processing in CAS:

```
/* an ERROR is generated */
data mycas.test / sessref="casauto";
  x = SYMGET('SYSDATE9');
  put x;
run;
```

For more information about macro processing in SAS Viya, see "Using the Macro Facility in SAS Viya" in *SAS Viya Macro Language: Reference*.

MODIFY, REMOVE, and REPLACE statements	These statements are not supported in a DATA step that is running in CAS and you cannot use them to modify an in-memory CAS table. You can use these statements in a SAS client-session and then load the results to CAS.
VARCHAR data type	There are some restrictions on uses of the VARCHAR data type. See "VARCHAR Data Type " on page 37 for more information.

WHERE= data set option	CAS does not support the WHERE= data set option for output tables. If you specify the WHERE= data set option on the output data set, the DATA step automatically runs in SAS rather than in CAS.
	
	For example, the following DATA step is processed in the SAS client session:
	
	`libname mycas cas;`
	`data mycas.air2(where=(air>450));`
	`  set mycas.air;`
	`run;`
	
	You can use the WHERE= option on the input data set for processing in CAS:
	
	`libname mycas cas;`
	`data mycas.air2;`
	`  set mycas.air(where=(air>450));`
	`run;`
	
	An alternative to using the WHERE statement is using a subsetting-IF statement in the body of the DATA step.
	
	`libname mycas cas;`
	`data mycas.air2;`
	`  set mycas.air;`
	`  if weight>450 then delete;`
	`run;`
	
	For more information, see "WHERE= Data Set Option" in *SAS Viya Data Set Options: Reference*.

zero-column tables	CAS does not support zero-column tables. You will get an error if you try to read a table in CAS that has no variables.
	
	`data mycas.hello;`
	` put "hello";`
	`run;`
	
	`NOTE: Running DATA step in Cloud Analytic Services.`
	`NOTE: The DATA step has no input data set and will run in a single thread.`
	`ERROR: The table hello in caslib CASUSER(lidave) must have at least one variable.`
	`NOTE: Duplicate messages output by DATA step:`
	`ERROR: The table hello in caslib CASUSER(lidave) must have at least one variable.`
	`ERROR: The action stopped due to errors.`
	
	The DATA step in SAS supports zero-column tables:
	
	`data work.hello;`
	` put "hello";`
	`run;`
	
	`NOTE: The data set WORK.HELLO has 1 observations and 0 variables.`
	`NOTE: DATA statement used (Total process time):`
	`  real time  0.00 seconds`
	`  cpu time  0.01 seconds`

DATA Step Processing Modes

In the SAS Viya environment, the DATA step runs either locally in a SAS client session or in CAS. When programming with the DATA step in SAS Viya, it is important to understand where the computation is taking place because performance can become an issue with very large data sets. Also, not all DATA step language elements are supported in both environments, so you need to know where the DATA step is processing to know what features are available.

SAS client session processing
executes outside the CAS environment in a local SAS session. This is the traditional SAS DATA step processing mode in which the DATA step runs in a single thread on the SAS Workspace Server using the V9 engine. The statements within the DATA step are processed in the SAS client session, but the output data is sent to CAS, where it exists as an in-memory CAS table. Some DATA step statements and some procedures, such as PROC PRINT and PROC CONTENTS might operate on CAS tables, but they are simply requesting information from CAS about the table. They are not actually processing data in the CAS server.

CAS server session processing
executes in the CAS server either in multiple threads or in a single thread. The data might be distributed over multiple machines or over data that exists on a single-machine CAS server. In either case, the DATA step executes in multiple threads by default, but it can be set to run in a single thread. Both the input and output tables must be specified using a CAS engine libref.

Role of the DATA Step in SAS Viya

If you are familiar with the SAS DATA step, then you know that its primary function is to create and manipulate SAS data sets for analysis. In SAS Viya, the SAS DATA step serves three distinct roles:

1. As a CAS *table manipulator* for creating and manipulating in-memory CAS tables. All data is in CAS and all processing is done in CAS. Both the input and output data must use a CAS engine libref and operate on CAS tables. Here is a code example that shows the DATA step processing in CAS.

```
libname mycas cas;
 data mycas.cars2;
    set mycas.cars;
    keep make weight comb;
    comb = (MPG_City * .55) + (MPG_Highway * .45);
  run;
```

2. As a *data loader* that uses the CAS engine to transfer rows of data to CAS.

   Note: You can also load SAS data sets into CAS using the CASUTIL procedure. See the example, "Load a Client-Side File" on page 4.

   The DATA step processing is done in the local SAS client session and not in CAS. After the data set is loaded into CAS, you can use a separate DATA step for execution in CAS, as shown in "as a CAS table manipulator" on page 29. Here is a code example that shows loading a SAS data set into CAS.

```
libname mycas cas;
data mycas.cars2;
     set sashelp.cars;
     keep make weight comb;
```

```
    comb = (MPG_City * .55) + (MPG_Highway * .45);
run;
```

3. As a *traditional* SAS DATA step - for manipulating SAS client-session (local) data and performing various data preparation tasks on local data in the SAS Studio client session. There is no interaction with CAS, and SAS uses only the V9 engine for data access. Here is a code example:

```
data work.mycars;
   set sashelp.cars;
   keep make weight comb;
   comb = (MPG_City * .55) + (MPG_Highway * .45);
run;
```

The following table lists summary information about how the DATA step can be used in SAS Viya.

	Manipulate Tables in CAS ***(Run the DATA Step in CAS)***	**Load SAS Data Sets into CAS** ***(Run the DATA Step in SAS)***	**Manipulate a SAS Data Set** ***(Run the DATA Step in SAS)***

Code Sample:	`data mycas.class2;`
	`set mycas.class;`
	`if age<15 then`
	`delete;`
	`run;`
		`data mycas.class2;`
		`set sashelp.class;`
		`if age<15 then`
		`delete;`
		`run;`
			`data class2;`
			`set sashelp.class;`
			`if age<15 then`
			`delete;`
			`run;`

Data Access Engine:	• LIBNAME *<libref>* CAS; • CAS engine libref specified on both input and output table names.	• LIBNAME *<libref>* CAS; • CAS engine libref specified on the output table name only.	• No LIBNAME statement required. SAS V9 Engine is used.
Input: Output:	in-memory CAS table in-memory CAS table	SAS data set in-memory CAS table	SAS data set (and other V9 engine supported data)
Processing is done:	in CAS server session	in SAS client session	in SAS client session
DATA types:	DOUBLE, CHAR, VARCHAR*	DOUBLE, CHAR, VARCHAR*	DOUBLE, CHAR
Session encoding:	The default session encoding in SAS Viya (in both SAS and CAS sessions) is UTF-8 See "UTF-8 Session Encoding and Migrating SAS Data Sets" on page 45 for more information.		
Language elements support:	See "Language Elements Support and Limitations" on page 24		

*VARCHAR data types are supported with the CAS engine only.

Controlling DATA Step Processing

Controlling Where the DATA Step Runs

In SAS Viya, the DATA step runs in CAS by default when the following conditions are met:

1. You specify the LIBNAME statement and specify CAS engine librefs on both the input and output tables.
2. You ensure that all statements are valid and supported in CAS. Language elements that are not supported in CAS are marked with a restriction on their syntax page. See "Language Elements Support and Limitations" on page 24 for more information.

CAUTION:
If you use a language element that is not supported in the CAS DATA step, then the DATA step will automatically run in SAS rather than generate an error.

You can use the DSACCEL= system option to control where the DATA step runs by default. The default value is DSACCEL=ANY, which causes the DATA step to run in CAS by default. Setting the DSACCEL= option to NONE causes the DATA step to automatically run in SAS instead of CAS.

In the following examples, both DATA steps run in CAS. In Example 1, the SESSREF= option is required because the DSACCEL option is set to NONE. In Example 2, the SESSREF= option is not required in the DATA statement since the DSACCEL option is set to ANY. This is the default setting in SAS Viya.

Example 1:

```
options dsaccel=none;
libname mycas cas;

data mycas.class2 / sessref=casauto;
   set mycas.class;
run;
```

Example 2:

```
options dsaccel=any;
libname mycas cas;

data mycas.class2 / sessref=casauto;
   set mycas.class;
run;
```

In the examples above, the DATA step is running on in-memory tables that were already loaded into CAS. For an example of loading a SAS data set into CAS using PROC CASUTIL, see "Load a Client-Side File" on page 4.

Note: When you specify DSACCEL=NONE, all DATA step processing is automatically done in the local SAS session. The SESSREF=<session-name> option in the DATA statement overrides this system option and allows that particular DATA step to process in CAS.

Determining Where the DATA Step Is Running

- Look at your log output. If you see the note **`Running DATA step in Cloud Analytic Services`**, then your DATA step is running in CAS.

```
57          data mycas.cars;
58             set mycas.cars;
59          run;

NOTE: Running DATA step in Cloud Analytic Services.
NOTE: The DATA step will run in multiple threads.
NOTE: There were 428 observations read from the table CARS in caslib CASUSERHDFS(usrname).
NOTE: The table cars in caslib CASUSERHDFS(usrname) has 428 observations and 15 variables.
NOTE: DATA statement used (Total process time):
      real time           0.25 seconds
      cpu time            0.02 seconds
```

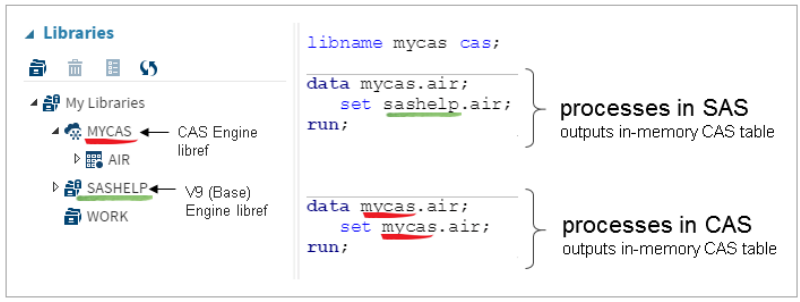
processes in CAS

```
58          data mycas.cars;
59             set sashelp.cars;
60          run;

NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: The data set MYCAS.CARS has 428 observations and 15 variables.
NOTE: DATA statement used (Total process time):
      real time           0.08 seconds
      cpu time            0.00 seconds
```

processes in SAS

- Look at the librefs specified on your input and output table names. Both the input and output tables must use the same CAS libref and all language elements in the DATA step must be supported in CAS. See "Language Elements Support and Limitations" on page 24 for more information.



- Specify OPTIONS MSGLEVEL = I and DSACCEL = ANY, and check the log for detailed information about where the DATA step was processed and why it processed there.

```
options dsaccel=any msglevel=i;
data mycas.air(where=(air>480));
   set mycas.air;
run;
```

```
58          options dsaccel=any msglevel=i;
 59
 60          data mycas.cars(where=(weight>6000));
 61             set mycas.cars;
 62          run;

 NOTE: The WHERE data set option on the DATA statement is not supported with
DATA step in Cloud Analytic Services.
 NOTE: Could not execute DATA step code in Cloud Analytic Services. Running
DATA step in the SAS client.
 NOTE: There were 3 observations read from the data set MYCAS.CARS.
 NOTE: The data set MYCAS.CARS has 3 observations and 15 variables.
 NOTE: DATA statement used (Total process time):
       real time           0.58 seconds
       cpu time            0.01 seconds
```

Controlling Threads in DATA Step Processing

A DATA step that has no input data runs in a single thread by default:

```
libname mycas cas;
     data mycas.class;
       x=1;
     run;
```

```
(Partial Log Output)
 NOTE: Running DATA step in Cloud Analytic Services.
 NOTE: The DATA step has no input data set and will run in a single thread
```

A DATA step that has input data runs in multiple threads by default on all available nodes. For example, suppose you have four worker nodes in a CAS cluster and each node supports 32 threads. Then the DATA step will run in 128 (4 x 32) threads. To control whether the DATA step runs in a single thread or in all available threads, specify the SINGLE= option in the DATA statement.

- SINGLE=YES specifies that the DATA step runs in a single thread:

```
libname mycas cas;
     data mycas.class / sessref="casauto" single=yes;
       set mycas.class;
     run;
```

- SINGLE=NO specifies that the DATA step runs in multiple threads:

```
libname mycas cas;
data mycas.example / sessref="casauto" single=no;
   x=1;
   put '# of threads is ' _nthreads_;
run;
```

- SINGLE=NOINPUT causes a DATA step without input tables to run in one thread instead of in all available threads in CAS. This is the default setting for a DATA step with no input data.
- THREADS= specifies the number of threads that the DATA step runs in on each node. A value of zero means to run using the maximum number of threads allowed. This option is for fine tuning and it might show different results depending on the number of CPU cores on each node.

See SESSREF= for syntax information related to this option.

Note: In the example above, ”CASAUTO” is the default CAS session that starts automatically when you log on to SAS Studio. See “Sessions” in *SAS Cloud Analytic Services: Fundamentals* for more information about CAS sessions.

BY-Group Processing and Ordering in the CAS DATA Step

How CAS Groups Data with BY Variables

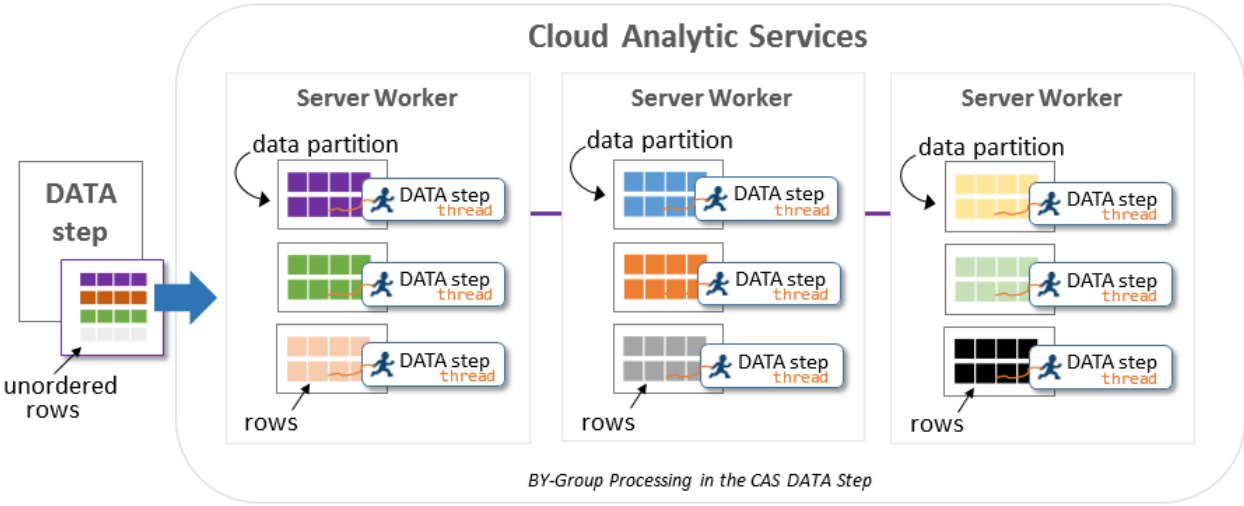
When the DATA step runs in a distributed CAS server, CAS distributes the input table across multiple servers. Each server gets only a portion of the rows in the table.

The BY statement in a SAS DATA step divides table data into groups of rows that share the same values for the BY variables. When you use the BY statement in a distributed CAS server, CAS groups the table rows based on the first BY variable, and then distributes the BY groups across multiple machines. In this way, rows within BY groups share the same server node.

Here are some key points to remember about BY-group processing in the CAS DATA step:

- CAS groups and distributes table rows based on the values of the first BY variable.
- If there is a format applied to the first BY variable, CAS groups the rows based on the *formatted* values of the first BY variable (rather than the raw values). If there is not a format applied to the first BY variable, CAS groups the rows based on the raw values.
- The DATA step orders data within each BY group based on the raw values of all the BY variables. When a BY group is defined, the DATA step orders only the rows within that BY group. There is no global ordering of BY groups.
- The DATA step runs in multiple threads on each node, one thread per BY group.
- Formats can have a noticeable impact on performance when they are used with BY variables. See "How BY Variables Affect Multithreaded DATA Step Execution" on page 34 for more information.

The following image shows how CAS organizes data into BY groups and distributes the groups across multiple machines in a distributed CAS server. The DATA step processes the rows on each node, one thread per BY group. Boxes with identical colors represent BY groups containing the same values.



How BY Variables Affect Multithreaded DATA Step Execution

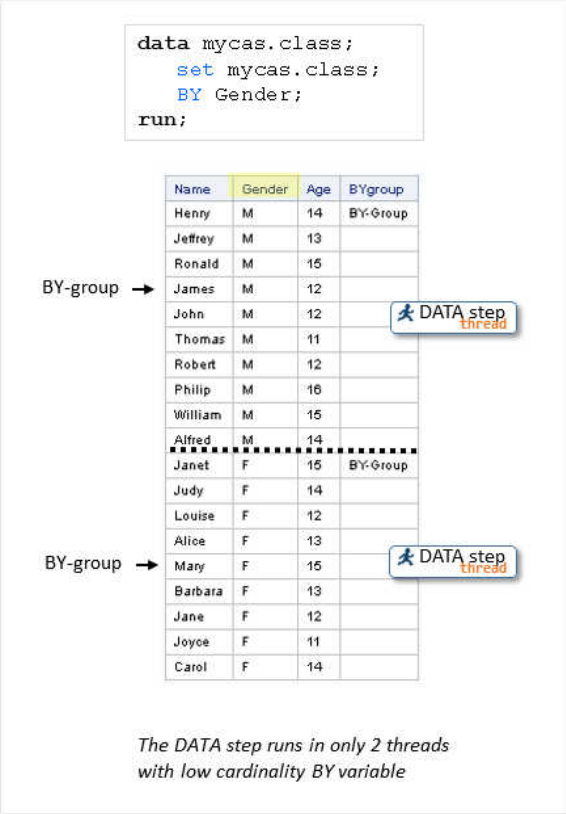
Because an entire BY group is operated on by a single DATA step thread, fewer threads are used if there are fewer BY groups. Fewer BY groups are created when the first BY variable has a low cardinality or is represented by a format with a low cardinality. Therefore, choose your first BY variable wisely and be aware of formats and their cardinality. You might need to assign or reassign formats to your BY variables to enhance performance.

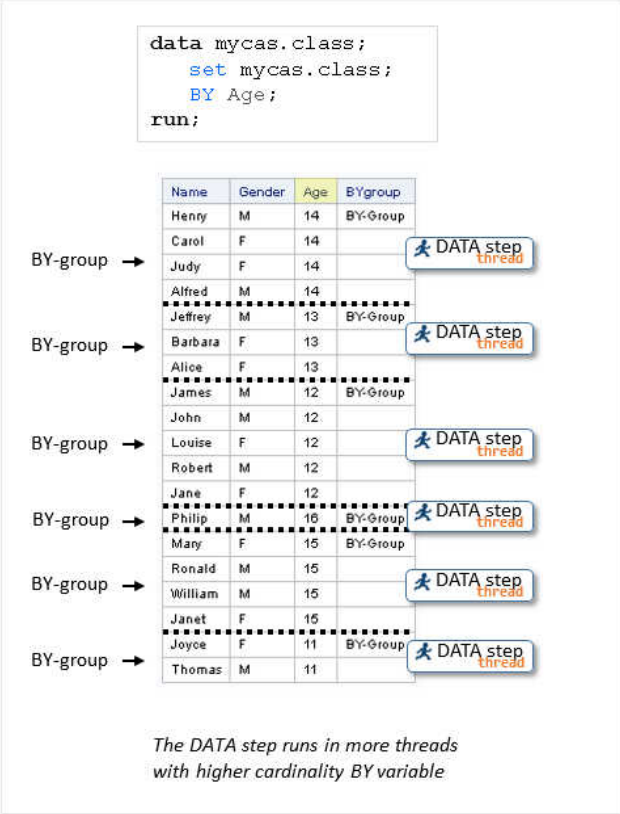
Here are some examples of low and high cardinality formats:

Formats with a High Cardinality	Formats with a Low Cardinality
• DATEw.	• QTRw.
• DATETIMEw.d	• MONNAME.
• DDMMYYw.	• WEEKDAYw.

The following image shows the relationship between the cardinality of the first BY variable and the number of threads that are spawned by the DATA step in BY-group processing.

Note: This image depicts a Bygroup column that shows the start of each BY group. For simplicity, the code used to create this column is not shown in the image. To see similar code used to identify BY groups, see "View DATA Step Processing Information Using Automatic Variables" on page 83.





Note: The DESCENDING option in the BY statement is not supported in a DATA step that is running in CAS.

BY-Group Processing versus Partitioning

BY-group processing in the CAS DATA step is an ad hoc grouping of data that is useful for performing one-time DATA step computations on small to medium tables. When you perform BY-group processing on a CAS table, the DATA step creates a temporary copy of the in-memory table that exists for the duration of the DATA step job. Once the DATA step finishes processing, the temporary table and all its BY groups disappear. Future data access to BY groups requires that the table is re-grouped and re-distributed each time. In other words, BY-group processing in the CAS DATA step does not affect the structure or distribution of the original in-memory table. BY-group processing is, therefore, more suitable for instantaneous DATA step computation.

The partitioning feature in CAS provides a more permanent and efficient solution for grouping and managing distributed data. The work required for partitioning does not have to be repeated as it does when using BY groups in the DATA step. When you partition a table as you load it using the Partition action, it becomes a partitioned in-memory table that can be accessed by subsequent operations.

To partition CAS tables, you use the Partition action. See "Partition table" in *SAS Cloud Analytic Services: System Programming Guide* in the Tables Action Set for examples and syntax for using the Partition action in CAS.

Other Methods for Partitioning, Grouping, and Ordering

Here are some other methods for ordering and grouping data:

- The ORDERBY= and GROUPBY= options in the CAS Partition action. See example "Partition a Table Using the Partition Action" in *SAS Cloud Analytic Services: System Programming Guide*.
- The ORDERBY= and GROUPBY= options in the PROC CASUTIL LOAD statement.
- The PARTITION= and ORDERBY= data set options in the SAS DATA statement. See example "Partition and Order a Table Using the DATA Step PARTITION= Data Set Option".

For an example of using the BY statement as described in this section, see "Group Variables in Descending Order Using the DATA Step BY Statement" and "Group and Order Variables Using the DATA Step BY Statement" on page 71.

DATA Step Actions

CAS actions are requests that are sent by the user to the CAS server to perform a task. You can use CAS actions to submit a DATA step. CAS actions are submitted within a PROC CAS block. For more information about CAS actions and the DATA step actions available in CAS, see "DATA Step" in *SAS Cloud Analytic Services: System Programming Guide*.

Data Types Supported in the CAS DATA Step

Overview

The DATA step in SAS Viya supports the storage of three data types. The following table provides information for each type:

Data Type	Description	Example	Missing Value

Fixed Length Character
CHAR(*n*)
	Stores a fixed-length character string, where *n* is the maximum number of bytes to store. The maximum number of bytes is required to store each value regardless of the actual size of the value.
	Range: 1–32767
		`data greets;`
		` length greet1 $16;`
		`  greet1="good dayg";`
		` format greet2 $char8.;`
		`  greet2="hi";`
		` attrib greet2`
		`  label="Greeting";`
		`run;`
			all blanks

Varying Length Character
VARCHAR(*n*)
	Stores a varying-length character string, where *n* is the maximum number of characters to store. **
	Range (UTF-8 encoding) 1–536,870,911
		`data mycas.info3;`
		`length lastname varchar(40);`
		`  set mycas.info2;`
		`  lastname="Adams";`
		`run;`
			all blanks, or zero length

DOUBLE (numeric)	Stores a numeric value, including dates and times, as a floating-point number.
	Range (SAS V9 Engine): 3–8 bytes
	Range (CAS Engine and CAS DATA step): 8 bytes *
		`data info2;`
		`set mycas.info2;`
		`length num 8;`
		`num = 25;`
		`run;`
			Any one of the following forms:
			`.`
			`.A-.Z`
			`._`

* If a shorter length is specified, then 8 bytes are used and a note is printed to the log.

** The CAS engine is required for storing VARCHAR variables in the output table of a DATA step. The DATA step can *read* CAS tables containing VARCHAR variables, but it cannot store them unless a CAS engine libref is specified on the output table.

See "SAS Cloud Analytic Services Data Types" in *SAS Cloud Analytic Services: Language Reference* for more information about data types in CAS. See "Convert Character Variables (Automatic)" on page 68 for an example that shows automatic data type conversion from VARCHAR to CHAR by the DATA step.

VARCHAR Data Type

Syntax

```
LENGTH variables(s) VARCHAR(length|*);
ARRAY arrayname[N] VARCHAR(length|*);
ARRAY arrayname[*] VARCHAR-variables;
```

variable(s)

specifies one or more variables that are assigned the type VARCHAR.

length

specifies a numeric constant that is the maximum number of characters stored in the VARCHAR variable. This value can up to 536,870,911 bytes in length.

```
length xyz varchar(32);
```

arrayname

specifies the name of the array. Defines the elements in an array as a list of VARCHAR variables.

When using a list of VARCHAR variables with the ARRAY statement, you can use the hyphen (–), colon / prefix, and double-dash lists:

```
array arr1[*] v1-v5;
array arr2[*] v:;
array arr3[*] v1--v5;
```

You cannot use VARCHAR character lists specified as _CHARACTER_.

N

describes the number and arrangement of elements in the array

*

specifies the maximum length allowed, which is currently 536,870,911 bytes. When assigning a character constant to a VARCHAR variable, the character constant is limited to 32767 bytes.

```
length xyz varchar(*);
```

Details

The VARCHAR type is a varying length character data type whose length represents the maximum number of characters you want to store. VARCHAR variables have the following characteristics:

- their length is measured in terms of characters rather than bytes
- their length varies depending on the values present

These characteristics are in contrast to those of the CHAR data type whose length is measured in terms of bytes and whose length is fixed. For example, a VARCHAR(20) can store *up to* 20 characters, but the actual storage used depends on the lengths of the individual values in the column.

For example, if a VARCHAR variable, **`columnName`**, is defined as a VARCHAR(10) column, this means that it can store up to 10 characters. But because the value "hello" contains only 5 characters, only 5 bytes of memory are allocated for the variable. A fixed-length CHAR column, on the other hand, takes up the defined number of bytes regardless of the actual size of the column.

Implicit data type conversion is supported for CHAR and NUMERIC types only and not for VARCHAR types. If you create a variable without explicitly declaring its type, then SAS will assign a type based on the values assigned to the variable. SAS assigns the CHAR data type to variables containing character data.

Range

Maximum length: 536,870,911 (this is due to the fact that the maximum length of a UTF-8 character is 4 bytes.) A VARCHAR variable can hold up to 536,870,911 bytes of data. The character encoding used in SAS and CAS is UTF-8, which has a maximum character size of 4 bytes. This means that up to 536,870,911 characters of data can be stored in a VARCHAR variable.

Tip

You can use the CALL MISSING routine to set a VARCHAR variable to missing:

```
if var1 = "abc" then call missing(var1);
```

You can use the MISSING function to test whether a VARCHAR variable is missing.

```
if missing(var2) then var2 = "missing";
```

See

"CALL MISSING Routine" in *SAS Viya Functions and CALL Routines: Reference*

"MISSING Function" in *SAS Viya Functions and CALL Routines: Reference*

Examples

1: **`Create a VARCHAR Using the LENGTH Statement`**

```
libname mycas;
data mycas.roman;
   length vc32 varchar(32);
      do i = 1 to 10;
         vc32 = put(i, ROMAN.);
         output;
      end;
run;
```

2: **`Create a VARCHAR Using the ARRAY Statement`**

```
data mycas.test;
   array test{*} varchar(*) a1 a2 a3 ('a','b','c');
   put test[1]; put test[2]; put test[3];
run;
```

Example

"Convert Character Variables (Automatic)" on page 68

When to Use a VARCHAR Data Type

The VARCHAR data type is useful in the following cases:

- the lengths of the character data vary a lot
- the longest strings are infrequent and require a fixed length of 64 bytes

In most cases, take advantage of the VARCHAR support. However, if values are consistently short, such as an ID column of airport codes, then a fixed-width CHAR variable uses less memory and runs faster. This is because VARCHAR values require 16 bytes plus the memory needed to store the VARCHAR value. So, if your values are always smaller than 16 bytes, you can save memory and processing time by using a CHAR type variable instead.

Restrictions for Using the VARCHAR Data Type in the DATA Step

VARCHAR is supported by the CAS engine but not by the SAS V9 engine. This means that the VARCHAR data type is supported in both the CAS DATA step and in the DATA step running in SAS. As long as the CAS engine is used for data access, VARCHAR variables are supported in the DATA step running in SAS and the DATA step running in CAS. If the DATA step is processing in SAS, then you must specify the CAS engine on the output data set to get full support of the VARCHAR data type.

Not all SAS language elements support the VARCHAR data type, even with the CAS engine. There are also differences in how some SAS language elements behave with VARCHAR variables. These limitations and behavior differences are listed in the table below.

Table 2.2 *Restrictions and Notable Behaviors for the VARCHAR Data Type in the SAS DATA Step*

Feature	Description
ATTRIB	You cannot use the ATTRIB statement to create VARCHAR variables.
BY statement	The BY statement uses a fixed width for VARCHAR variables. Using a VARCHAR(*) type in the BY statement might cause unexpected results.
Formats	The width of VARCHAR formats is measured in bytes rather than characters.
Functions	When passing a character value to a function, numbers indicating a length or position that are passed to the function or returned by the function are in units of bytes. When passing a VARCHAR variable to functions, these numbers are in units of characters. Of note, this includes the INDEX and SUBSTR functions. See "Index CHAR and VARCHAR Character Strings" on page 65 for a related example.
KEY= on SET and MODIFY statements	VARCHAR variables are not supported by the KEY= option in either the SET or MODIFY statements.

PUT statement (to ODS output)	VARCHAR variables are not supported with the PUT statement when the DATA step outputs to ODS. For example, you get an error if you submit the following:
	`data _null_;`
	`  length firstname varchar(20) lastname varchar(20);`
	`  file print ods;`
	`  put _ods_;`
	`run;`

VARCHAR	In SAS Viya, the VARCHAR data type is supported by the CAS engine but not by the V9 engine. This means that to create or store a VARCHAR variable, you must use the CAS engine. The SAS DATA step (with the V9 engine) can read data containing VARCHAR variables but it converts and stores them as CHAR data types.
Variable Lists	Selecting a character variable range for character variable lists (for example, *a-character-f*) is not supported for VARCHAR variables because VARCHAR variables are not fixed-width character variables. For example, you cannot specify VARCHAR variables using the following shorthand forms: _CHARACTER_ *var1*–*varN* *var1*–CHARACTER-*varN* sum(of *variable-name-prefix*:)
V9 Engine	The SAS V9 Engine does not support the VARCHAR data type. The V9 Engine supports only the CHAR and NUMERIC (DOUBLE) data types.

Data Type Conversion

Implicit Variable Declaration

SAS does not always require that you declare a variable before you use it. You can create a new variable and use it for the first time in an assignment statement without having to explicitly declare its type or length. When you create a variable this way, SAS determines the type based on the values that you assign to it. If you assign a string value to the variable, then the variable is implicitly defined as a CHAR type. If you assign an integer value to the variable, then it becomes a NUMERIC type.

```
/* Data type and length for variables x and y are set implicitely by SAS */
data test;
   x=1;
   y='hello';
run;
proc contents data=test; run;
/* PROC CONTENTS output showing automatically defined data types */
```

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
1	x	Num	8
2	y	Char	5

Implicit data typing is supported for the CHAR and NUMERIC data types in the SAS DATA step and in the CAS runCode DATA step action. It is not supported for the VARCHAR data type. In other words, to use a VARCHAR type in your program, you must explicitly define the variable as a VARCHAR type using either the LENGTH or ARRAY statement.

For more information about SAS variables and implicit data typing, see Resulting Variable Types and Lengths Produced When They Are Not Explicitly Set.

How a Variable's Length Is Determined When Converted from VARCHAR to CHAR

When a variable is converted from a VARCHAR to a CHAR, the length of the CHAR is determined based on how the VARCHAR length was defined. If the VARCHAR variable was defined with the maximum length, then the VARCHAR is converted to a CHAR with a length of 32767 bytes. If a VARCHAR is defined with a specific length, then it is converted to a CHAR that is 4 times the length of the VARCHAR.

- Maximum length:

  `xyz varchar(*)` converts to a CHAR with a length of 32767 bytes.

- Specific length:

  `xyz varchar(20)` converts to a CHAR with a length of 80, which is 4 times the length of the VARCHAR variable.

Note: The VARCHAR data type is not supported by the SAS V9 engine. A DATA step that is running in SAS (using the V9 engine) automatically converts any VARCHAR values it encounters to CHAR values.

SAS performs this multiplication by four so that all possible VARCHAR values, including multibyte characters, can be stored in a CHAR type without truncation occurring.

```
/* Create a Maximum Length VARCHAR variable and a */
/* VARCHAR Variable with a Length of 20 */
data mycas.test;
   length maxLength varchar(*);
   length length20 varchar(20);
   maxLength="maximum";
   length20="twenty";
run;

/* Run PROC CONTENTS to get length information about the VARCHAR variables */
proc contents data=mycas.test; run;
```

Output 2.1 *PROC CONTENTS Output Showing Length Information for VARCHAR Variables in CAS DATA Step*

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	length20	Varchar	20
1	maxLength	Varchar	.

```
/* Read the CAS table containing the VARCHAR variables into a SAS data set */
/* VARCHARs are converted to CHARS */
data test;
      set mycas.test;
run;

/* Run PROC CONTENTS for information about how variable lengths were converted */
proc contents data=work.test; run;
```

Output 2.2 *PROC CONTENTS Output Showing Length Converted from VARCHAR to CHAR Type Variables in SAS DATA Step*

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	length20	Char	80
1	maxLength	Char	7

```
NOTE: One or more variables were converted because the data type is not
supported by the V9 engine.
For more details, run with options MSGLEVEL=I.
 NOTE: There were 1 observations read from the data set MYCAS.TEST.
 NOTE: The data set WORK.TEST has 1 observations and 2 variables.
```

Rules for Names

Variable and Data Set Names in the DATA Step

The rules for naming variables and tables when running the DATA step in SAS and in CAS are as follows:

- Names can be up to 32 bytes in length.
- Names can contain alpha-numeric characters and the underscore (_) character.
- Names can contain special characters **/ \ * ? " < > | : -** if they are specified as name literals (in quotations followed by n).

Here is an example showing these rules:

```
data _null_ / sessref="casauto";
      greater_than_8_but_less_than_32 = 100;
      'blanks & special chars'n = 'hello';
      put greater_than_8_but_less_than_32
      put 'blanks & special chars'n;
run;
```

Note: These rules are identical to the rules for names in SAS 9.4 when VALIDVARNAME=ANY and VALIDMEMNAME=EXTEND. For more information, see Rules for Names.

Automatic Variables

Automatic variables are created by the DATA step and saved in memory, but they are not included in the output data set being created. The values of automatic variables are retained from one iteration of the DATA step to the next, rather than set to missing. To

see an example that uses automatic variables see "View DATA Step Processing Information Using Automatic Variables" on page 83.

The following automatic variables are created by every DATA step that runs in a SAS client session or in CAS. Some of these examples use the following data set:

```
libname mycas cas;
data mycas.test;
   input x y;
datalines;
1 1
2 3
3 g
4 4
;
```

ERROR

is valid in the SAS client session and in the CAS server session.

ERROR returns a 0 by default if no errors are found. It returns a 1 when it encounters an error. Errors include input data errors, conversion errors, or math errors, such as division by 0. You can use the value of this variable to help locate errors in data records and print error messages to the SAS log.

In the following example, the IF statement writes to the SAS log during each iteration of the DATA step. The DATA step writes the contents of the input record when an error is encountered:

Example:

```
data mycas.test;
   set mycas.test;
   if _error_ = 1 then
   put "****** Error in row " _n_ " ******";
run;
```

```
NOTE: Invalid data for y in line 64 3-3.
****** Error in row 3  ******
RULE:      ----+----1----+----2----+----3----+----4----+----5----+-
64         3 g
x=3 y=. _ERROR_=1 _N_=3
NOTE: The data set MYCAS.TEST has 4 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

_N_

is initially set to 1. Each time the DATA step loops past the DATA statement, the variable _N_ increments by 1. The value of _N_ represents the number of times the DATA step has iterated, which translates to the number of rows in a data set. You can use the PUT statement with the _ALL_ argument to print the values of _N_ and _ERROR_ to the SAS log.

In a multi-node CAS server session, _N_ returns the number of DATA step iterations (table rows) per worker node per thread. For example, if an eight-row table is distributed across four worker nodes (with each node holding 2 rows), then the log displays all rows processed on each node:

Example 1:

```
data mycas.test;
   set mycas.test;
   put "This is iteration (row) # " _n_
```

```
       "on " _hostname_;
run;
```

```
NOTE: Running DATA step in Cloud Analytic Services.
 NOTE: The DATA step will run in multiple threads.
 This is iteration (row) # 1 on srvr004
 This is iteration (row) # 2 on srvr004
 This is iteration (row) # 1 on srvr005
 This is iteration (row) # 2 on srvr005
 This is iteration (row) # 1 on srvr002
 This is iteration (row) # 2 on srvr002
 This is iteration (row) # 1 on srvr003
 This is iteration (row) # 2 on srvr003
```

Example 2:

```
data mycas.test;
   set mycas.test;
   put "Iteration # " _n_ "of DATA step running in thread " _threadid_;
run;
```

```
 Iteration # 1 of DATA step running in thread 33
 Iteration # 2 of DATA step running in thread 33
 Iteration # 1 of DATA step running in thread 1
 Iteration # 2 of DATA step running in thread 1
 Iteration # 1 of DATA step running in thread 65
 Iteration # 2 of DATA step running in thread 65
 Iteration # 1 of DATA step running in thread 97
 Iteration # 2 of DATA step running in thread 97
```

HOSTNAME

returns the name of the worker node or host that the DATA step is running on. In a DATA step that is processing in SAS, _HOSTNAME_ is the name of the machine SAS is running on.

```
data _null_ ;
   x=1;
   hostname = _hostname_;
   put hostname;
run;
```

```
NOTE: Running DATA step in Cloud Analytic Services.
Server5
Server6
Server1
Server2
Server3
Server4
NOTE: Duplicate messages output by DATA step:
Server5  (occurred 32 times)
Server6  (occurred 32 times)
Server1  (occurred 32 times)
Server2  (occurred 32 times)
Server3  (occurred 32 times)
Server4  (occurred 32 times)
```

NTHREADS

returns the number of DATA step threads running in a CAS server session. In a DATA step that is processing in SAS, _NTHREADS_ is always 1.

```
data mycas.cars;
   set sashelp.cars;
   keep make model weight type;
run;
data mycas.cars / sessref="casauto" single=no;
   set mycas.cars;
   put "This is iteration (row) # " _n_
       "The DATA step is running in " _nthreads_ "threads";
run;
```

```
NOTE: Running DATA step in Cloud Analytic Services.
This is iteration (row) # 1 The DATA step is running in 16  threads
This is iteration (row) # 2 The DATA step is running in 16  threads
This is iteration (row) # 3 The DATA step is running in 16  threads
This is iteration (row) # 4 The DATA step is running in 16  threads
This is iteration (row) # 5 The DATA step is running in 16  threads
This is iteration (row) # 6 The DATA step is running in 16  threads
This is iteration (row) # 7 The DATA step is running in 16  threads
This is iteration (row) # 8 The DATA step is running in 16  threads
This is iteration (row) # 9 The DATA step is running in 16  threads
```

```
data mycas.cars / sessref="casauto" single=yes;
   set mycas.cars;
   put "This is iteration (row) # " _n_
       "The DATA step is running in " _nthreads_ "threads";
run;
```

```
NOTE: Running DATA step in Cloud Analytic Services.
This is iteration (row) # 1 The DATA step is running in 1  thread
This is iteration (row) # 2 The DATA step is running in 1  thread
This is iteration (row) # 3 The DATA step is running in 1  thread
This is iteration (row) # 4 The DATA step is running in 1  thread
This is iteration (row) # 5 The DATA step is running in 1  thread
This is iteration (row) # 6 The DATA step is running in 1  thread
This is iteration (row) # 7 The DATA step is running in 1  thread
This is iteration (row) # 8 The DATA step is running in 1  thread
This is iteration (row) # 9 The DATA step is running in 1  thread
This is iteration (row) # 10 The DATA step is running in 1  thread
```

THREADID
returns the number that is associated with the thread that the DATA step is running in a CAS server session. In a DATA step that is processing in SAS, _THREADID_ is always 1.

UTF-8 Session Encoding and Migrating SAS Data Sets

The default session encoding for all processing modes and environments in SAS Viya is UTF-8. UTF-8 is a character encoding that follows the Unicode standard and is capable of encoding all possible characters in the Unicode standard. In SAS 9 and other prior releases, the default session encoding is Latin1 for UNIX and Linux and wLatin1 for Windows environments. When you read data sets into SAS Viya from these earlier environments, you might see log messages indicating the differences between the two encodings.

SAS/CONNECT can convert data from a non-UTF-8 encoded SAS session to the UTF-8 format that SAS Viya requires, and it is required for moving data from other SAS

deployments into SAS Viya. SAS/CONNECT is not included with a standard SAS Data Mining and Machine Learning order. You must order it separately. If you order SAS/CONNECT, the required commands to install it are automatically included in your purchase. Here are some resources for handling the migration of your legacy SAS data into the SAS Viya environment:

- *SAS/CONNECT for SAS Viya: User's Guide*
- "LIBNAME Statement, CVP Engine" in *SAS Viya Statements: Reference*
- "COPY" in *SAS Viya Data Management and Utility Procedures Guide*

Examples

Prerequisites

Follow the instructions in "Tip: Automatically Connect and Generate Librefs" in *SAS Viya Quick Start* to run the examples in this section.

Examples by Category

Basic

- "Run the DATA Step in CAS" on page 47
- "Run the DATA Step in CAS on a CAS Table" on page 49

Table and Row Management

- "Convert a CAS Table to a SAS Data Set" on page 55
- "Load a SAS Data Set into CAS as a Distributed Table" on page 52
- "Combine Tables Using the BY Statement" on page 56
- "Score an In-Memory CAS Table" on page 59
- "Create a DATA Step View" on page 60

Variables Management

- "Create a VARCHAR Variable Using the LENGTH Statement" on page 63
- "Impute Missing Values Using the LAG Function" on page 64
- "Index CHAR and VARCHAR Character Strings" on page 65
- "Keep Variables in a CAS DATA Step" on page 67
- "Convert Character Variables (Automatic)" on page 68
- "Group Variables in Descending Order Using the DATA Step BY Statement" on page 75
- "Create a DATA Step Hash Object with VARCHAR Key and Data Variables" on page 79

DATA Step Session Management

Additional Information

- Many of the examples used in this document use data sets that can be found in the Sashelp library. Some examples use comma delimited files that can be found at http://support.sas.com/documentation/onlinedoc/viya/examples.htm.
- Disclaimer: SAS may reference other websites or content or resources for use at Customer's sole discretion. SAS has no control over any websites or resources that are provided by companies or persons other than SAS. Customer acknowledges and agrees that SAS is not responsible for the availability or use of any such external sites or resources, and does not endorse any advertising, products, or other materials on or available from such websites or resources. Customer acknowledges and agrees that SAS is not liable for any loss or damage that may be incurred by Customer or its end users as a result of the availability or use of those external sites or resources, or as a result of any reliance placed by Customer or its end users on the completeness, accuracy, or existence of any advertising, products, or other materials on, or available from, such websites or resources.

Run the DATA Step in CAS

Example

The following example shows how to run a DATA step in CAS. This DATA step is processed in CAS because there is no input data set, but the output is an in-memory CAS table. The DATA step executes in a single thread in the CAS server session because there is no input data. CAS tables require at least one variable, so an assignment statement is used to create a variable.

```
libname mycas cas;
data mycas.hello;
   put 'Hello from ' _hostname_
       'thread # '   _threadid_;
   x=1;
run;
```

The output table is an in-memory CAS table that contains one row and one column with a value of 1. Here is the log output:

```
 NOTE: Running DATA step in Cloud Analytic Services.
 NOTE: The DATA step has no input data set and will run in a single thread.
 Hello from server01 thread #1
 NOTE: The table hello in caslib CASUSER(****user) has 1 observations and 1
variables.
 NOTE: DATA statement used (Total process time):
       real time           0.81 seconds
       cpu time            0.00 seconds
```

Key Ideas

- Running a DATA step in CAS means that the DATA step code and all the statements within it are executing in CAS in a CAS server session. In general, statements that run in the traditional SAS DATA step run the same as they do in CAS. For information about how the DATA step runs in CAS, see "Running the DATA Step" on page 22.
- For a DATA step to run in CAS, the following must be true:
  - Both the input and output data must be CAS tables. CAS tables are named using a CAS engine libref.
  - The output CAS table must contain at least 1 variable: CAS does not support zero-column tables.
- It is recommended that you use the CASUTIL procedure to load your SAS data sets into CAS. For more information, see "CASUTIL" in *SAS Cloud Analytic Services: Language Reference* for more information.
- The DATA step automatically runs in CAS when you use a CAS engine libref on the input and output data sets. It also automatically runs in CAS when you do not specify an input data set and specify output data using a CAS engine libref. For more information, see "Controlling DATA Step Processing" on page 31.
- Not all language elements are supported in a CAS DATA step. See "Language Elements Support and Limitations" on page 24 for information about what language elements are supported in the CAS DATA step.

See Also

- Chapter 1, "Accessing Data," on page 1
- "CASUTIL" in *SAS Cloud Analytic Services: Language Reference*
- "SESSREF=cas-session-reference-name" in *SAS Viya Statements: Reference*
- "DATA Step Processing Modes" on page 29

Run the DATA Step in CAS on a CAS Table

Example

To run the DATA step in CAS using input data, both the input and output must be in the form of a CAS table. In the following example, the "CASUTIL" in *SAS Cloud Analytic Services: Language Reference* is used to load a SAS data set into CAS. The SAS data set name is preceded by the CAS engine libref, **mycas**.

```
libname mycas cas;            /* 1 */

proc casutil outcaslib="casuser";
   load data=sashelp.cars;    /* 2 */
run;

data mycas.cars2;             /* 3 */
   set mycas.cars;            /* 4 */
   if mpg_city > 25 then eff='Y';
      else eff='N';
   put 'Thread number: '  _threadid_
       'on worker node '  _hostname_;
run;
```

1. Create a CAS engine libref named MyCas.
2. Use PROC CASUTIL to load the SASHELP data set to CAS. This creates an in-memory CAS table named Cars. The OUTCASLIB= option loads the CAS table to the CASUSER caslib.
3. Specify a CAS table as the output data using the CAS engine libref.
4. Specify a CAS table as the input data using the CAS engine libref.

Key Ideas

- Running a DATA step in CAS means that the DATA step code and all the statements within it are executing in CAS. For information about how the DATA step runs in CAS, see "Running the DATA Step" on page 22.
- For a DATA step to run in CAS, the following must be true:
  - Both the input and output data must be CAS tables and use a CAS engine libref in the table name.
  - The output CAS table must contain at least 1 variable. CAS does not support zero-column tables.
- It is recommended that you use the CASUTIL procedure to load your SAS data sets into CAS. For more information, see "CASUTIL" in *SAS Cloud Analytic Services: Language Reference* for more information.
- The DATA step automatically runs in CAS when you use a CAS engine libref on the input and output data sets. It also automatically runs in CAS when you do not specify an input data set and specify output data using a CAS engine libref. For more information, see "Controlling DATA Step Processing" on page 31.
- Not all language elements are supported in a CAS DATA step. See "Language Elements Support and Limitations" on page 24."Language Elements Support and Limitations" on page 24 for information about what language elements are supported in the CAS DATA step.

See Also

- Chapter 1, "Accessing Data," on page 1
- "CASUTIL" in *SAS Cloud Analytic Services: Language Reference*
- "SESSREF=cas-session-reference-name" in *SAS Viya Statements: Reference*
- "DATA Step Processing Modes" on page 29

Save a SAS Data Set to CAS as a Distributed Table

Example

You cannot use the DATA step to save a CAS table to disk on the CAS server. You must use the CASUTIL procedure for this. The following example uses data from the Sashelp.Cars data set to show how to load a distributed table into CAS using PROC CASUTIL. The table is loaded as an in-memory CAS table. It then shows how use the DATA step in CAS to keep a limited number of variables from the original data set. Finally, it shows how to use PROC CASUTIL and the default caslib to save the in-memory CAS table to disk.

```
libname mycas cas;
caslib _all_ assign;

data mycas.mycars;
```

```
   set sashelp.cars;
run;

proc casutil incaslib="casuserhdfs";
   save casdata="mycars" outcaslib="casuserhdfs" replace;
run;
```

Here is the Log output

```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
 56
 57         data mycars;
 58         set sashelp.cars;
 59         run;

 NOTE: There were 428 observations read from the data set SASHELP.CARS.
 NOTE: The data set WORK.MYCARS has 428 observations and 15 variables.
 NOTE: DATA statement used (Total process time):
       real time           0.03 seconds
       cpu time            0.01 seconds


 60         proc casutil incaslib="casuserhdfs";
 NOTE: The UUID 'c0c8df2c-d9ec-b348-a79c-9676f2bfc759' is connected using
session CASAUTO.
 61         save casdata="mycars" outcaslib="casuserhdfs" replace;
 NOTE: Cloud Analytic Services saved the file mycars.sashdat to HDFS in caslib
CASUSERHDFS(lidave).
 NOTE: The Cloud Analytic Services server processed the request in 4.119785
seconds.
 62         run;
  74
```

You can also drop a table using PROC CASUTIL:

```
proc casutil incaslib="casuserhdfs";
   droptable casdata="mycars";
run;
```

```
1          OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
 56
 57         proc casutil incaslib="casuserhdfs";
 NOTE: The UUID 'c0c8df2c-d9ec-b348-a79c-9676f2bfc759' is connected using
session CASAUTO.
 58         droptable casdata="mycars";
 NOTE: Cloud Analytic Services dropped table mycars from caslib
CASUSERHDFS(lidave).
 NOTE: The Cloud Analytic Services server processed the request in 0.010862
seconds.
 59         run;
```

Key Ideas

- You must use PROC CASUTIL to save in-memory SAS data permanently to CAS.
- You can use the KEEP and DROP statements to manage the numbers of variables in your output data.

See Also

- "SAS Data Connector to SAS Data Sets" in *SAS Cloud Analytic Services: Language Reference*
- "Save an In-Memory Table" on page 13
- "CASUTIL" in *SAS Cloud Analytic Services: Language Reference*
- "DROP= Data Set Option" in *SAS Viya Data Set Options: Reference*
- "KEEP= Data Set Option" in *SAS Viya Data Set Options: Reference*

Load a SAS Data Set into CAS as a Distributed Table

Example

The following example loads a SAS data set into CAS as an in-memory CAS table. It then shows how to load the table as a distributed table with a different name. The LOAD statement in the CASUTIL procedure uses the default CASUSERHDFS library.

```
libname mycas cas;
caslib _all_ assign;
data mycas.mycars;
   set sashelp.cars;
   keep make type;
run;
proc casutil incaslib="casuserhdfs";
   load casdata="mycars.sashdat" casout="mycars2" replace;
run;
```

Note that caslib _all_ assign causes CAS to copy all data sets in your SAS libraries to your CASUSER and CASUSERHDFS libraries.

```
 1         OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
 56
 57        data mycas.mycars;
 58           set sashelp.cars;
 59           keep make type;
 60        run;

 NOTE: There were 428 observations read from the data set SASHELP.CARS.
 NOTE: The data set MYCAS.MYCARS has 428 observations and 2 variables.
 NOTE: DATA statement used (Total process time):
       real time           0.20 seconds
       cpu time            0.02 seconds


 61        proc casutil incaslib="casuserhdfs";
 NOTE: The UUID 'c0c8df2c-d9ec-b348-a79c-9676f2bfc759' is connected using
session CASAUTO.
 62        load casdata="mycars.sashdat" casout="mycars2" replace;
 NOTE: Cloud Analytic Services made the HDFS file mycars.sashdat available as
table MYCARS2 in caslib CASUSERHDFS(lidave).
 NOTE: The Cloud Analytic Services server processed the request in 11.722372
seconds.
 63        run;
```

Key Ideas

- You can use the KEEP and DROP statements to manage the numbers of variables in your output data.
- If you save the SAS data set onto the CAS server, you can later load it in parallel from the server rather than loading it serially from the client. Server-side loading of data is faster than loading data serially from the client. When you load data from the client serially, it is loaded one row at a time. For more information about parallel and serial loading, see "Client-Side Data Access" in *SAS Cloud Analytic Services: Fundamentals* and "Server-Side Data Access" in *SAS Cloud Analytic Services: Fundamentals*.

See Also

- "Load a Database Table" on page 12
- "Save an In-Memory Table" on page 13
- "Load a Server-Side File" on page 5
- "CASUTIL" in *SAS Cloud Analytic Services: Language Reference*

Load a CSV File into CAS and Save as a CAS Table

Example

The following example shows how to load a comma-separated value file (.csv) into CAS using the DATA step.

This example uses the file names.csv file, which can be found here: http://support.sas.com/documentation/onlinedoc/viya/examples.htm.

```
data mycas.names;
  infile '/u/myfiles/names.csv' dsd truncover;                /* 1 */
  input BRTH_YR :$10. GNDR :$10. ETHCTY :$10. NM :$10.
            CNT :$10.  RNK :$10.;                             /* 2 */
run;

proc casutil incaslib="casuserhdfs";                          /* 3 */
   save casdata="names" outcaslib="casuserhdfs" replace;
   list;
run;
```

Below is a partial view of the CSV file and its contents:

```
BRTH_YR,GNDR,ETHCTY,NM,CNT,RNK
2011,FEMALE,HISPANIC,GERALDINE,13,75
2011,FEMALE,HISPANIC,GIA,21,67
2011,FEMALE,HISPANIC,GIANNA,49,42
2011,FEMALE,HISPANIC,GISELLE,38,51
           . . . <more rows>
2014,MALE,WHITE NON HISPANIC,Youssef,24,88
2014,MALE,WHITE NON HISPANIC,Yusuf,16,96
2014,MALE,WHITE NON HISPANIC,Zachary,90,39
2014,MALE,WHITE NON HISPANIC,Zev,49,65
```

1. Load the external comma-separated file using the INFILE statement. Specify a CAS engine libref on the output table. The TRUNCOVER option allows SAS to correctly read in variable-length records. Variables without any values assigned are set to missing.
2. Specify the INPUT statement to list the column names and read them in as informats.
3. Save the in-memory CAS table as a distributed table.

Key Ideas

- You can load external files directly into CAS by using an INFILE statement and a CAS engine libref on the output table.
- You can load external files into CAS using the CASUTIL procedure with the LOAD FILE statement. The following examples show how to load files into CAS: Load a Client-Side File on page 4 and Load a Server-Side File on page 5.

See Also

- INFILE Statement in *SAS(R) 9.4 Statements: Reference*
- "Overview of SAS Informats" in *SAS Viya Formats and Informats: Reference*

Convert a CAS Table to a SAS Data Set

Example

The following example shows how to use the DATA step to convert an in-memory CAS table to a SAS data set. The first DATA step creates an in-memory CAS table that can be used for the conversion in this example. The second DATA step reads in the CAS table and creates a SAS data set in the local SAS Work library.

```
libname mycas cas;            /* 1 */
data mycas.earnings;          /* 2 */
   Amount=1000;               /* 3 */
   Rate=.075/12;
   do month=1 to 12;
      Earned +(amount+earned)*(rate);
   end;
   run;
proc print data=mycas.earnings;run;

libname mysas "/u/myFiles/";   /* 4 */
data mysas.earnings;
   set mycas.earnings;
run;
```

1 Specify a LIBNAME statement to create a CAS engine libref.

2 Specify a CAS table as the output table. This DATA step runs in CAS.

3 Create variables for the table and calculate interest earned using a DO statement.

4 Create a libref named for storing the data set. The libref Mysas represents the physical location in which the Earnings data set is stored.

5 Read in the CAS table and write it out it as a SAS data set to the SAS library that you created.

Obs	Amount	Rate	month	Earned
1	1000	.00625	13	77.6326

Key Ideas

- When you specify a CAS engine libref on the output data set in a DATA step that contains no input data, the DATA step runs in CAS. See "CAS server session processing " on page 29 for information about DATA step processing in CAS.
- The CAS DATA step (or the DATA step running in CAS) works only on in-memory CAS tables. Therefore, input and output data must be in-memory CAS tables.
- Tables in CAS must contain at least 1 variable.
- The DATA step automatically runs in CAS when you use CAS engine librefs on the input and output data sets, or when you do not specify an input data set and specify output data using a CAS engine libref. For more information, see "Controlling DATA Step Processing" on page 31.
- Not all language elements are supported in a CAS DATA step. See "Language Elements Support and Limitations" on page 24 for information about what language elements are supported in the CAS DATA step.

See Also

- "LIBNAME Statement" in *SAS Viya Statements: Reference*
- Chapter 1, "Accessing Data," on page 4
- "DATA Statement" in *SAS Viya Statements: Reference*
- "CAS server session processing " on page 29

Combine Tables Using the BY Statement

Example

The following example creates two data sets and then loads them into CAS tables using PROC CASUTIL. The first DATA step combines two data sets by appending one to the other. The second DATA step interleaves two data sets using one BY variable (Common) and the third DATA step interleaves two data sets using two BY variables (Common and Number).

```
libname mycas cas;                  /* 1 */

data animal1;                       /* 2 */
input Common $ Animal $ 3-8
      Number 10-11;
datalines;
a Ant    1
b Bird   2
c Cat    3
d Dog    4
e Eagle  5
f Frog   6
g Goose  7
```

```
h Hawk   8
i Impala 9
;
data plant1;                            /* 3 */
input Common $ Plant $ 3-10
      Number 12-13;
datalines;
a Grape    1
c Hazelnut 2
e Indigo   3
g Jicama   4
i Kale     5
;

data append;                            /* 4 */
set animal1 plant1;
run;
proc print data=append;
title '1. Plant1 Appended to Animal1
          in a ';
title2 'DATA Step Running in SAS'; run;

data interleaveByCommon;                /* 5 */
set animal1 plant1;
by common; run;
proc print data=interleaveByCommon;
title '2. Animals and Plants Interleaved
          by Variable COMMON';
title2 'in a DATA Step Running in SAS'; /
run;

data interleaveByCommonNumber;          /* 6 */
set animal1 plant1;
by common Number; run;
proc print data=interleaveByCommonNumber;
 title '3. Animals and Plants Interleaved
          by Common and Number';
title2 'in a DATA Step Running in SAS';
run;

proc casutil incaslib="casuserhdfs"     /* 7 */
             outcaslib="casuserhdfs";
  load data=animal1 replace;
  load data=plant1 replace;
run;

data mycas.appendCas ;                  /* 8 */
set mycas.animal1 mycas.plant1;
hostname=_hostname_; run;
proc print data=mycas.appendCas;
title  '4. Plant1 Appended to Animal1 in a';
title2 'Multithreaded DATA Step
        Running in CAS';
run;
```

```
data mycas.byCommon;
set mycas.animal1 mycas.plant1;          /* 9 */
by common; hostname=_hostname_; run;
proc print data=mycas.byCommon;
title '5. Animal and Plant Combined
          by Common';
title2 'in a Multithreaded DATA Step
        Running in CAS';
run;

data mycas.ByCommonNumber;               /* 10 */
set mycas.animal1 mycas.plant1;
by common Number; hostname=_hostname_;  run;
proc print data=mycas.ByCommonNumber;
title '6. Animal and Plant Combined
         by Common and Number';
title2 'in a DATA Step Running in CAS'; run;
```

1 Create a CAS engine libref.

2 Create the first data set.

3 Create the second data set.

4 Append the Plant data set to the Animal data set by specifying the SET statement. No BY statement is used.

5 Interleave the two data sets using the BY statement with 1 variable.

6 Interleave the two data sets using the BY statement with 2 variables.

7 Load the data sets into CAS.

8 Combine the two data sets by specifying only the SET statement in a DATA step running in CAS. No BY statement is used.

9 Combine the two data sets using the BY statement with 1 variable in a CAS DATA step.

10 Combine the two data sets using the BY statement with 2 variables in a CAS DATA step.

Key Ideas

- The order that rows are added to a CAS table are not necessarily the order in which they are processed. In Base SAS, the V9 engine guarantees that the order that rows are input is the order the rows will be read and stored. CAS tables do not make this guarantee. CAS tables are split among multiple grid nodes. When partitioning occurs, BY groups are formed on a single node from rows that are on multiple workers.
- The order of rows within a partition is not stable. If you need a stable ordering, use more BY variables. For example, if you use the BY statement BY ID; the rows are grouped by formatted value for ID, then ordered by raw value for ID. The order the rows were added to the table is not considered. If you use the BY statement BY ID DATE;, the rows are grouped by the formatted value of ID, then ordered by the raw values for ID and DATE.
- The SAS V9 engine guarantees the ordering of table rows. CAS does not make this guarantee. the ordering of rows in tables. See "BY-Group Processing and Ordering in the CAS DATA Step" on page 33. Ordering may be implemented in parallel on different rows of the table. In most parallel sorts, common keys are brought together with no information about their original order (unless that is implemented in the sort key, either explicitly or implicitly).
- The BY statement combines the rows in the two data sets that have the same values for the BY variable.

See Also

- "BY Statement" in *SAS Viya Statements: Reference*
- "MERGE Statement" in *SAS Viya Statements: Reference*
- "BY-Group Processing and Ordering in the CAS DATA Step" on page 33
- "ORDERBY= Data Set Option" in *SAS Cloud Analytic Services: Language Reference*
- "BY-Group Processing and Ordering in the CAS DATA Step" on page 33

Score an In-Memory CAS Table

Example

The following example loads a SASHELP data set into CAS and then scores the data using a conditional IF statement.

```
libname mycas cas;              /* 1 */

proc casutil;                   /* 2 */
   load data=sashelp.baseball promote;
run;
data mycas.baseball;            /* 3 */
   set mycas.baseball;
   if nHits > 150
     then score=1;              /* 4 */
```

```
run;
```

1. Create a CAS engine libref to access the data in CAS.
2. Load the SASHELP.BASEBALL data set into CAS as an in-memory CAS table.
3. Run the DATA step in CAS on the table. Use the CAS engine libref (Mycas) that you created with the LIBNAME statement in Step 1 to name the input and output tables.
4. Execute the score code.

Key Ideas

- When you run the DATA step in the CAS server, you are running it in multiple threads on in-memory data, which enables faster processing.
- To run the DATA step in a CAS server, you must specify a CAS engine libref on both the input and output data sets. Also, you must ensure that all the language elements in the DATA step are valid for the CAS server. See "Language Elements Support and Limitations" on page 24 and "Restrictions for DATA Step Processing in CAS" on page 25 for information about language element support.

Create a DATA Step View

Example

The following example shows how to create a DATA step view. First, a SAS data set named Class is created from raw data using the DATALINES statement. Then, a SAS DATA step view is created from the Class data set. The VIEW= option in the DATA statement is used to define DATA step views.

```
libname mycas cas;

data class;
input name $ sex $ age height weight;
datalines;
Alfred  M   14  69.0    122.5
Alice   F   13  56.5     94.0
Barbara F   13  65.3    113.0
Carol   F   14  62.8    112.5
Henry   M   14  63.5    116.5
James   M   12  57.3     99.0
Jane    F   12  59.8     96.5
;

data BMI / view=BMI;
   set class;
   BMI = weight / height**2 * 703;
run;
```

Here is the log output for the code sample above describing how the DATA step View named Work.BMI was saved in the local SAS library.

```
NOTE: DATA STEP view saved on file WORK.BMI.
 NOTE: A stored DATA STEP view cannot run under a different operating system.
 NOTE: DATA statement used (Total process time):
       real time           0.00 seconds
       cpu time            0.01 seconds
```

Next, the example loads the Class data set into CAS to create an in-memory CAS table. A new VARCHAR variable is added to the table to show later how DATA step views work with VARCHAR variables.

```
data mycas.class;
   set class;
   length note varchar(30);
   if sex="F" then do;
   note="female";
   end;
run;

data BMI / view=BMI;
set mycas.class;
   BMI = weight / height**2 * 704;
   format BMI 6.2;
run;

proc print data=BMI;
run;
```

Obs	name	sex	age	height	weight	note	BMI
1	Alfred	M	14	69.0	122.5		18.11
2	Henry	M	14	63.5	116.5		20.34
3	Alice	F	13	56.5	94.0	female	20.73
4	James	M	12	57.3	99.0		21.23
5	Barbara	F	13	65.3	113.0	female	18.66
6	Jane	F	12	59.8	96.5	female	19.00
7	Carol	F	14	62.8	112.5	female	20.08

To see how the SAS DATA step view handled the VARCHAR variable, the CONTENTS procedure is specified:

```
proc contents data=BMI;
run;
```

The CONTENTS Procedure

Data Set Name	WORK.BMI	Observations	.
Member Type	VIEW	Variables	7
Engine	SASDSV	Indexes	0
Created	08/09/2016 17:08:57	Observation Length	64
Last Modified	08/09/2016 17:08:57	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	Default		
Encoding	Default		

Engine/Host Dependent Information	
DATA Step view type	INPUT

Alphabetic List of Variables and Attributes

#	Variable	Type	Len
7	BMI	Num	8
3	age	Num	8
4	height	Num	8
1	name	Char	8
6	note	Varchar	30
2	sex	Char	8
5	weight	Num	8

Key Ideas

- You can use the DATA step to create a SAS view from a CAS table. When using a CAS table as input, you must run the DATA step locally, in a Base SAS session and not in a CAS server session.
- Views can be stored only in a Base SAS library (a library created using the V9 engine). The VIEW option in the DATA statement is not supported with the DATA step in Cloud Analytic Services. This means that you cannot use a DATA step that is running in CAS to create a SAS view. However, you can create a SAS view using a DATA step that is running in SAS and that is using a CAS table as input.
- Views containing VARCHAR variables can be stored in a Base SAS library. The SAS DATA step can read and return VARCHAR variable values that are stored in SAS views. However, the DATA step cannot read and return VARCHAR variables that are contained in CAS tables or other data sources.
- A DATA step view can include VARCHAR variables but a SAS data set cannot.
- Even though a view is stored in a Base engine library, views can read data sets stored in non-Base engine libraries. You can write a view that is stored in a Base engine library and accesses data sets in a CAS engine library.
- Update views are not supported.

See Also

- SAS Views in *SAS(R) 9.4 Language Reference: Concepts, Fifth Edition*

Create a VARCHAR Variable Using the LENGTH Statement

Example

In the following example, the CAS engine is used with the LENGTH statement to create a VARCHAR variable and a CHAR variable. The VARCHAR variable, X, has a length of 30 and the CHAR variable, Y, also has a length of 30.

```
libname mycas cas;
data mycas.string;
   length x varchar(30);
   length y $30;
   x = 'abc'; y = 'def';
run;

proc contents data=mycas.string;
run;
```

Engine/Host Dependent Information	
Data Limit	100MB

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
1	x	Varchar	30
2	y	Char	30

Key Ideas

- The VARCHAR data type is not supported by the SAS V9 engine. You must use the CAS engine when creating a VARCHAR in SAS Viya. To use the CAS engine with the DATA step, specify the CAS libref on the output data set. The example, "Convert a CAS Table to a SAS Data Set" on page 55, shows how to use the CAS engine to create a CAS table as output.
- The length of VARCHAR variables is determined based on the number of characters the string contains. The length of CHAR variables is determined based on the number of bytes the characters in the string requires.
- It is usually better to declare character strings as VARCHAR types than to declare them as CHAR types to avoid truncation of data in columns. Truncation can occur, for example, when multi-byte characters are stored in a CHAR data type. For example, if the DATA step converts character variables from UTF-8 to Wlatin2, then the variable length (in bytes) might not be long enough and values are truncated.
- If the data is consistently short, such as in an ID column or in two-letter state abbreviations, consider using a VARCHAR with a fixed length of 16 bytes.
- With smaller variables that consist of integers with up to three digits, such as in three-letter airport codes, VARCHAR variables can increase memory use.

See Also

- "DATA Types" in *An Introduction to SAS Viya Programming for SAS 9 Programmers*
- "Data Types Supported in the CAS DATA Step" on page 36
- "SAS Cloud Analytic Services Data Types" in *SAS Cloud Analytic Services: Language Reference*

Impute Missing Values Using the LAG Function

Example

The following example uses the LAG function to impute missing values in data. The missing price is imputed based on a 10% increase from the price of the previous year. If the price of the previous year is also missing, then the missing price is imputed based on a 20% increase from the price of the previous year. The following data set mycas.example is first created from raw input data using the DATALINES statement and written out to an in-memory CAS table. The LAG function compares values between rows in a table. Therefore, in a distributed CAS server, where rows are located on separate machines, the LAG function does not perform as expected. To use the LAG function in a distributed CAS server, specify the SINGLE=YES statement to ensure that the data is confined to a single node.

```
libname mycas cas;
data mycas.example;
input county $ year price;
datalines;
1001 2001 200000
1001 2002 220000
1001 2003 250000
1001 2004 280000
1001 2005 310000
1002 2001 220000
1002 2002 240000
1002 2003 270000
1002 2004 300000
1002 2005 340000
1003 2001 280000
1003 2002 300000
1003 2003 330000
1003 2004 370000
1003 2005 410000
;

data mycas.example2 (drop=lag_price lag2_price lag_county lag2_county)
                    / sessref="casauto" single=yes;
   set mycas.example;
      lag_price = lag(price);
      lag2_price = lag2(price);
      lag_county = lag(county);
```

```
        lag2_county = lag2(county);
           if price NE . then price_impute = price;
           else if price = . and county = lag_county and lag_price ne .
              then price_impute = lag_price*1.1;
                 else if price = . and lag_price = . and county = lag2_county
                  and lag2_price NE . then price_impute = lag2_price*1.2;
run;
proc print data=mycas.example2;
run;
```

Obs	county	year	price	price_impute
1	1001	2001	200000	200000
2	1001	2002	.	220000
3	1001	2003	.	240000
4	1001	2004	280000	280000
5	1001	2005	310000	310000
6	1002	2001	.	.
7	1002	2002	240000	240000
8	1002	2003	270000	270000
9	1002	2004	300000	300000
10	1002	2005	340000	340000
11	1003	2001	280000	280000
12	1003	2002	300000	300000
13	1003	2003	330000	330000
14	1003	2004	370000	370000
15	1003	2005	.	407000

Key Ideas

- For more information about multithreaded processing and inter-row dependencies, see "Inter-row Dependencies and Multithreaded DATA Step Execution" on page 24

See Also

- "LAG Function" in *SAS Viya Functions and CALL Routines: Reference*
- "Inter-row Dependencies and Multithreaded DATA Step Execution" on page 24

Index CHAR and VARCHAR Character Strings

Example

In the following example, the LENGTH statement creates a VARCHAR variable and a CHAR variable. Both variables are defined with a length of 10. The variables contain identical values (a string, 'abc'). The INDEX function returns the position in the string that the target value first appears. As you might expect, the INDEX function returns a

value of 2 for both data types. This is because the variables X and Y do not contain multi-byte characters.

```
libname mycas.cas;

data mycas.string;
   length x varchar(10);
   length y $10;
   x = 'abc'; y = 'abc';
   xi = index(x,'b');
   yi = index(y,'b');
   put "xi = " xi;
   put "yi = " yi;
run;
```

Log 2.1 *Log Output*

```
xi = 2
yi = 2
```

However, in the next DATA step, the INDEX function is used with multi-byte Chinese characters in which each of the 3 characters takes 3 bytes. The statement **`xi = index(x,'样')`** tells SAS to return the position of the character '样' in the 3-character string '榜样榜'. Even though the VARCHAR and CHAR variables contain identical string values, ('榜样榜'), the INDEX function returns different values for each type. This is because the VARCHAR data type uses character-length semantics and the CHAR data type uses byte-length semantics.

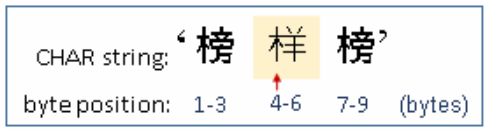
```
data mycas.string2;
  length x varchar(10);
  length y $10;
  x = '榜样榜';  y = '榜样榜';
  xi = index(x,'样');
  yi = index(y,'样');
  put "xi = " xi;
  put "yi = " yi;
run;
```

Log 2.2 *Log Output*

```
xi = 2
yi = 4
```

Therefore, it is important to remember that the INDEX function returns the **character** position when it indexes VARCHAR data types:

And it returns the **byte** position when it indexes CHAR data types.



Key Ideas

- The VARCHAR data type is not supported by the SAS V9 engine. You must use the CAS engine when creating a VARCHAR in SAS Viya. When you use the CAS engine with the DATA step, specify the CAS libref on the output data set.
- VARCHAR uses character semantics. This means that the width of a column in a table is specified in character units rather than in bytes. Some characters, like the ones in this example, require more than one byte. VARCHAR variables are also varying length variables. This means that they use as many bytes as necessary to store up to 10 characters, even if some of those characters require 3 or 4 bytes to store them.
- VARCHAR variables can store lengths up to 40 bytes.
- VARCHAR variables can increase memory use when the column contains only 2- or 3-digit integer values, such as a three-letter airport.

Note: The INDEX function uses character-based positional indexing on VARCHAR variables, and it uses byte-based indexing on CHAR variables. The INDEX function is useful for finding a substring in a parent string. For example, to find the position of the string "**hat**" in "**abhatcd**" you specify **h = index('abhatcd', 'hat');** and the function returns h = 3, because the first letter of the "**hat**" string is the third from the left starting with 1.

See Also

- "INDEX Function" in *SAS Viya Functions and CALL Routines: Reference*

Keep Variables in a CAS DATA Step

Example

In this example, the KEEP statement is used in a CAS DATA step to keep only a select set of variables in the Sashelp.Cars data set. The WHERE= data set option is used in the SET statement to filter data from the Cars data set.

```
libname mycas cas;                        /* 1 */

data mycas.cars;                          /* 2 */
   set data=sashelp.cars;
run;

data mycas.bigcars;
   set mycas.cars(where=(weight>6000));   /* 3 */
   keep make model type;                  /* 4 */
run;
```

```
proc print data=mycas.bigcars; run;      /* 5 */
```

1 Create a CAS engine libref to load data to CAS.

2 Load the Sashelp.cars data set into CAS using the DATA step.

3 Filter the data on the input data set using the WHERE statement.

4 Keep only three variables.

5 Run a PROC PRINT to view the output in the SAS Studio Results Tab.

Key Ideas

- Both the DROP and KEEP statements are supported in a CAS DATA step.
- When the DATA step is running in CAS, the WHERE= data set option is supported only for the input data set.
- The WHERE= data set option is not supported in the DATA statement in a DATA step that is running in CAS. For example, this code will generate an error:

```
data mycas.cars(where=(weight>6000));
   set mycas.cars;
run;
```

- The WHERE= data set option is supported in the SET statement in a DATA step that is running in SAS (regardless of whether it is loading to CAS using a libref). For example, the following code is valid because the processing is being done in SAS rather than CAS:

```
data mycas.cars(where=(weight>6000));
   set sashelp.cars;
run;
```

See Also

- "DROP Statement" in *SAS Viya Statements: Reference*
- "KEEP Statement" in *SAS Viya Statements: Reference*
- "WHERE= Data Set Option" in *SAS Viya Data Set Options: Reference*
- "WHERE= Data Set Option" in *SAS Viya Data Set Options: Reference*
- "DATA Step Processing Modes" on page 29

Convert Character Variables (Automatic)

Example

The following example shows how a VARCHAR variable is automatically converted to a CHAR type variable when it is processed in a SAS DATA step.

```
data mycas.test;
     length x varchar(*);
```

```
      length y varchar(20);
      x="hello";
      y="goodbye";
run;
proc contents data=mycas.test;
run;

/* transform the CAS table into a SAS data set */
data test;
   set mycas.test;
run;

proc contents data=test;
run;
```

Log 2.3 *Log Output*

```
NOTE: One or more variables were converted because the data type is not
supported by the V9 engine.
      For more details, run with options MSGLEVEL=I.
```

Output 2.3 *PROC CONTENTS Output for CAS Table "Mycas.Test"*

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	length20	Varchar	20
1	maxLength	Varchar	.

Output 2.4 *PROC CONTENTS Output for SAS Data Set "Test"*

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	length20	Char	80
1	maxLength	Char	7

Key Ideas

- The length of the CHAR variables depends on how the VARCHAR lengths are defined.
- If a VARCHAR variable is defined for its maximum length (for example, **`x varchar(*)`**), then the VARCHAR will be converted to a CHAR type with a length of 32,767 bytes.
- If a VARCHAR is defined with a specific length, then it will be converted to a CHAR type whose length is 4 times that of the VARCHAR. Also, the length will be measured in bytes rather than in characters. For example, a VARCHAR variable with a length 20 characters **`x varchar(20)`** is converted to a CHAR variable with a length of 80 bytes (4 x 20).

Order Rows Using the ORDERBY= Data Set Option

Example

In this example, a SAS data set is loaded into a CAS table using PROC CASUTIL. The DATA step runs in CAS to order the data in the table by the number of home runs using the ORDERBY= CAS data set option.

```
libname mycas cas;                        /*1*/
proc casutil outcaslib="casuser";         /*2*/
   load data=sashelp.baseball;
run;

data mycas.baseball;
   set mycas.baseball (orderby=(nHome)); /*3*/
   keep name nHits salary team;
run;
proc print data=mycas.baseball;
run;
```

1 Create a CAS engine libref for data access to CAS.

2 Load the SASHELP.BASEBALL data set to a CAS table.

3 In a CAS DATA step, read in the data set and group the rows by the nHome variable.

Obs	Name	Team	nHits	Salary
1	Allanson, Andy	Cleveland	66	.
2	Ashby, Alan	Houston	81	475.00
3	Davis, Alan	Seattle	130	480.00
4	Dawson, Andre	Montreal	141	500.00
5	Galarraga, Andres	Montreal	87	91.50
6	Griffin, Alfredo	Oakland	169	750.00
7	Newman, Al	Montreal	37	70.00
8	Salazar, Argenis	Kansas City	73	100.00
9	Thomas, Andres	Atlanta	81	75.00
10	Thornton, Andre	Cleveland	92	1100.00

You can also order the rows when you use PROC CASUTIL and the ORDERBY= option to load the data set to CAS:

```
proc casutil outcaslib="casuser";
   load data=sashelp.baseball
      casout="baseball" replace
   orderby=(team name nHome);
run;
proc print data=mycas.baseball;
run;
```

See Also

- Chapter 1, "Accessing Data," on page 1
- "ORDERBY= Data Set Option" in *SAS Cloud Analytic Services: Language Reference*

Group and Order Variables Using the DATA Step BY Statement

Example

This example shows how to use the BY statement in a DATA step that is running in CAS to group and order variables in a table. The example shows the behavior of BY group processing in a distributed CAS server. The BY statement is specified in the first DATA step to simply group and distribute the table rows by Make. The second DATA step specifies a second BY variable, Combined. CAS groups the data by Make, as before, then orders the data by Make and Combined within each BY group.

```
proc casutil;
   load data=sashelp.cars
   outcaslib='casuserhdfs'
   replace;
run; quit;

data mycas.cars2(replace=yes) ;
   set mycas.cars; by make;
   format Combined 2.;
   Combined = (MPG_Highway*.45) + (MPG_City*.55);
   if type NE 'SUV' or origin NE 'USA' then delete;
   keep make model type MPG_City MPG_Highway Combined;
run;
proc print data=mycas.cars2;
   title 'Combined Averate MPG for SUVs in the USA';
   title2 'Grouped BY Make (only)'; run; title ' ';
run;

/* Show grouped and ordered output */
data mycas.cars2;
   set mycas.cars2; by make Combined;
   keep make model type MPG_City MPG_Highway Combined;
run;
proc print data=mycas.cars2;
   title 'Grouped BY Make, Ordered by Make and Combined';
run;

/* Show grouped and ordered output along with By groups on */
/* each node to see how ordering is done */
data mycas.cars2;
   set mycas.cars2; by make Combined;
   keep make model type MPG_City MPG_Highway Combined Node Group;
```

```
   if first.make then Group = 'BY group (first row of)';
   Node=_hostname_;
run;
proc print data=mycas.cars2;
   title1 'Grouped BY Make, Ordered by Make and Combined';
   title2 '(Ordering is done with each BY group on each node)';
run;
```

Combined Averate MPG for SUVs in the USA
Grouped BY Make (only)

Obs	Make	Model	Type	MPG_City	MPG_Highway	Combined
1	Dodge	Durango SLT	SUV	15	21	18
2	Buick	Rainier	SUV	15	21	18
3	Buick	Rendezvous CX	SUV	19	26	22
4	Saturn	VUE	SUV	21	26	23
5	GMC	Yukon 1500 SLE	SUV	16	19	17
6	GMC	Yukon XL 2500 SLT	SUV	13	17	15
7	GMC	Envoy XUV SLE	SUV	15	19	17
8	Chevrolet	Tahoe LT	SUV	14	18	16
9	Chevrolet	Tracker	SUV	19	22	20
10	Chevrolet	Suburban 1500 LT	SUV	14	18	16
11	Chevrolet	TrailBlazer LT	SUV	16	21	18
12	Ford	Explorer XLT V6	SUV	15	20	17
13	Ford	Excursion 6.8 XLT	SUV	10	13	11
14	Ford	Expedition 4.6 XLT	SUV	15	19	17
15	Ford	Escape XLS	SUV	18	23	20
16	Jeep	Liberty Sport	SUV	20	24	22
17	Jeep	Wrangler Sahara convertible 2dr	SUV	16	19	17
18	Jeep	Grand Cherokee Laredo	SUV	16	21	18
19	Pontiac	Aztekt	SUV	19	26	22
20	Mercury	Mountaineer	SUV	16	21	18
21	Hummer	H2	SUV	10	12	11
22	Cadillac	SRX V8	SUV	16	21	18
23	Cadillac	Escalade	SUV	14	18	16
24	Lincoln	Aviator Ultimate	SUV	13	18	15
25	Lincoln	Navigator Luxury	SUV	13	18	15

Grouped BY Make, Ordered by Make and Combined

Obs	Make	Model	Type	MPG_City	MPG_Highway	Combined
1	Dodge	Durango SLT	SUV	15	21	18
2	Buick	Rainier	SUV	15	21	18
3	Buick	Rendezvous CX	SUV	19	26	22
4	Saturn	VUE	SUV	21	26	23
5	GMC	Yukon XL 2500 SLT	SUV	13	17	15
6	GMC	Envoy XUV SLE	SUV	15	19	17
7	GMC	Yukon 1500 SLE	SUV	16	19	17
8	Chevrolet	Tahoe LT	SUV	14	18	16
9	Chevrolet	Suburban 1500 LT	SUV	14	18	16
10	Chevrolet	TrailBlazer LT	SUV	16	21	18
11	Chevrolet	Tracker	SUV	19	22	20
12	Ford	Excursion 6.8 XLT	SUV	10	13	11
13	Ford	Expedition 4.6 XLT	SUV	15	19	17
14	Ford	Explorer XLT V6	SUV	15	20	17
15	Ford	Escape XLS	SUV	18	23	20
16	Jeep	Wrangler Sahara convertible 2dr	SUV	16	19	17
17	Jeep	Grand Cherokee Laredo	SUV	16	21	18
18	Jeep	Liberty Sport	SUV	20	24	22
19	Pontiac	Aztekt	SUV	19	26	22
20	Mercury	Mountaineer	SUV	16	21	18
21	Hummer	H2	SUV	10	12	11
22	Cadillac	Escalade	SUV	14	18	16
23	Cadillac	SRX V8	SUV	16	21	18
24	Lincoln	Navigator Luxury	SUV	13	18	15
25	Lincoln	Aviator Ultimate	SUV	13	18	15

Grouped BY Make, Ordered by Make and Combined
(Ordering is done with each BY group on each node)

Obs	Make	Model	Type	MPG_City	MPG_Highway	Combined	Group	Node
1	Dodge	Durango SLT	SUV	15	21	18	BY group (first row of)	server02
2	Buick	Rainier	SUV	15	21	18	BY group (first row of)	server02
3	Buick	Rendezvous CX	SUV	19	26	22		server02
4	Saturn	VUE	SUV	21	26	23	BY group (first row of)	server02
5	GMC	Yukon XL 2500 SLT	SUV	13	17	15	BY group (first row of)	server03
6	GMC	Envoy XUV SLE	SUV	15	19	17		server03
7	GMC	Yukon 1500 SLE	SUV	16	19	17		server03
8	Chevrolet	Suburban 1500 LT	SUV	14	18	16	BY group (first row of)	server03
9	Chevrolet	Tahoe LT	SUV	14	18	16		server03
10	Chevrolet	TrailBlazer LT	SUV	16	21	18		server03
11	Chevrolet	Tracker	SUV	19	22	20		server03
12	Ford	Excursion 6.8 XLT	SUV	10	13	11	BY group (first row of)	server03
13	Ford	Expedition 4.6 XLT	SUV	15	19	17		server03
14	Ford	Explorer XLT V6	SUV	15	20	17		server03
15	Ford	Escape XLS	SUV	18	23	20		server03
16	Jeep	Wrangler Sahara convertible 2dr	SUV	16	19	17	BY group (first row of)	server03
17	Jeep	Grand Cherokee Laredo	SUV	16	21	18		server03
18	Jeep	Liberty Sport	SUV	20	24	22		server03
19	Pontiac	Aztekt	SUV	19	26	22	BY group (first row of)	server03
20	Mercury	Mountaineer	SUV	16	21	18	BY group (first row of)	server04
21	Hummer	H2	SUV	10	12	11	BY group (first row of)	server04
22	Cadillac	Escalade	SUV	14	18	16	BY group (first row of)	server04
23	Cadillac	SRX V8	SUV	16	21	18		server04
24	Lincoln	Navigator Luxury	SUV	13	18	15	BY group (first row of)	server05
25	Lincoln	Aviator Ultimate	SUV	13	18	15		server05

Key Ideas

- In SAS, when using the BY statement in a DATA step, you must first sort the data: This is also true for a DATA step that is running in a single thread in CAS.

```
proc sort data=table1; by var1; run;

data table2;
   set table1;
   by var1;
run;
```

- You do not have to pre-sort data to use the BY statement in a multithreaded CAS DATA step. CAS performs implicit ordering on BY variables. For example, the following DATA step is valid and pre-sorting is not required:

```
data mycas.table2;
   set mycas.table1;
   by var1;
run;
```

See Also

- "BY Statement" in *SAS Viya Statements: Reference*

- "SORT" in *SAS Viya Data Management and Utility Procedures Guide*

Group Variables in Descending Order Using the DATA Step BY Statement

Example

This first example shows how to use the BY statement in a SAS DATA step to group the variable Age in descending order. This DATA step is running in the local SAS client (not in CAS). To do this, the Age variable must first be sorted using PROC SORT. The DESCENDING option is in the PROC SORT step.

```
proc sort data=sashelp.class out=classAge;
   descending age;
run;
data classAgeOrder;
   set classAge;
   by descending age;
run;
```

The next example takes the output from the first example and uses it as the input data set. This example shows how specifying the DESCENDING option in a CAS DATA step causes the DATA step to run in SAS rather than in CAS. This happens because the DESCENDING option in the BY statement is not supported in a DATA step that is running in CAS. Instead of generating an error, the DATA step automatically converts to processing in a local SAS session.

```
libname mycas cas;

data mycas.classAge;
   set sashelp.class;
run;
data mycas.classAge2;    /* 1 */
   set mycas.classAge;
   by age;              /* 2 */
run;
```

1 Run the DATA step in CAS by specifying a CAS engine libref on both the input and output tables.

2 Specify the BY statement to group the table rows by Age.

You can see that the DATA step did not run in CAS by looking at the log. There is no note indicating that the DATA step ran in CAS:

```
NOTE: There were 19 observations read from the data set SASHELP.CLASS.
 NOTE: The data set MYCAS.CLASSAGE has 19 observations and 5 variables.
```

The final example shows how to correctly use the BY statement in a CAS DATA step. The DATA step runs in CAS because no invalid or unsupported options are specified (for example, the DESCENDING option). Note also that a sort is not required for the BY variable when used in the CAS DATA step. You do not have to pre-sort BY variables in a CAS DATA step.

```
data mycas.classAge2;
   set mycas.classAge;
   by age;
run;
proc print data=mycas.classAge2; run;
```

You can check the log to confirm that the DATA step processed in CAS:

```
NOTE: Running DATA step in Cloud Analytic Services.
 NOTE: The DATA step will run in multiple threads.
 NOTE: There were 19 observations read from the table CLASSAGE2 in caslib
CASUSER(username).
 NOTE: The table classAge2 in caslib CASUSER(username) has 19 observations and
5 variables.
```

Key Ideas

- The DESCENDING option in the BY statement is not supported in a DATA step that is running in CAS. However, you can specify the DESCENDING option in a DATA step that is processing in SAS using the V9 engine.
- You will not get an error if you try to use the DESCENDING option in a CAS DATA step. Instead, the DATA step will automatically convert to running in SAS and it will run successfully if all other syntax is valid. You can tell if your DATA step ran in SAS by looking at the log. If you do not see the following note, then the DATA step ran in SAS rather than CAS.

  ```
  NOTE: Running DATA step in Cloud Analytic
        Services
  ```

- In SAS, when using the BY statement in a DATA step, you must first sort the data: This is also true for a DATA step that is running in a single thread in CAS.

  ```
  proc sort data=table1; by var1; run;

  data table2;
     set table1; by var1;
  run;
  ```

- You do not have to pre-sort data to use the BY statement in a multithreaded CAS DATA step. CAS performs implicit ordering on BY variables. For example, the following DATA step is valid and pre-sorting is not required:

  ```
  data mycas.table2;
     set mycas.table1;
     by var1;
  run;
  ```

See Also

- "BY Statement" in *SAS Viya Statements: Reference*
- "SORT" in *SAS Viya Data Management and Utility Procedures Guide*

Partition and Order a Table Using the DATA Step PARTITION= Data Set Option

Example

The following example uses the PARTITION= option to divide and distribute the table based on the value for the Type variable. It uses the ORDERBY= option to sort the data within each partition. The data within each partition is ordered by Make. This example uses the Cars data set in the Sashelp library.

```
data mycas.cars3;
   set sashelp.cars;
run;

data mycas.cars3(partition=(type) orderby=(make));
   set mycas.cars3;
run;
proc print data=mycas.cars3;
   title  'Partitioning';
   title2 'Combined Averate MPG for SUVs in the USA';
   title3 'Partitioned By Type and Ordered By Make ';
run;
```

Partitioning
Combined Averate MPG for SUVs in the USA
Partitioned By Type and Ordered By Make

Obs	Make	Model	Type	MPG_City	MPG_Highway	Combined
1	Buick	Rendezvous CX	SUV	19	26	22
2	Buick	Rainier	SUV	15	21	18
3	Cadillac	SRX V8	SUV	16	21	18
4	Cadillac	Escalade	SUV	14	18	16
5	Chevrolet	TrailBlazer LT	SUV	16	21	18
6	Chevrolet	Suburban 1500 LT	SUV	14	18	16
7	Chevrolet	Tracker	SUV	19	22	20
8	Chevrolet	Tahoe LT	SUV	14	18	16
9	Dodge	Durango SLT	SUV	15	21	18
10	Ford	Escape XLS	SUV	18	23	20
11	Ford	Expedition 4.6 XLT	SUV	15	19	17
12	Ford	Excursion 6.8 XLT	SUV	10	13	11
13	Ford	Explorer XLT V6	SUV	15	20	17
14	GMC	Yukon 1500 SLE	SUV	16	19	17
15	GMC	Yukon XL 2500 SLT	SUV	13	17	15
16	GMC	Envoy XUV SLE	SUV	15	19	17
17	Hummer	H2	SUV	10	12	11
18	Jeep	Liberty Sport	SUV	20	24	22
19	Jeep	Grand Cherokee Laredo	SUV	16	21	18
20	Jeep	Wrangler Sahara convertible 2dr	SUV	16	19	17
21	Lincoln	Navigator Luxury	SUV	13	18	15
22	Lincoln	Aviator Ultimate	SUV	13	18	15
23	Mercury	Mountaineer	SUV	16	21	18
24	Pontiac	Aztekt	SUV	19	26	22
25	Saturn	VUE	SUV	21	26	23

Key Ideas

- The PARTITION= data set option specifies the list of partitioning variables for an he output table.
- You must specify the PARTITION= option before you can specify the ORDERBY= option.
- CAS partitions the table based on the formatted value of the PARTITION= variable (rather than the raw value). So, for example, a column of date values that are formatted using the Qtr. format (quarters in a year) will be divided into 4 partitions, one partition for each quarter that the date value falls into.
- Observations that share the same partition key are arranged together on the same worker node in SAS Cloud Analytic Services.

See Also

- "PARTITION= Data Set Option" in *SAS Cloud Analytic Services: Language Reference*
- "ORDERBY= Data Set Option" in *SAS Cloud Analytic Services: Language Reference*
- "BY-Group Processing versus Partitioning" on page 35

• "Other Methods for Partitioning, Grouping, and Ordering" on page 36

Create a DATA Step Hash Object with VARCHAR Key and Data Variables

Example

In the following example, a hash object is created in which VARCHAR variables are defined for the KEY and DATA values. The DATA step processes in CAS.

```
libname mycas cas;

/* Simple add and iterate */
data _null_ &dataopts;
  length vc1 vc2 varchar(*);

  dcl hash h();
  h.definekey('vc1');
  h.definedata('vc2');
  h.definedone();

  vc1 = 'abcd';    vc2 = 'ABCD';    h.add();
  vc1 = 'efghijk'; vc2 = 'EFGHIJK'; h.add();
  vc1 = 'lmn';     vc2 = 'LMN';     h.add();

  do vc1 = 'lmn', 'opqrst', 'efghijk', 'abcd';
    rc = h.find();
    if rc ^= 0 then
      put 'Didn''t find key ' vc1=;
    else
      put _all_;
  end;

  put '-----------------------------';

  dcl hiter hi('h');
  rc = hi.next();
  do until (rc ^= 0);
    put _all_;
    rc = hi.next();
  end;
run;

/* Add with parameters */
data _null_ &dataopts;
  length vc1 vc2 varchar(*);
  length vc1b vc2b varchar(*);

  dcl hash h();
  h.definekey('vc1');
  h.definedata('vc2');
  h.definedone();
```

```
    vc1b = 'abcd';    vc2b = 'ABCD';    h.add(key:vc1b, data:vc2b);
    vc1b = 'efghijk'; vc2b = 'EFGHIJK'; h.add(key:vc1b, data:vc2b);
    vc1b = 'lmn';     vc2b = 'LMN';     h.add(key:vc1b, data:vc2b);

    do vc1b = 'lmn', 'opqrst', 'efghijk', 'abcd';
      rc = h.find(key:vc1b);
      if rc ^= 0 then
        put 'Didn''t find key ' vc1b=;
      else
        put _all_;
    end;

    put '-----------------------------';

    dcl hiter hi('h');
    rc = hi.next();
    do until (rc ^= 0);
      put _all_;
      rc = hi.next();
    end;
  run;
```

```
Partial Log Output
31         libname c cas;
NOTE: Libref C was successfully assigned as follows:
      Engine:        CAS
      Physical Name: 6738b390-2073-504a-9a4c-454463e60aea


NOTE: Running DATA step in Cloud Analytic Services.
vc1=lmn vc2=LMN rc=0 _ERROR_=0 _N_=1
Didn't find key vc1=opqrst
vc1=efghijk vc2=EFGHIJK rc=0 _ERROR_=0 _N_=1
vc1=abcd vc2=ABCD rc=0 _ERROR_=0 _N_=1
-----------------------------
vc1=abcd vc2=EFGHIJK rc=0 _ERROR_=0 _N_=1
vc1=abcd vc2=LMN rc=0 _ERROR_=0 _N_=1
NOTE: Duplicate messages output by DATA step:
vc1=abcd vc2=ABCD rc=0 _ERROR_=0 _N_=1  (occurred 2 times)
NOTE: DATA statement used (Total process time):
      real time           0.20 seconds
      cpu time            0.03 seconds
```

Key Ideas

- You can use VARCHAR variables for KEY and DATA values when defining a hash object for a DATA step running in CAS or in SAS. VARCHAR variables are supported with all hash object methods where character values are supported.
- Ordered hashes are not supported in CAS (for example, **`dcl hash h(ordered:'yes');`** is not supported in the DATA step running in CAS.
- A fixed-width 32-byte SHA-256 hash digest value is computed for VARCHAR keys.

See Also

- *SAS Viya Component Objects: Reference*

Run the DATA Step in a Single Thread

Example

The following example shows how to use the SINGLE= option to run the DATA step in a single thread. This option is useful for running functions that have inter-row dependencies.

```
libname mycas cas;

data mycas.class;                                   /* 1 */
   set sashelp.class;
run;

data mycas.class2 / sessref="casauto"               /* 2 */
                    single=yes;                     /* 3 */
   set mycas.class;
   put "The # of Threads is " _nthreads_;           /* 4 */
run;
```

1 Load the Class data set from the Sashelp library into CAS.

2 Specify the SESSREF= option to identify the session. Casauto is the default session.

3 Specify the SINGLE=YES option.

4 Specify the automatic variable, _NTHREADS_ with the PUT statement to verify the number of threads that are processing.

Key Ideas

- The DATA step in CAS automatically runs in multiple threads on every available computer in your system, with one exception: a DATA step that has no input data runs in a single thread by default.
- A SAS DATA step (a DATA step that is not executing in the CAS server) always runs in a single thread.
- Casauto is the default server session that is started when you log on to SAS Studio. For more information about CAS sessions, see "Sessions" in *SAS Cloud Analytic Services: Fundamentals*..

See Also

- "Inter-row Dependencies and Multithreaded DATA Step Execution" on page 24
- "SESSREF=cas-session-reference-name" in *SAS Viya Statements: Reference*

- "DATA Statement" in *SAS Viya Statements: Reference*
- "Running the DATA Step" on page 22

Run the DATA Step in Multiple Threads

Example

The following example shows how to run the DATA step in multiple threads. Since this is the default behavior for the DATA step in CAS, no option is specified. The second DATA step runs in CAS in multiple threads because a CAS engine libref is specified on both the input and output data sets.

```
libname mycas cas;

data mycas.class;                                   /* 1  */
   set sashelp.class;
run;

data mycas.class2;                                  /* 2  */
   set mycas.class;
   put "The # of Threads is " _nthreads_;           /* 3  */
run;
```

1 Load the Class data set from the Sashelp library into CAS.

2 You do not have to specify SINGLE=NO since this is the default for the DATA step in CAS.

3 Specify the automatic variable, _NTHREADS_ with the PUT statement to verify that the DATA step is running in multiple threads.

Log 2.4 *Partial Log Output*

```
NOTE: Running DATA step in Cloud Analytic Services.
The # of Threads is 4
The # of Threads is 4
The # of Threads is 4
The # of Threads is 4
```

Or, since the DATA step runs by default in multiple threads on all available nodes, you can simply specify the following:

```
libname mycas cas;

data mycas.class;
   set sashelp.class;
run;

data mycas.class2;
   set mycas.class;
   put "The # of Threads is " _nthreads_;
run;
```

Key Ideas

- The DATA step in CAS automatically runs in multiple threads on every available computer in your system, with one exception: a DATA step that has no input data runs in a single thread by default.
- A SAS DATA step (a DATA step that is not executing in the CAS server) always runs in a single thread.
- Casauto is the default server session that is started when you log on to SAS Studio. For more information about CAS sessions, see "Sessions" in *SAS Cloud Analytic Services: Fundamentals*

See Also

- "SESSREF=cas-session-reference-name" in *SAS Viya Statements: Reference*
- "DATA Statement" in *SAS Viya Statements: Reference*
- "Running the DATA Step" on page 22

View DATA Step Processing Information Using Automatic Variables

Example

This first example uses the _THREADID_ automatic variable to get information about a multithreaded DATA step that contains two BY variables. When the DATA step runs in CAS, the table is grouped and distributed by the first BY variable, Make. Each BY group is then ordered by the second BY variable, Type. The FIRST. automatic variable is used to show the distinct BY groups in the output table.

```
data mycas.cars;
    set sashelp.cars;
run;

data mycas.cars;
   set mycas.cars(where=(weight>4000));
   by make type;
   if first.make then first="BY Group";
   keep make type first threadid;
   threadid = _threadid_;
run;
proc print data=mycas.cars;
  title 'Cars By Make By Type - Multithreaded'; run; title;
run;
```

Cars By Make By Type - Multithreaded

Obs	Make	Type	first	threadid
1	BMW	Sedan	BY Group	2
2	BMW	Sedan		2
3	BMW	SUV		2
4	BMW	SUV		2
5	Isuzu	SUV	BY Group	3
6	Jaguar	Sedan	BY Group	11
7	Jaguar	Sports		11
8	Cadillac	Sedan	BY Group	20
9	Cadillac	SUV		20
10	Cadillac	SUV		20
11	Cadillac	Truck		20
12	Dodge	Sedan	BY Group	23
13	Dodge	SUV		23
14	Dodge	Truck		23
15	Infiniti	Wagon	BY Group	23
16	Infiniti	Wagon		23
17	Buick	SUV	BY Group	29
18	Buick	SUV		29
19	Volkswagen	Sedan	BY Group	32
20	Volkswagen	Sedan		32
21	Volkswagen	SUV		32
22	Volkswagen	Wagon		32
23	Pontiac	Sedan	BY Group	43
24	Land Rover	SUV	BY Group	44
25	Land Rover	SUV		44
26	Mercedes-Benz	Sedan	BY Group	46
27	Mercedes-Benz	Sedan		46
28	Mercedes-Benz	Sedan		46
29	Mercedes-Benz	Sedan		46
30	Mercedes-Benz	Sports		46
31	Mercedes-Benz	Sports		46
32	Mercedes-Benz	Sports		46
33	Mercedes-Benz	SUV		46

Key Ideas

- The DATA step automatically runs in multiple threads in a distributed CAS server. This is the case even in a single-machine CAS environment. By default, the DATA step runs in all available threads on every computer node in the cluster.
- You can specify SINGLE=YES to force the DATA step to run in a single thread in CAS. Single-threaded DATA step processing is useful on small- to medium-sized tables where performance is not an issue and where preserving order or maintaining inter-row dependencies is important.
- When the DATA step runs in CAS, the same DATA step program is replicated across the CAS cluster. On each node in the cluster, the DATA step runs multiple threads. Each thread runs on only a portion of the table.
- The DATA step automatically runs in CAS when you specify a CAS engine libref on both the input and output data sets.
- There are restrictions on what language elements you can use in a DATA step running in CAS. For information about these restrictions, see "Language Elements Support and Limitations" on page 24.

See Also

- "Automatic Variables" on page 42

- "DATA Statement" in *SAS Viya Statements: Reference*
- "Controlling DATA Step Processing" on page 31
- "About SAS Cloud Analytic Services Sessions" in *SAS Cloud Analytic Services: Fundamentals*

Use the Macro Facility to Generate CAS DATA Step Code

Example

The following example uses a SAS macro to generate SAS code. The macro takes the tables listed in the TABLES= parameter and writes them out as a SAS data set. For more information about macro processing in SAS Viya, see "Using the Macro Facility in SAS Viya" in *SAS Viya Macro Language: Reference*.

```
libname mycas cas;                                          /* 1 */
data mycas.class;                                           /* 2 */
   set sashelp.class; run;
data mycas.cars;
   set sashelp.cars; run;
data mycas.air;
   set sashelp.air; run;

%macro load(tables=);                                       /* 3 */
libname mysas "/u/mycas/";                                  /* 4 */
%let n=%sysfunc(countw(&tables,%str( )));                   /* 5 */
%do i=1 %to &n;                                             /* 6 */
   %let dsn=%scan(&tables,&i);
   data mysas.&dsn;                                         /* 7 */
      set mycas.&dsn;
   run;
%end;
%mend;                                                      /* 8 */
%load(tables=class cars air);
```

1. Create a CAS engine libref to access the CAS tables.
2. Load three SASHELP data sets to CAS.
3. Start the macro definition.
4. Create a SAS libref.
5. Set *n* equal to the number of tables.
6. Loop through *n* times (for each table).
7. Create a SAS data set from each CAS table.
8. End macro definition.

Log 2.5 *Partial Log Output for Macro*

```
NOTE: Libref MYSAS was successfully assigned as follows:
      Engine:        V9
      Physical Name: /u/mycas

 NOTE: There were 19 observations read from the data set MYCAS.CLASS.
 NOTE: The data set MYSAS.CLASS has 19 observations and 5 variables.
 NOTE: DATA statement used (Total process time):
       real time           0.02 seconds
       cpu time            0.01 seconds



 NOTE: There were 428 observations read from the data set MYCAS.CARS.
 NOTE: The data set MYSAS.CARS has 428 observations and 15 variables.
 NOTE: DATA statement used (Total process time):
       real time           0.02 seconds
       cpu time            0.00 seconds



 NOTE: There were 144 observations read from the data set MYCAS.AIR.
 NOTE: The data set MYSAS.AIR has 144 observations and 2 variables.
 NOTE: DATA statement used (Total process time):
       real time           0.02 seconds
       cpu time            0.01 seconds
```

See Also

- "Using the Macro Facility in SAS Viya" in *SAS Viya Macro Language: Reference*

DATA Step Program Walk-through

Example

The following example shows the steps that you need to perform to run the DATA step in CAS from the SAS Studio client application. See "Tip: Automatically Connect and Generate Librefs" in *SAS Viya Quick Start* for instructions on how to add the CAS and CASLIB statements to your Autoexec File. These statements automatically connect you to a CAS session every time you log on to SAS Studio.

1. **Sign in to SAS Studio**.

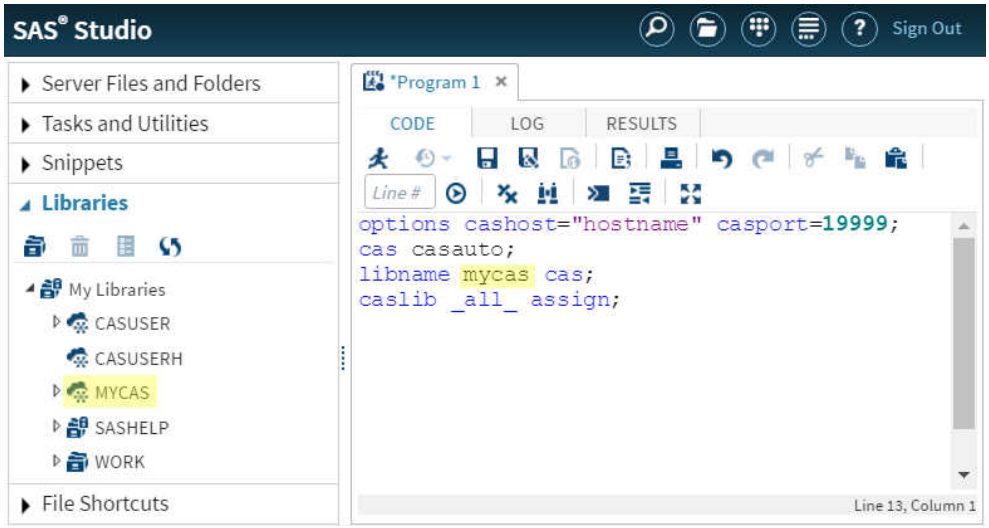
   Open SAS Studio from a URL in the form of **`http://hostname:port`**. Sign in using the user-ID and password for your operating system account.

2. **Start a CAS session**.

   Activate the default CAS session, Casauto, using the CAS statement. Specify the LIBNAME statement to create a SAS library that is associated with the CAS engine. Mycas is the CAS engine libref. The CASLIB _ALL_ ASSIGN statement maps caslibs to SAS libraries.

   ```
   cas casauto;
   libname mycas cas;
   ```

```
caslib _all_ assign;
```



Here is the Log output:

```
1   options cashost="server1" casport=19999;
 2         cas casauto;
 NOTE: The session CASAUTO connected successfully to Cloud Analytic Services
server1 using port 19999. The UUID is
       06c4f9f5-3592-7a42-8c0e-ed3dc0a78e19. The user is ****** and the active
caslib is CASUSERHDFS(******).
 NOTE: The SAS option SESSREF was updated with the value CASAUTO.
 NOTE: The SAS macro _SESSREF_ was updated with the value CASAUTO.
 NOTE: The session is using 4 workers.
 3         libname mycas cas;
 NOTE: Libref MYCAS was successfully assigned as follows:
       Engine:        CAS
       Physical Name: 06c4f9f5-3592-7a42-8c0e-ed3dc0a78e19
 4         caslib _all_ assign;
 NOTE: CASLIB CASUSER(******) for session CASAUTO will be mapped to SAS Library
CASUSER.
 NOTE: CASLIB CASUSERHDFS(*****) for session CASAUTO will be mapped to SAS
Library CASUSERH.
```

3. **Load data to CAS**.

   Use the CASUTIL procedure to load data to CAS. The data is loaded from the Sashelp library in the local SAS client session into an in-memory table in CAS. The DATA step processes in SAS. The output table, Cars, is loaded into the CAS server as an in-memory table.

```
proc casutil;
   load data=sashelp.cars
   outcaslib='casuserhdfs'
   replace;
run; quit;
```

   The output table is an in-memory CAS table that contains 428 rows and 15 columns. Here are the results that are automatically displayed in the Output Data tab:

Table: MYCAS.CARS2 View: Column names Filter: (none)

Columns

- Select all
- Make
- Model
- Type
- Origin
- DriveTrain
- MSRP
- Invoice
- EngineSize
- Cylinders
- Horsepower

Total rows: 428 Total columns: 15

	Make	Model	Type	Origin	DriveTr.
1	Acura	MDX	SUV	Asia	All
2	Acura	NSX coupe 2dr manual S	Sports	Asia	Rear
3	Audi	A6 3.0 4dr	Sedan	Europe	Front
4	Audi	A8 L Quattro 4dr	Sedan	Europe	All
5	Audi	A6 3.0 Avant Quattro	Wagon	Europe	All
6	BMW	325Ci convertible 2dr	Sedan	Europe	Rear
7	BMW	330Ci convertible 2dr	Sedan	Europe	Rear
8	BMW	M3 convertible 2dr	Sports	Europe	Rear
9	Buick	Century Custom 4dr	Sedan	USA	Front
10	Buick	Park Avenue Ultra 4dr	Sedan	USA	Front
11	Cadillac	Seville SLS 4dr	Sedan	USA	Front
12	Chevrolet	Tracker	SUV	USA	Front

Here is the log output:

```
NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: The data set MYCAS.CARS has 428 observations and 15 variables.
```

4. **Run the DATA step in CAS**. The DATA step runs in CAS in multiple threads by default. When the DATA step runs in SAS Viya, it runs where the data is stored. So, to run the DATA step in CAS, you must first load the table into memory as was done in Step 3 on page 87.

```
data mycas.cars2(promote=yes);                          /*  a  */
   set mycas.cars;                                      /*  b  */
      format Combined 2.;
      Combined = (MPG_Highway*.45) + (MPG_City*.55); /*  c  */
      if type="SUV" or origin="USA";                    /*  d  */
      keep make type MPG_City MPG_Highway Combined;
run;
title 'Combined Average MPG for SUVs in the USA By Make';
proc print data=mycas.cars2; run;                       /*  e  */
```

a Run the DATA step in CAS and specify Cars2 as the output table. Promote the output table to global scope by specifying the PROMOTE= data set option.

b Specify the in-memory table that you loaded in Step 3 on page 87, as the input table.

c Create a new variable, Combined, to calculate the combined average miles per gallon for SUVs in the USA.

d Specify the KEEP statement to keep some variables in the output table and drop the rest. Use a subsetting IF statement to subset the data and select only rows containing SUV for Type and USA for Origin.

e Print the results.

Here is the PROC PRINT output.

Combined Average MPG for SUVs in the USA By Make

Obs	Make	MPG_City	MPG_Highway	Combined
1	Cadillac	14	18	16
2	Cadillac	16	21	18
3	Saturn	21	26	23
4	Dodge	15	21	18
5	Buick	15	21	18
6	Buick	19	26	22
7	Pontiac	19	26	22
8	Mercury	16	21	18
9	Lincoln	13	18	15
10	Lincoln	13	18	15
11	Chevrolet	19	22	20
12	Chevrolet	14	18	16
13	Chevrolet	16	21	18
14	Chevrolet	14	18	16
15	GMC	16	19	17
16	GMC	13	17	15
17	GMC	15	19	17
18	Jeep	20	24	22
19	Jeep	16	19	17
20	Jeep	16	21	18
21	Hummer	10	12	11
22	Ford	18	23	20
23	Ford	15	20	17
24	Ford	10	13	11
25	Ford	15	19	17

Here is the Log output:

```
NOTE: Running DATA step in Cloud Analytic Services.
 NOTE: The DATA step will run in multiple threads.
 NOTE: There were 428 observations read from the table CARS in caslib
CASUSERHDFS(*****).
 NOTE: The table cars2 in caslib CASUSERHDFS(*****) has 25 observations and 5
variables.
 NOTE: DATA statement used (Total process time):
       real time           0.28 seconds
       cpu time            0.02 seconds
```

5. **Save the table, list information about the table, and drop the table from memory.**

   The in-memory CAS table, Cars2, can be saved permanently to disk in the server. Use the SAVE statement in the CASUTIL procedure to save the table. Use the CONTENTS statement to view information about the table. The REPLACE statement enables you to run the same procedure again and overwrite the table without an error.

```
proc casutil;
       save casdata='cars2'
       outcaslib='casuserhdfs' replace;
            contents casdata='cars2';
       droptable casdata='cars2';
run; quit;
```

Combined Average MPG for SUVs in the USA By Make

The CASUTIL Procedure

Table Information for Caslib CASUSERHDFS([redacted])

Table Name	Number of Rows	Number of Columns	NLS encoding	Created	Last Modified	Promoted Table	Repeated Table	View	Source Name	Compressed
CARS2	0	5	utf-8	14Sep2016:10:37:05	14Sep2016:10:37:05	Yes	No	No	_T_A4A44B2D_7FC2A68FCB20	No

Detail Information for cars2 in Caslib CASUSERHDFS([redacted]).

Node	Number of Blocks	Active Blocks	Rows	Fixed Data size	Variable Data size	Blocks Mapped	Memory Mapped	Blocks Unmapped	Memory Unmapped	Blocks Allocated	Memory Allocated
ALL	0	0	0	0	0	0	0	0	0	0	0

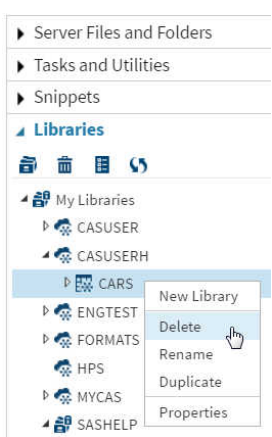
Column Information for CARS2 in Caslib CASUSERHDFS([redacted])

Column	Label	Type	Length	Formatted Length
Make		char	13	13
Type		char	8	8
MPG_City	MPG (City)	double	8	12
MPG_Highway	MPG (Highway)	double	8	12
Combined		double	8	2

Here is the Log output:

```
NOTE: The UUID 'ff29c82f-b23b-8b4e-b9d5-9ae21c7dd4fd' is connected using
session CASAUTO.
20      save casdata='cars2' outcaslib='casuserhdfs' replace;
NOTE: Cloud Analytic Services saved the file cars2.sashdat to HDFS in caslib
      CASUSERHDFS(*****).
NOTE: The Cloud Analytic Services server processed the request in 2.98294
seconds.
22           contents casdata='cars2';
NOTE: Cloud Analytic Services processed the combined requests in 0.026388
seconds.
23          droptable casdata='cars2';
NOTE: Cloud Analytic Services dropped table cars2 from caslib
CASUSERHDFS(*****).
NOTE: The Cloud Analytic Services server processed the request in 0.008735
seconds.
```

Note: You can easily delete tables by using the SAS Studio interface. Select the **Libraries tab** in the left pane, right-click on the table that you want to delete, and select delete from the pop-up menu.



Here is the entire program together.

```
cas casauto;
libname mycas cas;
caslib _all_ assign;
```

```
proc casutil;
   load data=sashelp.cars
   outcaslib='casuserhdfs'
   replace;
run; quit;
data mycas.cars2(promote=yes);
   set mycas.cars;
      format Combined 2.;
      Combined = (MPG_Highway*.45) + (MPG_City*.55);
      if type="SUV" and origin="USA";
      keep make type MPG_City MPG_Highway Combined;
      by make;
run;
proc print data=mycas.cars2;
title 'Combined Average MPG for SUVs in the USA By Make';
run;
proc casutil;
    save casdata='cars2'
      outcaslib='casuserhdfs' replace;
          contents casdata='cars2';
          droptable casdata='cars2';
run; quit;
```

Key Ideas

- You can use many of the same statements that you use in a traditional SAS DATA step in CAS. For example, the SET, KEEP, and DROP statements perform the same functions in CAS as they do in a SAS DATA step. For a complete list of supported language elements, see "Language Elements Support and Limitations" on page 24.
- When the DATA step runs in SAS Viya, it runs where the tables are stored. Running a DATA step in CAS means that the DATA step code and all the statements within it are executing in a CAS server session. For information about how the DATA step runs in CAS, see "DATA Step Processing Modes" on page 29.
- When the DATA step runs in CAS, it runs in all available threads by default. See "Multithreaded DATA Step Processing in CAS" on page 22 for more information.
- For a DATA step to run in CAS, the following must be true:
  - If the DATA step has both input and output data, both the input and output data must be CAS tables that use CAS engine librefs: **`data mycas.hello; set mycas.goodbye; run;`**
  - If the DATA step has no input table, the output table must be a CAS table that contains at least 1 variable: **`data mycas.hello; x=1; run;`**
- A caslib is a container for both the files in your personal directory space and the in-memory tables that you load from your directory.
- It is recommended that you use the CASUTIL procedure to load your SAS data sets into CAS. For more information, see "CASUTIL" in *SAS Cloud Analytic Services: Language Reference*.
- You can control where the DATA step runs by default by specifying the DSACCEL= system option. See "Controlling DATA Step Processing" on page 31 for more information.

See Also

- Chapter 1, "Accessing Data," on page 1
- "CASUTIL" in *SAS Cloud Analytic Services: Language Reference*
- SESSREF= in *SAS® Viya™ Statements: Reference*
- "Caslibs Scope" in *SAS Cloud Analytic Services: Fundamentals*

Chapter 3

Working with User-Defined Formats

Common Tasks for User-Defined Formats

Table 3.1 *Common Tasks for User-Defined Formats*

Task	Sample Syntax

Add a format to SAS and SAS Cloud Analytic Services.	`proc format lib=work.formats`
	`  casfmtlib="fmtlib1";`
	`  value fmtname`
	`  ...`
	`  ;`
	`run;`

List your format search path.	`cas casauto listfmtsearch;`
List the format libraries that are available.	`cas casauto listformats;`
Add format library to your search path.	`cas casauto fmtsearch=(fmtlib1);`
List the format names in all format libraries. *Note:* You can limit the results to a single format library with the FMTLIBNAME= option.	`cas casauto listformats members;`

Copy existing formats to a SAS Cloud Analytic Services format library.	`libname existing '/path/...';`
	
	`catname work.mycat(existing.format1`
	`  existing.format2 ...);`
	
	`proc format library=work.mycat`
	`  cntlout=temp;`
	`run;`
	
	`proc format cntlin=temp`
	`  casfmtlib='fmtlib1';`
	`run;`

Persist a format library as a SASHDAT file.	`cas casauto savefmtlib`
	`  fmtlibname=fmtlib1`
	`  caslib=casuser`
	`  table=fmtlibfile;`

Add a format library from a SASHDAT file.
Tip: By default, ADDFMTLIB appends the format library to the search path. You can specify POSITION=INSERT to add the format library to the beginning of the search path.
Note: Some sites have a large number of formats that should always be available. Administrators should use the **cas.fmtcaslib=** and **cas.addfmtlib=** server configuration file options to add those formats automatically when the server starts.
	`cas casauto addfmtlib`
	`  fmtlibname=fmtlib2`
	`  caslib=casuser`
	`  table=fmtlibfile;`

User-Defined Format Basics

SAS provides formats for controlling how variables are printed. You can use PROC FORMAT in your SAS programs to create user-defined formats. PROC FORMAT supports the creation of user-defined formats in catalogs on the SAS client and it supports adding formats and format libraries for a CAS session.

The basics for working with formats on the SAS client are described in the FORMAT procedure. The information in this section covers how to work with user-defined formats with SAS Cloud Analytic Services.

A format library has a name and can be temporary (session-scope) for the duration of your programming session. The alternative is that it can be set to have global scope and exist for as long as the server is running. You can persist a format library in a SASHDAT file and you can add formats to your CAS session from a persisted file.

Your administrator can configure the server so that the following occur when the server starts:

- format libraries are added and promoted from persisted format libraries (libraries that are persisted as SASHDAT files in a caslib's data source).
- during session start up for each connection, the format search path is updated to include the format libraries.

The following table identifies some of the ways that you might want to work with formats and summarizes methods to achieve the outcome.

Desired Outcome	Method	Format Library Scope
Formats are available to your CAS session only.	• PROC FORMAT with the CASFMTLB= option • CAS statement with the ADDFMTLIB option	Session scope
Formats are available to all CAS sessions connected to the server. Users must use FMTSEACH= in each session to add the format library to the search path.	• CAS statement with the ADDFMTLIB and PROMOTE options • CAS statement with the PROMOTEFMTLIB	Global scope
Formats are made available to all CAS sessions when the server starts. The format library is automatically added to each user's search path.	• Add **cas.fmtcaslib=** and **cas.addfmtlib=** option in the server configuration file	Global scope

Permanent Formats Supplied by SAS

In addition to user-defined formats, SAS provides formats that can be used with SAS Cloud Analytic Services. For a list of formats that are supported, see *SAS Viya Formats and Informats: Reference*.

How Are Format Libraries Different than Format Catalogs?

Format catalogs are accessed by the SAS client and are used for formatting values in procedure and DATA steps that run on the SAS client. This applies to procedures like PROC PRINT, PROC SQL, and others.

Format libraries are used within SAS Cloud Analytic Services. These are server-side formats that the server uses when an analysis is performed according to formatted values. For example, when a procedure or DATA step that runs in the server works on formatted values, the format must be available in a format library. This applies to procedures like PROC CARDINALITY, PROC BINNING, and others. These procedures are documented in *SAS Visual Data Mining and Machine Learning: Data Mining and Machine Learning Procedures*.

You must assign a format to a variable before the table is loaded into the server. After a table is in memory, you cannot assign a format. You can do this with the FORMAT statement in a DATA step or with the CASUTIL procedure.

See Also

- See add multiple format catalogs to a CAS session for an example of migrating your personal format catalogs to format libraries.
- See add a format library from a file on page 99 to see how to work with a format library after you save one as a file.
- See "Migrate User-Defined Format Catalogs to Format Libraries" in the administration documentation if your site has format catalogs that are used widely by many programmers.

Add and Save a User-Defined Format

Example

```
cas casauto sessopts=(caslib="casuser");

proc format library=work.formats casfmtlib="casformats";              /* 1 */
  value enginesize
    low - <2.7 = "Very economical"
    2.7 - <4.1 = "Small"
```

```
    4.1 - <5.5 = "Medium"
    5.5 - <6.9 = "Large"
    6.9 - high = "Very large"
;

cas casauto savefmtlib fmtlibname=casformats                          /* 2 */
   table=enginefmt replace;

proc casutil;
  format enginesize enginesize.;                                      /* 3 */
  load data=sashelp.cars casout="cars" replace;
  contents casdata="cars";
quit;

libname mycas cas;                                                    /* 4 */

proc mdsummary data=mycas.cars;
  var mpg_highway;
  groupby enginesize / out=mycas.mpg_hwy_by_size;                     /* 5 */
run;

proc print data=mycas.mpg_hwy_by_size;
  var enginesize--_mean_;
run;

/*
 * run the following CASUTIL code if you plan
 * to run the next example that adds a format
 * library from a file.
 */
proc casutil;
  save casdata="cars" casout="cars_formatted";
  droptable casdata="cars";
quit;
```

1 The FORMAT procedure is used to create a format that is named Enginesize. On the SAS client, the format is temporary and stored in the Work library. The CASFMTLIB= option is used to add the same format to your CAS session in a format library that is named Casformats.

2 The CAS statement is used with the SAVEFMTLIB option to persist the format as a SASHDAT file in the data source that is associated with the active caslib. The CASUSER caslib is typically a PATH type and uses a file system. The TABLE= option specifies the name and results in a file that is named enginefmt.sashdat.

3 The FORMAT statement is used with the CASUTIL procedure to assign the Enginesize format to the Enginesize variable. The format is applied to the in-memory instance of the Cars table. See partial results for the CONTENTS statement.

4 A CAS LIBNAME engine libref is assigned so that SAS procedures can access the Cars table that is in memory on the server. The assignment does not specify a caslib, so the active caslib is used.

5 The MDSUMMARY procedure provides descriptive statistics for the Cars table. The results are grouped by the formatted values of the Enginesize variable (five values) instead of the numeric value. As a result, the output table, Mpg_hwy_by_size, has five rows.

Results

The following display shows the partial results for the CONTENTS statement in the CASUTIL procedure. The results show that the EngineSize column uses the Enginesize format.

Figure 3.1 *Partial Results for the CONTENTS Statement of the CASUTIL procedure*

Column Information for CARS in Caslib CASUSER(					
Column	**Label**	**Type**	**Length**	**Formatted Length**	**Format**
Make		char	13	13	
Model		char	40	40	
Type		char	8	8	
Origin		char	6	6	
DriveTrain		char	5	5	
MSRP		double	8	8	DOLLAR
Invoice		double	8	8	DOLLAR
EngineSize	Engine Size (L)	double	8	15	ENGINESIZE
Cylinders		double	8	12	
Horsepower		double	8	12	
MPG_City	MPG (City)	double	8	12	
MPG_Highway	MPG (Highway)	double	8	12	
Weight	Weight (LBS)	double	8	12	
Wheelbase	Wheelbase (IN)	double	8	12	
Length	Length (IN)	double	8	12	

The following display shows the first few columns from the Mpg_hwy_by_size table.

Figure 3.2 *Selected Columns from the Mpg_Hwy_By_Size Table*

Obs	EngineSize	EngineSize_f	_Column_	_Min_	_Max_	_NObs_	_NMiss_	_Mean_
1	Very large	Very large	MPG_Highway	20	20	1	0	20
2	Large	Large	MPG_Highway	12	25	15	0	18.6
3	Medium	Medium	MPG_Highway	14	28	80	0	21.925
4	Small	Small	MPG_Highway	17	32	173	0	25.76300578
5	Very economical	Very economical	MPG_Highway	21	66	159	0	31.314465409

Key Ideas

- Use PROC FORMAT with the CASFMTLIB= option to add formats to your CAS session automatically.
- You can save all the formats in a format library in a SASHDAT file with the CAS statement and the SAVEFMTLIB option. Only caslibs with path-based data sources can be used, such as PATH, DNFS, and HDFS.
- Two ways to use formats that are saved in a SASHDAT file are as follows:
  - Add them to individual programming sessions as shown in the "Add a Format Library from a File" on page 99 example.
  - An administrator can make the formats available as a global-scope format library for all sessions by setting the **cas.fmtcaslib=** and **cas.addfmtlib=** values in the server configuration file.

Add a Format Library from a File

Prerequisites

If you ran the code in the add a user-defined format example, then run the following code to remove the format from the SAS client and the CAS session.

```
proc datasets lib=work memtype=catalog kill;
quit;

cas casauto dropfmtlib fmtlibname=casformats;
```

Example

When you use the CAS statement with the SAVEFMTLIB option, you create a SASHDAT file that includes the formats in a format library. One requirement is that you specify a caslib with a data source type that supports saving files. The data source types are PATH, DNFS, and HDFS.

This example shows the reverse of that process, adding a format library from a SASHDAT file.

```
cas casauto addfmtlib fmtlibname=fmtlib1                    /* 1 */
   caslib=casuser
   table=enginefmt;

proc casutil;                                               /* 2 */
  contents casdata="cars_formatted.sashdat";
  load casdata="cars_formatted.sashdat"
       casout="cars_formatted";

libname mycas cas;

proc mdsummary data=mycas.cars_formatted;
```

```
  var weight length;
  groupby enginesize / out=mycas.weight_len_by_size;                    /* 3 */
run;

proc print data=mycas.weight_len_by_size;
  var enginesize -- _mean_;
  by _column_;
run;
```

1 The CAS statement is used with the ADDFMTLIB option to add a format library to the CAS session. The file, Enginefmt.sashdat, is specified in the TABLE= option. The file is relative to the path for the CASUSER caslib.

2 The CONTENTS statement specifies the on-disk name, Cars_formatted.sashdat. As a result, the column information shows the format, Enginesize, that was associated with the column when the table was saved.

3 Because the Enginesize format has five values and the VAR statement specifies two columns, the MDSUMMARY procedure creates an output table, Weight_len_by_size, with 10 rows.

Results

The following display shows the results of the PRINT procedure. The rows are grouped by the two different values of _Column_.

Column=Length

Obs	EngineSize	EngineSize_f	_Column_	_Min_	_Max_	_NObs_	_NMiss_	_Mean_
1	1	Very economi	Length	143	194	159	0	176.19496855
2	8	Very large	Length	176	176	1	0	176
3	6	Large	Length	179	238	15	0	203.73333333
4	4	Medium	Length	178	230	80	0	198.325
5	3	Small	Length	150	219	173	0	188.7283237

Column=Weight

Obs	EngineSize	EngineSize_f	_Column_	_Min_	_Max_	_NObs_	_NMiss_	_Mean_
6	8	Very large	Weight	3410	3410	1	0	3410
7	1	Very economi	Weight	1850	3903	159	0	2951.163522
8	6	Large	Weight	3246	7190	15	0	4993.4
9	3	Small	Weight	2811	5086	173	0	3652.6242775
10	4	Medium	Weight	3347	5969	80	0	4398.925

List Format Libraries, Formats, and Search Path

Example

```
proc format lib=work.formats casfmtlib="casformats";                /* 1 */
  value ynm
  1='yes'
  2='no'
  3='maybe';
run;

proc format lib=work.formats casfmtlib="fmtlib2";                   /* 2 */
  value sml
  1='small'
  2='medium'
  3='large';
run;

cas casauto listformats;                                            /* 3 */

cas casauto listfmtranges fmtname=ynm;                              /* 4 */

cas casauto dropfmtlib fmtlibname=casformats;                       /* 5 */

cas casauto listfmtsearch;                                          /* 6 */
```

1. The FORMAT procedure is used to create a format that is named Ynm. The format is added to the Work.formats catalog on the SAS client. The format is also added to the CAS session in a format library that is named Casformats.
2. The FORMAT procedure is used again to create a format that is named Sml. The format is added to the Work.formats catalog on the SAS client. The format is also added to the CAS session in a format library that is named Fmtlib2.
3. The LISTFORMATS option is used to list the format libraries that are available to the CAS session. Both Casformats and Fmtlib2 are shown in the SAS log.
4. The LISTFMTSRANGES option is used with FMTNAME= to print a brief view of the format values for the Ynm format in the SAS log..
5. The DROPFMTLIB option is used to drop the Casformats format library. By default, the format library name remains in the format search path. You can specify the FMTSEARCHREMOVE option to remove the format library from the format search.
6. The LISTFMTSEARCH option is used to display the format libraries that are in the search path. In this example, because the Casformats library was dropped, but the FMTSEARCHREMOVE option was not used, the name remains in the search path. "Not Found" is used to indicate that Casformats is not available as a session-scope or global-scope format library.

SAS Log

```
NOTE: This session is in interactive mode.

 1          proc format lib=work.formats casfmtlib="casformats";
 NOTE: Both CAS-based formats and catalog-based formats will be written.
 2            value ynm
 3            1='yes'
 4            2='no'
 5            3='maybe';
 NOTE: Format library CASFORMATS added. Format search update using parameter
APPEND completed.
 NOTE: Format YNM has been output.
 6          run;
 7
 8          proc format lib=work.formats casfmtlib="fmtlib2";
 NOTE: Both CAS-based formats and catalog-based formats will be written.
 9            value sml
 10           1='small'
 11           2='medium'
 12           3='large';
 NOTE: Format library FMTLIB2 added. Format search update using parameter APPEND
completed.
 NOTE: Format SML has been output.
 13         run;
 14
 15         cas casauto listformats;
 NOTE: Fmtlib = FMTLIB2
          Scope = Session
          Fmtsearch = YES
 NOTE: Fmtlib = CASFORMATS
          Scope = Session
          Fmtsearch = YES
 16
 17         cas casauto listfmtranges fmtname=ynm;
 NOTE: Format Name = YNM
          Range = 1=yes
 NOTE: Format Name =
          Range = 2=no
 NOTE: Format Name =
          Range = 3=maybe
 NOTE: Request to LISTFMTRANGES YNM completed for session CASAUTO.
 18
 19         cas casauto dropfmtlib fmtlibname=casformats;
 NOTE: Request to DROPFMTLIB CASFORMATS completed for session CASAUTO.
 20
 21         cas casauto listfmtsearch;
 NOTE: FmtLibName = CASFORMATS
          Scope = Not Found
 NOTE: FmtLibName = FMTLIB2
          Scope = Session
 NOTE: Request to LISTFMTSEARCH completed for session CASAUTO.
```

Key Ideas

- When you use the FORMAT procedure with the CASFMTLIB= option, the format library and formats are automatically made available to your CAS session.
- You can use the LISTFMTSEARCH option for the CAS statement to display the format libraries that are in the search path.
- You can print a brief view of format values in the SAS log with the LISTFMTRANGES and FMTNAME= options. The format name that you specify must be in the search path.

Add Locale-Specific Format Libraries

Example

```
/* For each locale, copy the catalogs one at a time. */
options locale=en_US;
proc format locale library=work.formats casfmtlib="casformats";
  value abc
    1='one'
    2='two'
    3='three';
run;

options locale=fr_FR;                                                    /* 1 */
proc format locale library=work.formats casfmtlib="casformats";
  value abc
    1='un'
    2='deux'
    3='trois';
run;

/* Confirm that the format is available in each locale. */
cas casauto sessopts=(locale="en_US");                                   /* 2 */
cas casauto listfmtranges fmtname=abc;

cas casauto sessopts=(locale="fr_FR");
cas casauto listfmtranges fmtname=abc;

cas casauto listformats members;                                         /* 3 */
```

1. The OPTIONS statement is used to set the locale for the SAS client to French. The subsequent PROC FORMAT step includes the LOCALE option that specifies to create a locale-specific format in the format catalog.
2. The SESSOPTS= option is used with the CAS statement to set the CAS session locale to United States English. The subsequent CAS statement prints a brief view of the format values in the SAS log.
3. The LISTFORMATS and MEMBERS options are used to list the format catalog, Casformats, and the format names, fr_fr-abc and en_us-abc. in the SAS log.

SAS Log

```
 NOTE: This session is in interactive mode.

 1          options locale=fr_fr;
 2          proc format locale library=work.formats casfmtlib="casformats";
 NOTE: Both CAS-based formats and catalog-based formats will be written.
 3            value abc
 4              1='un'
 5              2='deux'
 6              3='trois';
 NOTE: Format library CASFORMATS added. Format search update using parameter
APPEND completed.
 NOTE: Format ABC has been written to WORK.FORMATS_fr_FR.
 7          run;
 8
 9          options locale=en_us;
 10         proc format locale library=work.formats casfmtlib="casformats";
 NOTE: Both CAS-based formats and catalog-based formats will be written.
 11           value abc
 12             1='one'
 13             2='two'
 14             3='three';
 NOTE: Format ABC has been written to WORK.FORMATS_en_US.
 15         run;
 16         cas casauto sessopts=(locale="en_US");
 NOTE: The CAS statement request to update one or more session options for
session CASAUTO completed.
 17         cas casauto listfmtranges fmtname=abc;
 NOTE: Format Name = ABC
          Range = 1=one
 NOTE: Format Name =
          Range = 2=two
 NOTE: Format Name =
          Range = 3=three
 NOTE: Request to LISTFMTRANGES ABC completed for session CASAUTO.
 18
 19         cas casauto sessopts=(locale="fr_FR");
 NOTE: The CAS statement request to update one or more session options for
session CASAUTO completed.
 20         cas casauto listfmtranges fmtname=abc;
 NOTE: Format Name = ABC
          Range = 1=un
 NOTE: Format Name =
          Range = 2=deux
 NOTE: Format Name =
          Range = 3=trois
 NOTE: Request to LISTFMTRANGES ABC completed for session CASAUTO.
 21
 22
 23         cas casauto listformats members;
 NOTE: Fmtlib = CASFORMATS
          Scope = Session
          Fmtsearch = YES
          Format = fr_fr-abc
          Format = en_us-abc
 NOTE: Request to LISTFORMAT completed for session CASAUTO.
```

Add Multiple Format Catalogs to a CAS Session

Example

The following example assumes that format catalogs are included in two locations, sasuser.formats and orion.mailfmt.

```
catname work.mycat(sasuser.formats orion.mailfmt);                    /* 1 */

proc format library=work.mycat cntlout=temp;                          /* 2 */
run;

proc format cntlin=temp casfmtlib='myfmtlib';                         /* 3 */
run;

cas casauto savefmtlib fmtlibname=myfmtlib                            /* 4 */
   caslib=casuser table=myfmtlib;
```

1. The CATNAME statement is used to combine the two catalogs into the Work.Mycat catalog.
2. The FORMAT procedure uses the combined catalog as input and creates a SAS data set on the SAS client that stores information about the formats.
3. The second PROC FORMAT step reads the SAS data set and adds a format library in your session that is named Myfmtlib.
4. You are not required to save the format library as a SASHDAT file. However, you might choose to save them if the catalogs are large and impact performance during the second PROC FORMAT step.

Promote a Format Library

Example

Example Code 3.1 *Add and Promote a Format Library*

```
proc format lib=work.formats casfmtlib="myfmtlib";                    /* 1 */
  value abc
    1 = 'yes'
    2 = 'no'
    3 = 'maybe'
    ;
run;

cas casauto promotefmtlib fmtlibname=myfmtlib replace;                /* 2 */
```

When you promote a format library, other users' sessions do not automatically add the format library to the format search path. You can reproduce another users' experience by opening a new tab in SAS Studio.

Example Code 3.2 *Add a Promoted Format Library to the Search Path*

```
/* from another tab */
cas casauto listformats;                                          /* 3 */

cas casauto fmtsearch=(myfmtlib) position=insert;

cas casauto listformats;                                          /* 4 */
```

1 The FORMAT procedure is used with the CASFMTLIB= option to add a format library named Myfmtlib to your CAS session. The format library is added with session scope.

2 The PROMOTEFMTLIB option is used to promote the format library to global scope. Other sessions that connect to the same server can access the Myfmtlib format library, but it is not automatically added to their format path.

3 The first use of the LISTFORMATS option shows that the Myfmtlib format library is available to the session, that it is not in the format search path.

4 After the FMTSEARCH= option is used to add the format library to the search path, the second use of LISTFORMATS shows that the format library has global scope and is

SAS Log

```
 NOTE: This session is in interactive mode.

 1          proc format lib=work.formats casfmtlib="myfmtlib";
 NOTE: Both CAS-based formats and catalog-based formats will be written.
 2            value abc
 3              1 = 'yes'
 4              2 = 'no'
 5              3 = 'maybe'
 6              ;
 NOTE: Format library MYFMTLIB added. Format search update using parameter
APPEND completed.
 NOTE: Format ABC has been output.
 7          run;
 8
 9          cas casauto promotefmtlib fmtlibname=myfmtlib replace;
 NOTE: Request to PROMOTEFMTLIB MYFMTLIB completed for session CASAUTO.
```

The SAS log for any session that runs the code to add the promoted format library to the search path is similar to the following.

```
NOTE: This session is in interactive mode.

1          /* from another tab */
2          cas casauto listformats;
NOTE: Fmtlib = MYFMTLIB
         Scope = Global
         Fmtsearch =
NOTE: Request to LISTFORMAT completed for session CASAUTO.
3          cas casauto fmtsearch=(myfmtlib) position=insert;
NOTE: Request to FMTSEARCH completed for session CASAUTO.
4          cas casauto listformats;
NOTE: Fmtlib = MYFMTLIB
         Scope = Session
         Fmtsearch = YES
NOTE: Fmtlib = MYFMTLIB
         Scope = Global
         Fmtsearch =
NOTE: Request to LISTFORMAT completed for session CASAUTO.
```

Key Ideas

- When you promote a format library from a programming session, the format library is available to other sessions. However, it is not automatically added to the search path for the other sessions.
- You can use the FMTSEARCH= option to add a format library to your search path.
- For frequently used formats that you want to be available for all programming sessions, you can save the format library as a SASHDAT file. Afterward, an administrator can modify the server configuration file to add the format library from the file when the server starts.