## Contents

### Chapter 1 / DATA Step Basics

- Common Tasks .................................................. 1
- The DATA Step and CAS ........................................ 2
- Running the DATA Step in CAS ................................. 3
- Processing Modes ............................................... 4
- Ordering in CAS ................................................ 8
- BY-Group Processing in CAS .................................. 13
- Language Elements Support .................................... 15
- CAS DATA Step Action ......................................... 18
- Data Types .................................................... 23
- Rules for Names ................................................ 24
- Automatic Variables ............................................ 24
- Avoiding Data Truncation ....................................... 27

### Chapter 2 / DATA Step Examples

- Set Up Code for Examples .................................... 33
- Run the DATA Step in CAS .................................... 34
- Run the DATA Step in CAS on a CAS Table ............... 35
- Load a CSV File into CAS and Save as a CAS Table .... 37
- Manage Tables .................................................. 39
- Manipulate Variables .......................................... 40
- Group Data ..................................................... 47
- Score an In-Memory CAS Table ............................... 69
- Run the DATA Step in a Single Thread .................... 79
- Redistribute a Table After Specifying the SINGLE= Option .................. 80
- View DATA Step Processing Information Using Automatic Variables .... 81
- Use the Macro Facility to Generate CAS DATA Step Code ........... 84
- DATA Step Program Walk-through ............................ 87
Contents
DATA Step Basics

Common Tasks .................................................. 2

The DATA Step and CAS ..................................... 3
Role of the DATA Step with CAS ............................. 3
Why Run the DATA Step in CAS? ............................ 4

Running the DATA Step in CAS ............................ 4
Requirements ................................................... 4
Recommendations ............................................... 5
Determining Where the DATA Step Is Running ........ 7
Controlling Where the DATA Step Runs by Default ... 7

Processing Modes .............................................. 8
Single versus Multithreaded Processing ................. 8
Distributed versus Non-Distributed Data ............... 8
Scenarios ......................................................... 9
Multithreaded Processing on Distributed Data ....... 9
Multithreaded Processing on a Single Machine .... 10
Single-threaded Processing ................................ 10
Controlling Threaded Processing ......................... 11
Inter-row Dependencies ..................................... 13

Ordering in CAS ................................................. 13

BY-Group Processing in CAS ............................... 15
How CAS Groups Data with BY Variables ............ 15
Identifying the First and Last Rows in BY-Groups .... 16
How BY Variables Affect Multithreaded DATA Step Execution 16
BY-Group Processing versus Partitioning ............ 18
Other Methods for Partitioning, Grouping, and Ordering .... 18

Language Elements Support ............................... 18
Supported Language Elements ............................. 18
Restrictions ..................................................... 19
Macro Processing ............................................. 21

CAS DATA Step Action ....................................... 23

Data Types ...................................................... 23

Rules for Names ................................................ 24

Automatic Variables ......................................... 24
Overview ....................................................... 24
_ERROR_ ......................................................... 24
_N_ ................................................................. 25
_HOSTNAME_ ................................................... 26
_NTHREADS_ .................................................... 26

Inter-row Dependencies ..................................... 13

Why Run the DATA Step in CAS? ........................ 4

Controlling Threaded Processing ......................... 11
Inter-row Dependencies ..................................... 13

BY-Group Processing versus Partitioning ............ 16
Other Methods for Partitioning, Grouping, and Ordering .... 18

Language Elements Support ............................... 18
Supported Language Elements ............................. 18
Restrictions ..................................................... 19
Macro Processing ............................................. 21

CAS DATA Step Action ....................................... 23

Data Types ...................................................... 23

Rules for Names ................................................ 24

Automatic Variables ......................................... 24
Overview ....................................................... 24
_ERROR_ ......................................................... 24
_N_ ................................................................. 25
_HOSTNAME_ ................................................... 26
_NTHREADS_ .................................................... 26
Common Tasks

The following table contains simplified code examples for performing common DATA step tasks. For more in-depth and detailed examples, see Chapter 2, “DATA Step Examples,” on page 33.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

<table>
<thead>
<tr>
<th>Task Description</th>
<th>Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run the DATA step in CAS</td>
<td><code>cas casauto; libname CAS-engine-libref cas; data CAS-engine-libref.table-name; put 'Hello from ' _hostname_ 'thread # ' _threadid_; x=1; run;</code></td>
</tr>
<tr>
<td>Load a SAS data set to CAS</td>
<td><code>cas casauto; libname SAS-engine-libref &quot;/file-path/&quot;; libname CAS-engine-libref cas; data CAS-engine-libref.table-name; set SAS-engine-libref.data-set-name; run;</code></td>
</tr>
<tr>
<td>Run the DATA step on a table in CAS</td>
<td><code>cas casauto; libname CAS-engine-libref cas; data CAS-engine-libref.table-name; set CAS-engine-libref.table-name; &lt;more-statements&gt;; run;</code></td>
</tr>
<tr>
<td>Create CHAR and VARCHAR variables</td>
<td><code>cas casauto; libname CAS-engine-libref cas; data CAS-engine-libref.table-name; length variable-name varchar(n); length variable-name $ n; run;</code></td>
</tr>
<tr>
<td>Promote a CAS table</td>
<td><code>cas casauto; libname CAS-engine-libref cas; data CAS-engine-libref.table-name(copies=1 promote=yes); set CAS-engine-libref.table-name; run;</code></td>
</tr>
</tbody>
</table>
The DATA Step and CAS

Role of the DATA Step with CAS

The DATA step can run in both Cloud Analytic Services (CAS) and in SAS. CAS is the cloud-based run-time environment that enables multithreaded, parallel DATA step execution. To understand more about what it means for the DATA step to run in CAS, see “Multithreaded Processing on Distributed Data” on page 9.

How you run the DATA step in CAS is almost identical to how you run it in SAS with the traditional DATA step. The DATA step in CAS supports most of the same language elements and provides much of the same functionality.

The DATA step serves two roles in the CAS environment:

- **Loading a SAS data set to CAS** – processing is done in SAS in a single thread in the SAS client session. A CAS engine libref is used to load the data after it has been processed in the SAS DATA step.

  ```sas
  libname mycas cas;
  data mycas.cars;
  set sashelp.cars;
  run;
  ```

- **Processing in CAS** – processing is done in CAS in multiple threads. Processing is done on an in-memory CAS table. Note that this is a separate process from loading the data and can be done only in a separate DATA step on data already loaded to CAS.

  ```sas
  libname mycas cas;
  data mycas.cars2;
  set mycas.cars;
  run;
  ```

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

You can use the CASUTIL procedure to load Excel, .csv, .txt, and other types of data to CAS. See “CASUTIL Procedure” in SAS Cloud Analytic Services: User’s Guide for more information. For an example of loading a .csv file to CAS, see “Load a CSV File into CAS” in SAS Cloud Analytic Services: User’s Guide.
Why Run the DATA Step in CAS?

Big data running in a single thread is slow. The DATA step runs faster in CAS because it can run in multiple threads, in parallel. In CAS, hundreds of threads are available, across several machines. When you program your DATA step to run in CAS, all of those threads become available for you to use to improve the performance with massively parallel processing. Parallel processing speeds the execution of big data by starting the DATA step on multiple machines and dividing processing workload among threads on these machines.

Running the DATA Step in CAS

Requirements

To run the DATA step in CAS on page 9, you must perform the following steps:

**Table 1.1 Requirements for Running the DATA Step in CAS**

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Sample Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Start a CAS session.</td>
<td>cas casauto; /* CAS session */</td>
</tr>
<tr>
<td>2 Create a CAS engine libref.</td>
<td>libname mycas cas; /* CAS engine libref */</td>
</tr>
<tr>
<td>3 Specify CAS engine librefs on both the input and output tables.</td>
<td>data mycas.random; /* CAS output table */</td>
</tr>
<tr>
<td>4 Use only CAS-supported language elements.</td>
<td>x=rand('UNIFORM'); /* CAS-supported function */</td>
</tr>
<tr>
<td>5 Ensure that there is at least one variable in the output table (CAS does not support zero-column tables).</td>
<td>run;</td>
</tr>
</tbody>
</table>

Note: Librefs in the CAS DATA step must be associated with the same CAS session.
Recommendations

Best Practices for Running the DATA Step in CAS

The following are recommended best practices for running the DATA step in CAS. To understand what it means for the DATA step to run in CAS, see "Multithreaded Processing on Distributed Data" on page 9.

- Check the log after the program runs to verify where the DATA step processed. The message "Running DATA step in Cloud Analytic Services" always appears in the log when the DATA step runs in CAS.
- Specify the MSGLEVEL=i system option to get more detailed information about where the DATA step ran.
- Specify the SESSREF= data set option if you want to ensure that the DATA step always runs in CAS.

For information about the language elements that are supported in CAS, see “Language Elements Support” on page 18.

DATA Step That Runs in SAS with an Unsupported Language Element

If all of the requirements for processing in CAS are not met, then the DATA step does not necessarily return error codes or stop executing. For example, if an unsupported language element is specified in the DATA step, but all other requirements have been met, then the DATA step defaults to running in SAS.

For example, this DATA step uses the RANUNI function, which is not supported for processing in CAS:

```sas
cas casauto;
libname mycas cas;

data mycas.randomU;
  x=ranuni(104);
run;
```

And, as the following log output indicates, the DATA step ran successfully, but it did not run in CAS:

```
NOTE: The data set MYCAS.RANDOMU has 1 observations and 1 variables.
```

DATA Step That Runs in CAS with a Supported Language Element

This DATA step uses the RAND function, which is supported for processing in CAS:

```sas
cas casauto;
libname mycas cas;
```
data mycas.random;
    x=rand('UNIFORM');
run;

And, as the following log output indicates, the DATA step ran successfully in CAS:

Running DATA step in Cloud Analytic Services:
NOTE: The DATA step has no input data set and will run in a single thread.
NOTE: The table random in caslib CASUSER(sasdemo) has 1 observations and 1 variables.

DATA Step with the MSGLEVEL=i System Option
You can specify i in the MSGLEVEL= system option to get more detailed information in the log about where the DATA step runs.

options msglevel=i;
ocas casauto;
libname mycas cas;

data mycas.randomU;
    x=ranuni(104); /* function is not supported for processing in CAS */
run;

data mycas.test;
    x = SYMGET('SYSDATE9'); /* macro is not supported for processing in CAS */
    put x;
run;

The following log output shows the extra information that is printed to the log when MSGLEVEL=i is specified:

NOTE: The data set MYCAS.RANDOMU has 1 observations and 1 variables.
28MAR2019
NOTE: The data set MYCAS.TEST has 1 observations and 1 variables

DATA Step with the SESSREF= Option
For DATA steps that you intend to always run in CAS, specify the SESSREF= option in the DATA statement to explicitly run the DATA step in CAS.

In this way, the DATA step either runs in CAS because all the requirements have been met, or it does not process at all. It also displays an error in the log. When you add this option in the DATA statement, you can be sure of where the DATA step is processing at all times. The following example uses the same unsupported DATA step function from the previous example but this time with the SESSREF= option specified in the DATA statement:
Example Code 1.1  DATA Step That Uses SESSREF= to Stop Processing When it Cannot Run in CAS

```sas
options msglevel=n;

cas casauto;
libname mycas cas;

data mycas.rand / sessref=casauto;
  x=ranuni(104);
run;
```

In this case, DATA step processing was stopped because the SESSREF= option forces the program to run in CAS rather than falling back to the SAS client for local processing. The following log output shows the error message:

```
NOTE: Running DATA step in Cloud Analytic Services.
ERROR: The function RANUNI is unknown, or cannot be accessed.
ERROR: The action stopped due to errors.
NOTE: The SAS System stopped processing this step because of errors.
```

Determining Where the DATA Step Is Running

The message "Running DATA step in Cloud Analytic Services" appears in the SAS log output when the DATA step runs in CAS.

Figure 1.1  Log Output for a DATA Step That Runs in CAS

```
NOTE: Running DATA step in Cloud Analytic Services.
NOTE: The DATA step will run in multiple threads.
```

Where the DATA step runs is based on where your tables are stored:

- If your data sets are stored in SAS, then the DATA step runs in SAS.
- If your input and output tables are stored in CAS, then the DATA step runs in CAS.
- If your tables are stored in a mix of locations, then the DATA step runs in SAS.

Note: If you do not see the message Running DATA step in Cloud Analytic Services in the SAS log, then the DATA step did not run in CAS. See “Best Practices for Running the DATA Step in CAS” on page 5 and “Requirements” on page 4 for more information.

Controlling Where the DATA Step Runs by Default

You can use the DSCAS | NODSCAS system option to control whether the DATA step runs in CAS. The default behavior is for the DATA step to run in CAS. This means that the DATA step first tries to run in CAS, and if it fails, it runs in SAS.
The DATA step fails to run in CAS if any of the requirements for running in CAS are not met. See “Running the DATA Step in CAS” on page 4 for a list of these requirements.

If you want the DATA step to run in SAS only, without first trying to run in CAS, you can specify the NODSCAS system option:

```sas
options nodscas;
```

data mycas.test; /* This DATA step runs in SAS */
   set mycas.air;
run;

Knowing and controlling where the DATA step is running is important for the following reasons:

- performance can be enhanced or degraded depending on the size of the table and where you run the DATA step. Running the DATA step in multiple threads in CAS is recommended for large tables.

- programs that rely on values retained from one row to the next (calculations that have inter-row dependencies) behave differently depending on where the DATA step runs. See “Inter-row Dependencies” on page 13.

- grouping and ordering of data is handled differently in the multithreaded CAS DATA step. Because each thread operates independently on a portion of the table, there is no global ordering of the table. See “BY-Group Processing in CAS” on page 15.

---

### Processing Modes

#### Single versus Multithreaded Processing

**Single-threaded**

the DATA step runs in SAS or in CAS in a single thread on an entire, non-distributed table. Running the DATA step in a single thread is not recommended for very large tables.

**Multithreaded (default)**

the DATA step runs in CAS in multiple threads in parallel. Each DATA step thread operates on a portion of the table. The DATA step can run in multiple threads in a single-machine (non-distributed) server or in a distributed server.

---

### Distributed versus Non-Distributed Data

**Distributed server**

a multi-node server environment in which CAS distributes the rows of a table across multiple physical machines. DATA step processing is distributed across the nodes so that the processing power of multiple CPUs running in parallel is
combined and runs on the partitioned table. This is ideal for very large tables that require a large amount of processing power.

**Single-Machine server**

A single-machine server in a non-distributed environment that is configured to run as the server. When you run the DATA step in multiple threads in a single-machine server, the processing is performed on each CPU core on that machine. This enables multithreaded, parallel DATA step execution.

For more information about CAS server architecture, see *SAS Cloud Analytic Services: Fundamentals*.

---

**Scenarios**

Scenarios for DATA step processing in CAS include the following:

- "Multithreaded Processing on Distributed Data"
- "Multithreaded Processing on a Single Machine"
- "Single-threaded Processing"

**Multithreaded Processing on Distributed Data**

A multithreaded DATA step is one in which the processing workload is shared among multiple, identical DATA steps that run concurrently on different parts of the table. In a distributed data environment, the table schema is duplicated on every node and each node holds different rows of the table.

In a distributed CAS server environment, DATA step processing is distributed across worker nodes so that the processing power of multiple CPUs running in parallel is combined. When the DATA step runs in a distributed server, it runs in multiple threads on each node. On each node, CAS divides the table into groups or blocks of data, and the DATA step runs on each group, allocating one DATA step thread per group. This is ideal for very large tables that require a large amount of processing power.
The following slideshow shows how a multithreaded DATA step behaves in a distributed CAS server.

View the slideshow in SAS Help Center.

The DATA step runs by default in multiple threads on every available node.

Note: Because the rows of a table are distributed across multiple machines and partitioned on each node in this way, each DATA step thread processes only a portion of the table. This separation of table rows might affect how some functions that have inter-row dependencies behave. See “Inter-row Dependencies” on page 13 for more information.

Multithreaded Processing on a Single Machine

In non-distributed CAS environments, a single-machine is configured to run as the CAS server. The single-machine contains multiple CPU cores to support multithreaded, parallel processing. When the DATA step runs in multiple threads on a single-machine server, the entire table is loaded onto this single machine. CAS partitions the table based on the number of cores available and each DATA step thread runs on each partition.

Splitting the table among threads in this way speeds up processing. Instead of having one DATA step thread work on the entire table, there are many DATA steps working on the table in parallel.

The following slideshow shows how the multithreaded DATA step behaves on a single-machine CAS server.

View the slideshow in SAS Help Center.

Single-threaded Processing

Single-threaded DATA step processing occurs when the DATA step runs in a single thread on an entire, non-distributed, non-partitioned table.

View the slideshow in SAS Help Center.

- Use the SINGLE= option to run the DATA step in a single thread.
- You can run the DATA step in a single thread, even when there are multiple CAS servers available. When you do this, processing is localized to a single server regardless of the number of available worker nodes.
- Single-threaded processing is useful when you need to compute sums across multiple rows. See “Sum a Variable across an Entire Table” on page 47 for an example of this.
- If you have read a table into CAS using the SINGLE=YES option or the SINGLE=NOINPUT option and you want to redistribute the table, you can use the Partition action to redistribute the table across all available worker nodes. See “Redistribute a Table After Specifying the SINGLE= Option” on page 81.
Running the DATA step in a single thread is not recommended for very large tables.

CAUTION
When you run the DATA step in a single thread in CAS, processing is localized to a single machine regardless of the number of available worker nodes in your environment. If a single-threaded DATA step processes large tables, then not only does the program run slowly, the machine executing the single thread can run out of disk or memory resources. Running the DATA step in a single thread is not recommended for very large tables.

Controlling Threaded Processing

Default Processing
A DATA step that runs in CAS, runs in all available threads by default. If the DATA step has no input table, then it runs in a single thread.

Example Code 1.2 The DATA Step Runs in Multiple Threads by Default When There Is an Input Table

data mycas.test;
  set mycas.test;
  put 'The DATA step is running in ' _nthreads_ ' threads';
run;

Output 1.1 Log Output

NOTE: Running DATA step in Cloud Analytic Services.
NOTE: The DATA step will run in multiple threads.
The DATA step is running in 192 threads

Example Code 1.3 A DATA Step with No Input CAS Table Runs in a Single Thread

data mycas.test;
  put 'The DATA step is running in ' _nthreads_ ' thread';
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.
The DATA step is running in 1 thread

The number of available threads depends on the number of active server nodes and CPU cores available on each node. The DATA step typically runs one DATA step thread per core.

For example, if your CAS configuration consists of four worker nodes and each node supports 32 threads (or cores), then the DATA step runs in 128 threads. The number of DATA step threads equals the number of nodes multiplied by the number of threads running on each node.
Controlling the Number of Threads

You can control whether the DATA step runs in a single thread or in all available threads by specifying the SINGLE= option in the DATA statement.

- **SINGLE=YES** specifies that the DATA step runs in a single thread.
  
  ```
  data mycas.class / single=yes;
  set mycas.class;
  run;
  ```

- **SINGLE=NO** specifies that the DATA step runs in all available threads. This is the default behavior as long as there is an INPUT CAS table specified:

  ```
  data mycas.class / single=no;
  set mycas.class;
  run;
  ```

- **SINGLE=NOINPUT** causes a DATA step without input tables to run in a single thread instead of in all available threads.

  NOINPUT is the default value for a DATA step that is running in CAS with no input tables when you do not specify the SESSREF= option.

- **THREADS=** specifies the number of threads that the DATA step runs in on each node. A value of zero means that the DATA step runs in 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.

If you have read a table into CAS using the SINGLE=YES option or the SINGLE=NOINPUT option and you want to redistribute the table to all worker nodes, you can use the Partition action, as shown below. For a more detailed example of redistributing a table, see "Redistribute a Table After Specifying the SINGLE= Option" on page 81.

```python
proc cas;
  table.partition / table="cars2",
    casout={name="cars", replace=true},
    pickerthreads=1;
run;
quit;
```

**CAUTION**

When you run the DATA step in a single thread in CAS, processing is localized to a single machine regardless of the number of available worker nodes in your environment. If a single-threaded DATA step processes large tables, then not only does the program run slowly, the machine executing the single thread can run out of disk or memory resources. Running the DATA step in a single thread is not recommended for very large tables.
Inter-row Dependencies

A multithreaded DATA step can reduce run time and improve performance. But 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 has inter-row dependencies because it retains a value from one row to the next. But, in a multithreaded DATA step, the rows in a table are not necessarily processed together in the same DATA step thread, nor is their order guaranteed. For more information about ordering, see “Ordering in CAS” on page 13.

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 inter-row dependent operations:

- RETAIN statement
- DIF and LAG functions
- .FIRST and .LAST automatic DATA step variables (see “How SAS Identifies the Beginning and End of a BY Group” in SAS DATA Step Statements: Reference)
- Sum statement
- Temporary arrays

The calculations performed by these types of statements are often referred to as full-table operations. Full-table operations require reading the entire contents of a table as a whole, rather than just reading selected portions.

Full-table operations can be performed in two steps: the first step involves performing a local operation, such as computing a partial sum. The second step involves performing a global operation, such as computing the final sum from a series of partial sums.

This two-step operation works well because it uses the processing power of all the threads to process the entire table and a single thread to process the smaller table subset.

For an example of this type of program, see Sum a Variable Across an Entire Table. on page 47

For more information about temporary arrays, see ARRAY Statement.

Ordering in CAS

In SAS, the order of variables in a data set is based on the order in which the variables are listed in the input data set. This implicit ordering of data by the DATA step does not occur in CAS. In CAS, there is no guarantee of ordering of data.

For example, suppose that you have the following SAS data set:
This data set is then read in by the following DATA step:

**Example Code 1.4  DATA Step in SAS**

```sas
data out;
  set in;
  by var1;
  if first.var1 and var2 = 'N';
run;
proc print data=out; run;
```

The DATA step reads the rows as they are listed in the input. Here are the results for the Example Code 1.4 on page 14 shown above:

<table>
<thead>
<tr>
<th>var1</th>
<th>var2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>N</td>
</tr>
</tbody>
</table>

Notice that there is one row of output that meets the conditions where "FIRST. and LAST. DATA Step Variables" in SAS Language Reference: Concepts FIRST. and LAST. DATA Step Variables (where first.var1=1 and var2='N').

When you read in this data set using the DATA step in CAS, CAS also groups the rows based on var1. But, the order of the rows within the BY group can vary. This variance causes the DATA step to produce different results each time that it is executed.

For example, the CAS DATA step might read the rows in this order:

<table>
<thead>
<tr>
<th>var1</th>
<th>var2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
</tr>
</tbody>
</table>

And, this produces different results than SAS. With this order, there is one output row in which var1=2 and var2='N'. Here are the results for the same DATA step in CAS:

<table>
<thead>
<tr>
<th>var1</th>
<th>var2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>N</td>
</tr>
</tbody>
</table>
One way to ensure stable ordering of data in this case is to add a sequence number to the rows as they are loaded into CAS.

*Example Code 1.5  DATA Step in SAS (Loads Data to CAS)*

```sas
data cas.mydata;
  set sas.mydata;
  seqno = _N_;  
run;
```

This DATA step produces the following table in CAS:

<table>
<thead>
<tr>
<th>var1</th>
<th>var2</th>
<th>seqno</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Y</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>N</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>N</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>6</td>
</tr>
</tbody>
</table>

You then specify the sequence number in the BY statement. When you do this, the DATA step groups the rows by `var1`, and then it orders the rows, first by `var1`, and then by `seqno`. This causes the rows to be read in the same order in CAS as in SAS.

*Example Code 1.6  DATA Step in CAS*

```sas
data cas.out;
  set cas.mydata;
  by var1 seqno;
  if first.var1 and var2 = 'N';
run;
```

To see an example of how you can add a unique sequence number to rows in a CAS table, see “Add a Unique Row Identifier” on page 51.

---

**BY-Group Processing in CAS**

**How CAS Groups Data with BY Variables**

When the DATA step runs in a distributed 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 of the BY variables. When you use the BY statement in a distributed server, CAS groups the rows based on the first BY variable. When CAS distributes the table across multiple nodes, it keeps the BY groups intact. That is, rows that share the same BY variables are stored together on a single machine.
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, then orders the rows within each BY group based on all the BY variables.
- On the CAS server, there is no guarantee of global ordering between BY groups. Each DATA step thread can group and order only the rows that are in the same data partition.
- If a format is applied to the first BY variable, then CAS creates BY groups 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 creates BY groups 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.
- 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. This is because CAS groups and distributes rows based on the format of the first BY variable. If a high cardinality format is applied to the first BY variable, then more threads are used. Conversely, if a low cardinality format is applied to the first BY variable, then fewer threads are used.

See the example in "How BY Variables Affect Multithreaded DATA Step Execution" on page 16 for an illustration of low and high-cardinality BY-groups.

For information about BY-Group processing in SAS, see “BY-Group Processing in the DATA Step” in SAS Language Reference: Concepts.

Identifying the First and Last Rows in BY-Groups

In the DATA step, SAS identifies the beginning and end of each BY group by creating two temporary variables. One temporary variable is created for the first row of the BY group and another temporary variable is created for the last row of the BY group. CAS creates a FIRST. and LAST. variable for every node. These variables can be used to manipulate how data is grouped, subdivided, and ordered.

Here are two examples that use the .FIRST variable to group rows in a table:

- "Group and Order Rows Using the DATA Step BY Statement" on page 74
- "Group Data by Two Variables" on page 77

For information about how the DATA step identifies BY groups and how you can manipulate them, see FIRST and LAST DATA Step Variables.

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:

<table>
<thead>
<tr>
<th>Formats with a High Cardinality</th>
<th>Formats with a Low Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATEw.</td>
<td>QTRw.</td>
</tr>
<tr>
<td>DATETIMEw.d</td>
<td>MONNAME.</td>
</tr>
<tr>
<td>DDMMYYw.</td>
<td>WEEKDAYw.</td>
</tr>
</tbody>
</table>

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 full code used to create this column is not shown in the image. To see the full example, see "View DATA Step Processing Information Using Automatic Variables" on page 84.
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 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 can use the partition action. See “Partition a Table Using the Partition Action” in SAS Viya: System Programming Guide 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:

- PARTITION action
- ORDERBY= and GROUPBY= options in the PROC CASUTIL LOAD statement
- PARTITION= and ORDERBY= data set options in the SAS DATA statement

For an example of using the BY statement as described in this section, see “Group and Order Rows Using the DATA Step BY Statement” on page 74.

Language Elements Support

Supported Language Elements

Not all SAS language elements are supported in CAS. Below are links to the reference documentation for the SAS DATA Step language. Each document contains a “By Category” table that lists the CAS supported language elements under the CAS table entry:

- “Data Set Options by Category” in SAS Data Set Options: Reference
Support for CAS is also listed on the individual language element syntax pages for all the DATA step language elements. The support is indicated the categories field as shown in the example document below:

**Figure 1.3  Example Documentation Showing CAS Support on Individual Syntax Page**

For CAS-specific DATA step language elements, see *SAS Cloud Analytic Services: User’s Guide*.

**Restrictions**

The following table contains restrictions on the use of some DATA step language elements in CAS:
<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>APPEND=</strong> data set option</td>
<td>You cannot set both the PROMOTE= and the APPEND= data set options to YES at the same time (in the same DATA step). You can set either PROMOTE= or APPEND= to YES. An error results if you set both to YES at the same time. For more information, see “APPEND= Data Set Option” in <em>SAS Cloud Analytic Services: User’s Guide</em>.</td>
</tr>
<tr>
<td><strong>BY variables</strong></td>
<td>BY variables are supported in the DATA step in CAS but there are some differences between how SAS performs group variables and how CAS groups them. See “Ordering in CAS” on page 13 and “BY-Group Processing in CAS” on page 15 for more information.</td>
</tr>
<tr>
<td><strong>BY statement, DESCENDING option</strong></td>
<td>The DESCENDING option in the BY statement is not supported on the first variable in a DATA step that is running in CAS. The DESCENDING option in the BY statement is supported in the second and subsequent variables in a CAS DATA step.</td>
</tr>
<tr>
<td><strong>FORMAT procedure FOR MAT procedure</strong></td>
<td>User-defined formats that are 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 DATA step fails before it tries to run in CAS. For more information, see “FORMAT Procedure” in <em>Base SAS Procedures Guide</em>.</td>
</tr>
</tbody>
</table>
| Formats | These formats are not supported in a DATA step that is running in CAS:  
- **WORDS**  
- **WORDF**  
- **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 | Not all functions are supported in a DATA step that is running in CAS. See *Functions and CALL Routines by Category* for a list of supported functions. |
| **Functions and Session Encoding** | The default encoding method for sessions in CAS is UTF-8. In a UTF–8 environment, data can contain multibyte characters that are equivalent to more than 1 byte. Some 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 “About Session Encoding” on page 27 for information about how session encoding affects behavior. For an example showing how some functions behave, see “Index CHAR and VARCHAR Character Strings” on page 62 for an example of this behavior. |
### Feature Description

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INFILE, INPUT, and DATALINES statements</strong></td>
<td>These language elements are not supported in a DATA step that is processing in CAS. You can use these statements in a SAS client session and then upload the results to CAS.</td>
</tr>
<tr>
<td><strong>Macros</strong></td>
<td>The SYMGET and SYMPUT functions are not supported in a DATA step that is running in CAS. For more information about macro processing in SAS, see “Macro Processing” on page 21.</td>
</tr>
<tr>
<td><strong>PROMOTE= data set option</strong></td>
<td>You can set either PROMOTE= or APPEND= to YES. An error results if you set both to YES at the same time.</td>
</tr>
<tr>
<td><strong>Statements:</strong> DATALINES INFILE INPUT MODIFY REMOVE REPLACE</td>
<td>These statements are not supported in a DATA step that is running in CAS. You cannot use them to modify an in-memory CAS table. You can use these statements in a SAS client session and then upload the results to CAS.</td>
</tr>
<tr>
<td><strong>VARCHAR data type</strong></td>
<td>There are some restrictions on using the VARCHAR data type. See “Data Types” in SAS Cloud Analytic Services: User’s Guide for more information.</td>
</tr>
<tr>
<td><strong>SAS Views</strong></td>
<td>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 SAS Cloud Analytic Services. For more information, see “Create a DATA Step View” on page 58 and “VIEW=view-name” in SAS DATA Step Statements: Reference.</td>
</tr>
<tr>
<td><strong>WHERE= data set option</strong></td>
<td>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. Specify the MSGLEVEL=i option to see more processing information in the log. You can specify the SESSREF= data set option to ensure that the DATA step always runs in CAS.</td>
</tr>
<tr>
<td><strong>zero-column tables</strong></td>
<td>CAS does not support zero-column tables. This is different from the DATA step in SAS, which does support zero-column tables. You will get an error if you try to create a table in CAS that has no variables.</td>
</tr>
</tbody>
</table>

---

## Macro Processing

### Support for Macro Processing

Macro code itself does not execute on the CAS server. It is precompiled and runs in the SAS client session. The code that the macro generates can be made to run in...
CAS, however. See “Examples” on page 22 for examples that show how you can use the SAS macro facility in SAS Viya.

The following DATA Step Functions for Macros are not supported in a DATA step that is running in CAS:

- SYMGET
- SYMGETN
- DOSUBL

The following DATA Step CALL Routines for Macros are not supported in a DATA step that is running in CAS:

- CALL EXECUTE Routine
- CALL SYMPUT Routine
- CALL SYMPUTN Routine
- CALL SYMPUTX Routine

**IMPORTANT** SAS does not necessarily return an error code or stop processing if you specify an unsupported language element, such as a SAS macro, in a CAS DATA step. Instead, the DATA step processes in the SAS client and sends the results to the CAS server. If you do not see the message *Running DATA step in Cloud Analytic Services* in the log, then the DATA step did not run in CAS.

---

**Note:** For more information about knowing where the DATA step is running, see *Recommendations For Running the DATA Step in CAS* on page 5.

---

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

**Examples**

You can use SAS macros in a variety of ways with the CAS server environment. For example, you can create macros that generate SAS code that executes in CAS. But, the macro code itself does not execute on the server.

Here are some examples showing how to use macros to execute the CAS procedure, the CASUTIL procedure, and the CAS DATA step.

**Example Code 1.7** Create a Macro for Uploading a Sashelp Data Set into CAS

```
%macro loadit(name=);
proc casutil;
  load data=sashelp.&name	noutcaslib='casuser'
    replace;
run; quit;
%mend loadit;

%loadit(name=cars)
```

**Example Code 1.8** Create a Macro for Saving a Sashelp Data Set in CAS

```
%macro loadsaveit(name=);
```

---
proc casutil;
  load data=sashelp.&name outcaslib='casuser'
    replace;
  save casdata="&name" replace;
run; quit;
%mend loadsavewithit;

%loadsavewithit(name=cars)

**Example Code 1.9**  
Create a Macro for Dropping a Table from Memory

```
%macro dropit(name=);
proc casutil incaslib="casuser";
  droptable casdata="&name";
  list files;
run; quit;
%mend dropit;

%dropit(name=cars)
```

**Example Code 1.10**  
Create a Macro for Getting Server Information Using the serverStatus Action

```
%macro aboutServer;
  serverStatus;
%mend;
proc cas;
  %aboutServer;
run;
```

---

**CAS DATA Step Action**

CAS actions are requests that are sent by the user to the server to perform a task. You can use CAS actions to submit a DATA step. CAS actions are submitted within a PROC CAS block.


---

**Data Types**

Rules for Names

For information about rules for names in CAS, see "Variable Names and Data Set Names in CAS Engine" in SAS Cloud Analytic Services: User’s Guide.

Automatic Variables

Overview

Automatic variables are system variables that are created automatically by the SAS DATA step and saved in memory. 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. For an example that uses automatic variables, see "View DATA Step Processing Information Using Automatic Variables" on page 84.

The following automatic variables are created by every DATA step that runs in a SAS client session or in CAS.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

_ERROR_

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

_ERROR_ is initialized to 0. It is set to 1 when an error is encountered. 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 to 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:

```
data mycas.test;
  input x y;
  if _error_ = 1 then
    put "****** Error in row " _n_ " ******";
  datalines;
  1 1
  2 3
  3 g
  4 4
```
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.

_N_

is an internal system variable that counts the iterations of the DATA step as it automatically loops through the rows of an input data set. The _N_ variable is initially set to 1 and increments by 1 each time the DATA step loops past the DATA statement. The DATA step loops past the DATA statement for every row that it encounters in the input data. Because the DATA step is a built-in loop that iterates through each row in a table, the _N_ variable can be used as a counter variable.

Note: The DATA step creates and stores the _N_ automatic variable internally but it does not include the variable in the output 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 server session, _N_ returns the number of DATA step iterations (table rows) per worker node, per thread. For example, suppose that an eight-row table is distributed across four worker nodes. If each node processes 2 row in a single thread, then the log displays all rows processed on each node:

```sas
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
_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.

```sas
data mycas.bigcars(where=(weight>5000));
  set sashelp.cars;
  keep make weight;
run;

data mycas.bigcars;
  set mycas.bigcars;
  keep make weight;
  put _hostname_;
run;
```

NOTE: Running DATA step in Cloud Analytic Services.
NOTE: The DATA step will run in multiple threads.
server006
server005
server007

NOTE: Duplicate messages output by DATA step:
server006  (occurred 3 times)
server005  (occurred 3 times)
server007  (occurred 2 times)
...

NOTE: There were 21 observations read from the table BIGCARS in caslib casuser(user-id).
NOTE: The table bigcars in caslib CASUSER(user-id) has 21 observations and 2 variables.

_NTHREADS_

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

```sas
data mycas.bigcars(where=(weight>5800));
  set sashelp.cars;
  keep make weight;
run;

data mycas.bigcars;
  set mycas.bigcars;
  put "The DATA step is running in " _nthreads_ "threads";
run;
```

NOTE: Running DATA step in Cloud Analytic Services.
NOTE: The DATA step will run in multiple threads.
The DATA step is running in 256  threads
NOTE: Duplicate messages output by DATA step:
The DATA step is running in 256  threads  (occurred 5 times)
_THREADID_

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

```sas
data mycas.bigcars(where=(weight>5800));
  set sashelp.cars;
  keep make weight;
run;

data mycas.bigcars;
  set mycas.bigcars;
  if _N_ = 1 then do;
    put "The ThreadID is " _threadid_ ;
  end;
run;
```

**NOTE:** Running DATA step in Cloud Analytic Services.
**NOTE:** The DATA step will run in multiple threads.
The ThreadID is 65
The ThreadID is 33
The ThreadID is 97
The ThreadID is 129
The ThreadID is 1

---

### Avoiding Data Truncation

#### About Session Encoding

If your SAS session uses an encoding method other than UTF-8, SAS automatically transcodes data that you move from SAS to CAS, and vice versa. Transcoding is the automatic process by which SAS converts data from one session encoding to another.

The CAS server uses UTF-8 session encoding. Your local SAS session can use any number of different encoding methods depending on your locale and the settings that are configured for your site. The default SAS session encoding is WLatin1 for Windows and Latin1 for UNIX.

The number of bytes for a character in your local SAS session encoding can be different from the number of bytes for the same character in CAS. For example, the characters é and è in a Latin1 session encoding require only one byte for storage but in UTF-8 encoding these same characters require 2 bytes for storage. This is because UTF-8 is a multi-byte character set (MBCS) while Latin1 is a single-byte character set (SBCS). Therefore, transcoding can result in data truncation.
Complete information about session encoding and how you can manage transcoding issues in SAS can be found in here:


Specific information about the VARCHAR data type can be found in “VARCHAR Data Type” in SAS Cloud Analytic Services: User’s Guide.

Determine the Session Encoding

To determine session encoding for your SAS session, specify the PROC OPTIONS statement:

```sas
proc options option=encoding; run;
```

The session encoding information is printed in the SAS log.

To determine session encoding for a specific SAS data set, specify the CONTENTS procedure:

```sas
proc contents data=sashelp.manage; run;
```

Examples of Truncation Errors

Overview

Truncation of the data can occur if your SAS session encoding is not UTF-8.

When you declare a variable in a CAS DATA step, the length of the variable is calculated in the SAS session then sent to CAS for processing. CAS creates a table with columns based on the variable length information from SAS. Truncation of the data can occur if your local SAS session is not compatible with the UTF-8 CAS session. Or, you might get a transcoding error in the SAS log.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

Example 1

In the following example, the local SAS session encoding is EUC-CN (Chinese) and the DATA step runs in CAS. The code is first partially compiled in the SAS client where information about the length of the variables is determined and then sent to CAS for processing. CAS creates a table based on these column widths.

In this example, the EUC-CN character set requires 2 bytes per character of storage in SAS. The length of the CHAR variable, c, is automatically set to 8 bytes (to enable saving 4 EUC-CN characters). SAS sends this length information to CAS and CAS creates a table column with a width of 8 bytes. When the DATA step runs in CAS, the value 我的例子 is assigned to the variable c. Truncation occurs because
我的例子 requires 3 bytes per character in UTF-8 (12 bytes total) and the variable has a length of only 8 bytes.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```sas
data mycas.example;
  c = '我的例子';
run;
```

**Solution:** Explicitly define the length of character variables to ensure that they are long enough to accommodate the lengths when they are converted to UTF-8 encoding in CAS. If your local session encoding uses a DBCS, then set the CHAR length to be equal to the original length times 2. In the solution below, the length of the CHAR is set to 16 bytes, enough to hold 4 multibyte characters.

```sas
data mycas.example;
  length c $ 16; /* in UTF-8 this variables might */
  c = '我的例子'; /* require up to 4 bytes per character */
run;
```

### Example 2

In the following example, the DATA step is running in CAS. The CHAR variable contains a Latin1 character with diacresis. The Latin1 character, ä, requires 1 byte in Latin1 and 2 bytes in UTF-8. SAS compiles the DATA step in the local SAS session, automatically declaring the variable c to be 4 bytes long, and sends information about the variable lengths to CAS. The value äbcd uses 5 bytes in UTF-8 in CAS. When this value is assigned to variable c, which was automatically declared to be 4 bytes long, truncation occurs.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```sas
data mycas.example;
  length c $4;
  c = 'äbcd'; /* length in Latin1 is 4 bytes and 5 bytes in UTF-8 */
run;
```

```
proc print data=mycas.example; run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>äbc</td>
</tr>
</tbody>
</table>

**Solution:** Increase the length of character variables to ensure that they are long enough to accommodate the lengths when they are converted to UTF-8 encoding in CAS.

```sas
data mycas.example2;
  length c $5;
  c = 'äbcd'; /* length in Latin1 is 4 bytes */
run;
```

```
proc print data=mycas.example2; run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>äbcd</td>
</tr>
</tbody>
</table>
**Solution:** Change the variable type to VARCHAR. VARCHAR variables are declared in units of characters instead of bytes, making it simpler to work with multibyte characters.

```plaintext
data mycas.example2;
  length vc varchar(*);
  vc = 'äbcd'; /* length in CAS UTF-8 is 5 */
run;
```

```plaintext
proc print data=mycas.example2;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>vc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>äbcd</td>
</tr>
</tbody>
</table>
```
```

**Note:** The NCHARMULTIPLIER=, TRANSCODE_FAIL, and ENCODING= data set options are not supported in a DATA step that runs in CAS. However, they can be used effectively in a DATA step that is loading to CAS to manage transcoding errors.

---

### Example 3

When you save a CAS table that contains a VARCHAR variable as a SAS data set, SAS automatically converts the variable to a CHAR variable. You will see the following in the log.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

**NOTE:** One or more variables were converted because the data type is not supported by the V9 engine.

If you load a SAS data set to CAS, you might get a transcoding error due to incompatible session encodings between SAS and CAS. When you transfer SAS data sets to CAS, SAS converts the data in your local SAS session into UTF-8 data for the CAS session.

```plaintext
ERROR: Some character data was truncated during transcoding in the dataset MYCAS.CITIES. Use of the NCHARMULTIPLIER option is recommended.
```

For example, the following program might cause an error if the local SAS session encoding is Latin1:

```plaintext
data cities;
  city = "Sévérac-le-Château"; /* 15 Latin1 characters totaling 15 bytes */
run;
```

```plaintext
data mycas.cities;
  set cities;
run;
```

**Solution:** Specify the NCHARMULTIPLIER= data set option on the output data set. This option increases the length of the transcoded variables.

```plaintext
data mycas.cities(NCHARMULTIPLIER=2);
  set cities;
run;
```
VARCHAR Length Conversion

Support

VARCHAR variables are not supported by the SAS V9 engine, so SAS automatically converts VARCHAR variables to CHAR variables.

If you try to store a data containing a VARCHAR in anything other than a CAS table, SAS automatically converts the VARCHAR variables to CHAR variables. When converting a variable from a VARCHAR(n) to a CHAR, SAS sets the length of the converted VARCHAR based on your local SAS session encoding. To calculate the length, SAS multiplies the current length of the VARCHAR, n, by the maximum value that a character’s length can be in the local SAS session encoding.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

Example 1

In this example, assume that the local SAS session encoding is Latin1.

```sas
proc options option=encoding; /* determine the encoding for your local session */ run;
data mycas.example;
  length c $10;
  length vc varchar(4);
  c = 'abcd';
  vc = 'abcd';
run;
proc contents data=mycas.example; run;
```

Here is the Log output for the OPTIONS procedure:

```
ENCODING=SQLATIN1 Specifies the default character-set encoding for the SAS session.
NOTE: PROCEDURE O
```

Here is the PROC CONTENTS output for the original CAS table:

```
data example;
  set mycas.example;
run;
proc contents data=example; run;
```

Here is the PROC CONTENTS output for the SAS data set with the VARCHAR variable converted to a CHAR variable. Notice how the length stays the same when
the local session encoding is Latin1. SAS calculates the length of the converted VARCHAR variable by multiplying the original length by the maximum number of bytes required to store characters in the character set. Latin1 is a single-byte character set, so the length is 4 x 1, or 4 bytes.

Example 2

For this example, use the program in “Example 1” on page 31 and assume that the local session encoding is UTF-8. When the VARCHAR variable is converted to a CHAR variable, the length of the CHAR variable is 4 times the length of the largest possible character in UTF–8. The largest possible number of bytes required to store a single character in UTF-8 is 4. So, the length is 4 x 4, or 16 bytes.

Here is the output when above is run in a local UTF-8 SAS session:

<table>
<thead>
<tr>
<th>Alphabetical List of Variables and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

More Information about SAS Session Encoding

- SAS® and UTF-8: Ultimately the Finest
- Multilingual Computing with SAS® 9.4
- Usage Note 51586: How to change the default SAS® session encoding to UTF-8
# DATA Step Examples

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set Up Code for Examples</strong></td>
<td></td>
</tr>
<tr>
<td>Set Up Code</td>
<td>34</td>
</tr>
<tr>
<td>User Interfaces</td>
<td>35</td>
</tr>
<tr>
<td><strong>Run the DATA Step in CAS</strong></td>
<td></td>
</tr>
<tr>
<td>Example Code</td>
<td>35</td>
</tr>
<tr>
<td>Key Ideas</td>
<td>36</td>
</tr>
<tr>
<td>See Also</td>
<td>36</td>
</tr>
<tr>
<td><strong>Run the DATA Step in CAS on a CAS Table</strong></td>
<td></td>
</tr>
<tr>
<td>Example Code</td>
<td>37</td>
</tr>
<tr>
<td>Key Ideas</td>
<td>38</td>
</tr>
<tr>
<td>See Also</td>
<td>38</td>
</tr>
<tr>
<td><strong>Load a CSV File into CAS and Save as a CAS Table</strong></td>
<td></td>
</tr>
<tr>
<td>Example Code</td>
<td>39</td>
</tr>
<tr>
<td>Key Ideas</td>
<td>40</td>
</tr>
<tr>
<td>See Also</td>
<td>40</td>
</tr>
<tr>
<td><strong>Manage Tables</strong></td>
<td>40</td>
</tr>
<tr>
<td>Convert a CAS Table to a SAS Data Set</td>
<td></td>
</tr>
<tr>
<td>Combine Tables Using the SET and BY Statements</td>
<td>42</td>
</tr>
<tr>
<td>Filter and Append Rows to a Promoted Table</td>
<td>45</td>
</tr>
<tr>
<td><strong>Manipulate Variables</strong></td>
<td>47</td>
</tr>
<tr>
<td>Sum a Variable across an Entire Table</td>
<td></td>
</tr>
<tr>
<td>Group and Sort Variables in Descending Order</td>
<td>49</td>
</tr>
<tr>
<td>Add a Unique Row Identifier</td>
<td>51</td>
</tr>
<tr>
<td>Create a VARCHAR Variable Using the LENGTH Statement</td>
<td>54</td>
</tr>
<tr>
<td>Impute Missing Values Using the LAG Function</td>
<td>56</td>
</tr>
<tr>
<td>Create a DATA Step View</td>
<td>58</td>
</tr>
<tr>
<td>Create a VARCHAR Using an ARRAY Statement</td>
<td>61</td>
</tr>
<tr>
<td>Index CHAR and VARCHAR Character Strings</td>
<td>62</td>
</tr>
<tr>
<td>Implicit Variable Type Conversion</td>
<td>64</td>
</tr>
<tr>
<td>Create a DATA Step Hash Object with VARCHAR Key and Data Variables</td>
<td>66</td>
</tr>
<tr>
<td><strong>Group Data</strong></td>
<td>69</td>
</tr>
<tr>
<td>Group and Order a CAS Table Using the PARTITION= Data Set Option</td>
<td></td>
</tr>
<tr>
<td>Group and Sort Variables in Descending Order</td>
<td>71</td>
</tr>
<tr>
<td>Group and Order Rows Using the DATA Step BY Statement</td>
<td>74</td>
</tr>
<tr>
<td>Group Data by Two Variables</td>
<td>77</td>
</tr>
<tr>
<td><strong>Score an In-Memory CAS Table</strong></td>
<td>79</td>
</tr>
<tr>
<td>Example Code</td>
<td>79</td>
</tr>
<tr>
<td>Key Ideas</td>
<td>80</td>
</tr>
<tr>
<td>See Also</td>
<td>80</td>
</tr>
</tbody>
</table>
Set Up Code for Examples

Set Up Code

If you are running SAS from the SAS Windowing Environment (Display Manager), then start with step 3 below.

1 Open SAS Studio from the URL in the form of http://hostname:port.

2 Sign in using your user ID and password for your operating system account.

3 In the Code Editor window, run the following program:

```sas
options cashost="cloud.example.com" casport=5570            /*1*/
cas casauto sessopts=(caslib='casuser');                    /*2*/
libname mycas cas;                                          /*3*/
caslib _all_ assign;                                        /*4*/
```

1 In the CASHOST= system option specify the host name of your CAS server in quotation marks. Specify the port number that the server is running on in the CASPORT= system option.

Note: If you are running SAS from SAS Studio, which connects to CAS automatically, then you might not need to manually connect to the CAS server and you can omit this OPTIONS statement.
2 Start a CAS session named Casauto and specify the personal caslib, Casuser, as the active caslib.

3 If you have not already done so, use the CAS LIBNAME statement to create a CAS engine libref.

4 Specify _ALL_ ASSIGN in the CASLIB statement to assign SAS librefs to existing caslibs. Assigning SAS librefs to caslibs enables you to see caslib names in the Libraries pane of the SAS Studio application, or, if you are using the SAS Windowing environment, then in the active libraries Explorer window.

**TIP** Specify the SESSREF= option option in the DATA statement to explicitly run the DATA step in CAS. Also, specify 1 in the MSGLEVEL= system option to get more detailed log results.

For more information about running the DATA step in CAS, see “Running the DATA Step in CAS” on page 4.

For more information about connecting to CAS, see the following resources:

- Getting Started with CASL Programming
- “Connecting to the CAS Server and Creating a CAS Session” in An Introduction to SAS Viya Programming

---

**User Interfaces**

Connection with the CAS server is automatically created when you use SAS Studio 5.2 on Linux and Windows, which is the web programming interface for SAS Viya. The connection information, the host name and port, are set during installation. You can view those values in the CASHOST and CASPORT system options using this code:

```
cas casauto;
```

---

**Run the DATA Step in CAS**

**Example Code**

The following example shows how to run the DATA step in CAS.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```
data mycas.hello; /*1*/
   put 'Hello from ' _hostname_ /*2*/
        'thread #' _threadid_; /*3*/
   x=1; /*4*/
run;
```
A CAS engine libref is specified in the DATA statement to create an in-memory CAS table named mycas.hello.

The _HOSTNAME_ automatic DATA step variable is specified in a PUT Statement to demonstrate how the DATA step runs in CAS in a distributed environment.

The _THREADID_ automatic DATA step variable is specified to demonstrate multithreaded processing in CAS.

CAS tables require at least one variable. An assignment statement is used to create a variable.

Output 2.1  Log Output Showing Multithreaded DATA Step Processing in a Distributed CAS Environment

```
NOTE: Running DATA step in Cloud Analytic Services.
Hello from server001 thread #2
Hello from server001 thread #3
Hello from server001 thread #4
Hello from server001 thread #1
NOTE: The table hello in caslib CASUSER(username) has 4 observations and 1 variables...
```

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. For information about controlling DATA processing, see “Controlling Threaded Processing” on page 11.
- For a DATA step to run in CAS, you must make sure you have met the requirements for processing in CAS on page 4.
- You can also use the CASUTIL procedure to load your SAS data sets into CAS. This method is usually more efficient for transferring large data sets to CAS. For more information, see “CASUTIL Procedure” in SAS Cloud Analytic Services: User’s Guide for more information.
- Not all language elements are supported in a CAS DATA step. See “Language Elements Support” on page 18 for information about what language elements are supported in the CAS DATA step.

See Also

- “Running the DATA Step in CAS”.
- “Automatic Variables” on page 24 (_THREADID_ and _HOSTNAME_).
Run the DATA Step in CAS on a CAS Table

Example Code

To run the DATA step in CAS using input data, both the input and output must be CAS tables and both tables must use the same CAS engine libref. In the following example, the CASUTIL Procedure is used to load a SAS data set into CAS as an in-memory table. The SAS data set name is preceded by the CAS engine libref, mycas.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```
proc casutil outcaslib='casuser';
  load data=sashelp.cars;    /* 1 */
run;

data mycas.cars2 / sessref=casauto;    /* 2 */
  set mycas.cars;           /* 3 */
  if mpg_city > 25 then eff='Y';
  else eff='N';
  put 'Thread number: '  _threadid_
    'on worker node '  _hostname_
run;
```

1 Use PROC CASUTIL to load the Sashelp.Cars data set to CAS. This creates an in-memory CAS table named Cars. The OUTCASLIB= option loads the CAS table to the CASUSER caslib.

2 Specify a CAS table as the output data using the CAS engine libref.

3 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 in a CAS server session. For information about controlling DATA processing, see “Controlling Threaded Processing” on page 11.

- For a DATA step to run in CAS, you must make sure you have met the requirements for processing in CAS on page 4.

- You can also load external files to CAS using the CASUTIL procedure with the LOAD FILE statement. The following examples show how to load files into CAS using PROC CASUTIL: “Load a CSV File into CAS” in SAS Cloud Analytic Services: User’s Guide, Load a Client-Side File and Load a Server-Side File.

- The DATA step automatically runs in CAS when you use a CAS engine libref on the input and output data sets. For more information about controlling where the DATA step runs by default, see “Controlling Where the DATA Step Runs by Default” on page 7.

- Not all language elements are supported in a CAS DATA step. See “Language Elements Support” on page 18 for information about what language elements are supported in the CAS DATA step.

See Also

- CASLIB statement option “_ALL_” in SAS Cloud Analytic Services: User’s Guide
Load a CSV File into CAS and Save as a CAS Table

Example Code

The following example shows how to load a comma-separated value file (CSV) into CAS using the DATA step. This example uses the names.csv file, which can be found here: [http://support.sas.com/documentation/onlinedoc/viya/examples.htm](http://support.sas.com/documentation/onlinedoc/viya/examples.htm).

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```sas
filename names url
   "http://support.sas.com/documentation/onlinedoc/viya/exampledatasets/names.csv";

data mycas.names;
   infile names dsd truncover firstobs=2;                  /* 2 */
   input BRTH_YR :$10. GNDR :$10. ETHCTY :$10. NM :$10.     /* 2 */
      CNT :$10. RNK :$10.;                         /* 2 */
   run;

data mycas.names;
   proc casutil incaslib='casuser';                          /* 3 */
      save casdata='names' outcaslib='casuser' replace;
      list;
      run;

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

<table>
<thead>
<tr>
<th>BRTH_YR</th>
<th>GNDR</th>
<th>ETHCTY</th>
<th>NM</th>
<th>CNT</th>
<th>RNK</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>FEMALE</td>
<td>HISPANIC</td>
<td>GERALDINE</td>
<td>13</td>
<td>75</td>
</tr>
<tr>
<td>2011</td>
<td>FEMALE</td>
<td>HISPANIC</td>
<td>GIA</td>
<td>21</td>
<td>67</td>
</tr>
<tr>
<td>2011</td>
<td>FEMALE</td>
<td>HISPANIC</td>
<td>GIANNA</td>
<td>49</td>
<td>42</td>
</tr>
<tr>
<td>2011</td>
<td>FEMALE</td>
<td>HISPANIC</td>
<td>GISELLE</td>
<td>38</td>
<td>51</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>MALE</td>
<td>WHITE</td>
<td>NON HISPANIC</td>
<td>Youssef,24,88</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>MALE</td>
<td>WHITE</td>
<td>NON HISPANIC</td>
<td>Yusuf,16,96</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>MALE</td>
<td>WHITE</td>
<td>NON HISPANIC</td>
<td>Zachary,90,39</td>
<td></td>
</tr>
<tr>
<td>2014</td>
<td>MALE</td>
<td>WHITE</td>
<td>NON HISPANIC</td>
<td>Zev,49,65</td>
<td></td>
</tr>
</tbody>
</table>

1 In SAS, load the external comma-separated file using the INFILE statement. Specify a CAS engine libref on the output table. The TRUNCOVER option enables 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 a permanent copy of the in-memory CAS table.
Key Ideas

- You can load external files to CAS by using an INFILE statement in the DATA step and a CAS engine libref on the output table.
- You can also load external files to CAS using the CASUTIL procedure with the LOAD FILE statement. The following examples show how to load files into CAS using PROC CASUTIL: “Load a CSV File into CAS” in SAS Cloud Analytic Services: User’s Guide, Load a Client-Side File and Load a Server-Side File.

See Also

- “FILENAME Statement: URL Access Method” in SAS Global Statements: Reference
- “INFILE Statement” in SAS DATA Step Statements: Reference
- “About Informats” in SAS Formats and Informats: Reference

Manage Tables

Convert a CAS Table to a SAS Data Set

Example Code

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.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

data mycas.earnings;
  Amount=1000;                             /* 1 */
  Rate=.075/12;
/* 1 */
do month=1 to 12;
    Earned + (amount+earned) * (rate);
end;
run;
proc print data=mycas.earnings; run;
libname mysas "/file-path"; /* 2 */
data mysas.earnings; /* 3 */
    set mycas.earnings;
run;

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

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

3 Read in the CAS table and write it out as a SAS data set to the SAS library.

<table>
<thead>
<tr>
<th>Obs</th>
<th>Amount</th>
<th>Rate</th>
<th>month</th>
<th>Earned</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1000</td>
<td>0.0625</td>
<td>13</td>
<td>77.625</td>
</tr>
</tbody>
</table>

Key Ideas

- A CAS DATA step that has an output table but no input table runs in a single thread by default. You can force the DATA step to run in multiple threads by specifying the SINGLE=NO option in the DATA statement. See “SINGLE=NO | NOINPUT | YES” in SAS DATA Step Statements: Reference for information about specifying the SINGLE= option.

- For more information about where the DATA step runs and controlling DATA step processing modes, see “Processing Modes” on page 8 and “Controlling Where the DATA Step Runs by Default” on page 7.

- The CAS DATA step works only on CAS tables.

- Not all language elements are supported in a CAS DATA step. See “Language Elements Support” on page 18 for information about what language elements are supported in the CAS DATA step.

See Also

- CASLIB statement option “_ALL_” in SAS Cloud Analytic Services: User’s Guide
- “LIBNAME Statement” in SAS Global Statements: Reference
- “DATA Statement” in SAS DATA Step Statements: Reference
Combine Tables Using the SET and BY Statements

Example Code

The following example creates two data sets and then loads them into CAS tables using PROC CASUTIL. The first DATA step combines the two data sets by concatenating one to the other. This is similar to appending tables to one another. The second DATA step interleaves the two data sets by adding the BY statement. The BY variables are Common and Number.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```sas
data animal1;                               ;/*1*/
  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;                                ;/*2*/
    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;                                ;/*3*/
    set animal1 plant1;
  run;
  proc print data=append;
    title '1. Plant1 Appended to Animal1 in a '
        'DATA Step Running in SAS'; run;
  data interleaveByCommon;                    ;/*4*/
    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;               /*
  5 */
  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="casuser"              /* 6 */
  outcaslib="casuser";
  load data=animal1 replace;
  load data=plant1 replace;
  run;

data mycas.appendCas / sessref=casauto;      /* 7 */
  set mycas.animal1 mycas.plant1;
  hostname=_hostname_; run;

proc print data=mycas.appendCas;
title '4. Plant1 Appended to Animal1 in a;
  Multithreaded DATA Step
  Running in CAS';
run;

data mycas.byCommon / sessref=casauto;      /* 8 */
  set mycas.animal1 mycas.plant1;           /*
  8 */
  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 / sessref=casauto; /* 9 */
  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 the first data set.
Create the second data set.

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

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

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

Load the data sets into CAS.

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

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

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

Key Ideas

- Concatenating data sets is the combining of two or more data sets, one after the other, into a single data set. Concatenating uses the SET statement with multiple input tables. All observations from the first data set are read, followed by all observations from the second data set.

- Interleaving uses a SET statement and a BY statement to combine multiple data sets into one new data set.

- Before you can interleave data sets, the data must be sorted by the same variable or variables that are used with the BY statement that accompanies your SET statement.

- The order of rows within a BY group is not stable within CAS. Unless you explicitly specify additional BY variables to order the rows, the ordering can change from one DATA step job to the next. If you need a stable ordering, explicitly specify additional BY variables. See “How CAS Groups Data with BY Variables” on page 15.

- The BY statement combines the rows in the two data sets that have the same values for the BY variable.

- Instead of concatenating tables, you can append them and produce the same results as concatenating. To avoid reading all the records in both tables, you can append one table to another by using the APPEND= data set option.

See Also

- CASLIB statement option “_ALL_” in SAS Cloud Analytic Services: User’s Guide
- “BY Statement” in SAS DATA Step Statements: Reference
- “MERGE Statement” in SAS DATA Step Statements: Reference
- “APPEND= Data Set Option” in SAS Cloud Analytic Services: User’s Guide
- “Combining SAS Data Sets: Methods” in SAS Language Reference: Concepts
Filter and Append Rows to a Promoted Table

Example Code

The following example appends a CAS table to a promoted CAS table by using the APPEND= data set option.

```sas
data mycas.class(promote=yes); /* 1 */
   set sashelp.class;
   where Weight>115; /* 2 */
run;

data mycas.addClass; /* 3 */
   input Name $ Sex $ Age Height Weight;
   datalines;
   Aaron M 13  64 112
   Becky F 16  68 125
   ;

data mycas.class(append=force); /* 4 */
   set mycas.addClass;
   where weight>115; /* 5 */
run;
proc print data=mycas.class; run; /* 6 */
```

1. Create the table mycas.class by reading in the SAS data set, class, from the Sashelp library. Promote the table to global scope by specifying the PROMOTE= data set option.

Note: You cannot specify the PROMOTE= data set option on an existing table that has already been promoted to global scope. For example, if you try to run this sample program again and the mycas.class data set is still in your mycas library, then an error will occur. You must drop or delete the promoted table before replacing it with a DATA statement that includes the PROMOTE= data set option. You can use the following code to delete the data set: proc datasets lib=mycas; delete class; run;

2. Use the WHERE Statement to select only those rows where the variable Age is greater than 15.

3. Create a second table by specifying the variable names and their types in the INPUT statement and the variable values in the DATALINES statement.

4. Append the mycas.addClass table to the mycas.class table by specifying the APPEND= CAS data set option in the DATA statement. Even though the tables contain the same variables, the variable Sex has different lengths in the two tables. In the table mycas.addClass, it has a length of 8 bytes and in the table mycas.addClass, it has a length of 1 byte. Therefore, you must specify the FORCE option in the APPEND= statement.
Specify the WHERE statement to select only the rows where weight is greater than 115.

Print the results.

**Output 2.2  Mycas.Class Table**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Sex</th>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Philip</td>
<td>M</td>
<td>15</td>
<td>72.0</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>Robert</td>
<td>M</td>
<td>12</td>
<td>84.8</td>
<td>128</td>
</tr>
<tr>
<td>3</td>
<td>Ronald</td>
<td>M</td>
<td>15</td>
<td>87.0</td>
<td>132</td>
</tr>
</tbody>
</table>

**Output 2.3  Mycas.addClass Table**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Sex</th>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Aaron</td>
<td>N</td>
<td>15</td>
<td>84</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>Becky</td>
<td>F</td>
<td>16</td>
<td>80</td>
<td>135</td>
</tr>
</tbody>
</table>

**Output 2.4  PROC PRINT Output Showing the Filtered Mycas.addClass Appended to the Mycas.Class Table**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Sex</th>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Philip</td>
<td>M</td>
<td>15</td>
<td>72.3</td>
<td>150</td>
</tr>
<tr>
<td>2</td>
<td>Robert</td>
<td>M</td>
<td>12</td>
<td>84.3</td>
<td>128</td>
</tr>
<tr>
<td>3</td>
<td>Becky</td>
<td>F</td>
<td>16</td>
<td>80.5</td>
<td>130</td>
</tr>
<tr>
<td>4</td>
<td>Ronald</td>
<td>M</td>
<td>15</td>
<td>87.3</td>
<td>133</td>
</tr>
</tbody>
</table>

Alternatively, you can change the length of the variable using the **LENGTH** statement:

``` SAS
data mycas.addClass;
  length Sex $1;
  set mycas.class;
run;
```

**Key Ideas**

- Unless you specify APPEND=FORCE, the lengths of the variables in the input tables must be the same.
- The FORCE option uses the character length of the in-memory table to determine the length of input observations. A truncation error occurs when the character length sizes of the input data exceeds the character length sizes of the output table.
- Variable attributes, such as name, length, and format, on the input data sets must be the same.
- You cannot specify the PROMOTE= data set option on an existing table that has already been promoted to global scope. You must drop or delete a promoted table before replacing it: `proc datasets lib=mycas; delete class; run;`
- The PROMOTE= and APPEND= options cannot be specified at the same time.
- You cannot append rows to a table that is both partitioned and ordered.
See Also
- “APPEND= Data Set Option” in SAS Cloud Analytic Services: User’s Guide
- “LENGTH Statement” in SAS DATA Step Statements: Reference
- “DATASETS Procedure” in Base SAS Procedures Guide

Manipulate Variables

Sum a Variable across an Entire Table

Example Code

This example sums a variable across all the rows of a very large, distributed table in CAS. To do this, the example uses a two-step process: In the first step, a DATA step runs in parallel on the “big data.” In the second step, the DATA step runs in a single thread on the smaller data set to gather the final sum. The data that is collected in the final sum is the total number of homeowners in the data set mycas.purchase.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```sas
/* Multiple Partial sums are computed by multiple threads */
data mycas.sums;
    retain homeowners_sum;        /* 1 */
    keep homeowners_sum;          /* 2 */
    set mycas.purchase end = done; /* 3 */
    if demog_ho = 1 then
        homeowners_sum + 1;       /* 4 */
    if done then output;          /* 5 */
run;
title 'DATA Step in CAS';
proc print data=mycas.sums; run;

data mycas.sums_all / single = yes; /* 6 */
    retain homeowners_sum_all;      /* 7 */
    keep homeowners_sum_all;        /* 8 */
    set mycas.sums end = done;      /* 9 */
    homeowners_sum_all + homeowners_sum; /* 10 */
    if done then output;
run;
title 'Final Sum';
proc print data=mycas.sums_all; run; /* 11 */
```

1 Retain the value of the homeowners_sum variable from one row to the next.
2 Keep only the homeowners_sum variable in the output table.
Read in the input table, mycas.purchase. Specify END= to create a flag for the conditional IF statement.

Create a sum of the variable Homeowners that includes the values in the rows that are being processed in that DATA step thread (on that node). Each thread calculates a partial sum for the rows that it processes. Store each sum for each thread total in homeowners_sum. See “Sum Statement” in SAS DATA Step Statements: Reference for more information.

When the last row is reached, SAS writes out the results.

Use the SINGLE=YES option to run the DATA step in a single thread.

Retain the value of the homeowners_sum_all variable from one row to the next.

Keep only the homeowners_sum_all variable in the output table.

Read in the input table, mycas.sums. Specify END= to create a flag for the conditional IF statement.

Create a total sum of Homeowners by adding the partial sums from the first DATA step. Because this DATA step runs in a single thread, the total sum of all Homeowners is calculated.

Print the results showing the total sum.

Key Ideas

- There is no sharing between DATA step threads.
- Each thread has its own set of rows from the table and computes values based on those rows.
- This solution speeds processing by running as much as possible in multiple threads.
- When you use a retained variable to hold a sum in a multithreaded DATA step, the variable contains only the sum of the values that the processing thread can see.

See Also

- “RETAIN Statement” in SAS DATA Step Statements: Reference
- “Sum Statement” in SAS DATA Step Statements: Reference
- “END=variable” in SAS DATA Step Statements: Reference
- CASLIB statement option “_ALL_ ” in SAS Cloud Analytic Services: User’s Guide
- “Inter-row Dependencies” on page 13.
Group and Sort Variables in Descending Order

Code Example

This example loads a Sashelp data set into CAS and then specifies the DESCENDING option in the BY statement to group the data by Region, Product, and Sales. The DATA step then orders the data by Product and Sales in descending order.

Note: The DESCENDING option in the BY statement is supported in a DATA step that is running in CAS for the second and subsequent variables specified in the BY statement only. The option is not supported for the first variable specified in the BY statement in a DATA step that is running in CAS.

See “Set Up Code” on page 34 for information about how to run this example.

```plaintext
options cashost="cloud.example.com" casport=5570;
cas casauto sessopts=(caslib='casuser');
libname mycas cas;
caslib _all_ assign;

data mycas.shoes;
  set sashelp.shoes;
  by Region;
  keep Region Product Subsidiary Sales;
run;

data mycas.shoes;
  set mycas.shoes;
run;

data mycas.shoes;
  set mycas.shoes;
  by Region
    descending Product
    descending Sales;
run;

proc print data=mycas.shoes;
  title 'Shoes By Region By Sales in Descending Order';
  var Region Product Sales Subsidiary;
run;
```
Here is the same example Using PROC CAS:

```sas
proc cas;
  datastep.runcode /
    code='data shoes;
      set shoes;
      by Region
        descending Product
        descending Sales;
      run;'
  run;
```

For more information about the DATA step run code action in PROC CAS, see “DATA Step Action Set Details” in SAS Viya: System Programming Guide.
Key Ideas

- The DESCENDING option in the BY statement is supported in a DATA step that is running in CAS for the second and subsequent variables specified in the BY statement only. The option is not supported for the first variable specified in the BY statement in a DATA step that is running in CAS.

- In CAS, there is no implicit ordering of data by the DATA step and there is no guarantee of ordering of data. See Ordering in CAS on page 13 for more information.

- When you use the BY statement in a distributed server, CAS groups the rows based on the first BY variable. When CAS distributes the table across multiple nodes, it keeps the BY groups intact. That is, rows that share the same BY variables are stored together on a single machine. For more information about BY-group processing in the CAS DATA step, see “BY-Group Processing in CAS” on page 15.

See Also

- “BY Statement” in SAS DATA Step Statements: Reference
- “DESCENDING” in SAS DATA Step Statements: Reference

Add a Unique Row Identifier

Example Code

The _N_ automatic DATA step variable holds the number of iterations that a DATA step makes as it reads in rows from a table or other data source. Since the SAS DATA step usually reads rows sequentially, one row at a time, _N_ is often used to create row numbers for observations in a SAS data set.

This works well as long as the DATA step is running in a single thread. When the DATA step runs in multiple threads, there are multiple ‘copies’ or threads of the DATA step running on a single table simultaneously. The _N_ values are repeated for every DATA step thread. Therefore, the _N_ value alone cannot be used to uniquely identify rows with multithreaded DATA step processing.

The following example shows how you can use the _N_ and _THREADID_ automatic variables to create unique row numbers for a table that is processed in multiple DATA step threads.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

First, data is created for the example by loading the Sashelp.Cars data set to an in-memory CAS table. For simplicity, the KEEP statement is used to limit the number of variables that are included in the output table.

```
/* Create a subset of the Cars data set and load it to CAS */
```
data mycas.cars1;
  set sashelp.cars;
  keep make type;
run;

In the next DATA step, the _N_ automatic variable is stored in the variable rowID. RowID counts the iterations within each DATA step thread. The _THREADID_ automatic variable is stored in the variable thread_ID. The _THREADID_ automatic variable assigns a number to each thread that the DATA step is running in.

data mycas.cars2;
  set mycas.cars1;
  rowID = _n_;
  threadID = _threadid_; run;

proc print data=mycas.cars2; run;

Notice how the _N_ value numbering repeats, starting again at 1 for the rows where thread_ID changes. Because the _N_ values are repeated across multiple threads, they are not unique table row numbers. Row 1, for example, has the same _N_ value as rows 55, row 109, and so on. The _THREADID_ automatic variable is also not unique since there are multiple rows assigned to a single thread. For example, rows 1 through 54 are all being processed by the same thread with an ID of 1, and rows 55 through 108 are running in thread 33.

Output 2.5 Partial PROC PRINT Output for Cars1 Showing How _N_ and _THREADID_ Values Are Not Unique in the Distributed Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>Make</th>
<th>Type</th>
<th>rowID</th>
<th>threadID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Acura</td>
<td>SUV</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Audi</td>
<td>Sedan</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Audi</td>
<td>Sedan</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>54</td>
<td>Volvo</td>
<td>Sedan</td>
<td>54</td>
<td>1</td>
</tr>
<tr>
<td>55</td>
<td>Acura</td>
<td>Sedan</td>
<td>1</td>
<td>33</td>
</tr>
<tr>
<td>56</td>
<td>Audi</td>
<td>Sedan</td>
<td>2</td>
<td>33</td>
</tr>
<tr>
<td>57</td>
<td>Audi</td>
<td>Sedan</td>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>108</td>
<td>Volvo</td>
<td>Sedan</td>
<td>54</td>
<td>33</td>
</tr>
<tr>
<td>109</td>
<td>Acura</td>
<td>Sedan</td>
<td>1</td>
<td>65</td>
</tr>
<tr>
<td>110</td>
<td>Audi</td>
<td>Sedan</td>
<td>2</td>
<td>65</td>
</tr>
<tr>
<td>111</td>
<td>Audi</td>
<td>Sedan</td>
<td>3</td>
<td>65</td>
</tr>
</tbody>
</table>

To create a unique row identifier for every row in the table, you can concatenate the values for _N_ with the values for _THREADID_:

data mycas.cars2;
  set mycas.cars1;
  rowID = _n_;
  threadID = _threadid_
  uniqueRowID = _n_ + (_threadid_ * 1E4);
run;
proc print data=mycas.cars2; run;
Here is the same example with a BY group defined:

data mycas.cars3;
   set mycas.cars1;
   rowID = _n_;  
   threadID = _threadid_;  
   uniqueRowID = _n_ + (_threadid_ * 1E4);  
   by make;  
run;
proc print data=mycas.cars3(obs=20); run;

Output 2.6  Partial PROC PRINT Output for Cars3 Table Showing Unique Row Identifiers with BY Groups

<table>
<thead>
<tr>
<th>Obs</th>
<th>Make</th>
<th>Type</th>
<th>rowID</th>
<th>threadID</th>
<th>uniqueRowID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Saturn</td>
<td>Sedan</td>
<td>1</td>
<td>8</td>
<td>80001</td>
</tr>
<tr>
<td>2</td>
<td>Saturn</td>
<td>Sedan</td>
<td>2</td>
<td>8</td>
<td>80002</td>
</tr>
<tr>
<td>3</td>
<td>Saturn</td>
<td>Sedan</td>
<td>3</td>
<td>8</td>
<td>80003</td>
</tr>
<tr>
<td>4</td>
<td>Saturn</td>
<td>Wagon</td>
<td>4</td>
<td>8</td>
<td>80004</td>
</tr>
<tr>
<td>5</td>
<td>Saturn</td>
<td>Sedan</td>
<td>5</td>
<td>8</td>
<td>80005</td>
</tr>
<tr>
<td>6</td>
<td>Saturn</td>
<td>Sedan</td>
<td>6</td>
<td>8</td>
<td>80006</td>
</tr>
<tr>
<td>7</td>
<td>Saturn</td>
<td>Sedan</td>
<td>7</td>
<td>8</td>
<td>80007</td>
</tr>
<tr>
<td>8</td>
<td>Saturn</td>
<td>SUV</td>
<td>8</td>
<td>8</td>
<td>80008</td>
</tr>
<tr>
<td>9</td>
<td>Saab</td>
<td>Sedan</td>
<td>1</td>
<td>14</td>
<td>140001</td>
</tr>
<tr>
<td>10</td>
<td>Saab</td>
<td>Sedan</td>
<td>2</td>
<td>14</td>
<td>140002</td>
</tr>
<tr>
<td>11</td>
<td>Saab</td>
<td>Wagon</td>
<td>3</td>
<td>14</td>
<td>140003</td>
</tr>
<tr>
<td>12</td>
<td>Saab</td>
<td>Sedan</td>
<td>4</td>
<td>14</td>
<td>140004</td>
</tr>
<tr>
<td>13</td>
<td>Saab</td>
<td>Sedan</td>
<td>5</td>
<td>14</td>
<td>140005</td>
</tr>
<tr>
<td>14</td>
<td>Saab</td>
<td>Sedan</td>
<td>6</td>
<td>14</td>
<td>140006</td>
</tr>
<tr>
<td>15</td>
<td>Saab</td>
<td>Sedan</td>
<td>7</td>
<td>14</td>
<td>140007</td>
</tr>
<tr>
<td>16</td>
<td>Subaru</td>
<td>Sports</td>
<td>1</td>
<td>19</td>
<td>190001</td>
</tr>
<tr>
<td>17</td>
<td>Subaru</td>
<td>Sedan</td>
<td>2</td>
<td>19</td>
<td>190002</td>
</tr>
<tr>
<td>18</td>
<td>Subaru</td>
<td>Truck</td>
<td>3</td>
<td>19</td>
<td>190003</td>
</tr>
<tr>
<td>19</td>
<td>Subaru</td>
<td>Sedan</td>
<td>4</td>
<td>19</td>
<td>190004</td>
</tr>
<tr>
<td>20</td>
<td>Subaru</td>
<td>Wagon</td>
<td>5</td>
<td>19</td>
<td>190005</td>
</tr>
</tbody>
</table>
Key Ideas

- The _THREADID_ automatic variable can be combined with the _N_ automatic variable to create a unique row identifier in a multithreaded DATA step.
- The _N_ automatic DATA step variable holds the number of iterations that a DATA step makes as it reads in rows from a table or other data source.
- The _N_ automatic variable is often used to add row numbers to observations in SAS data sets.
- The _N_ automatic variable values alone are not unique in a multithreaded DATA step and cannot be used to uniquely identify rows in a table.
- The _THREADID_ automatic variable returns the number that is associated with the thread that the DATA step is running in a server session.
- There is no sharing between DATA step threads in a multithreaded DATA step. Each thread has its own set of rows from the table and computes values based on those rows.

See Also

- "_N_" on page 25
- "_THREADID_" on page 27

Create a VARCHAR Variable Using the LENGTH Statement

Example Code

In the following example, the first LENGTH statement creates a VARCHAR variable, x, with a length of 30 characters. The second LENGTH statement creates a CHAR variable, y, with a length of 30 bytes.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

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

proc contents data=mycas.string;
run;
```
If you read the table into a SAS data set and then run PROC CONTENTS on the output data set, you can see how SAS converts the variable.

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

When SAS converts a VARCHAR to a CHAR in automatic data type conversion, SAS determines the length of the CHAR variable based on the SAS session encoding. If the local session encoding uses a single-byte character set, then SAS multiplies the original length by 1 to get the length of the converted CHAR variable. The length is multiplied by 1 in the calculation because 1 is the largest possible value for any character in a single-byte character set.

**Output 2.7  PROC CONTENTS Output When the Local SAS Session Encoding Is Single Byte (Latin1)**

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Max Bytes Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>Varchar</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>Char</td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

If the local session encoding is a multi-byte encoding such as UTF-8, then SAS multiplies the length times the longest possible value of any character in the character set. The example here shows the length of the converted VARCHAR when the local session encoding is UTF-8. Because 4 bytes is the largest possible value for any character in UTF-8, the length is calculated as the original length, 30, times 4 (120).

**Output 2.8  PROC CONTENTS Output When the Local SAS Session Encoding Is Multi-byte (UTF-8)**

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>Char</td>
<td>120</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>Char</td>
<td>30</td>
</tr>
</tbody>
</table>
Key Ideas

- The VARCHAR data type is not supported by the SAS V9 engine. Therefore, to store a VARCHAR variable in a table, you must use a CAS engine libref on the output table.

- 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 40, 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 that the string contains. The length of CHAR variables is determined based on the number of bytes that the characters in the string require.

- 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 WLatin1, 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

- “Avoiding Data Truncation” on page 27
- “VARCHAR Data Type” in SAS Cloud Analytic Services: User’s Guide
- “Create a VARCHAR Variable Using the LENGTH Statement” on page 54.

Impute Missing Values Using the LAG Function

Example Code

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 for PRICE between the rows in the Mycas.Example 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.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

data mycas.example;
   input county $ year price;
datalines;
  1001 2001 200000
  1001 2002 .
  1001 2003 .
  1001 2004 280000
  1001 2005 310000
  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 .
; 

data mycas.example2 (drop=lag_price lag2_price lag_county lag2_county)
   / single=yes;
   set mycas.example;
   by county year;
   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;
Key Ideas

- For more information about multithreaded processing and inter-row dependencies, see “Inter-row Dependencies”.
- For information about the LAG function, see “LAG Function” in SAS Functions and CALL Routines: Reference.

See Also

- CASLIB statement option “_ALL_” in SAS Cloud Analytic Services: User’s Guide
- “LAG Function” in SAS Functions and CALL Routines: Reference
- “Inter-row Dependencies”

Create a DATA Step View

Example Code

The following example shows how to create a DATA step view. First, an in-memory CAS table named Mycas.Class is created from the Class data set in the Sashelp library. The SET statement selects all rows where Sex=M. A VARCHAR variable, Note, is created so that any Males who are 12 years old and taller than 60 inches are highlighted. See “Key Ideas” on page 61 for more information about DATA step views in CAS.
For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

data mycas.class;                       /* 1 */
   set sashelp.class(where=(sex='M'));  /* 2 */
   length Note varchar(10);             /* 3 */
   if age="12" and height>60 then do;   /* 4 */
      Note="stature";                    /* 5 */
   end;
run;

data BMI / view=BMI;                    /* 6 */
   set mycas.class;                     /* 7 */
   BMI = weight / height**2 * 703;      /* 8 */
   format BMI 6.2;                      /* 9 */
run;
proc print data=BMI; run;               /* 10 */

1  Create an in-memory CAS table named Mycas.Class.
2  Use the Class data set in the Sashelp library as the input data. The SET statement selects all rows where Sex=M using the WHERE statement. The rows are selected from the input data set before the output table is created in CAS.
3  Create a VARCHAR variable with a length of 10 characters.
4  Use a subsetting IF statement to select rows where Age =12 and Height is greater than 60.
5  If Age=12 and Height is greater than 60, then set the Note variable to "stature."
6  Create a SAS DATA step view named BMI from the Mycas.Class table. The VIEW= option in the DATA statement defines the view.
7  Read the Mycas.Class table using the SET statement.
8  Calculate the BMI based on the Weight and Height variable values and store the result in the variable BMI.
9  Format the BMI value using the FORMAT statement.
10 Print the BMI value using the PRINT procedure.

Here is the log and PROC PRINT output.

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

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

---

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Sex</th>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
<th>Note</th>
<th>BMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alfred</td>
<td>M</td>
<td>14</td>
<td>69.0</td>
<td>112.5</td>
<td></td>
<td>16.61</td>
</tr>
<tr>
<td>2</td>
<td>Robert</td>
<td>M</td>
<td>12</td>
<td>64.8</td>
<td>128.0</td>
<td>stature</td>
<td>21.43</td>
</tr>
<tr>
<td>3</td>
<td>Henry</td>
<td>M</td>
<td>14</td>
<td>63.5</td>
<td>102.5</td>
<td></td>
<td>17.87</td>
</tr>
<tr>
<td>4</td>
<td>Ronald</td>
<td>M</td>
<td>15</td>
<td>67.0</td>
<td>133.0</td>
<td></td>
<td>20.83</td>
</tr>
<tr>
<td>5</td>
<td>James</td>
<td>M</td>
<td>12</td>
<td>57.3</td>
<td>83.0</td>
<td></td>
<td>17.77</td>
</tr>
<tr>
<td>6</td>
<td>Thomas</td>
<td>M</td>
<td>11</td>
<td>57.5</td>
<td>85.0</td>
<td></td>
<td>18.07</td>
</tr>
<tr>
<td>7</td>
<td>Jeffrey</td>
<td>M</td>
<td>13</td>
<td>62.5</td>
<td>84.0</td>
<td></td>
<td>15.12</td>
</tr>
<tr>
<td>8</td>
<td>William</td>
<td>M</td>
<td>15</td>
<td>66.5</td>
<td>112.0</td>
<td></td>
<td>17.80</td>
</tr>
<tr>
<td>9</td>
<td>John</td>
<td>M</td>
<td>12</td>
<td>59.0</td>
<td>99.5</td>
<td></td>
<td>20.09</td>
</tr>
<tr>
<td>10</td>
<td>Philip</td>
<td>M</td>
<td>16</td>
<td>72.0</td>
<td>150.0</td>
<td></td>
<td>20.34</td>
</tr>
</tbody>
</table>

---

The CONTENTS Procedure

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>WORK.BMI</th>
<th>Observations</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Type</td>
<td>VIEW</td>
<td>Variables</td>
<td>7</td>
</tr>
<tr>
<td>Engine</td>
<td>SASDSV</td>
<td>Indexes</td>
<td>0</td>
</tr>
<tr>
<td>Created</td>
<td>08/14/2017 14:08:39</td>
<td>Observation Length</td>
<td>51</td>
</tr>
<tr>
<td>Last Modified</td>
<td>08/14/2017 14:08:39</td>
<td>Deleted Observations</td>
<td>0</td>
</tr>
<tr>
<td>Protection</td>
<td>Compressed</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Data Set Type</td>
<td>Sorted</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Representation</td>
<td>Default</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td>Default</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Alphabetic List of Variables and Attributes

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Age</td>
<td>Num</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>BMI</td>
<td>Num</td>
<td>8</td>
<td>6.2</td>
</tr>
<tr>
<td>4</td>
<td>Height</td>
<td>Num</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Name</td>
<td>Char</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Note</td>
<td>Char</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Sex</td>
<td>Char</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Weight</td>
<td>Num</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
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 SAS 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 that contain 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

- CASLIB statement option “_ALL_” in SAS Cloud Analytic Services: User’s Guide
- “SAS Views” in SAS Language Reference: Concepts

Create a VARCHAR Using an ARRAY Statement

Example Code

In the following example, three VARCHAR variables are created with a maximum length for the array, Test.

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

- For more information about array processing in the DATA step, see “Array Processing” in SAS Language Reference: Concepts.

See Also

- CASLIB statement option “_ALL_” in SAS Cloud Analytic Services: User’s Guide
- “Data Types” in SAS Cloud Analytic Services: User’s Guide

Index CHAR and VARCHAR Character Strings

Example Code

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 ('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.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

Note: Assume that the local SAS session encoding is UTF-8.

data mycas.string / sessref=casauto;
  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;
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 \( xi = \text{index}(x,'样') \) 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 measures data in terms of characters and the CHAR data type measures data in terms of bytes.

```
data mycas.string2 / sessref=casauto;
  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;
```

### Output 2.11 Log Output

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

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

<table>
<thead>
<tr>
<th>VARCHAR string: ‘榜样榜’</th>
<th>character position: 1 2 3 (characters)</th>
</tr>
</thead>
</table>

And it returns the byte position when it indexes CHAR data types:

<table>
<thead>
<tr>
<th>CHAR string: ‘榜样榜’</th>
<th>byte position: 1-3 4-6 7-9 (bytes)</th>
</tr>
</thead>
</table>
Key Ideas

- 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 \[ h = \text{index('abhatcd', 'hat')}; \] The function returns \( h = 3 \), because the first letter of the "hat" string is the third from the left starting with 1.

- The VARCHAR data type is not supported by the SAS V9 engine. Therefore, to store a VARCHAR variable in a table, you must use a CAS engine libref on the output table.

- The VARCHAR data type 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 no more than 3 or 4 bytes.

- The length of a CHAR variable when it is converted from a VARCHAR variable is based on your local session encoding. For more information, see "Avoiding Data Truncation" on page 27.

- VARCHAR variables can increase memory use when the column contains only 2- or 3-digit integer values, such as a three-letter airport. For more information about when to use a VARCHAR data type, see "When to Use a VARCHAR Data Type" in SAS Cloud Analytic Services: User's Guide.

See Also

- "INDEX Function" in SAS Functions and CALL Routines: Reference
- "Data Types" in SAS Cloud Analytic Services: User’s Guide

Implicit Variable Type Conversion

Example Code

The following example shows how a VARCHAR variable is automatically converted to a CHAR type variable when it is saved as a SAS data set. The DATA step supports the processing of VARCHAR data, but only the CAS engine supports the VARCHAR data type. The first DATA step is shown for the purposes of creating data containing a VARCHAR for this example.
For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```sas
data mycas.test;                                       /* 1 */
    length x varchar(*);
    length y varchar(20);
    x="hello";
    y="goodbye";
run;
proc contents data=mycas.test;
run;

data simple;                                          /* 2 */
    set mycas.test;
run;

proc contents data=simple;                            /* 3 */
run;
```

1 Create a CAS table that contains a VARCHAR variable for the example. The in-memory table, Mycas.Test, contains two VARCHAR variables.

2 Save the CAS table as a SAS data set. Note there is no CAS engine libref on the output data set.

3 View the contents of the data set to see how VARCHAR variables are converted to CHAR variables.

### Alphabetic List of Variables and Attributes

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len Bytes</th>
<th>Chars</th>
<th>Max Bytes Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>Varchar</td>
<td></td>
<td>.</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>Varchar</td>
<td>20</td>
<td>20</td>
<td>7</td>
</tr>
</tbody>
</table>

### Alphabetic List of Variables and Attributes

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>Char</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>y</td>
<td>Char</td>
<td>20</td>
</tr>
</tbody>
</table>

**Example Code 2.1 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=1.
Key Ideas

- You cannot create a VARCHAR variable using an assignment statement alone. SAS sets the type to CHAR if the value on the right side of the assignment is a character string. To create a VARCHAR variable, you must define it in either a LENGTH statement or in an ARRAY statement.

- 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 is converted to a CHAR type with a length of 32,767 bytes.

- If a VARCHAR variable is defined with a specific length and then is converted to a CHAR, SAS determines the length for the CHAR based on the local SAS session encoding. SAS calculates the length of the converted VARCHAR based on the SAS session encoding. See “Avoiding Data Truncation” on page 27 and “VARCHAR Length with Implicit Type Conversion” in SAS Cloud Analytic Services: User’s Guide for more information.

See Also

- CASLIB statement option “_ALL_” in SAS Cloud Analytic Services: User’s Guide
- “VARCHAR Data Type” in SAS Cloud Analytic Services: User’s Guide

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

Example Code

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. The LET macro statement substitutes the word `dataopts` for the SESSREF= option. When `&dataopts` is specified in the DATA statement, this causes the DATA step to run in CAS.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

Example Code 2.1 Simple Add and Iterate

```sas
%let dataopts=%str(/sessref=casauto);
/* Simple add and iterate */
data _null_ &dataopts;
length vc1 vc2 varchar(*);
```
Manipulate Variables 67

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

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

Output 2.12  Partial Log Output for Simple Add and Iterate

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 
vcl=abcd vc2=ABCD rc=0 _ERROR_=0 _N_=1

--------------------
vc1=abcd vc2=EFUHJKIJK rc=0 _ERROR_=0 _N_=1
vc1=abcd vc2=LMN rc=0 _ERROR_=0 _N_=1
NOTE: Duplicate messages output by DATA step:
vc1=lmn vc2=LMN rc=0 _ERROR_=0 _N_=1 (occurred 5744 times)
Didn't find key vc1=opqrst (occurred 5744 times)
vc1=efghijk vc2=EFGHIJK rc=0 _ERROR_=0 _N_=1 (occurred 5744 times)
vc1=abcd vc2=ABCD rc=0 _ERROR_=0 _N_=1 (occurred 11488 times)
-------------------- (occurred 5744 times)
vc1=abcd vc2=EFUHJKIJK rc=0 _ERROR_=0 _N_=1 (occurred 5744 times)
vc1=abcd vc2=LMN rc=0 _ERROR_=0 _N_=1 (occurred 5744 times)

NOTE: DATA statement used (Total process time):
real time 2.18 seconds
cpu time 0.12 seconds

Example Code 2.2  Add and Iterate With Parameters

/* Add with parameters */
data _null_ _dataopts;
   length vc1b vc2b varchar(*);

dcl hash h();
h.definekey('vc1b');
h.definedata('vc2b');
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;

Output 2.13  Partial Log Output for Simple Add and Iterate with Parameters

NOTE: Running DATA step in Cloud Analytic Services.
vc1b=lmn vc2b=LMN rc=0 _ERROR_=0 _N_=1
Didn't find key vc1b=opqrst
vc1b=efghijk vc2b=EFGHIJK rc=0 _ERROR_=0 _N_=1
vc1b=abcd vc2b=ABCD rc=0 _ERROR_=0 _N_=1
-----------------------------
vc1b=abcd vc2b=EFGHIJK rc=0 _ERROR_=0 _N_=1
vc1b=abcd vc2b=LMN rc=0 _ERROR_=0 _N_=1
NOTE: Duplicate messages output by DATA step:
vc1b=lmn vc2b=LMN rc=0 _ERROR_=0 _N_=1 (occurred 5744 times)
Didn't find key vc1b=opqrst (occurred 5744 times)
vc1b=efghijk vc2b=EFGHIJK rc=0 _ERROR_=0 _N_=1 (occurred 5744 times)
vc1b=abcd vc2b=ABCD rc=0 _ERROR_=0 _N_=1 (occurred 11488 times)
----------------------------- (occurred 5744 times)
vc1b=abcd vc2b=EFGHIJK rc=0 _ERROR_=0 _N_=1 (occurred 5744 times)
vc1b=abcd vc2b=LMN rc=0 _ERROR_=0 _N_=1 (occurred 5744 times)
NOTE: DATA statement used (Total process time):
    real time 2.17 seconds
    cpu time 0.07 seconds

Key Ideas

- You can use VARCHAR variables for KEY and DATA values when defining a hash object for a DATA step that is 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 that is running in CAS.
- A fixed-width, 32-byte SHA-256 hash digest value is computed for VARCHAR keys.
Group Data

Group and Order a CAS Table Using the PARTITION= Data Set Option

Example Code

In this example, a SAS data set from the Sashelp library is loaded into a distributed CAS server using PROC CASUTIL. In the DATA step, the table is partitioned and then ordered within each partition using the PARTITION= and ORDERBY= data set options.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

Note: To use the ORDERBY= option, you must first specify the PARTITION= option.

```
proc casutil outcaslib='casuser';                            /*1*/
   load data=sashelp.baseball replace;
run;

data mycas.baseball (partition=(team) orderby=(nHome name)); /*2*/
   set mycas.baseball;
   keep team name nHome;
run;
proc print data=mycas.baseball;
run;
```

1 Load the Sashelp.Baseball data set to a distributed CAS table.
2 Use the PARTITION= data set option to partition the table by team. This groups the rows that contain the same value for Team together, onto the same server node. The ORDERBY= option then orders the rows within each partition by nHome and Name.
Output 2.14  Partial Output for Partitioned and Ordered Baseball Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Team</th>
<th>nHome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Quinones, Rey</td>
<td>Boston</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Romero, Ed</td>
<td>Boston</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Barrett, Marty</td>
<td>Boston</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Boggs, Wade</td>
<td>Boston</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Armas, Tony</td>
<td>Boston</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>Gedman, Rich</td>
<td>Boston</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>Buckner, Bill</td>
<td>Boston</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>Rice, Jim</td>
<td>Boston</td>
<td>20</td>
</tr>
<tr>
<td>9</td>
<td>Evans, Dwight</td>
<td>Boston</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>Baylor, Don</td>
<td>Boston</td>
<td>31</td>
</tr>
<tr>
<td>11</td>
<td>Cruz, Julio</td>
<td>Chicago</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>Cangelosi, John</td>
<td>Chicago</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>Guillen, Ozzie</td>
<td>Chicago</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>Borilla, Bobby</td>
<td>Chicago</td>
<td>3</td>
</tr>
<tr>
<td>15</td>
<td>Tolleson, Wayne</td>
<td>Chicago</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>Dernier, Bob</td>
<td>Chicago</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>Boston, Daryl</td>
<td>Chicago</td>
<td>5</td>
</tr>
<tr>
<td>18</td>
<td>Haiston, Jerry</td>
<td>Chicago</td>
<td>5</td>
</tr>
<tr>
<td>19</td>
<td>Mumphrey, Jerry</td>
<td>Chicago</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>Skinner, Joel</td>
<td>Chicago</td>
<td>5</td>
</tr>
</tbody>
</table>

Note: There is no global ordering of rows across nodes or across partitions. You should not see rows for the same team spread across different nodes or different partitions. However, you might see the global order vary. The partitioning is done first, then the ordering is done within each partition.
Key Ideas

- A partition is a group of rows that share the same values for a specified variable, or a partition key. Rows that share the same partition key are arranged together on the same worker node. After the rows are partitioned, they are ordered within each node. Rows are not ordered globally across nodes unless you run the DATA step in a single thread. See "Run the DATA Step in a Single Thread" on page 80 to see an example of running the DATA step in a single thread.

- A CAS table can be partitioned using the PARTITION= data set option and ordered within each partition using the ORDERBY= option.

- You must specify the PARTITION= option before you can specify the ORDERBY= option.

- CAS partitions tables based on the formatted values of the PARTITION= variable (rather than the raw values). For example, a table containing dates that are formatted with the QTR format is divided into four partitions, one partition for each yearly quarter.

- If user-defined formats are used, then the format for the specified key variable must be available to the CAS server when the table is loaded into memory.

See Also

- “Ordering in CAS” on page 13
- “PARTITION= Data Set Option” in SAS Cloud Analytic Services: User’s Guide
- “ORDERBY= Data Set Option” in SAS Cloud Analytic Services: User’s Guide
- SAS Cloud Analytic Services: User-Defined Formats

Group and Sort Variables in Descending Order

Code Example

This example loads a Sashelp data set into CAS and then specifies the DESCENDING option in the BY statement to group the data by Region, Product, and Sales. The DATA step then orders the data by Product and Sales in descending order.

---

Note: The DESCENDING option in the BY statement is supported in a DATA step that is running in CAS for the second and subsequent variables specified in the BY
statement only. The option is not supported for the first variable specified in the BY statement in a DATA step that is running in CAS.

See “Set Up Code” on page 34 for information about how to run this example.

```plaintext
options cashost="cloud.example.com" casport=5570;
cas casauto sessopts=(caslib='casuser');
libname mycas cas;
caslib _all_ assign;

data mycas.shoes;
  set sashelp.shoes;
  by Region;
  keep Region Product Subsidiary Sales;
run;

data mycas.shoes;
  set mycas.shoes;
run;

data mycas.shoes;
  set mycas.shoes;
  by Region;
  descending Product
  descending Sales;
run;

proc print data=mycas.shoes;
  title 'Shoes By Region By Sales in Descending Order';
  var Region Product Sales Subsidiary;
run;
```
Figure 2.2  Partial PROC PRINT Output for the Shoes Table, Grouped by Region, Product, and Sales, and Ordered in Descending Order by Product and Sales

Shoes By Region By Sales in Descending Order

<table>
<thead>
<tr>
<th>Obs</th>
<th>Region</th>
<th>Product</th>
<th>Sales</th>
<th>Subsidiary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>South America</td>
<td>Women's Dress</td>
<td>$144,302</td>
<td>Caracas</td>
</tr>
<tr>
<td>2</td>
<td>South America</td>
<td>Women's Dress</td>
<td>$71,736</td>
<td>Bogota</td>
</tr>
<tr>
<td>3</td>
<td>South America</td>
<td>Women's Dress</td>
<td>$68,837</td>
<td>Sao Paulo</td>
</tr>
<tr>
<td>4</td>
<td>South America</td>
<td>Women's Dress</td>
<td>$29,059</td>
<td>La Paz</td>
</tr>
<tr>
<td>5</td>
<td>South America</td>
<td>Women's Dress</td>
<td>$21,435</td>
<td>Santiago</td>
</tr>
<tr>
<td>6</td>
<td>South America</td>
<td>Women's Dress</td>
<td>$21,434</td>
<td>Montevideo</td>
</tr>
<tr>
<td>7</td>
<td>South America</td>
<td>Women's Dress</td>
<td>$21,261</td>
<td>Buenos Aires</td>
</tr>
<tr>
<td>8</td>
<td>South America</td>
<td>Women's Casual</td>
<td>$48,250</td>
<td>La Paz</td>
</tr>
<tr>
<td>9</td>
<td>South America</td>
<td>Women's Casual</td>
<td>$42,654</td>
<td>Caracas</td>
</tr>
<tr>
<td>10</td>
<td>South America</td>
<td>Women's Casual</td>
<td>$36,791</td>
<td>Sao Paulo</td>
</tr>
<tr>
<td>11</td>
<td>South America</td>
<td>Women's Casual</td>
<td>$28,343</td>
<td>Montevideo</td>
</tr>
<tr>
<td>12</td>
<td>South America</td>
<td>Women's Casual</td>
<td>$15,470</td>
<td>Buenos Aires</td>
</tr>
<tr>
<td>13</td>
<td>South America</td>
<td>Women's Casual</td>
<td>$5,709</td>
<td>Santiago</td>
</tr>
<tr>
<td>14</td>
<td>South America</td>
<td>Sport Shoe</td>
<td>$8,365</td>
<td>Montevideo</td>
</tr>
<tr>
<td>15</td>
<td>South America</td>
<td>Sport Shoe</td>
<td>$8,095</td>
<td>La Paz</td>
</tr>
<tr>
<td>16</td>
<td>South America</td>
<td>Sport Shoe</td>
<td>$4,729</td>
<td>Bogota</td>
</tr>
<tr>
<td>17</td>
<td>South America</td>
<td>Sport Shoe</td>
<td>$4,194</td>
<td>Santiago</td>
</tr>
<tr>
<td>18</td>
<td>South America</td>
<td>Sport Shoe</td>
<td>$3,545</td>
<td>Buenos Aires</td>
</tr>
<tr>
<td>19</td>
<td>South America</td>
<td>Sport Shoe</td>
<td>$2,427</td>
<td>Sao Paulo</td>
</tr>
<tr>
<td>20</td>
<td>South America</td>
<td>Sport Shoe</td>
<td>$1,716</td>
<td>Caracas</td>
</tr>
<tr>
<td>21</td>
<td>South America</td>
<td>Slipper</td>
<td>$132,853</td>
<td>Caracas</td>
</tr>
<tr>
<td>22</td>
<td>South America</td>
<td>Slipper</td>
<td>$125,753</td>
<td>La Paz</td>
</tr>
<tr>
<td>23</td>
<td>South America</td>
<td>Slipper</td>
<td>$80,525</td>
<td>Montevideo</td>
</tr>
<tr>
<td>24</td>
<td>South America</td>
<td>Slipper</td>
<td>$54,768</td>
<td>Sao Paulo</td>
</tr>
</tbody>
</table>

Here is the same example Using PROC CAS:

```sas
proc cas;
    datastep.runcode /
       code='data shoes;
       set shoes;
       by Region
           descending Product
           descending Sales;
       run;'
run;
```

For more information about the DATA step run code action in PROC CAS, see "DATA Step Action Set Details" in SAS Viya: System Programming Guide.
Key Ideas

- The DESCENDING option in the BY statement is supported in a DATA step that is running in CAS for the second and subsequent variables specified in the BY statement only. The option is not supported for the first variable specified in the BY statement in a DATA step that is running in CAS.

- In CAS, there is no implicit ordering of data by the DATA step and there is no guarantee of ordering of data. See Ordering in CAS on page 13 for more information.

- When you use the BY statement in a distributed server, CAS groups the rows based on the first BY variable. When CAS distributes the table across multiple nodes, it keeps the BY groups intact. That is, rows that share the same BY variables are stored together on a single machine. For more information about BY-group processing in the CAS DATA step, see “BY-Group Processing in CAS” on page 15.

See Also

- “BY Statement” in SAS DATA Step Statements: Reference
- “DESCENDING” in SAS DATA Step Statements: Reference

Group and Order Rows Using the DATA Step BY Statement

Example Code

This example shows how to use the BY statement in a CAS DATA step to group and order variables in a table. The BY statement groups the table rows by Make and then orders the rows within each group by Type. The example also shows how to concatenate values from multiple rows into one row. The values for car Type are concatenated into a single variable so that they can be grouped into one row. In this way, all the CarStyles for each make of car are listed together in one row, in order.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```sas
proc casutil; /*1*/
load data=sashelp.cars
  outcaslib='casuser'
  replace;
run; quit;

data mycas.cars2;
  length CarStyles varchar(*); /*2*/
  retain CarStyles; /*3*/
  set mycas.cars;
  by make type; /*4*/
```
if first.make then CarStyles = trim(type); /* 5 */
    else CarStyles = CarStyles || "," || trim(type); /* 6 */
if last.make then output; /* 7 */
keep make CarStyles; /* 8 */
run;

proc print data=mycas.cars2; var Make CarStyles; /* 9 */
run;

1 Load the Sashelp.Cars data set to CAS.
2 Create a new variable, CarStyles, to be the aggregation variable to hold the concatenated values from other rows.
3 Retain the values for CarStyles, which are set equal to the existing variable Type. Type represents the style of car (SUV, wagon, and so on). For more information about retaining variables in the DATA step, see “RETAIN Statement” in SAS DATA Step Statements: Reference.
4 Create BY groups based on the Make of car and the style (type) of car. The values are grouped and ordered by the variables Make and Type.
5 If the DATA step encounters the first row in the BY group, then the variable CarStyles is set equal to the value for Type. The value stored in the variable CarStyles is retained from the previous row.
6 If the DATA step does not encounter the first or last row of the BY group, then the variable CarStyles equals the retained value from the previous row, Type. These values are separated by concatenating a comma to the end of the first value. The TRIM function removes trailing blanks.
7 If the DATA step encounters the last row of the BY group, then execute the OUTPUT statement and write the contents of the current row to the output table.
8 Keep only the variables Make and CarStyles. The variable Type is not needed because it is now stored in the variable CarStyles.
9 Print the output.
### Key Ideas

- In SAS, when using the BY statement in a DATA step, you must first sort the data. This requirement is also true for a DATA step that is running in a single thread in CAS.<br>
- 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.<br>
- On the CAS server, there is no guarantee of global ordering between BY groups. Each DATA step thread can group and order only the rows within each BY group. When a BY group is defined, the DATA step runs in multiple threads, one thread per BY group. Ordering is done only within each BY group. For more information about BY-group processing with the DATA step in CAS, see “How CAS Groups Data with BY Variables” on page 15.
Group Data by Two Variables

Example Code

This example shows how to group a table by two variables and keep only the unique rows in each group. The example first concatenates the values from two variables into a single new string variable, then orders by that new variable, keeping only the first in each group.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```
proc casutil;                                         /* 1 */
   load data=sashelp.cars
   outcaslib='casuser'
   replace;
run; quit;

data mycas.grouped;                                   /* 2 */
   set mycas.cars;
   keep make cylinders groupkey;
   groupkey = cats('make', make, 'cyl', cylinders);  /* 3 */
run;

data mycas.final;                                     /* 4 */
   set mycas.grouped;
   by groupkey;                                       /* 5 */
   if first.groupkey then output;                     /* 6 */
   drop groupkey;                                     /* 7 */
run;
```

1 Load the Sashelp.Cars data set to CAS.
2 Specify the loaded CAS table as the input table and create a new table called mycas.grouped.
3 Specify the CATS function to concatenate the values in the Make and Cyl columns and store them as a concatenated string in the variable Groupkey. The CATS function also removes leading and trailing blanks.
4 Run another DATA step to isolate the second group-by variable.
5 Group the table rows by the variable Groupkey. Note: When running the DATA step in CAS, you do not have to pre-sort the data before grouping using the BY statement.

6 Specify the FIRST. automatic variable in an IF statement to identify the beginning of each BY-group. Specify the OUTPUT statement to write only the first row of each group to the output table.

7 Drop the Groupkey column.

Key Ideas

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

- 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.

- On the CAS server, there is no guarantee of global ordering between BY groups. Each DATA step thread can group and order only the rows within each BY group. When a BY group is defined, the DATA step runs in multiple threads, one thread per BY group. Ordering is done only within each BY group. For more information about BY-group processing with the DATA step in CAS, see “How CAS Groups Data with BY Variables” on page 15.

See Also

- “Ordering in CAS” on page 13
- “How CAS Groups Data with BY Variables” on page 15
- “BY Statement” in SAS DATA Step Statements: Reference
- “RETAIN Statement” in SAS DATA Step Statements: Reference
- “IF-THEN/ELSE Statement” in SAS DATA Step Statements: Reference
- “How SAS Identifies the Beginning and End of a BY Group” in SAS DATA Step Statements: Reference
- “Combining Multiple Observations and Grouping Them Based on One BY Value” in SAS DATA Step Statements: Reference
- CASLIB statement option “_ALL_ ” in SAS Cloud Analytic Services: User’s Guide
Score an In-Memory CAS Table

Example Code

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

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```plaintext
proc casutil;                           /* 1 */
    load data=sashelp.baseball replace;
run;

data mycas.baseball / sessref=casauto;  /* 2 */
    set mycas.baseball(where=(team='Boston'));
    if nHits > 125
        then score=1;                       /* 3 */
    keep Name Team nHits score;
run;

proc print data=mycas.baseball; run;    /* 4 */
```

1. Load the Sashelp.Baseball data set into CAS as an in-memory CAS table.
2. 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.
3. Execute the score code.
4. Run the PRINT procedure to view the output table showing the scored values.

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Team</th>
<th>nHits</th>
<th>score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Buckner, Bill</td>
<td>Boston</td>
<td>168</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Romero, Ed</td>
<td>Boston</td>
<td>49</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>Gedman, Rich</td>
<td>Boston</td>
<td>119</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>Rice, Jim</td>
<td>Boston</td>
<td>200</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Quinones, Rey</td>
<td>Boston</td>
<td>68</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>Armas, Tony</td>
<td>Boston</td>
<td>112</td>
<td>.</td>
</tr>
<tr>
<td>7</td>
<td>Boggs, Wade</td>
<td>Boston</td>
<td>207</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Evans, Dwight</td>
<td>Boston</td>
<td>137</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>Barrett, Marty</td>
<td>Boston</td>
<td>179</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Baylor, Don</td>
<td>Boston</td>
<td>139</td>
<td>1</td>
</tr>
</tbody>
</table>
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.
- For a DATA step to run in CAS, you must make sure you have met the requirements for processing in CAS on page 4.

See Also

- CASLIB statement option “_ALL_” in SAS Cloud Analytic Services: User’s Guide
- “WHERE Statement” in SAS DATA Step Statements: Reference

Run the DATA Step in a Single Thread

Example Code

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.

```
data mycas.shoes / single=yes;                        /* 1 */
  set mycas.shoes;
  put "The number of Threads is " _nthreads_;        /* 2 */
run;
```

1 Using the Mycas.Shoes data set from the previous example, load the Mycas.Shoes table. Specify the SINGLE=YES option.

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

Output 2.16  Partial Log Output

```
NOTE: Running DATA step in Cloud Analytic Services.
The number of Threads is 1
```
Key Ideas

- The DATA step in CAS automatically runs in multiple threads on every available computer in your system by default. To run the DATA step in multiple threads, specify the SINGLE=YES option.
- A SAS DATA step (a DATA step that is not executing in CAS) always runs in a single thread.
- For more information about CAS sessions, see “Sessions” in SAS Cloud Analytic Services: Fundamentals.

See Also

- SESSOPTS=
- SINGLE= in SAS DATA Step Statements: Reference
- “Inter-row Dependencies” on page 13
- “DATA Statement” in SAS DATA Step Statements: Reference
- CASLIB statement option “_ALL_” in SAS Cloud Analytic Services: User’s Guide

Redistribute a Table After Specifying the SINGLE= Option

Example Code

The following example shows how to redistribute a CAS table to all available worker nodes after the SINGLE= option has been specified.

```sas
data mycas.prdsal2; set sashelp.prdsal2; run;           /*1*/
proc cas;
  table.tabledetails / level="node" name="prdsal2";      /*2*/
run;
quit;
```

1 For the example, load the SAS data set Sashelp.prdsal2 to CAS.
Run the TableDetails action on the table to verify that all available CAS worker nodes are being used.

### Output 2.17

Partial Results for TableDetails Action Showing Multiple Worker Nodes Processing Table Prdsal2

<table>
<thead>
<tr>
<th>Node</th>
<th>Number of Blocks</th>
<th>Active Blocks</th>
<th>Rows</th>
<th>Filled Data size</th>
<th>Variable Data size</th>
<th>Blocks Mapped</th>
<th>Memory Mapped</th>
<th>Blocks Unmapped</th>
<th>Memory Unmapped</th>
<th>Blocks Allocated</th>
<th>Memory Allocated</th>
<th>Index Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>server03.abc.com</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>648</td>
<td>0</td>
<td>1</td>
<td>524</td>
<td>1</td>
<td>524</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>server04.abc.com</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>648</td>
<td>0</td>
<td>1</td>
<td>524</td>
<td>1</td>
<td>524</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>server06.abc.com</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>648</td>
<td>0</td>
<td>1</td>
<td>524</td>
<td>1</td>
<td>524</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>server07.abc.com</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>648</td>
<td>0</td>
<td>1</td>
<td>524</td>
<td>1</td>
<td>524</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>server09.abc.com</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>648</td>
<td>0</td>
<td>1</td>
<td>524</td>
<td>1</td>
<td>524</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

All available CAS workers hold distributed table.

Each CAS worker contains 9 rows.

```plaintext
data mycas.predict / single=yes;                   /* 1 */
   set mycas.prdsal2;
   TotalA+Actual;
   TotalP+Predict;                                   /* 2 */
run;
proc cas;
   table.tabledetails /                            /* 3 */
      level="node"
      name="predict";
   table.tableinfo / table="predict";
run;
```

1. Specify **YES** in the **SINGLE=** option in the DATA statement to limit processing of the entire table to a single thread. The entire table is loaded into memory on a single CAS worker node and processed in a single thread on that node. Create a new output table named **predict**.

2. Specify the **Sum statement** to store running totals for the variables **Actual** and **Predict**.

3. Run the **tableDetails action** and the **tableInfo action** on the table to verify that processing is limited to a single node.
Redistribute a Table After Specifying the SINGLE= Option

### Output 2.18 Partial Results for TableDetails Action Showing Table Predict on a Single Worker Node

<table>
<thead>
<tr>
<th>Node</th>
<th>Number of Blocks</th>
<th>Active Blocks</th>
<th>Rows</th>
<th>Fixed Data size</th>
<th>Variable Data size</th>
<th>Blocks Mapped</th>
<th>Memory Mapped</th>
<th>Blocks Unmapped</th>
<th>Memory Unmapped</th>
<th>Blocks Allocated</th>
<th>Memory Allocated</th>
<th>Index Size</th>
<th>Compressed Size</th>
<th>Compression Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>server55.abc.com</td>
<td>1</td>
<td>1</td>
<td>1152</td>
<td>161578</td>
<td>1</td>
<td>161562</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>server7.abc.com</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

1. Use the partition action to redistribute the table. This will ensure that you are getting maximum efficiency from all the worker nodes in your environment.

2. Run the TableDetails action on the table to verify that the table is distributed across all nodes.

### Output 2.19 Partial Results for TableDetails Action Showing Table Re-partitioned onto Multiple Worker Nodes

```
proc cas;
  table.partition / table="predict", /*1*/
  casout={name="predict", replace=true},
  pickerthreads=1;
  table.tabledetails / /*2*/
    level="node"
    name="predict";
  casout={name="predict", replace=true};
run;
quit;
```

1. Use the partition action to redistribute the table. This will ensure that you are getting maximum efficiency from all the worker nodes in your environment.

2. Run the TableDetails action on the table to verify that the table is distributed across all nodes.
Key Ideas

- The DATA step in CAS automatically runs in multiple threads on every available computer in your system by default. To run the DATA step in multiple threads, specify the SINGLE=YES option.

- When you run the DATA step in a single thread in CAS, processing is localized to a single machine regardless of the number of available worker nodes in your environment. If a single-threaded DATA step processes large tables, then not only does the program run slowly, the machine executing the single thread can run out of disk or memory resources. Running the DATA step in a single thread is not recommended for very large tables.

- A SAS DATA step (a DATA step that is not executing in CAS) always runs in a single thread.

See Also

- "DATA Step Action Set" in SAS Viya: System Programming Guide
- "Single-threaded Processing" on page 10
- SINGLE= in SAS DATA Step Statements: Reference

View DATA Step Processing Information Using Automatic Variables

Example Code

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.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```sas
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;

You can also get information about the node (host) name and the number of threads the DATA step is running in by specifying the _HOSTNAME_ and _NTHREADS_ automatic variables. In the following example, the host name and thread count results are printed in the SAS log.

data mycas.cars;
  set mycas.cars;
  put _hostname_;  
  put _nthreads_; 
run;
### Output 2.20  Log Output for View DATA Step Processing Information Using Automatic Variables

```sas
1 OPTIONS NONOTES NOSTIMER NOSOURCE NOSYNTAXCHECK;
72
73 data mycas.cars;
74 set mycas.cars;
75 put _hostname_;  
76 put _nthreads_;  
77 run;
```

NOTE: Running DATA step in Cloud Analytic Services.
NOTE: The DATA step will run in multiple threads.

srvrnode002
srvrnode005
srvrnode007
192
srvrnode003	srvrnode004	srvrnode006

NOTE: Duplicate messages output by DATA step:

srvrnode002  (occurred 22 times)
srvrnode005  (occurred 27 times)
srvrnode007  (occurred 12 times)
192  (occurred 103 times)
srvrnode003  (occurred 17 times)
srvrnode004  (occurred 14 times)
srvrnode006  (occurred 11 times)

NOTE: There were 103 observations read from the table CARS in caslib CASUSER(sasdemo).

NOTE: The table cars in caslib CASUSER(sasdemo) has 103 observations and 4 variables.

NOTE: DATA statement used (Total process time):

real time 0.42 seconds

### Key Ideas

- A DATA step that has an input table automatically runs in multiple threads in CAS. 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.

- 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.

- There are restrictions on what language elements you can use in a DATA step running in CAS. For information about these restrictions, see “Restrictions” on page 19.
Use the Macro Facility to Generate CAS DATA Step Code

Example Code

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. Macros compile on the SAS server.

For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

```sas
data mycas.class; /* 1 */
    set sashelp.class; run;
data mycas.cars;
    set sashelp.cars; run;
data mycas.air;
    set sashelp.air; run;
libname mysas "\u\file-path\mycas";   /* 2 */
%macro load(tables=);
%let n=%sysfunc(countw(&tables,%str( )));    /* 3 */
%do i=1 %to &n;                       /* 4 */
    %let dsn=%scan(&tables,&i);
    data mysas.&dsn;                   /* 5 */
    set mycas.&dsn; run;
%end;
%end;                                 /* 6 */
%load(tables=class cars air);         /* 7 */
```

1. Use the DATA step to load three SASHELP data sets to CAS.
2. Create a SAS libref.
Start the macro definition.

Set \( n \) equal to the number of tables.

Loop through \( n \) times (for each table).

Create a SAS data set from each CAS table.

End macro definition.

**Example Code 2.2  Partial Log Output for Macro**

```latex
NOTE: Libref MYSAS was successfully assigned as follows:
      Engine:        V9
      Physical Name: /file-path/mycas

NOTE: There were 19 observations read from the data set MYCAS.CLASS.
NOTE: The data set MYCAS.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 MYCAS.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 MYCAS.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 Macro Language: Reference*
- “CAS Statement” in *SAS Cloud Analytic Services: User’s Guide*
- CASLIB statement option “_ALL_” in *SAS Cloud Analytic Services: User’s Guide*

**DATA Step Program Walk-through**

**Example Code**

The following example shows the steps that you need to perform to run the DATA step in CAS.
For information about how to run the examples in these sections, see Set Up Code For Examples on page 34.

1 Start a CAS session.

NOTE: ‘CASUSER(sasdemo)’ is now the active caslib.
NOTE: The CAS statement request to update one or more session options for session CASAUTO completed.
386  caslib _all_ assign;
NOTE: A SAS Library associated with a caslib can only reference library member names that conform
to SAS Library naming conventions.
NOTE: CASLIB CASUSER(sasdemo) for session CASAUTO will be mapped to SAS Library CASUSER.
NOTE: CASLIB CASUSERHDFS(sasuser) for session CASAUTO will be mapped to SAS Library CASUSERH.
387  libname mycas cas;
NOTE: Libref MYCAS was successfully assigned as follows:
   Engine:        CAS

The SESSOPTS= option in the CAS statement is used with the CASLIB= session option to ensure that the CASUSER personal caslib is set as the active caslib.

The CASLIB _ALL_ ASSIGN maps caslibs to SAS libraries. Caslibs provide a way to access in-memory tables and an associated data source. In this example, the personal caslib CASUSER is being used, so no CASLIB statement is needed. For more information about the CASUSER and CASUSERHDFS personal caslibs, see Personal, Predefined, and Custom Caslibs in SAS Cloud Analytic Services: Fundamentals: “Caslibs, Files, and Tables” in SAS Cloud Analytic Services: Fundamentals.

The LIBNAME CAS statement creates a SAS library that is associated with the CAS engine. Mycas is the CAS engine libref.

2 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='casuser'
   replace;
run; quit;

Here is the log output:

NOTE: The UUID 'session-UUID' is connected using session CASAUTO.
318     load data=sashelp.cars
319     outcaslib='casuser'
320     replace;
NOTE: SASHELP.CARS was successfully added to the "casuser" caslib as "CARS".

3 Run the DATA step in CAS. The DATA step runs in CAS in multiple threads by default.
data mycas.cars2(promote=yes)/sessref=casauto;  /* 1 */
set mycas.cars;                                     /* 2 */
format Combined 2.;
Combined = (MPG_Highway*.45) + (MPG_City*.55);    /* 3 */
if type="SUV" and origin="USA";                   /* 4 */
keep make type MPG_City MPG_Highway Combined;
by make;
run;
title 'Combined Average MPG for SUVs in the USA By Make';
proc print data=mycas.cars2; run;                  /* 5 */

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. Global scope lets other sessions access the table.
b Specify the in-memory table that you loaded in step 2. Load data to CAS, on page 89 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

<table>
<thead>
<tr>
<th>Obs</th>
<th>Make</th>
<th>Type</th>
<th>MPG_City</th>
<th>MPG_Highway</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cadillac</td>
<td>SUV</td>
<td>16</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>2</td>
<td>Cadillac</td>
<td>SUV</td>
<td>14</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>Saturn</td>
<td>SUV</td>
<td>21</td>
<td>26</td>
<td>23</td>
</tr>
<tr>
<td>4</td>
<td>Dodge</td>
<td>SUV</td>
<td>15</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>Buick</td>
<td>SUV</td>
<td>19</td>
<td>26</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Buick</td>
<td>SUV</td>
<td>15</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Pontiac</td>
<td>SUV</td>
<td>19</td>
<td>26</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>Mercury</td>
<td>SUV</td>
<td>16</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>Lincoln</td>
<td>SUV</td>
<td>13</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>Lincoln</td>
<td>SUV</td>
<td>13</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>Chevrolet</td>
<td>SUV</td>
<td>19</td>
<td>22</td>
<td>20</td>
</tr>
<tr>
<td>12</td>
<td>Chevrolet</td>
<td>SUV</td>
<td>14</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>Chevrolet</td>
<td>SUV</td>
<td>16</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>14</td>
<td>Chevrolet</td>
<td>SUV</td>
<td>14</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>15</td>
<td>GMC</td>
<td>SUV</td>
<td>16</td>
<td>19</td>
<td>17</td>
</tr>
<tr>
<td>16</td>
<td>GMC</td>
<td>SUV</td>
<td>13</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>GMC</td>
<td>SUV</td>
<td>15</td>
<td>19</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>Jeep</td>
<td>SUV</td>
<td>20</td>
<td>24</td>
<td>22</td>
</tr>
<tr>
<td>19</td>
<td>Jeep</td>
<td>SUV</td>
<td>16</td>
<td>19</td>
<td>17</td>
</tr>
<tr>
<td>20</td>
<td>Jeep</td>
<td>SUV</td>
<td>16</td>
<td>21</td>
<td>18</td>
</tr>
<tr>
<td>21</td>
<td>Hummer</td>
<td>SUV</td>
<td>10</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>22</td>
<td>Ford</td>
<td>SUV</td>
<td>16</td>
<td>23</td>
<td>20</td>
</tr>
<tr>
<td>23</td>
<td>Ford</td>
<td>SUV</td>
<td>10</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>24</td>
<td>Ford</td>
<td>SUV</td>
<td>15</td>
<td>20</td>
<td>17</td>
</tr>
<tr>
<td>25</td>
<td>Ford</td>
<td>SUV</td>
<td>15</td>
<td>19</td>
<td>17</td>
</tr>
</tbody>
</table>
Here is the log output:

348  data mycas.cars2(promote=yes)/sessref=casauto;
349    set mycas.cars;
350 format Combined 2.;
351    Combined = (MPG_Highway*.45) + (MPG_City*.55);
352    if type="SUV" and origin="USA";
353    keep make type MPG_City MPG_Highway Combined;
354    by make;
355  run;

NOTE: Running DATA step in Cloud Analytic Services.
NOTE: There were 428 observations read from the table CARS in caslib CASUSER(sasdemo).
NOTE: The table cars2 in caslib CASUSER(sasdemo) has 25 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time           0.34 seconds
      cpu time            0.03 seconds

357  title 'Combined Average MPG for SUVs in the USA By Make';
358  proc print data=mycas.cars2; run;

NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.11 seconds
      cpu time            0.04 seconds

4  Save the table and list information about the table.

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='casuser' replace;
  contents casdata='cars2';
run; quit;

Here is the log output:
5 Drop the promoted table from memory. You cannot replace a promoted table with the same name, so you must drop the table first if you want to replace it.

```
proc casutil;
  droptable casdata='cars2';
run; quit;
```

**Note:** You can also delete tables in SAS Studio by using 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.

```
options cashost="srvrnode001" casport=10380;
cas casauto sessopts=(caslib='casuser');
libname mycas cas;
caslib _all_ assign;
proc casutil;
  load data=sashelp.cars outcaslib='casuser' replace;
run; quit;

data mycas.cars2(promote=yes)/sessref=casauto;
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;
```
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" on page 18.

- When the DATA step runs in CAS, 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 "The DATA Step and CAS" on page 3.

- When the DATA step runs in CAS, it runs in all available threads by default. See "Controlling Threaded Processing" on page 11 for more information about controlling threaded processing.

- For a list of requirements for running the DATA step in CAS, see "Running the DATA Step in CAS" on page 4.

- You can use the CASUTIL procedure to do more than load your SAS data sets into CAS. For more information, see "CASUTIL Procedure" in SAS Cloud Analytic Services: User’s Guide.

See Also

- An Introduction to SAS Viya Programming