# SAS® LASR™ Analytic Server 2.81: Reference Guide

SAS® Documentation
September 19, 2017

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *SAS® LASR™ Analytic Server 2.81: Reference Guide*. Cary, NC: SAS Institute Inc.

**SAS® LASR™ Analytic Server 2.81: Reference Guide**



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

September 2017



2.81-P1:inmsref

# Contents

# What's New In SAS LASR Analytic Server

## Overview

SAS LASR Analytic Server 2.81 includes the following changes:

- Distributed servers can use separate networks for internal and external communication. This enhancement is provided by the SAS High-Performance Analytics Infrastructure that is used by the server.
- When the FORCESIGNER option is used to load a table, the user ID that starts the server is reported as the table owner.
- The IMSTAT procedure is enhanced to provide error messages when unsupported data set options are specified.

## Enhancements for the FORCESIGNER Option

The SIGNER and FORCESIGNER options enable administrators to set data access controls in SAS metadata. The SIGNER= option is used to specify a URI to the SAS LASR Authorization Service. The FORCESIGNER option forces the server to use access controls that are set in SAS metadata.

In previous releases, when a table is loaded into the server and the FORCESIGNER option is used, no table owner was identified. In this release, the user ID that starts the server is reported as the table owner.

When the server is deployed with SAS Visual Analytics, the administrator web application reports the same user ID in the **Loaded By** column.

## Enhancements to the IMSTAT Procedure

In previous releases, the following data set options were silently ignored when they were specified in the the PROC IMSTAT statement or the TABLE statement. These options are not supported by the server. In this release, a warning message is printed to the SAS log when they are used.

- OBS=
- FIRSTOBS=

- DROP=
- KEEP=
- RENAME=

*Chapter 1*

# Introduction to the SAS LASR Analytic Server

# What Is SAS LASR Analytic Server?

The SAS LASR Analytic Server is an analytic platform that provides a secure, multi-user environment for concurrent access to data that is loaded into memory. The server can take advantage of a distributed computing environment by distributing data and the workload among multiple machines and performing massively parallel processing. The server can also be deployed on a single machine where the workload and data volumes do not demand a distributed computing environment.

The server handles both big data and smaller sets of data, and it is designed with a high-performance, multi-threaded, analytic code. The server processes client requests at extraordinarily high speeds due to the combination of hardware and software that is designed for rapid access to tables in memory. By loading tables into memory for analytic processing, the server enables business analysts to explore data and discover relationships in data at the speed of RAM.

The server can also perform text analysis on unstructured data. The unstructured data is loaded to memory in the form of a table, with one document in each row. The TEXTPARSE statement in the IMSTAT procedure can then provide similar analysis to what is available with the HPTMINE procedure.

Another use for the analytic platform that the server provides is to create a recommender system. Creating recommender systems introduces the concept of an application in the

server. The recommender system contains the application and might contain four or five tables. Each of the tables can be used in different ways, depending on the task and which method you apply. For example, making an item-based prediction for a nearest-neighbor method requires different data structures than a singular-value decomposition. You can associate a particular method or a set of methods with the application. You can execute one method or an ensemble. The flexibility provided by the server enables you to add and drop methods from the application. As a modeler, you want to explore and evaluate with different methods and different parameter configurations for the methods until you have optimized the system for your purposes. Then, you can deploy the recommender system in an online scoring environment.

The architecture for the server was originally designed for optimal performance in a distributed computing environment. A distributed server runs on multiple machines. A typical distributed configuration is to use a series of blades as a cluster. Each blade contains both local storage and large amounts of memory. Local storage is used to store large data sets in distributed form. Data is loaded into memory and made available so that clients can quickly access that data.

For distributed deployments, having local storage available on machines is critical in order to store large data sets in a distributed form. The server supports the Hadoop Distributed File System (HDFS) as a co-located data provider. HDFS is used because the server can read from and write to HDFS in parallel. In addition, HDFS provides replication for data redundancy. HDFS stores data as blocks in distributed form on the blades and the replication provides failover capabilities.

In a distributed deployment, the server also supports some third-party vendor databases as co-located data providers. Teradata Data Warehouse Appliance and Greenplum Data Computing Appliance are massively parallel processing database appliances. You can install the SAS LASR Analytic Server software on some of the machines in either appliance. The server can read in parallel from the data in the appliance.

For the SAS LASR Analytic Server 1.6 release (concurrent with the SAS Visual Analytics 6.1 release) the server supports a non-distributed deployment. A non-distributed server can perform the same in-memory analytic operations as a distributed server. However, a non-distributed deployment does not support parallel I/O from HDFS or third-party vendor appliances.

# How Does the SAS LASR Analytic Server Work?

## *Distributed SAS LASR Analytic Server*

The server provides a client/server environment where the client connects to the server, sends requests to the server, and receives results back from the server. The server-side environment is a distributed computing environment. A typical deployment is to use a series of blades in a cluster. In addition to using a homogeneous hardware profile, the software installation is also homogeneous. The same operating system is used throughout and the same SAS software is installed on each blade that is used for the server. In order for the software on each blade to share the workload and still act as a single server, the SAS software that is installed on each blade implements the Message Passing Interface (MPI). The MPI implementation is used to enable communication between the blades.

After a client connection is authenticated, the server performs the operations requested by the client. Any request (for example, a request for summary statistics) that is authorized will execute. After the server completes the request, there is no trace of the

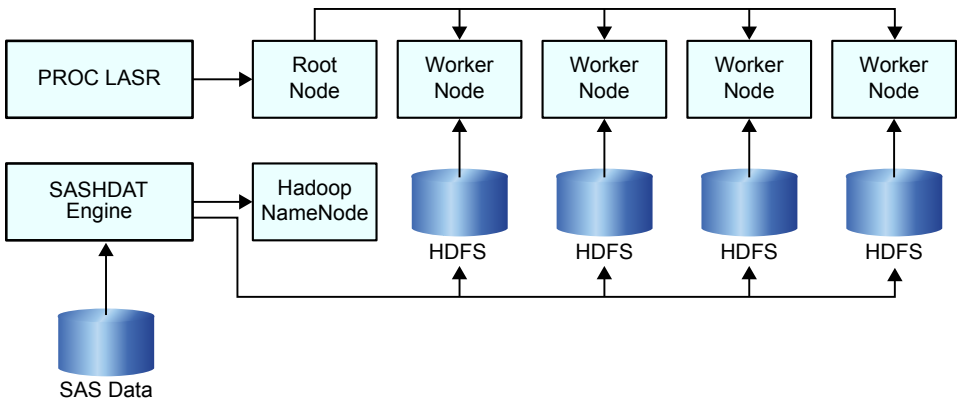
request. Every client request is executed in parallel at extraordinarily high speeds, and client communication with the server is practically instantaneous and seamless.

There are two ways to load data into a distributed server:

- **load data from tables and data sets.** You can start a server instance and directly load tables into the server by using the SAS LASR Analytic Server engine or the LASR procedure from a SAS session that has a network connection to the cluster. Any data source that can be accessed with a SAS engine can be loaded into memory. The data is transferred to the *root node* of the server and the root node distributes the data to the *worker nodes*. You can also append rows to an in-memory table with the SAS LASR Analytic Server engine.
- **load tables from a co-located data provider.**
  - Tables can be read from the Hadoop Distributed File System (HDFS) or an NFS-mounted distributed file system. You can use the SASHDAT engine to add tables to HDFS. When a table is added to HDFS, it is divided into blocks that are distributed across the machines in the cluster. The server software is designed to read data in parallel from HDFS. When used to read data from HDFS, the LASR procedure causes the worker nodes to read the blocks of data that are local to the machine.
  - Tables can also be read from a third-party vendor database. For distributed databases like Teradata and Greenplum, the SAS LASR Analytic Server can access the data in the appliance.

The following figure shows the relationship of the root node, the worker nodes, and how they interact when working with large data sets in HDFS. As described in the previous list, the LASR procedure communicates with the root node and the root node directs the worker nodes to read data in parallel from HDFS. The figure also indicates how the SASHDAT engine is used to transfer data.

***Figure 1.1*** *Relationship of PROC LASR and the SASHDAT Engine*




*Note:* The preceding figure shows a distributed architecture that uses HDFS. For deployments that use a third-party vendor database, the architecture is also distributed, but different procedures and software components are used for distributing the data.

After the data is loaded into memory on the server, it resides in memory until the table is unloaded or the server terminates. After the table is in memory, client applications that are authorized to access the table can send requests to the server and receive the results from the server.

In-memory tables can be saved. You can use the SAS LASR Analytic Server engine to save an in-memory table as a SAS data set or as any other output that a SAS engine can use. This method of using an engine transfers the data across the network connection. For large tables, saving to HDFS is supported with the LASR and IMSTAT procedures. This strategy saves the data in parallel and keeps the data on the cluster.

### *Non-Distributed SAS LASR Analytic Server*

Most of the features that are available with a distributed deployment also apply to the non-distributed deployment too. Any limitations are related to the reduced functionality of using a single-machine rather than a distributed computing environment.

In a non-distributed deployment, the server acts in a client/server fashion where the client sends requests to the server and receives results back. The server performs the analytic operations on the tables that are loaded in to memory. As a result, the processing times are very fast and the results are delivered almost instantaneously.

You can load tables to a non-distributed server with the SAS LASR Analytic Server engine. Any data source that SAS can access can be used for input and the SAS LASR Analytic Server engine can store the data as an in-memory table. The engine also supports appending data.

You can save in-memory tables by using the SAS LASR Analytic Server engine. The tables can be saved as a SAS data set or as any other output that a SAS engine can use.

The LASR procedure cannot be used with a non-distributed server. The procedure operates in a distributed computing environment only.

## About the SAS Plug-ins for Hadoop

SAS has discontinued delivery of SAS High-Performance Deployment for Hadoop. Instead, SAS offers the SAS Plug-ins for Hadoop package that enables SAS LASR Analytic Server to work with SASHDAT files in HDFS. The package includes the following:

- two JAR files that provide services that run inside Hadoop (**`sas.lasr.jar`** and **`sas.lasr.hadoop.jar`**)
- an executable file, **saslasrfd**, that facilitates reading data in parallel into SAS LASR Analytic Server
- a JAR file that enables resource management with YARN (**`sas.grid.provider.yarn.jar`**)

The software enables a cluster to support parallel I/O with SAS LASR Analytic Server as well as operating with the SASHDAT engine.

### *See Also*

*SAS High-Performance Analytics Infrastructure: Installation and Configuration Guide*

## Benefits of Using the Hadoop Distributed File System

Loading data from disk to memory is efficient when the SAS LASR Analytic Server is co-located with a distributed data provider. The Hadoop Distributed File System (HDFS) acts as a co-located data provider. HDFS offers some key benefits:

- **Parallel I/O.** The SAS LASR Analytic Server can read data in parallel at very impressive rates from a co-located data provider.
- **Data redundancy.** By default, two copies of the data are stored in HDFS. If a machine in the cluster becomes unavailable or fails, the SAS LASR Analytic Server instance on another machine in the cluster retrieves the data from a redundant block and loads the data into memory.
- **Homogeneous block distribution.** HDFS stores files in blocks. The SAS implementation enables a homogeneous block distribution that results in balanced memory utilization across the SAS LASR Analytic Server and reduces execution time.

## MapR Distribution for Apache Hadoop

### *How Does SAS LASR Analytic Server Work with MapR?*

The machines that are used for SAS LASR Analytic Server act as NFS clients to mount the MapR cluster. No client software is required.

At installation time, there is a configuration prompt that is related to specifying the mount point, such as **`/mapr/my.cluster.com`**. When the path is specified, SAS LASR Analytic Server and the SASHDAT engine use the specified path as a root directory path. For example, if the directory **`/mapr/my.cluster.com/hps`** exists and **`/mapr/my.cluster.com`** was specified as the mount point, then the following SASHDAT engine LIBNAME statement refers to the **`/hps`** directory:

```
options set=GRIDHOST="grid001.example.com" set=GRIDINSTALLLOC="/opt/TKGrid";
libname hps sashdat path="/hps";
```

### *How is working with MapR different from working with HDFS?*

Working with MapR instead of HDFS is largely transparent. All the benefits of working with HDFS apply to working with MapR.

- SAS programs that use the SASHDAT engine to distribute data can run without modification.
- SAS programs that use the LASR procedure to load tables from HDFS can load data over the NFS mounts from MapR without modification.

From an architectural or network topology standpoint, a deployment that is co-located with HDFS has to run the server on every machine in the Hadoop cluster. With MapR, the number of machines used for the server does not need to be the same as the number of machines used for MapR. The use of NFS also means that the server can be co-

located and use the same machines as MapR or the MapR cluster can be remote. In either case, NFS provides the advantages of co-located data access.

## Sample Code

The following code sample shows how the following components are used with the MapR distribution:

- SAS LASR Analytic Server engine
- SASHDAT engine
- LASR procedure
- IMSTAT procedure

```
options set=GRIDHOST="grid001.example.com" set=GRIDINSTALLLOC="/opt/TKGrid";

proc lasr create port=10010;
  performance nodes=all;
run;

libname example sasiola tag="hps" port=10010;  1
libname hps sashdat path="/hps";  2

data hps.prdsale(squeeze=yes copies=2);  3
  set sashelp.prdsale;
run;

/*
proc datasets lib=hps;  4
quit;
*/

proc lasr add data=hps.prdsale noclass port=10010;  5
run;

proc imstat data=example.prdsale;
  tableinfo;
  columninfo;
  numrows;
  fetch / format to=5;  6
run;

  where prodtype="FURNITURE";
  save path="/hps/prdsalefurn.sashdat" fullpath copies=2;  7
run;
quit;
```

1 The example libref is created. It references the server that was started with the PROC LASR CREATE statement. The libref uses TAG="hps" because the PATH= option in the next step is a directory named **/hps**.

2 The Hps libref references the **/hps** directory in MapR-FS. No special options are required to work with MapR.

3 A DATA step is used to distribute the Sashelp.Prdsale data set. All SASHDAT engine data set options are supported. No special options are required to work with MapR.

4 You can use the DATASETS procedure to list the SASHDAT tables. The engine ignores files that are not SASHDAT tables.

5 The LASR procedure loads the table to memory from MapR transparently over the NFS mounts.

6 The example libref is used with the IMSTAT procedure to reference the in-memory table. All the analytic and management capabilities of the server are available to operate on the in-memory table.

7 The WHERE clause sets a limit on the Prodtype variable. The SAVE statement creates a new SASHDAT table in MapR-FS that include observations that pass the WHERE clause.

***Output 1.1*** *Results for PROC LASR ADD*

**The LASR Procedure**

| Performance Information | |
|---|---|
| Host Node | [illegible] |
| Execution Mode | Distributed |
| Number of Compute Nodes | 4 |

| Data Access Information | | | |
|---|---|---|---|
| Data | Engine | Role | Path |
| HPS.PRDSALE | SASHDAT | Input | Parallel, Symmetric |

***Output 1.2*** *Results for TABLEINFO, COLUMNINFO, and NUMROWS*

**The IMSTAT Procedure**

| Table Information | | | | | | |
|---|---|---|---|---|---|---|
| Table Name | Label | NLS encoding | Number of Rows | Number of Columns | Owner | Created |
| HPS.PRDSALE | Product Sales Data | latin1 | 1440 | 10 | [illegible] | Tue Mar 31 14:14:53 2015 |

| Column Information for Table HPS.PRDSALE | | | | | |
|---|---|---|---|---|---|
| Id | Column | Type | Length | Format | Label |
| 1 | ACTUAL | Num | 8 | DOLLAR12.2 | Actual Sales |
| 2 | PREDICT | Num | 8 | DOLLAR12.2 | Predicted Sales |
| 3 | COUNTRY | Char | 10 | $CHAR10. | Country |
| 4 | REGION | Char | 10 | $CHAR10. | Region |
| 5 | DIVISION | Char | 10 | $CHAR10. | Division |
| 6 | PRODTYPE | Char | 10 | $CHAR10. | Product type |
| 7 | PRODUCT | Char | 10 | $CHAR10. | Product |
| 8 | QUARTER | Num | 8 | F8. | Quarter |
| 9 | YEAR | Num | 8 | F4. | Year |
| 10 | MONTH | Num | 8 | MONNAME3. | Month |

| Number of Rows Action for Table HPS.PRDSALE |
|---|
| Number of Records |
| 1440 |

***Output 1.3*** *Results for FETCH and SAVE*

| Selected Records from Table HPS.PRDSALE | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| ACTUAL | PREDICT | COUNTRY | REGION | DIVISION | PRODTYPE | PRODUCT | QUARTER | YEAR | MONTH |
| $925.00 | $850.00 | CANADA | EAST | EDUCATION | FURNITURE | SOFA | 1 | 1993 | Jan |
| $999.00 | $297.00 | CANADA | EAST | EDUCATION | FURNITURE | SOFA | 1 | 1993 | Feb |
| $608.00 | $846.00 | CANADA | EAST | EDUCATION | FURNITURE | SOFA | 1 | 1993 | Mar |
| $642.00 | $533.00 | CANADA | EAST | EDUCATION | FURNITURE | SOFA | 2 | 1993 | Apr |
| $656.00 | $646.00 | CANADA | EAST | EDUCATION | FURNITURE | SOFA | 2 | 1993 | May |

| Information from Saving Table HPS.PRDSALE | |
|---|---|
| Path | /hps/prdsalefurn.sashdat |
| Number of Records | 576 |
| Block Size (kBytes) | 172 |

# Components of the SAS LASR Analytic Server

## About the Components

The following sections identify some software components and interactions for SAS LASR Analytic Server.

## Root Node

When the SAS client initiates contact with the *grid host* to start a SAS LASR Analytic Server instance, the SAS software on that machine takes on the role of distributing and coordinating the workload. This role is in contrast to a worker node. This term applies to a distributed SAS LASR Analytic Server only.

## Worker Nodes

This is the role of the software that receives the workload from the root node. When a table is loaded into memory, the root node distributes the data to the worker nodes and they load the data into memory. If you are using a co-located data provider, each worker node reads the portion of the data that is local to the machine. The data is loaded into memory and requests that are sent to root node are distributed to the worker nodes. The worker nodes perform the analytic tasks on the data that is loaded in memory on the machine and then return the results to the root node. This term applies to a distributed SAS LASR Analytic Server only.

## In-Memory Tables

SAS LASR Analytic Server performs analytics on tables that are in-memory only. Typically, large tables are read from a co-located data provider by worker nodes. The tables are loaded quickly because each worker node is able read a portion of the data from local storage. Once the portion of the table is in memory on each worker node, the server instance is able to perform the analytic operations that are requested by the client. The analytic tasks that are performed by the worker nodes are done on the in-memory data only.

## Signature Files

SAS LASR Analytic Server uses two types of signature files, server signature files and table signature files. These files are used as a security mechanism for server management and for access to data in a server. When a server instance is started, a directory is specified on the PATH= option to the LASR procedure. The specified directory must exist on the machine that is specified as GRIDHOST= environment variable.

In order to start a server, the user must have Write access to the directory in order to be able to create the server signature file. In order stop a server, the user must have Read access to the server signature file so that it can be removed from the directory.

In order to load and unload tables on a server, the user must have Read access to the server signature file in order to interact with the server. Write permission to the directory

is needed to create the table signature file when loading a table and to delete the table signature file when unloading the table.

### *Server Description Files*

*Note:* Most administrators prefer to use the PORT= option in the LASR procedure rather than use server description files.

If you specify a filename in the CREATE= option in the LASR procedure, then you start a SAS LASR Analytic Server instance, the LASR procedure creates two files:

- a server description file
- a server signature file (described in the previous section)

The server description file contains information such as the host names of the machines that are used by the server instance and signature file information.

In the LASR procedure, the server description file is specified with the CREATE= option. The server description file is created on the SAS client machine that invoked PROC LASR.

## Administering the SAS LASR Analytic Server

### *Administering a Distributed Server*

Basic administration of a distributed SAS LASR Analytic Server can be performed with the LASR procedure from a SAS session. Server instances are started and stopped with the LASR procedure. The LASR procedure can be used to load and unload tables from memory though the SAS LASR Analytic Server engine also provides that ability.

The SASHDAT engine is used to add and delete tables from the Hadoop Distributed File System (HDFS). The tables are stored in the SASHDAT file format. You can use the DATASETS procedure with the engine to display information about tables that are stored in HDFS.

The HPDS2 procedure has a specific purpose for use with SAS LASR Analytic Server. In this deployment, the procedure is used to distribute data to the machines in an appliance. After the data are distributed, the SAS LASR Analytic Server can read the data in parallel from each of the machines in the appliance.

### *Administering a Non-Distributed Server*

A non-distributed SAS LASR Analytic Server runs on a single machine. A non-distributed server is started and stopped with the SAS LASR Analytic Server engine. A server is started with the STARTSERVER= option in the LIBNAME statement. The server is stopped when one of the following occurs:

- The libref is cleared (for example, **`libname lasrsvr clear;`**).
- The SAS program and session that started the server ends. You can use the SERVERWAIT statement in the VASMP procedure to keep the SAS program (and the server) running.
- The server receives a termination request from the SERVERTERM statement in the VASMP procedure.

A non-distributed deployment does not include a distributed computing environment. As a result, a non-distributed server does not support a co-located data provider. Tables are loaded and unloaded from memory with the SAS LASR Analytic Server engine only.

### Common Administration Features

As described in the previous sections, the different architecture for distributed and non-distributed servers requires different methods for starting, stopping, and managing tables with servers. However, the IMSTAT procedure works with distributed and non-distributed servers to provide administrators with information about server instances. The statements that provide information that can be of interest to administrators are as follows:

- SERVERINFO
- TABLEINFO

Administrators might also be interested in the SERVERPARM statement. You can use this statement to adjust the number of requests that are processed concurrently. You might reduce the number of concurrent requests if the number of concurrent users causes the server to consume too many sockets from the operating system.

### Features Available in SAS Visual Analytics Administrator

SAS LASR Analytic Server is an important part of SAS Visual Analytics. SAS Visual Analytics Administrator is a web application that provides an intuitive graphical interface for server management. You can use the application to start and stop server instances, as well as load and unload tables from the servers. Once a server is started, you can view information about libraries and tables that are associated with the server. The application also indicates whether a table is in-memory or whether it is unloaded.

For deployments that are co-located with Hadoop, an HDFS explorer enables you to browse the tables that are stored in HDFS. Once tables are stored in HDFS, you can to load them into memory in a server instance. Because SAS uses the special SASHDAT file format for the data that is stored in HDFS, the HDFS explorer also provides information about the columns, row count, and block distribution.

### Understanding Server Run Time

By default, servers are started and run indefinitely. However, in order to conserve the hardware resources in a distributed computing environment, server instances can be configured to exit after a period of inactivity. This feature applies to distributed SAS LASR Analytic Server deployments only. You specify the inactivity duration with the LIFETIME= option when you start the server.

When the LIFETIME= option is used, each time a server is accessed, such as to view data or perform an analysis, the run time for the server is reset to zero. Each second that a server is unused, the run timer increments to count the number of inactive seconds. If the run timer reaches the maximum run time, the server exits. All the previously used hardware resources become available to the remaining server instances.

## Distributing Data

### SAS Plug-ins for Hadoop

SAS provides the SAS Plug-ins for Hadoop that you can use to configure a Hadoop cluster as a co-located data provider. The SAS LASR Analytic Server software and the plug-ins are installed on the same hosts in the cluster. The SASHDAT engine can be used to distribute data to HDFS.

For more information, see Chapter 12, "Using the SASHDAT Engine," on page 437.

### PROC HPDS2 for Big Data

For deployments that use Greenplum or Teradata, the HPDS2 procedure can be used to distribute large data sets to the machines in the appliance. The procedure provides an easy-to-use and efficient method for transferring large data sets.

For deployments that use Greenplum, the procedure is more efficient than using a DATA step with the SAS/ACCESS Interface to Greenplum and is an alternative to using the gpfdist utility.

The SAS/ACCESS Interface for the database must be configured on the client machine. It is important to distribute the data as evenly as possible so that the SAS LASR Analytic Server has an even workload when the data is read into memory.

The following code sample shows a LIBNAME statement and an example of the HPDS2 procedure for adding tables to Greenplum.

```
libname source "/data/marketing/2012";

libname target greenplm
    server = "grid001.example.com"
    user = dbuser
    password = dbpass
    schema = public
    database = template1
    dbcommit=1000000;

proc hpds2 data = source.mktdata
    out = target.mktdata (distributed_by = 'distributed randomly'); 1

    performance host = "grid001.example.com"
        install = "/opt/TKGrid";;

    data DS2GTF.out;
        method run();
            set DS2GTF.in;
        end;
    enddata;
run;

proc hpds2 data = source.mkdata2
    out = target.mkdata2 (dbtype=(id='int')
        distributed_by='distributed by (id)'); 2

    performance host = "grid001.example.com"
        install = "/opt/TKGrid";
```

```
    data DS2GTF.out;
        method run();
            set DS2GTF.in;
        end;
    enddata;
run;
```

1 The rows of data from the input data set are distributed randomly to Greenplum.

2 The ID column in the input data set is identified as being an integer data type. The rows of data are distributed based on the value of the ID column.

For information about the HPDS2 procedure, see the *Base SAS Procedures Guide: High-Performance Procedures*. The procedure documentation is available from http://support.sas.com/documentation/cdl/en/prochp/66409/HTML/default/viewer.htm#prochp_hpds2_toc.htm.

### Bulkload for Teradata

The SAS/ACCESS Interface to Teradata supports a bulk-load feature. With this feature, a DATA step is as efficient at transferring data as the HPDS2 procedure.

The following code sample shows a LIBNAME statement and two DATA steps for adding tables to Teradata.

```
libname tdlib teradata
    server="dbc.example.com"
    database=hps
    user=dbuser
    password=dbpass
    bulkload=yes; 1

data tdlib.order_fact;
  set work.order_fact;
run;

data tdlib.product_dim (dbtype=(partno='int') 2
    dbcreate_table_opts='primary index(partno)'); 3
  set work.product_dim;
run;

data tdlib.salecode(dbtype=(_day='int' fpop='varchar(2)')
    bulkload=yes
    dbcreate_table_opts='primary index(_day,fpop)'); 4
    set work.salecode;
run;

data tdlib.automation(bulkload=yes
    dbcommit=1000000 5
    dbcreate_table_opts='unique primary index(obsnum)'); 6
    set automation;
    obsnum = _n_;
run;
```

1 Specify the BULKLOAD=YES option. This option is shown as a LIBNAME option but you can specify it as a data set option.

2 Specify a data type of **int** for the variable named partno.

3 Specify to use the variable named partno as the distribution key for the table.

4 Specify to use the variables that are named _day and fpop as a distribution key for the table that is named salecode.

5 Specify the DBCOMMIT= option when you are loading many rows. This option interacts with the BULKLOAD= option to perform checkpointing. Checkpointing provides known synchronization points if a failure occurs during the loading process.

6 Specify the **UNIQUE** keyword in the table options to indicate that the primary key is unique. This keyword can improve table loading performance.

### *Smaller Data Sets*

You can use a DATA step to add smaller data sets to Greenplum or Teradata. Transferring small data sets does not need to be especially efficient. The SAS/ACCESS Interface for the database must be configured on the client machine.

The following code sample shows a LIBNAME statement and DATA steps for adding tables to Greenplum.

```
libname gplib greenplm server="grid001.example.com"
    database=hps
    schema=public
    user=dbuser
    password=dbpass;

data gplib.automation(distributed_by='distributed randomly'); 1
    set work.automation;
run;

data gplib.results(dbtype=(rep='int') 2
    distributed_by='distributed by (rep)') 3;
    set work.results;
run;

data gplib.salecode(dbtype=(day='int' fpop='varchar(2)') 4
    distributed_by='distributed by day,fpop'); 5
    set work.salecode;
run;
```

1 Specify a random distribution of the data. This data set option is for the SAS/ACCESS Interface to Greenplum.

2 Specify a data type of **int** for the variable named rep.

3 Specify to use the variable named rep as the distribution key for the table that is named results.

4 Specify a data type of **int** for the variable named day and a data type of **varchar(2)** for the variable named fpop.

5 Specify to use the combination of variables day and fpop as the distribution key for the table that is named salecode.

The following code sample shows a LIBNAME statement and a DATA step for adding a table to Teradata.

```
libname tdlib teradata server="dbc.example.com"
    database=hps
    user=dbuser
    password=dbpass;
```

```
data tdlib.parts_dim;
    set work.parts_dim;
run;
```

For Teradata, the SAS statements are very similar to the syntax for bulk loading. For more information, see "Bulkload for Teradata" on page 14.

### See Also

*SAS/ACCESS for Relational Databases: Reference*

## Passwordless SSH

### What Is Passwordless SSH?

SSH is a network protocol that allows data to be exchanged using a secure channel between two networked devices. Passwordless SSH enables an identity to connect from one device to another without specifying a password. The identity can log on without a credential challenge, or it can invoke commands on the other device without a credential challenge.

### Who Needs Passwordless SSH?

For a non-distributed server, passwordless SSH is not applicable.

For a distributed server, the requirements for passwordless SSH are as follows:

- Each user that needs to start and stop servers and load and unload tables must have an account that is configured for passwordless SSH (on each machine in the cluster).
- If you use automated loading, the service account under which the scheduled task runs must be configured for passwordless SSH (on each machine in the cluster). This is necessary to perform tasks such as starting and stopping the server and loading and unloading tables.
- For deployments that include SAS Visual Analytics, the service account for SAS LASR Analytic Server Monitor must be configured for passwordless SSH (on each machine in the cluster). This is necessary to monitor hardware resources and processes for a distributed SAS LASR Analytic Server. This service account can be the same as the SAS installer account.

### How to Set Up Passwordless SSH

You can use a point-and-click interface to generate SSH keys and configure them for passwordless SSH automatically for administrator accounts. See the SAS High-Performance Computing Management Console: User's Guide.

Here are some tips:

- In the SAS High-Performance Computing Management Console, be sure to select the **Generate and Propagate SSH Keys** option on the Create User page. This ensures that passwordless SSH is configured correctly for the account.

- After you add user or group accounts to the machines in the cluster, you must restart the HDFS service if it is co-located. An error message such as the following indicates that a user is not recognized:

  ```
  ERROR: host02.example.com (192.168.1.240) User does not belong to  .
  ```

### *Generate SSH Keys Manually*

The recommended method is to use the SAS High-Performance Computing Management Console to generate SSH keys (as described in the preceding topic).

If you must generate SSH keys manually (for example, for existing user IDs), use the following steps:

1. Generate a private/public key pair on a Linux system. Enter the following command to generate the keys and avoid using a passphrase:

   ```
   ssh-keygen -t rsa -P ""
   ```

2. After the keys are generated, if passwordless SSH is required, then add the public key to the list of authorized keys by entering this command on the command line:

   ```
   cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
   ```

3. Check permissions on the **.ssh** directory and the files in your **.ssh** directory. The directory must be readable and writable by you only. The id_rsa file must be readable by you only. To verify access, enter the following command, and check the results:

   ```
   ls -asl ~/.ssh
   ```

   ```
   4 drwx------ 2 datamgr datamgr 4096 Jan 23 10:27 . a
   4 drwx------ 4 datamgr datamgr 4096 Jan 12 19:09 ..
   4 -rw-r--r-- 1 datamgr datamgr  397 Jan 23 10:27 authorized_keys
   4 -rw------- 1 datamgr datamgr  1675 Jan 23 10:00 id_rsa b
   4 -rw-r--r-- 1 datamgr datamgr  397 Jan 13 10:00 id_rsa.pub
   4 -rw-r--r-- 1 datamgr datamgr 1705 Jan 23 10:27 known_hosts
   ```

   1 The directory permissions for the .**ssh** directory indicate that access is denied for all users other than the directory owner.

   2 The id_rsa file is the private key. Read access and Write access are available to the file owner only.

*Note:* If the machines in the cluster are not configured to access the home directories for the users, create local home directories for the users. Copy the **.ssh** directory for each user to his or her local home directory. Make sure that the permissions are preserved.

### *About Passwordless SSH and Windows Clients*

If you need to access a distributed SAS LASR Analytic Server from a Windows client, then you need to perform the following steps to copy your SSH keys to the Windows machine.

To copy your SSH keys to a Windows machine:

1. Determine your Windows home directory. Enter the following command in a command window:

   ```
   echo %HOMEDRIVE%%HOMEPATH%
   ```

The results are typically something like **C:\Users\sasdemo**.

2. You can use Windows Explorer to drag-and-drop the **.ssh** directory from your UNIX home directory, or you can use a command like the following to copy it:

```
xcopy driverLetter:\.ssh\* "%HOMEDRIVE%%HOMEPATH%\.ssh" /s /i
```

These steps are typically necessary for deployments that use SAS Studio on a Windows client or SAS solutions that use Windows machines for the server tier.

### Troubleshooting

If access problems occur, use the following steps to help diagnose any SSH configuration errors:

1. Impersonate the user or ask the user to perform the following command that requires passwordless SSH:

```
/opt/TKGrid/bin/simsh hostname
```

If each of the machines in the cluster responds with a host name, then no passwordless SSH configuration error exists.

2. As root, log on to one of the machines in the cluster and monitor the logon access:

```
tail -f /var/log/secure
```

3. Review the messages in the **/var/log/secure** file. The following example shows that the file system access permissions for **/home/sas** are not set correctly:

```
Mar 14 22:12:36 hostname sshd[11235]: pam_unix(sshd:session): session opened
for user root by (uid=0)
Mar 14 22:12:57 hostname sshd[11266]: Authentication refused: bad ownership or
modes for directory /home/sas
```

## Memory Management

### About Physical and Virtual Memory

The amount of memory on a machine is the physical memory. The amount of memory that can be used by an application can be larger, because the operating system can provide virtual memory. Virtual memory makes the machine appear to have more memory available than there actually is, by sharing physical memory between applications when they need it and by using disk space as memory.

When memory is not used and other applications need to allocate memory, the operating system pages out the memory that is not currently needed to support the other applications. When the paged-out memory is needed again, some other memory needs to be paged out. Paging means to write some of the contents of memory onto a disk.

Paging does affect performance, but some amount of paging is acceptable. Using virtual memory enables you to access tables that exceed the amount of physical memory on the machine. So long as the time to write pages to the disk and read them from the disk is short, the server performance is good.

One advantage of SASHDAT tables that are read from HDFS is that the server performs the most efficient paging of memory.

### How Does the Server Use Memory for Tables?

When you load a table to memory with the SAS LASR Analytic Server engine, the server allocates physical memory to store the rows of data. This applies to both distributed and non-distributed servers.

When a distributed server loads a table from HDFS to memory with the LASR procedure, the server defers reading the rows of data into physical memory. You can direct the server to perform an aggressive memory allocation scheme at load time with the READAHEAD option for the PROC LASR statement.

*Note:* When a distributed server loads a table from either the Greenplum Data Computing Appliance or the Teradata Data Warehouse Appliance, physical memory is allocated for the rows of data. This is true even when the data provider is co-located.

### How Else Does the Server Use Memory?

Physical memory is used when the server performs analytic operations such as summarizing a table. The amount of memory that a particular operation requires typically depends on the cardinality of the data. In most cases, the cardinality of the data is not known until the analysis is requested. When the server performs in-memory analytics, the following characteristics affect the amount of physical memory that is used:

- Operations that use group-by variables can use more memory than operations that do not. The amount of memory that is required is not known without knowing the number of group-by variable combinations that are in the data.
- The memory utilization pattern on the worker nodes can change drastically depending on the distribution of the data across the worker nodes. The distribution of the data affects the size of intermediate result sets that are merged across the network.

Some requests, especially with high-cardinality variables, can generate large result sets. To enable interactive near-real-time work with high cardinality problems, the server allocates memory for data structures that speed performance. The following list identifies some of these uses:

- The performance for traversing and querying a decision tree is best when the tree is stored in the server.
- Paging through group-by results when you have a million groups is best done by storing the group-by structure in a temporary table in the server. The temporary table is then used to look up groups for the next page of results to deliver to the client.

### Monitoring Server Memory Use

SAS LASR Analytic Server 2.5 introduces the _T_LASRMEMORY and _T_TABLEMEMORY tables. These tables contain information about server memory

usage and table memory usage. The tables are always available with any SASIOLA engine libref because each table is created dynamically when you access the table.

***Table 1.1*** *Column Descriptions for the _T_LASRMEMORY Table*

| Column Name | Data Type | Description |
|---|---|---|
| Hostname | Character (64) | Identifies the machine. |
| CommitSize * | Numeric | Amount of memory that the memory manager has committed for the server. |
| WorkingSet * | Numeric | Amount of memory that is physically mapped to the process context for the server. |
| VirtualMemory ** | Numeric | Amount of virtual memory that is used by the server. |
| ResidentMemory ** | Numeric | Amount of physical memory currently in use by the server process. |
| AllocatedMemory | Numeric | Amount of memory that is used by the server, including memory that is used for tables. |
| TableAllocatedMemory | Numeric | Amount of memory that is used for table storage. |
| ChildSMPTableMemory | Numeric | Amount of memory that is used by a child non-distributed server for full copies of tables. |
| ChildSMPVirtualMemory | Numeric | Amount of virtual memory that is used by a child non-distributed server. |
| ChildSMPResidentMemory | Numeric | Amount of physical memory currently in use by a child non-distributed server. |

* Applies to non-distributed servers on Windows only. These column names align with terms used in the Microsoft Windows Resource Monitor.

** Applies to distributed and non-distributed servers on Linux only.

To view the table, you can use a program like the following:

```
libname example sasiola host="grid001.example.com" port=10010 tag=hps;

proc imstat;
    table example._T_LASRMEMORY;
    fetch;
quit;

/* Alternatively, use the PRINT procedure */
data lasrmemory;
  set example._T_LASRMEMORY;
run;

proc print data=lasrmemory;
    title "Non-distributed Server Memory Use";
    format _numeric_ sizekmg9.2;
run;
```

The previous program generates output like the following example.

***Figure 1.2*** *Contents of _T_LASRMEMORY for a Non-Distributed Server*

**Non-distributed Server Memory Usage**

| Obs | Hostname | CommitSize | WorkingSet | AllocatedMemory | TableAllocatedMemory |
|---|---|---|---|---|---|
| 1 | [illegible] | 174.52MB | 182.54MB | 44.24MB | 17.30MB |

For a distributed server, you might want to sum the values from each machine. See the following example:

```
libname example sasiola host="grid001.example.com" port=10010 tag=hps;

data distributed;
  set example._T_LASRMEMORY;
run;

proc print data=distributed;
  title "Distributed Server Memory Usage";
  format _numeric_ sizekmg9.2;
  sum    _numeric_;
run;
```

In the following display, notice that the first machine uses much less memory than the others. This is because of the following reasons:

- The first machine is the root node of a distributed server. The root node does not store rows of data from tables that are loaded into a distributed server.
- A child non-distributed server is started on the same machine as the root node for providing high-volume access to small tables. However, 0 KB is used for child non-distributed server table memory because the server did not place full copies of tables on that machine.

As more tables are added for high-volume access, this can lead to additional full copies of tables using memory on the root node as well as the worker nodes.

***Figure 1.3*** *Contents of _T_LASRMEMORY for a Distributed Server*

**Distributed Server Memory Usage**

| Obs | Hostname | VirtualMemory | ResidentMemory | AllocatedMemory | TableAllocatedMemory | ChildSMPTableMemory | ChildSMPVirtualMemory | ChildSMPResidentMemory |
|---|---|---|---|---|---|---|---|---|
| 1 | [illegible] | 659.24MB | 36.49MB | 222.21MB | 0.00KB | 0.00KB | 242.67MB | 7.64MB |
| 2 | [illegible] | 18.63GB | 36.27MB | 125.98MB | 10.28MB | 1.58MB | 575.97MB | 19.44MB |
| 3 | [illegible] | 18.46GB | 34.88MB | 125.98MB | 10.28MB | 1.58MB | 333.30MB | 11.80MB |
| 4 | [illegible] | 18.45GB | 34.81MB | 125.98MB | 10.28MB | 1.58MB | 333.30MB | 11.80MB |
| 5 | [illegible] | 18.45GB | 34.71MB | 125.98MB | 10.28MB | 0.00KB | 242.67MB | 7.64MB |
| | | **74.64GB** | **177.17MB** | **726.12MB** | **41.12MB** | **4.75MB** | **1.69GB** | **58.32MB** |

### Monitoring Table Memory Use

The _T_TABLEMEMORY table provides information about the amount of memory that is used for tables. The table is always available with any SASIOLA engine libref because the table is created dynamically when you access the table.

***Table 1.2*** *Column Descriptions for the _T_TABLEMEMORY Table*

| Column Name | Data Type | Description |
|---|---|---|
| Hostname | Character (64) | Identifies the machine. |
| Tablename | Character(64) | Identifies the table. |
| InMemorySize | Numeric | Amount of memory that is needed to store the table in memory. |
| UncompressedSize | Numeric | Amount of memory that is used by the table when it is not compressed. |
| CompressedSize | Numeric | Amount of memory that is used by the table when it is compressed. |
| TableAllocatedMemory | Numeric | Amount of memory that is used for table storage. |
| NumberRecords | Numeric | Number of rows from the table that are on the machine. |
| UseCount | Numeric | Number of processes that are using the table.<br>When the value is zero and the table is dropped, the memory is immediately freed. If the count is greater than zero and the table is dropped, the memory is not freed until the count drops to zero. |
| RecordLength | Numeric | Amount of memory that is used to store one row of the table. |
| ComputedColLength | Numeric | Amount of memory that is used to store columns that are created with the COMPUTE statement of the IMSTAT procedure. |
| InMemoryMappedSize | Numeric | Amount of memory that is mapped to a SASHDAT table. |
| ChildSMPTableMemory | Numeric | Amount of memory that is used by a child non-distributed server for full copies of tables. This field applies shown for distributed servers only. For more information, see "High Volume Access to Smaller Tables" on page 27. |

Memory that is used by temporary tables is not included in the calculations. They are excluded because temporary tables are typically either dropped after an analysis is performed or they are made available for general use with the PROMOTE statement. After a table is promoted, it is included in the memory use calculations.

For SASHDAT tables, the InMemoryMappedSize matches the InMemorySize. The TableAllocatedMemory value represents internal memory structures for the table and classification levels if the NOCLASS option was not specified.

To view the table, you can use a program like the following:

```
libname example sasiola host="grid001.example.com" port=10010 tag=hps;
```

```
%let sizecols = InMemorySize UncompressedSize
                CompressedSize TableAllocatedMemory
                InMemoryMappedSize ChildSMPTableMemory;
%let countcols = NumberRecords UseCount RecordLength ComputedColLength;

data tablemem;
    set example._T_TABLEMEMORY;
run;

proc print data=tablemem;
    title "Non-distributed Server Table Memory Usage";
    format &sizecols.  sizekmg9.2;
    format &countcols. 8.;
    sum _numeric_;
run;
```

*Note:* Even though the example uses the TAG=HPS option in the LIBNAME statement, the contents of the _T_TABLEMEMORY table include the memory used by all tables in the server.

The previous program generates output like the following example. In this example, the server has two in-memory tables.

***Figure 1.4*** *Contents of _T_TABLEMEMORY for a Non-distributed Server*

**Non-distributed Server Table Memory Usage**

| Obs | Hostname | Tablename | InMemorySize | UncompressedSize | CompressedSize | TableAllocatedMemory | NumberRecords | UseCount | RecordLength | ComputedColLength | InMemoryMappedSize |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | [illegible] | HPS.AIR | 51.19KB | 2.25KB | . | 59.81KB | 144 | 0 | 16 | 0 | 0.00KB |
| 2 | [illegible] | HPS.PRDSAL3 | 1.60MB | 1.58MB | . | 17.24MB | 11520 | 0 | 144 | 0 | 0.00KB |
| | | | **1.65MB** | **1.58MB** | **0.00KB** | **17.29MB** | **11664** | **0** | **160** | **0** | **0.00KB** |

For a distributed server, the output is similar, but the table includes a row for each machine. In the following example, a five-machine cluster has two in-memory tables, Energy and Prdsal3. The root node of the cluster is shown in the first two rows. The root node never holds rows of data and always indicates that 0 KB used is for TableAllocatedMemory.

***Figure 1.5*** *Contents of _T_TABLEMEMORY for a Distributed Server*

**Distributed Server Table Memory Usage**

| Obs | Hostname | Tablename | InMemorySize | UncompressedSize | CompressedSize | TableAllocatedMemory | NumberRecords | UseCount | RecordLength | ComputedColLength | InMemoryMappedSize | ChildSMPTableMemory |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | [illegible] | HPS.ENERGY | 0.00KB | 0.00KB | . | 0.00KB | 0 | 0 | 0 | 0 | 0.00KB | 0.00KB |
| 2 | [illegible] | HPS.PRDSAL3 | 0.00KB | 0.00KB | . | 0.00KB | 0 | 0 | 0 | 0 | 0.00KB | 0.00KB |
| 3 | [illegible] | HPS.ENERGY | 17.69GB | 17.69GB | . | 9.85MB | 118415 | 0 | 160368 | 0 | 17.69GB | 0.00KB |
| 4 | [illegible] | HPS.PRDSAL3 | 405.00KB | 405.00KB | . | 434.69KB | 2880 | 0 | 144 | 0 | 0.00KB | 1.58MB |
| 5 | [illegible] | HPS.ENERGY | 17.62GB | 17.62GB | . | 9.85MB | 117958 | 0 | 160368 | 0 | 17.62GB | 0.00KB |
| 6 | [illegible] | HPS.PRDSAL3 | 405.00KB | 405.00KB | . | 434.69KB | 2880 | 0 | 144 | 0 | 0.00KB | 1.58MB |
| 7 | [illegible] | HPS.ENERGY | 17.52GB | 17.52GB | . | 9.85MB | 117337 | 0 | 160368 | 0 | 17.52GB | 0.00KB |
| 8 | [illegible] | HPS.PRDSAL3 | 405.00KB | 405.00KB | . | 434.69KB | 2880 | 0 | 144 | 0 | 0.00KB | 1.58MB |
| 9 | [illegible] | HPS.ENERGY | 17.52GB | 17.52GB | . | 9.85MB | 117338 | 0 | 160368 | 0 | 17.52GB | 0.00KB |
| 10 | [illegible] | HPS.PRDSAL3 | 405.00KB | 405.00KB | . | 434.69KB | 2880 | 0 | 144 | 0 | 0.00KB | 0.00KB |
| | | | **70.35GB** | **70.35GB** | **0.00KB** | **41.10MB** | **482568** | **0** | **642048** | **0** | **70.35GB** | **4.75MB** |

The Energy table is a SASHDAT table. As a result, the value for the TableAllocatedMemory column is much less than the InMemorySize column because the memory is used only while the server operates on the table. In contrast, the Prdsal3 table is a small table and was loaded with the SASIOLA engine. As a result, the rows are in memory all the time and the overhead for the table structure makes the TableAllocatedMemory size greater than the InMemorySize value. There is some overhead for all tables, it is just more apparent with smaller tables.

The Prdsal3 table was also loaded with the FULLCOPYTO= data set option and a value of 3. The ChildSMPTableMemory column includes three rows that are a little more than 1.5 MB. This indicates the three machines that were selected to hold full copies of the Prdsal3 table.

### Accessing the Memory Tables from Other Applications

The _T_LASRMEMORY and _T_TABLEMEMORY tables do not exist until the table is referenced with the IMSTAT procedure or a libref from a SASIOLA LIBNAME engine. For example, the tables are not listed from PROC DATASETS or with the TABLEINFO statement in the IMSTAT procedure.

To access the information from other applications, especially SAS applications that rely on SAS metadata for tables, you can run a DATA step like the following example. The output table can be registered in SAS metadata and you can also manage the output table as part of an ETL process.

```
libname example sasiola host="grid001.example.com" port=10010
    tag=hps signer="https://server.example.com/SASLASRAuthorization";

options dsaccel=any msglevel=i;
data example.servermem(append=yes);
    set example._T_LASRMEMORY;
    dttm = datetime();
run;
```

In the example, the output table, servermem, is stored in the server and can be registered in SAS metadata to become available for reporting with applications like SAS Visual Analytics. If the server is secured with SAS LASR Authorization Service, then the SIGNER= option is needed and the account that runs a program like the example must have metadata-layer permissions. For more information about the permissions, see *SAS Visual Analytics: Administration Guide*.

### Managing Memory

The following list identifies some of the options that SAS provides for managing memory:

- You can use the TABLEMEM= option to specify a threshold for physical memory utilization.
- You can use the EXTERNALMEM= option to specify a threshold for memory utilization for SAS High-Performance Analytics procedures and other external processes.

By default, whenever the amount of physical memory in use rises above 75% of the total memory available on a node of a distributed server, adding tables (including temporary ones), appending rows, or any other operation that consumes memory for storing data fails.

If the machine has already crossed the threshold, your requests to add data are immediately rejected. If you attempt to add a table and the server crosses the threshold as the data is added, the server removes the table that you attempted to add and frees the memory. Similarly, if you attempt to append rows and the server crosses the threshold during the request, the entire append request fails. The table remains as it was before the append was attempted.

You can specify the threshold when you start a server with the TABLEMEM= option in the PROC LASR statement or alter it for a running server with the SERVERPARM statement in the VASMP procedure. By default, TABLEMEM=75 (%).

*Note:* The memory that is consumed by tables loaded from HDFS do not count toward the TABLEMEM= limit.

Be aware that the TABLEMEM= option does not specify the percentage of memory that can be filled with tables. The memory consumption is measured across all processes of a machine.

A separate memory setting can be applied to external processes that transfer data from a server on a worker node. SAS High-Performance Analytics procedures and SAS Cloud Analytic Services can do this. If you set the EXTERNALMEM= option in the PROC LASR statement or through the SERVERPARM statement in the IMSTAT procedure, then you are specifying the threshold of total memory (expressed as a percentage) at which the server stops sending data to an external process. The default value, 75% of memory, is useful to prevent a high-performance procedure from reading enough memory that the server begins to swap memory. If the high-performance procedure or external processes are running on a separate cluster, then you can increase this threshold to 100%.

### See Also


- "TABLEMEM=*pct*" on page 51
- "EXTERNALMEM=*pct*" on page 47


## Data Partitioning and Ordering

### Overview of Partitioning

By default, partitioning is not used and data are distributed in a round-robin algorithm. This applies to SASHDAT engine as well as SAS LASR Analytic Server. In general, this works well so that each machine in a distributed server has an even workload.

However, there are some data access patterns that can take advantage of partitioning. When a table is partitioned in a distributed server, all of the rows that match the partition key are on a single machine. If the data access pattern matches the partitioning (for example, analyzing data by Customer_ID partitioning the data by Customer_ID), then the server can direct the work to just the one machine. This can speed up analytic processing because the server knows where the data are.

However, if the data access pattern does not match the partitioning, processing times might slow. This might be due to the uneven distribution of data that can cause the server to wait on the most heavily loaded machine.

*Note:* You can partition tables in non-distributed SAS LASR Analytic Server deployments. However, all the partitions are kept on the single machine because there is no distributed computing environment.

### Understanding Partition Keys

Partition keys in SASHDAT files and in-memory tables are constructed based on the formatted values of the partition variables. The formatted values are derived using

internationalization and localization rules. (All formatted values in the server follow the internationalization and localization rules.)

All observations that compare equal in the (concatenated) formatted key belong to the same partition. This enables you to partition based on numeric variables. For example, you can partition based on binning formats or date and time variables use date and time formats.

A multi-variable partition still has a single value for the key. If you partition according to three variables, the server constructs a single character key based on the three variables. The formatted values of the three variables appear in the order in which the variables were specified in the PARTITION= data set option. For example, partitioning a table by the character variable REGION and the numeric variable DATE, where DATE is formatted with a MONNAME3. format:

```
data hdfslib.sales(partition=(region date) replace=yes);
   format date monname3.;
   set work.sales;
run;
```

The partition keys might resemble EastJan, NorthJan, NorthFeb, WestMar, and so on. It is important to remember that partition keys are created only for the variable combinations that occur in the data. It is also important to understand that the partition key is not a sorting of Date (formatted as MONNAME3.) within Region. For information about ordering, see "Ordering within Partitions" on page 26.

If the formats for the partition keys are user-defined, they are transferred to the LASR Analytic Server when the table is loaded to memory. Be aware that if you use user-defined formats to partition a SASHDAT file, the definition of the user-defined format is not stored in the SASHDAT file. Only the name of the user-defined format is stored in the SASHDAT file. When you load the SASHDAT file to a server, you need to provide the XML definition of the user-defined format to the server. You can do this with the FMTLIBXML= option to the LASR procedure at server start-up or with the PROC LASR ADD request.

### Ordering within Partitions

Ordering of records within a partition is implemented in the SASHDAT engine and the SAS LASR Analytic Server. You can order within a partition by one or more variables and the organization is hierarchical—that is ordering by A and B implies that the levels of A vary slower than those of B (B is ordered within A).

Ordering requires partitioning. The sort order of character variables uses national language collation and is sensitive to locale. The ordering is based on the raw values of the order-by variables. This is in contrast to the formation of partition keys, which is based on formatted values.

When a table that is partitioned and ordered in HDFS is loaded into memory on the server, the partitioning and ordering is maintained. You can append to in-memory tables that are partitioned and ordered. However, this does require a re-ordering of the observations after the observations are transferred to the server.

# High Volume Access to Smaller Tables

## Applicability and Special Considerations

The information about providing high-volume access to smaller tables applies to distributed servers only.

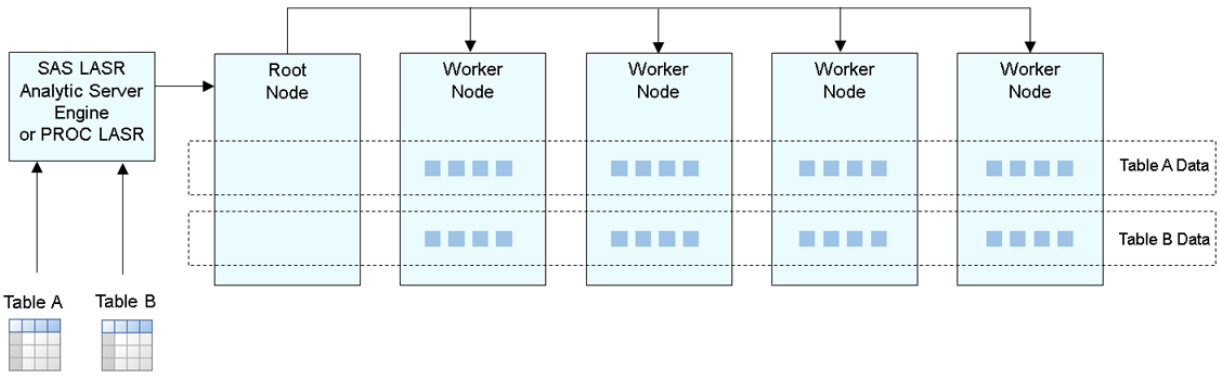
A distributed server scales to handle large tables by adding more machines to the cluster. This is commonly referred to as scaling horizontally. However, for smaller tables, adding more machines to the cluster to support a large number of concurrent users is not effective. Every request for data requires communication between the machines in the server to process the request. For smaller tables, typically between 2G and 20G, the time spent in communication is large compared to the computation time of the request.

The SAS LASR Analytic Server 2.6 release introduces a strategy for high-volume requests for smaller tables. The following sections provide more details.

## Depictions of Data Distribution

The following figure shows the conventional method of data distribution for a distributed server. As data are read with the LASR procedure or the SAS LASR Analytic Server engine, rows are distributed in a round-robin pattern to the machines in the cluster. (There is one exception—for partitioned tables, all the rows with the same partition key are placed on a single machine.) This results in an even workload and best performance for most scenarios.

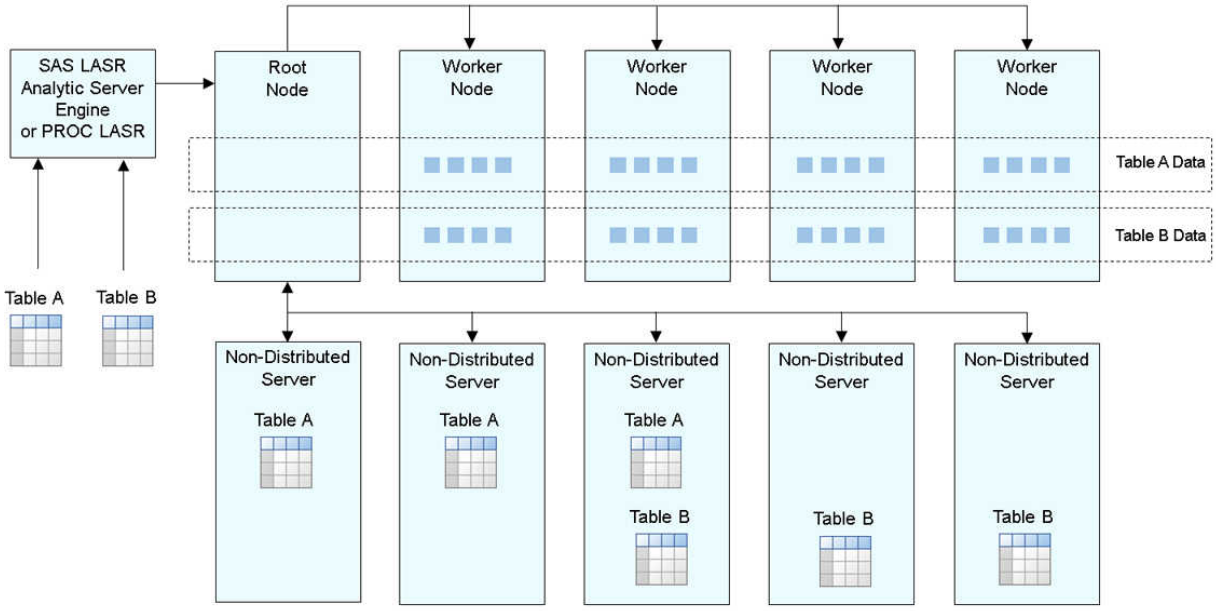
***Figure 1.6*** *Conventional Data Distribution*




The following figure represents the strategy for providing high-volume access to smaller tables. The rows from the tables are distributed in the same conventional fashion as

before. However, a non-distributed server is started on each of the machines and a full copy of the smaller tables is loaded into a user-specified number of servers.

***Figure 1.7*** *Full Copies of Tables Using Non-Distributed Servers*




As the root node accepts requests for data access to one of the smaller tables, the root node directs the request to one of the non-distributed servers with the table. Because a non-distributed server has a full copy of the table, it can process the request independent of other machines. The results are then sent back through the root node to the client.

## Sample Code

The following shows an example of using the FULLCOPYTO= option with the LASR procedure. You can use the option with co-located SASHDAT files or with a libref to any SAS engine.

```
options set=GRIDHOST="grid001.example.com" set=GRIDINSTALLLOC="/opt/TKGrid_REP";

libname hdfs sashdat path="/hps";

proc lasr add data=hdfs.iris fullcopyto=3 port=10010 ;
run;

libname example sasiola port=10010 tag="hps";

data example.prdsale(fullcopyto=5);
  set sashelp.prdsale;
run;

proc imstat;
  table example._T_TABLEMEMORY;
```

```
        where childsmptablememory > 0 and
           (tablename eq "HPS.IRIS" or tablename eq "HPS.PRDSALE");
        fetch hostname tablename childsmptablememory / format;
quit;
```

The results of the FETCH statement are shown in the following figure. The Iris table is on three machines and the Prdsale table is on machines. These results were generated on a cluster that has more than 8 machines. As a result, no machine has more than one full copy of a table. If the cluster were smaller or if the FULLCOPYTO= value were increased, then the results would show a host with more than one table.

**The IMSTAT Procedure**

| Selected Records from Table _T_TABLEMEMORY | | |
|---|---|---|
| **Hostname** | **Tablename** | **ChildSMPTableMemory** |
| [illegible]002 | HPS.IRIS | 7200 |
| [illegible]003 | HPS.IRIS | 7200 |
| [illegible]004 | HPS.IRIS | 7200 |
| [illegible]005 | HPS.PRDSALE | 172800 |
| [illegible]006 | HPS.PRDSALE | 172800 |
| [illegible]007 | HPS.PRDSALE | 172800 |
| [illegible]008 | HPS.PRDSALE | 172800 |
| [illegible]010 | HPS.PRDSALE | 172800 |

Requests for the Iris table are load-balanced across machines 002 to 004. Requests for the Prdsale table are load-balanced across machines 005 to 010. Information about the _T_TABLEMEMORY table is provided in "Memory Management" on page 18.

## Details

- Requests for a table are load-balanced across the machines that have a full copy of the table.
- Programming statements (or requests from applications like SAS Visual Analytics) that create temporary tables result in creating a temporary table on the non-distributed server only. Subsequent requests for data from the temporary table are directed to the same non-distributed server.
- The DROPTABLE statement runs on the distributed server first and then any non-distributed servers that have full copies.
- The PROMOTE statement results in a regular in-memory table on the distributed server and the non-distributed server that had the temporary table.
- The tables provide Read-Only access. Statements that attempt to modify the table, such as UPDATE, return an error.
- Statements that use more than one input table (such as the SCHEMA statement) run on the distributed server only. If an input table exists on a non-distributed server, then the table is sent to the distributed server before the processing begins.
- Some statements can run in the distribute server only, such as the SAVE statement. For these statements, if the table exists on a non-distributed server only, then the table is sent to the distributed server before the processing begins.
- The non-distributed servers continue to run until the distributed server is stopped.

- The non-distributed servers do not generate any logging records. If you have enabled logging, the log for the distributed server includes the requests that it handles and the requests that are sent to the non-distributed servers.

For information about enabling high-volume access to smaller tables in a metadata environment, see *SAS Visual Analytics: Administration Guide*.

# Data Compression

## Overview of Data Compression

SAS LASR Analytic Server supports compression for in-memory tables. All the analytic statements, such as PERCENTILES, LOGISTIC, and so on, in the IMSTAT procedure are supported for compressed tables as well as regular, uncompressed, tables. Clients like SAS Visual Analytics can also operate on compressed tables as well.

All compression is performed by the server. In other words, when you transfer a table to the server in a DATA step and specify the SQUEEZE= data set option, the rows are sent to the server as is, and the server compresses the rows. The server uses the zlib compression algorithm that is described in RFC 1950, "ZLIB Compressed Data Format Specification."

All data in a row, both character and number variables, are compressed. Every row in a table is compressed, the server does not support some rows in compressed form and others as uncompressed. The server can report the uncompressed size of the table, the compressed size, and the compression ratio.

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

## Compressed Tables and the DATA Step

The following example shows how to use the SQUEEZE= data set option for the SAS LASR Analytic Server.

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

```
libname example sasiola host="grid001.example.com" port=10010 tag=hps;

data example.prdsale(squeeze=yes);
    set sashelp.prdsale;
run;
```

After the table is loaded to memory, you can access the compressed table with the Example.Prdsale libref.

The server supports the APPEND= option for compressed tables. The following example shows how to add new rows (uncompressed) to the compressed table:

***Example Code 1.2*** *Appending Rows to a Compressed Table*

```
data example.prdsale(append=yes);
```

```
    somelib.newrows;
run;
```

Because the Example.Prdsale table is already compressed, the new rows are automatically compressed as they are appended to the table. Specifying SQUEEZE= with APPEND= has no effect. If the table is compressed, the server compresses the new rows. If the table is not compressed, the server does not compress the new rows (even if SQUEEZE=YES is specified). The compressed or uncompressed state of the table determines how the rows are appended.

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

***Example Code 1.3*** *Creating a Partitioned and Compressed Table*

```
data example.iris(partition=(species) squeeze=yes);
    set sashelp.iris;
run;

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

In the first DATA statement, the Iris data set is loaded to memory on the server and is partitioned by the formatted values of the Species variable. The table is also compressed. In the second DATA statement, the table is appended to with more rows. Because the in-memory table is already partitioned and compressed, the new rows are automatically partitioned and compressed when they are appended.

## *Compressed Tables and the LASR Procedure*

The LASR procedure is used for loading data to memory on distributed SAS LASR Analytic Server.

The following example shows how to read a SAS data set and compress it in memory on the server:

```
proc lasr add data=sashelp.prdsale port=10010 squeeze;
    performance host="grid001.example.com";
run;
```

The example uses the SQUEEZE option to read the Prdsale data set from the Sashelp library and compress it in-memory on the server. Be aware that you must specify the SQUEEZE option for each table that you want to load in compressed form. You cannot specify SQUEEZE with the CREATE option when you start a server and have the server automatically compress all tables.

The SQUEEZE option works when reading data from SAS/ACCESS engines, too. The resulting in-memory table is compressed whether you read data serially from a standard database or you read data in parallel from a distributed database like Greenplum Database.

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

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

If you specify a WHERE clause and the SQUEEZE option, then the server evaluates the WHERE clause as it reads data from HDFS and compresses the rows that meet the

WHERE clause criteria. The memory efficiencies of the SASHDAT table format are forfeited in this scenario because the server had to apply the WHERE clause.

If you do not specify a WHERE clause, then the server ignores the SQUEEZE option and relies on whether the SASHDAT table is compressed. If the SASHDAT table is compressed, then the in-memory representation of the table is also compressed. If the SASHDAT table is not compressed, then the in-memory representation is not compressed either. The server ignores the option so that it can keep the memory efficiencies of the SASHDAT table format—when a SASHDAT table is loaded to memory, the in-memory representation is identical to the on-disk representation.

### Performance Considerations

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

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

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

### Interactions

The interactions for compressed tables and SAS programs are as follows:

- You can use a compressed table in programs like any other table.
- You can define calculated columns for compressed tables with the COMPUTE statement or with the TEMPNAMES= and TEMPEXPRESS= options.
- You can use SIGNER= security with compressed tables.
- You can append to compressed tables with the SET statement or the APPEND= data set option. This is also supported for compressed tables that have partitioning. However, you cannot append to a compressed table that is partitioned and has an ORDERBY specification.
- You can use the UPDATE statement with a compressed table.
- You can use a compressed table with a statement in the IMSTAT procedure that produces a temporary table. Whether the resulting temporary table is compressed depends on whether you specify the TEMPSQUEEZE option in the IMSTAT procedure. The following statements in the IMSTAT procedure support creating temporary tables:
  - ACCESS
  - AGGREGATE
  - BALANCE

  - CLUSTER
  - CROSSTAB
  - DECISIONTREE
  - DISTINCT
  - GENMODEL
  - GLM
  - GROUPBY
  - LOGISTIC
  - MDSUMMARY
  - PARTITION
  - PERCENTILE
  - RANDOMWOODS
  - SCHEMA (does not support creating compressed temporary tables)
  - SCORE
  - SUMMARY
- You can use the BALANCE and PARTITION statements to create a compressed table when the TEMPSQUEEZE option is used. Applying orderby and compression at the same time has a significant performance penalty.
- You can use the UNCOMPRESS statement with a compressed and partitioned table. The temporary table that is created is partitioned according to the original table.
- You can use DELETEROWS and the PURGE option with a compressed table. There is a significant performance penalty for the PURGE option.
- You can create an empty table in compressed form, using either a DATA step or the CREATETABLE statement of the IMSTAT procedure. Although the table has no rows, rows that are appended to it later are compressed.
- You can use the SCORE statement with a compressed table.
- You can use a WHERE clause with the COMPRESS and UNCOMPRESS statements to move a subset of the rows to the temporary table.
- You can use the SAVE statement to save a compressed table to a SASHDAT table that is either compressed or uncompressed.
- You can use the SAVE statement to save an uncompressed table to a SASHDAT table that is in either compressed or uncompressed.
- You can specify the ORDERBY= option with the FETCH statement to read from a compressed table. The ORDERBY= option has a performance penalty.
- You can specify a sort order for a compressed table. Applying the sort has a performance penalty.
- You can use the SET statement to append a compressed table to an uncompressed table. The new rows are uncompressed because the base table is uncompressed.
- You can use the SET statement to append an uncompressed table to a compressed table. The new rows are compressed because the base table is compressed.
- You can use the UNCOMPRESS statement to create an uncompressed temporary table. The uncompressed table provides better performance.

### Limitations

The limitations for working with compressed tables are as follows:

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

## Data Encryption

### Overview

The SAS LASR Analytic Server 2.6 release introduces on-disk encryption for SASHDAT tables. SASHDAT tables are created with the SASHDAT engine or are saves to disk by the SAS LASR Analytic Server. AES encryption with 256-bit keys is used.

To comply with import and export restrictions, the encryption software is delivered in an installation program that is separate from the program that installs the server. The SAS TKGrid Encryption Extension is installed with the TKGrid_Sec_x86_64.sh file. For information about installation, see *SAS High-Performance Analytics Infrastructure: Installation and Configuration Guide*.

The data in a SASHDAT file is encrypted. The header is not encrypted and this enables procedures like PROC CONTENTS and components like HDFS browsers can show column information.

*Note:* If you implement encryption and find that performance suffers due to the processing required to decrypt data, consider anonymizing the data before it is transferred to the cluster and avoiding encryption.

For deployments that use SAS metadata, consider the following items:

- Passphrases are not preserved during promotion. After the initial import of an encrypted SASHDAT library or server, you must use SAS Management Console to re-apply the passphrase in the target environment.
- If you export or copy metadata for a SASHDAT library with encryption properties, the encryption key is ignored. You must use SAS Management Console to re-apply the passphrase.

### Example: SAS Metadata Environment

In a deployment that uses SAS metadata, such as SAS Visual Analytics, administrators can register the encryption settings in SAS metadata. The metadata for the connection

object for the Hadoop server must enable the SAS LASR authorization service. Users must have the Read permission on the SASHDAT table that is registered in metadata. Additional considerations are described in *SAS Visual Analytics: Administration Guide*.

```
options set=GRIDHOST="grid001.example.com" set=GRIDINSTALLLOC="/opt/TKGrid";
options metaserver="server.example.com" metaport=8561;  1
options metauser=sasdemo metapass="secret";

libname hdfs sashdat path="/hps" signer=
     "https://server.example.com/SASLASRAuthorization";  2

data hdfs.heart(replace=yes);  3
   set sashelp.heart;
run;

proc lasr create port=10010
    signer="https://server.example.com/SASLASRAuthorization";  4
  performance nodes=all;
run;

proc lasr add data=hdfs.heart signer=
    "https://server.example.com/SASLASRAuthorization"  5
    signerfilepolicy noclass port=10010 verbose;
run;

libname example sasiola tag="hps"
    signer="https://server.example.com/SASLASRAuthorization";

proc imstat signer="https://server.example.com/SASLASRAuthorization";
  table example.heart;
  save fullpath path="/hps/heart2" signerfilepolicy replace;  6
run;
```

1. The metadata-related options enable the SAS session to communicate with the SAS Metadata Server and to read encryption settings that are stored in metadata.
2. The SIGNER= option is used so that the engine can determine the metadata settings that are associated with a library. This enables the engine to exchange keys with the metadata server for decrypting tables as they are read. The library encryption settings also determine when an in-memory table should be encrypted as it is saved as a SASHDAT file.
3. If a SASHDAT engine library is registered in metadata that specifies encryption settings for a Hadoop server on host grid001.example.com (the GRIDHOST environment variable) and directory **/hps**(the PATH= option), then the Heart table is read from Sashelp and written to **/hps/heart.sashdat** in encrypted form.
4. In a metadata environment, a server must be started with the SIGNER= option.
5. The encrypted **/hps/heart.sashdat** file is decrypted and loaded to memory by the server.
6. Because the /hps directory that is associated with the Hadoop server is associated with encryption settings in SAS metadata (the same circumstance as item 3), the **/hps/heart2.sashdat** file is created with encryption.

### Example: Environment without Metadata

In a deployment that does not use SAS metadata, programmers can specify passphrases themselves.

```
options set=GRIDHOST="grid001.example.com" set=GRIDINSTALLLOC="/opt/TKGrid";

libname hdfs sashdat path="/hps";

data hdfs.heart(replace=yes encrypt=aes encryptkey="secret");
   set sashelp.heart;
run;

proc lasr create port=10010;
  performance nodes=all;
run;

proc lasr add data=hdfs.heart encryptkey="secret" noclass port=10010 verbose;
run;

libname example sasiola tag="hps";

proc imstat;
  table example.heart;
  save fullpath path="/hps/heart2" encryptkey="moresecret" replace;
run;
```

If you write programs with passphrases in a metadata environment, it is possible to specify ENCRYPT=AES and the passphrase in the ENCRYPTKEY= option instead of using the SIGNER= option. However, be aware that in the metadata environment, the passphrase is managed at the level of the Hadoop server or SASHDAT library. If you encrypt files in a directory with more than one passphrase, some files cannot be opened in a metadata environment.

As a best practice, if you are working with a metadata environment, using the SIGNER= option and managing passphrases in metadata is simpler than specifying passphrases in programs.

In order to decrypt a SASHDAT table, the server must read the entire table into memory at load time. The use of a WHERE clause with to subset the rows at load time with PROC LASR ADD prevents the server from loading the table.

### Encryption and Compression

Encrypted SASHDAT files are always uncompressed when they are loaded into the server. You can use compression to conserve disk space for an encrypted SASHDAT file. However, compressing an encrypted SASHDAT file does not conserve memory. Before an encrypted file is loaded, it must be decrypted—and decryption requires that the data be uncompressed.

# Server Logging

## Understanding Logging

Logging is an optional feature that can be enabled when a server instance is started with the LASR procedure. In order to conserve disk space, the default behavior for the server is to delete log files when the server exits. You can override this behavior with the KEEPLOG suboption to the LOGGING option when you start the server. You can also override this behavior with a suboption to the STOP option when you stop the server.

The server writes logs files on the grid host machine. The default directory for log files is **/tmp**. You can specify a different directory in the LOGGING option when you start the server instance. The log filename is the same as server signature file with a .log suffix (for example, LASR.924998214.28622.saslasr.log).

### See Also


- LOGGING option for the LASR procedure on page 49
- "Example 2: Starting a Server with Logging Options" on page 55
- "Starting and Stopping Non-Distributed Servers" on page 41


## What is Logged?

When a server is started with the LOGGING option, the server opens the log file immediately, but does not generate a log record to indicate that the server started. As clients like SAS Visual Analytics Explorer make requests to the server for data, the server writes a log record.

The server writes a log record when a request is received and completed by the server. The server does not write log records for activities that do not contact the server (for example, ending the SAS session).

A user that is configured with passwordless SSH to access the machines in the cluster, but who is not authorized to use a server instance is denied access. The denial is logged with the message **You do not have sufficient authorization to add tables to this LASR Analytic Server**. However, if a user is not configured correctly to access the machines in the cluster, communication with the server is prevented by the operating system. The request does not reach the server. In this second case, the server does not write a log record because the server does not receive the request.

## Log Record Format

The following file content shows an example of three log records. Line breaks are added for readability. Each record is written on a single line and fields are separated by commas. Each field is a name-value pair.

***File 1.1*** *Sample Log File Records*

```
ID=1,PID=28622,SASTime=1658782485.36,Time=Tue Jul 24 20:54:45 2012,User=sasdemo,
Host=grid001,LASRServer=/tmp/LASR.924998214.28622.saslasr,Port=56925,
```

```
RawCmd=action=ClassLevels name=DEPT.GRP1.PRDSALE "NlsenCoding=62",
ExeCmd=action=ClassLevels name=DEPT.GRP1.PRDSALE "NlsenCoding=62",JnlMsg=,
StatusMsg=Command successfully completed.,RunTime=        2.17

ID=2,PID=28622,SASTime=1658782593.09,Time=Tue Jul 24 20:56:33 2012,User=sasdemo,
Host=grid001,LASRServer=/tmp/LASR.924998214.28622.saslasr,Port=56925,
RawCmd=action=BoxPlot name=DEPT.GRP1.PRDSALE,
ExeCmd=action=BoxPlot name=DEPT.GRP1.PRDSALE,JnlMsg=,
StatusMsg=Command successfully completed.,RunTime=        0.12

ID=3,PID=28622,SASTime=1658825361.76,Time=Wed Jul 25 08:49:21 2012,User=sasdemo,
Host=grid001,LASRServer=/tmp/LASR.924998214.28622.saslasr,Port=56925,
RawCmd=action=APPEND_TABLE     ,ExeCmd=action=APPEND_TABLE     ,JnlMsg=,
StatusMsg=Command successfully completed.,RunTime=        0.09
```

***Table 1.3*** *Log Record Fields*

| Field Name | Description |
|---|---|
| ID | specifies a unique identifier for the action. |
| PID | specifies the operating system process identifier for the server. |
| SASTime | specifies the local time of execution in SAS datetime format. |
| Time | specifies the local time of execution as a date and time string. |
| User | specifies the user ID that started the server. |
| Host | specifies the host name of the grid host machine. |
| LASRServer | specifies the server signature file. |
| Port | specifies the network port number on which the server listens. |
| RawCmd | specifies the request that is received by the server. |
| ExeCmd | specifies the command that the server executes. This value can include default substitutions or adjustments to the RawCmd (for example, completion of variable lists). |
| JnlMsg | specifies an error message that is buffered in a journal object. |
| StatusMsg | specifies the status completion message. |
| RunTime | specifies the processing duration (in seconds). |

The server uses a journal object to buffer messages that can be localized. The format for the JnlMsg value is **`n-m:text`**.

n
: is an integer that specifies the message is the **`n`**th in the journal.

m
: is an integer that specifies the message severity.

text
: is a text string that specifies the error.

### Sample JnlMsg Values

```
JnlMsg=1-4:ERROR: The variable c1 in table WORK.EMPTY must be
numeric for this analysis.

JnlMsg=2-4:ERROR: You do not have sufficient authorization to add
tables to this LASR Analytic Server.
```

*Chapter 2*

# Non-Distributed SAS LASR Analytic Server



## About Non-Distributed SAS LASR Analytic Server

In a non-distributed deployment, the SAS LASR Analytic Server runs on a single machine. All of the in-memory analytic features that are available for the distributed deployment are also available for the non-distributed server.

One key difference has to do with reading and writing data. Because the server does not use a distributed computing environment, the server cannot be co-located with a data provider. The server does not read data in parallel and does not write SASHDAT files to HDFS.

## Starting and Stopping Non-Distributed Servers

### *Starting Servers*

Non-distributed servers are started and stopped with the SAS LASR Analytic Server engine. Starting a server requires the STARTSERVER= LIBNAME option.

To start a server:

***Example Code 2.1*** *Starting a Non-Distributed Server*

```
libname server1 sasiola
    startserver=( 2
        path="c:\temp"
        keeplog=yes maxlogsize=20 2
    )
    host=localhost 3
    port=10010 4
```

```
    tag='hps';
```

1. The STARTSERVER= option indicates to start a server. For information about the options, see "STARTSERVER= YES | NOSTARTSERVER =(*non-distributed-server-options*)" on page 421.
2. The KEEPLOG= option implies the LOGGING option and prevents the server from removing the log file when the server exits. The MAXLOGSIZE= option specifies to use up to 20 MB for the log file before the file is rolled over.
3. The HOST= specification is optional.
4. If you do not specify a PORT= value, then the server starts on a random port and sets the LASRPORT macro variable to the network port number.

Submitting the previous LIBNAME statement from a SAS session starts a server and the server remains running as long as the SAS session remains running. In a batch environment where you want to start a server for client/server use by other users, follow the LIBNAME statement with the following VASMP procedure statements:

***Example Code 2.2*** *SERVERWAIT Statement for the VASMP Procedure*

```
proc vasmp;
    serverwait port=10010; 1
quit;
```

1. The SERVERWAIT statement causes the server to continue running and wait for a termination request.

When a non-distributed SAS LASR Analytic Server is used in a metadata environment like SAS Visual Analytics, the SIGNER= option enables the server to enforce the permissions that are set in metadata. The values for the HOST= and PORT= options must match the host name and network port number that are specified for the server in metadata.

```
libname server1 sasiola startserver=(path="/tmp")
    host="server.example.com" port=10010 tag='hps'
    signer="http://server.example.com/SASLASRAuthorization";
```

For information about using SAS LASR Analytic Server in a metadata environment, see *SAS Visual Analytics: Administration Guide*.

If you want to use a script for starting a server, then include the STARTSERVER= LIBNAME option and the SERVERWAIT statement for the VASMP procedure in the program. Start one server only and do not include additional SAS statements after the QUIT statement for the VASMP procedure. If additional statements are included, they can prevent the SAS session from terminating (after receiving a SERVERTERM request). This can prevent the SAS session from freeing memory resources that were used by the server. It is best to restrict the program to starting the server only.

## Stopping Servers

Stopping a server is performed by clearing the libref that was used to start the server (if you start the server from a SAS session and keep the session running) or with the SERVERTERM statement.

To stop a server from the same SAS session that started it:

***Example Code 2.3*** *Stopping a Non-Distributed Server with the LIBNAME CLEAR Option*

```
libname server1 clear;
```

To stop a server from a different SAS session, use the SERVERTERM statement:

***Example Code 2.4*** *SERVERTERM Statement for the VASMP Procedure*

```
proc vasmp;
    serverterm host="server.example.com" port=10010;
quit;
```

*Note:* Exiting the SAS session that started the server also terminates the server because all librefs are automatically cleared at the end of a SAS session.

## Loading and Unloading Tables for Non-Distributed Servers

Tables are loaded into memory in a non-distributed server with the SAS LASR Analytic Server engine. A DATA step can be used. The following example demonstrates loading the Prdsale table into memory after starting a server on port 10010.

To load a table to memory:

***Example Code 2.5*** *Loading a Table to Memory for Non-Distributed Servers*

```
libname server1 sasiola start port=10010 tag='hps';

data server1.prdsale;
    set sashelp.prdsale;
run;
```

*Note:* The SASIOLA engine can always be used to connect to a server on the local host, as shown in this example. However, for connections to remote hosts, the engine supports connections to Linux hosts only. For more information, see "Understanding How the SAS LASR Analytic Server Engine Works" on page 415.

You can unload a table from memory with the DATASETS procedure:

***Example Code 2.6*** *Unloading a Table with the DATASETS Procedure*

```
proc datasets lib=server1;
    delete prdsale;
quit;
```

*Chapter 3*

# LASR Procedure



## Overview: LASR Procedure

### *What Does the LASR Procedure Do?*

The LASR procedure is used to start, stop, and load and unload tables from a distributed SAS LASR Analytic Server. The LASR procedure can also be used to save in-memory tables to HDFS.

### *Data Sources*

The LASR procedure can transfer data from any data source that SAS can read and load it into memory on the SAS LASR Analytic Server. However, the LASR procedure can also be used to make the server read data from a co-located data provider. The HDFS that is part of Hadoop provides a co-located data provider. When the data is co-located, each machine that is used by the server instance reads the portion of the data that is

local. Because the read is local and because the machines read in parallel, very large tables are read quickly.

In order to read tables from a third-party vendor database, the client machine must be configured with the native database client software and the SAS/ACCESS Interface software for the database.

# Syntax: LASR Procedure

**PROC LASR** *server-options*;
  **PERFORMANCE** *performance-options*;
  **REMOVE** *table-specification*;
  **SAVE** *table-specification* / *save-options*;

| Statement | Task | Example |
|---|---|---|
| PROC LASR | Start a server. | Ex. 1 |
| PROC LASR | Start a server with logging. | Ex. 2 |
| PROC LASR | Using the SASHDAT engine. | Ex. 3 |
| PROC LASR | Load a table from Teradata to memory | Ex. 4 |
| PROC LASR | Load a table from Greenplum to memory | Ex. 5 |
| REMOVE | Unload a table from memory. | Ex. 6 |
| PROC LASR | Stop a server. | Ex. 7 |
| PROC LASR | Working with user-defined formats. | Ex. 8 |
| PROC LASR | Working with user-defined formats and the FMTLIBXML= option. | Ex. 9 |
| SAVE | Save a table to HDFS. | Ex. 10 |

## PROC LASR Statement

Controls the SAS LASR Analytic Server.

### Syntax

**PROC LASR** *server-options*;

#### *Server Options*

These options control how the server starts, stops, and operates with data.

**ADD**

specifies to load a table to the SAS LASR Analytic Server. The data to load is identified by the DATA= option or the HDFS= option.

You can also add tables to memory with the SAS LASR Analytic Server engine. An important difference between using the LASR procedure and the engine is that the procedure has the ability to load data in parallel.

**CONCURRENT=*maximum-requests***

specifies the number of concurrent requests that can execute in the server. This option does not reject connections or requests that exceed *maximum-requests*. When *maximum-requests* is reached, the additional requests are queued and then processed in first-in-first-out order.

After the server is running, you can adjust this value in a SERVERPARM statement with the VASMP procedure.

| | |
|---|---|
| **Alias** | NACTIONS= |
| **Default** | 20 |

**CREATE <="*server-description-file*">**

specifies to start a server. The optional *server-description-file* argument specifies the fully qualified path to a file. The file is created when the server starts. Enclose the value in quotation marks. The fully qualified path is limited to 200 characters. The server description file is assigned to the LASRLAST macro variable.

If you do not specify a server description file, then you can use the PORT= option to specify the network port number. In either case, the LASRPORT macro variable is updated with the network port number that the server uses for communication.

**DATA=*libref.member-name***

specifies the data to load into the SAS LASR Analytic Server.

**DETAILS= TABLES | ALL**

specifies the information to return. Use TABLES to retrieve the table names, NLS encoding, row count, owner, and the table load time. The ALL value provides the previous information and adds the MPI rank and host name for each machine in the server.

The information always includes the performance information. This information includes the host name for the grid host, the grid installation location, and the number of machines in the server.

**EXTERNALMEM=*pct***

specifies the percentage of memory that can be allocated before the server stops transferring data to external processes such as external actions and the SAS High-Performance Analytics procedures. If the percentage is exceeded, the server stops transferring data.

| | |
|---|---|
| **Default** | 75 |
| **Tip** | If the external processes are running on a separate cluster, you can increase this value to 100%. |

**FMTLIBXML**

specifies the file reference for a format stream. For more information, see "Example 8: Working with User-Defined Formats" on page 60.

**FORCE**
specifies that a server should be started even if the server description file specified in the CREATE= option already exists. The procedure attempts to stop the server process that is described in the existing server description file and then the file is overwritten with the details for the new server.

**Restriction** Use this option with the CREATE= option only.

**FORCESIGNER**
specifies whether to rely on the SAS LASR Authorization service or to create a signature file for data that is loaded into the server.

If you specify this option, then the server does not create a signature file for a table when the data is loaded into the server. This forces the server to rely on the SAS LASR Authorization service for data access decisions. This option enables sites that run the server as a service account and manage access controls in SAS metadata. The service account is identified as the table owner, although access control decisions are managed with SAS metadata.

**FULLCOPYTO=*n***
specifies the number of full copies of a table to load into child non-distributed servers. For more information, see "High Volume Access to Smaller Tables" on page 27.

**HDFS(*HDFS-options*)**
specifies the parameters for the SASHDAT file to load from HDFS.

**TIP** Instead of specifying the HDFS option and parameters, you can use the ADD and DATA= options with a SASHDAT engine library.

**FILE=**
specifies the fully qualified path to the SASHDAT file. Enclose the value in quotation marks. The filename is converted to lowercase and the SASHDAT file in HDFS must be named in lowercase.

**Alias** PATH=

**LABEL=**
specifies the description to assign to the table. This value is used to override the label that was associated with the data set before it was stored in HDFS. If this option is not specified, then the label that was associated with the data set is used. Enclose the value in quotation marks.

**DIRECT**
specifies that the data is loaded directly from HDFS into memory. This option provides a significant performance improvement. With this option, the user account ID that is used to start the server process is used to create the table signature file.

**Alias** HADOOP=

**Interaction** To load encrypted SASHDAT files, you must specify the DIRECT suboption when you use the HDFS option.

**LIFETIME=*maximum-runtime* <(*active-time*)>**
specifies the duration of the server process, in seconds. If you do not specify this option, the server runs indefinitely.

***maximum-runtime***
When the *maximum-runtime* is specified without an *active-time* value, the server exits after *maximum-runtime* seconds.

***active-time***
When the *maximum-runtime* and *active-time* values are specified, the server runs for *maximum-runtime* seconds and then starts a run timer with an inactivity time-out of *active-time* seconds. When the server is contacted with a request, the run timer is reset to zero. Each second that the server is unused, the run timer increments to count the number of inactive seconds. If the run timer reaches the *active-time*, the server exits.

**LOGGING <(*log-options*)>**
The log file is named lasr.log.

**CLF**
specifies to use the common log format for log files. This format is a standardized text file format that is frequently analyzed by web analysis software. Specifying this option implies the LOGGING option.

**KEEPLOG**
specifies to keep the log files when the server exits instead of deleting them. By default, the log files are removed when the server exits.If you did not specify this option when the server was started, you can specify it as an option to the STOP option.

**MAXFILESIZE=**
specifies the maximum log file size, in megabytes, for a log file. When the log file reaches the specified size, a new log file is created and named with a sequentially assigned index number (for example, `.log.1`). The default value is 100 megabytes.

**TIP** Do not include an MB or M suffix when you specify the size.

**MAXROLLNUM=**
specifies the maximum number of log files to create. When the maximum has been reached, the server begins to overwrite existing log files. The oldest log file is overwritten first. The default value is 10.

**OSENCODING**
specifies that the log file is produced with the operating system encoding of the SAS LASR Analytic Server root node. This option is useful when the server is run in a different encoding than the operating system, but you want a log file that is readable in the server operating system.

**PATH='*log-file-directory*'**
specifies the fully qualified path to the directory to use for server log files. The default value is `/tmp`.

**MERGELIMIT=*n***
specifies that when the number of unique values in a numeric GROUPBY variable exceeds *n*, the variable is automatically binned and the GROUPBY structure is determined based on the binned values of the variable, rather than the unique formatted values.

For example, if you specify MERGELIMIT=500, any numeric GROUPBY variable with more than 500 unique formatted values is binned. Instead of returning results for more than 500 groups, the results are returned for the bins. You can specify the number of bins with the MERGEBINS= option.

**NOCLASS**
specifies that all character variables are not to be treated implicitly as classification variables. Without this option, all character variables are implicitly treated as classification variables. The performance for loading tables is improved when this option is used.

**PATH="*signature-file-path*"**
specifies the directory to use for storing the server and table signature files. The specified directory must exist on the machine that is specified in the GRIDHOST= environment variable.

**PORT=*integer***
specifies the network port number to use for communicating with the server. You can specify a port number with the CREATE option to start a server on the specified port.

**Interaction** Do not specify the PORT= option in the LASR procedure statement with a LASRSERVER= option in the PERFORMANCE statement.

**READAHEAD**
specifies for the server to be more aggressive in reading memory pages during the mapping phase when tables are loaded from HDFS. Loading the table takes more time with this option, but the first access of the table is faster.

**Engine** SASHDAT engine

**SERVERPERMISSIONS=*mode***
specifies the permission setting for accessing the server instance. The mode value is specified as an integer value such as 755. The mode corresponds to the mode values that are used for UNIX file access permissions.

**Alias** SERVERPERM=

**Range** 600 to 777

**Interaction** You can use this option with the CREATE option when you start a server.

**SIGNER="*authorization-web-service-uri*"**
specifies the URI for the SAS LASR Authorization web service. The web service is provided by the SAS Visual Analytics software. For more information, see *SAS Visual Analytics: Administration Guide*.

**Example** SIGNER="https://server.example.com/SASLASRAuthorization"

**SIGNERFILEPOLICY**
specifies to apply the encryption policies stored in metadata to the file.

**See** For more information, see "SIGNERFILEPOLICY Data Set Option" on page 454.

**SQUEEZE**
specifies to compress the table as it is added to the server. You can specify this option when you use the ADD and DATA= options.

Be aware that specifying the option when you start a server does not result in compressing all tables that are added to the server. The option applies only to the table that is loaded when CREATE is also specified, or to the table that is added when the ADD option is specified. In other words, the SQUEEZE option is not a persistent option for the lifetime of the server.

**Alias** COMPRESS

**Interaction** This option is ignored when reading SASHDAT tables from HDFS.

**See** "Compressed Tables and the LASR Procedure" on page 31

**STOP <(*stop-options*)>**

terminates a SAS LASR Analytic Server. The server instance is specified in the LASRSERVER= option that identifies a server description file, or it is determined from the LASRLAST macro variable. Once the server instance receives a request to stop, the server does not accept new connections.

**IMMEDIATE**

specifies to stop the server without waiting for current requests to complete. Without this option, termination requests are queued and can be queued behind a long-running request.

**Alias** NOW

**KEEPLOG**

specifies to keep log files that are created with the LOGGING option.

**Alias** TERM

**TABLEMEM=*pct***

specifies the percentage of memory that can be allocated before the server rejects requests to add tables or append data. If the percentage is exceeded, adding a table or appending rows to tables fails. These operations continue to fail until the percentage is reset or the memory usage on the server drops below the threshold.

This option has no effect for non-distributed servers. For non-distributed servers, the memory limits can be controlled with the MEMSIZE system option.

*Note:* The specified *pct* value does not specify the percentage of memory allocated to in-memory tables. It is the percentage of all memory used by the entire machine that—if exceeded—prevents further addition of data to the server. The memory used is not measured at the process or user level, it is computed for the entire machine. In other words, if operating system processes allocate a lot of memory, then loading tables into the server might fail. The threshold is not affected by memory that is associated with SASHDAT tables that are loaded from HDFS.

**Alias** MEMLOAD=

**Default** 75

**TABLEPERMISSIONS=*mode***

specifies the permission setting for accessing a table. The mode value is specified as an integer value such as 755. The mode corresponds to the mode values that are used for UNIX file access permissions.

**Alias** TABLEPERM=

**Range** 600 to 777

**Interaction** You can use this option with the ADD option when you load a table to memory.

**VERBOSE**
specifies to request additional information about starting a server or connecting to a server in the SAS log. This information can be helpful to diagnose environment configuration issues.

**Alias** GRIDMSG

## PERFORMANCE Statement

The PERFORMANCE statement defines performance parameters for multithreaded and distributed computing.

**Examples:** "Example 4: Load a Table from Teradata to Memory" on page 57
"Example 6: Unload a Table from Memory" on page 59

### Syntax

**PERFORMANCE** *performance-options*;

#### *Performance Statement Options*

**COMMIT=**
specifies that periodic updates are written to the SAS log when observations are sent from the client to the server instance. Whenever the number of observations sent exceeds an integer multiple of the COMMIT= size, a message is written to the SAS log. The message indicates the actual number of observations distributed and not an integer multiple of the COMMIT= size.

**HOST=**
specifies the grid host to use for the server instance. Enclose the host name in quotation marks. If you do not specify the HOST= option, it is determined from the GRIDHOST= environment variable.

**Alias** GRIDHOST=

**INSTALL=**
specifies the path to the TKGrid software on the grid host. If you do not specify this option, it is determined from the GRIDINSTALLLOC= environment variable.

**Alias** INSTALLOC=

**LASRSERVER=**
specifies the server to use. Provide the fully qualified path to the server description file.

**Alias** LASR=

**MODE= SYM | ASYM**
specifies whether the server runs in symmetric (SYM) mode or asymmetric (ASYM) mode. Asymmetric mode is useful when you want to read data in parallel from a massively parallel processing (MPP) data appliance that is not co-located with the server.

**Alias** GRIDMODE=

**Default** SYM

**Note** The only time you need to specify this option is when you want to run in symmetric mode alongside a Teradata database. In all other cases, this option is not needed, and the value that you specify is ignored.

**NODES=**
specifies the number of machines in the cluster to use for the server instance. Specify ALL to calculate the number automatically.

**Alias** NNODES=

**Restriction** This option has no effect when you use a third-party vendor database as a co-located data provider and you specify the CREATE= and DATA= options in the PROC LASR statement. When you use a third-party vendor database as a co-located data provider, you must use all of the machines to read data from the database.

**NTHREADS=**
specifies the number of threads for analytic computations and overrides the SAS system option THREADS | NOTHREADS. By default, the server uses one thread for each CPU core that is available on each machine in the cluster. Use this option to throttle the number of CPU cores that are used on each machine.

The maximum number of concurrent threads is controlled by the SAS software license.

*Note:* The SAS system options THREADS | NOTHREADS apply to the client machine that issues the PROC LASR statement. They do not apply to the machines in the cluster.

**TIMEOUT=**
specifies the time in seconds for the LASR procedure to wait for a connection to the grid host and establish a connection back to the client. The default value is 120 seconds. If jobs are submitted through workload management tools that might suspend access to the grid host for a longer period, you might want to increase the value.

# REMOVE Statement

The REMOVE statement is used to unload a table from memory.

## Syntax

**REMOVE** *table-specification*;

### Required Argument

***table-specification***
specifies the table to unload from memory. For a table that was loaded from a SAS library, the table specification is the same libref.*member-name* that was used to load the table. For a table that was loaded from HDFS, the table specification is the same as the HDFS path to the table, but is delimited with periods (.) instead of slashes (/). For a table that was loaded from the / directory in HDFS, the table specification is HADOOP.*TABLENAME*.

## SAVE Statement

The SAVE statement is used to save an in-memory table to HDFS.

### Syntax

**SAVE** *table-specification* / *save-options*;

#### Required Arguments

***table-specification***

specifies the table that is in memory. For a table that was loaded from a SAS library with the procedure, the table specification is the same libref.*member-name* that was used to load the table. For a table that was loaded from HDFS, the table specification is the same as the HDFS path to the table, but is delimited with periods (.) instead of slashes (/). For a table that was loaded from the / directory in HDFS, the table specification is HADOOP.*TABLENAME*.

***save-options***

specifies the options for saving the file in HDFS.

**BLOCKSIZE=**

specifies the block size to use for distributing the data set. Suffix values are B (bytes), K (kilobytes), M (megabytes), and G (gigabytes). The default block size is 32M.

**Alias** BLOCK=

**COPIES=*n***

specifies the number of replications to make for the data set (beyond the original blocks). The default value is 1.

**FULLPATH**

specifies that the value for the PATH= option specifies the full path for the file, including the filename.

**PATH='*HDFS-path*'**

specifies the directory in HDFS in which to store the SASHDAT file. The value is case sensitive. The filename for the SASHDAT file that is stored in the path is always lowercase.

*Note:* If the PATH= option is not specified, the server attempts to save the table in the **`/user/userid`** directory. The *userid* is the user ID that started the server instance.

**REPLACE**

specifies that the SASHDAT file should be overwritten if it already exists.

# Examples: LASR Procedure

## Example 1: Start a Server

### Details

This PROC LASR example demonstrates starting a server instance on network port number 10010. Once the server instance is started, the LASRPORT macro variable in the SAS session is set.

### Program

```
option set=GRIDHOST="grid001.example.com"; 1
option set=GRIDINSTALLLOC="/opt/TKGrid";

proc lasr create port=10010 2
    path="/tmp" noclass;

    performance nodes=all;
run;
```

### Program Description

1. The GRIDHOST= and GRIDINSTALLLOC= environment variables are used to identify the machine to connect to and the location of the SAS High-Performance Analytics environment. Do not include any trailing spaces (such as "/opt/TKGrid ") when you specify the path in GRIDINSTALLLOC=. The server will not start if the path includes trailing spaces.
2. The CREATE option is required and the PORT= option specifies the network port number to use.

## Example 2: Starting a Server with Logging Options

### Details

This PROC LASR example demonstrates how to start a server instance and specify logging options.

### Program

```
option set=GRIDHOST="grid001.example.com";
option set=GRIDINSTALLLOC="/opt/TKGrid";

proc lasr
    create port=10010
    path="/tmp"
```

```
    noclass
    logging(path="/opt/logs" maxfilesize=5 keeplog clf);

    performance nodes=all;
run;
```

### Program Description

The logging statement modifies the default logging behavior. Log files are written to **/opt/logs** instead of the default directory, **/tmp**. The log files are rolled over when they reach five megabytes. The KEEPLOG option is used to keep the log files when the server exits rather than delete them.

## Example 3: Using the SASHDAT Engine

### Details

The LASR procedure can load tables to memory from HDFS with the SASHDAT engine. This use is similar to using the HDFS option with the procedure, but has the advantage that you can use FORMAT statements and data set options.

### Program

```
option set=GRIDHOST="grid001.example.com"; 1
option set=GRIDINSTALLLOC="/opt/TKGrid";

libname grp1 sashdat path="/dept/grp1"; 2

proc lasr create port=10010 noclass;
    performance nodes=all;
run;

proc lasr add data=grp1.sales2012 port=10010;
    format predict $dollar20. 3
           actual $dollar20.;

run;

proc lasr add data=grp1.sales2013(where=(region="West")) port=10010; 4
run;
```

### Program Description

1. The GRIDHOST= and GRIDINSTALLLOC= environment variables are used by the LASR procedure and the GRIDHOST= option is also used by the LIBNAME statement.
2. The SASHDAT engine uses the GRIDHOST= environment variable to determine the host name for the NameNode. The PATH= option is used to specify the directory in HDFS.
3. The FORMAT statement is used to override the format name in HDFS for the variable.

4. The WHERE clause subsets the Sales2013 table. Only the rows with Region equal to "West" are read into memory. The WHERE clause is useful for subsetting data, but it does not take advantage of the memory efficiencies that are normally used with SASHDAT tables.

If the table in HDFS has variables that are associated with user-defined formats, then you must have the user-defined formats available in the format catalog search order.

## Example 4: Load a Table from Teradata to Memory

### Details

This PROC LASR example demonstrates how to load a table to memory from Teradata. The native database client for Teradata and SAS/ACCESS Interface to Teradata must be installed and configured on the client machine.

```
option set=GRIDHOST="grid001.example.com";
option set=GRIDINSTALLLOC="/opt/TKGrid_REP";

libname tdlib teradata server="dbccop1.example.com" 1
    database=hps user=dbc password=dbcpass;

proc lasr create port=10010 data=tdlib.tableone noclass;  2
    performance nodes=all;
run;

proc lasr add data=tdlib.tabletwo (label = "Table description") 3
    port=10010;
    format revenue dollar20.2
           units comma9; 4
run;

libname example sasiola port=10010 tag="tdlib"; 5

proc imstat;
  tableinfo / host="grid001.example.com" port=10010;
run;
  table example.tableone;  6
  columninfo;
quit;
```

### Program Description

1. The SERVER= option in the LIBNAME statement specifies the host name for the Teradata database.
2. The first PROC LASR statement uses the CREATE option to start a server on network port 10010. The DATA= option is used to load table tableOne from Teradata into the server.
3. The input data set option, LABEL=, associates the description with the data in the server instance. This option causes a warning in the SAS log because the SAS/ACCESS Interface to Teradata does not support data set labels.
4. SAS formats are applied with the FORMAT statement. Specifying the variable formats is useful for DBMS tables because database systems do not store formats.

5. The SASIOLA LIBNAME statement is used to access the tables in the server. The TAG= option is set to Tdlib so that it matches the libref that was used in the PROC LASR statements when tables were loaded into the server.
6. The Example libref is used with the IMSTAT procedure to access tables in the server.

## Example 5: Load a Table from Greenplum to Memory

### Details

This PROC LASR example demonstrates how to load a table to memory from Greenplum. The ODBC drivers and SAS/ACCESS Interface to Greenplum must be installed and configured on the client machine.

```
libname gplib greenplm server="mdw.example.com" 1
    database=hps user=dbuser password=dbpass;

proc lasr create port=10010
    data=gplib.sometable
    path="/tmp";

    performance host="mdw.example.com" 2
        install = "/opt/TKGrid";

run;

proc lasr add data=gplib.tabletwo (label = "Table description") 3
    port=10010;
    format y x1-x15 5.4
           dt date9.; 4
run;
```

### Program Description

1. The SERVER= option in the LIBNAME statement specifies the host name for the Greenplum database.
2. The HOST= option in the PERFORMANCE statement specifies the host name of the Greenplum master host.
3. The input data set option, LABEL=, associates the description with the data in the server instance. This option causes a warning in the SAS log because the SAS/ACCESS Interface to Greenplum does not support data set labels.
4. SAS formats are applied with the FORMAT statement. Specifying the variable formats is useful for DBMS tables because database systems do not store formats.

## Example 6: Unload a Table from Memory

### Details

This PROC LASR example demonstrates how to unload tables from memory. The first REMOVE statement applies to tables that were loaded from HDFS. The second REMOVE statement is typical for tables that are loaded from SAS libraries.

### Program

```
libname finance "/data/finance/2011/";

proc lasr port=10010;
    remove user.sales.2011.q4; 1
    remove finance.trans; 2
    performance host="grid001.example.com"
        install="/opt/TKGrid";
run;
```

### Program Description

1. This REMOVE statement specifies a table that was loaded from HDFS.
2. The libref and member name for a SAS data set are specified in this REMOVE statement example.

## Example 7: Stopping a Server

### Details

This PROC LASR example demonstrates stopping a server instance.

### Program

```
option set=GRIDHOST="grid001.example.com";
option set=GRIDINSTALLLOC="/opt/TKGrid";

proc lasr term port=10010;
run;
```

### Program Description

The server instance listening on port 10010 is stopped.

## Example 8: Working with User-Defined Formats

### Details

By default, when user-defined formats are used with the server, the LASR procedure automatically uses these formats. The formats must be available in the format catalog search order. You can use the FMTSEARCH= system option to specify the format catalog search order. The LASR procedure converts the formats to an XML representation and transfers them to the server with the data.

### Program

```
proc format library=myfmts;
     value YesNo        1='Yes'        0='No';
     value checkThis    1='ThisisOne'  2='ThisisTwo';
     value $cityChar    1='Portage'    2='Kinston';
run;

options fmtsearch=(myfmts); 1

proc lasr add data=orsdm.profit_company_product_year port=10010;
    format city $cityChar.; 2

    performance host="grid001.example.com"
        install="/opt/TKGrid"
        nodes=ALL;
run;
```

### Program Description

1. The user-defined formats are available to the LASR procedure because they are added to the format catalog search order.
2. When the $cityChar. format is applied to the city variable, the LASR procedure converts the formats to XML, and transfers the format information and the data to the server.

Be aware that user-defined formats that specify an existing format in the format-or-range specification are not supported. This is referred to as using a format as a label.

For example, when the FETCH statement is used with the FORMAT option in the following code, the server does not support using another format (MMDDYY10.) as a label. Other clients, such as a SAS Visual Analytics cannot use the in-memory table either if the client requests formatted values.

```
proc format library=myfmts;
     /* This pattern is unsupported and results in an error. */
     value mdy other = [mmddyy10.];
run;

options fmtsearch=(myfmts);

libname example sasiola host="grid001.example.com" port=10010 tag='hps';
```

```
data example.samp;
  format y mdy.;
  input x y;
  datalines;
1 19724
2 19725
3 19726
;
run;

proc imstat data=example.samp;
  fetch / format;
run;
```

```
ERROR: The format MDY could not be loaded in the LASR Analytic Server.
WARNING: The LASR Analytic Server action request for the FETCH statement did not
         complete successfully.
```

## Example 9: Working with User-Defined Formats and the FMTLIBXML= Option

### Details

As explained in the previous example, the LASR procedure can use any format so long as the format is in the format catalog search order. The procedure automatically converts the format information to XML and transfers it to the server with the data. However, if the same formats are used many times, it is more efficient to convert the formats to XML manually and use the FMTLIBXML= option.

You can use the FORMAT procedure to write formats to an XML fileref. Then, you can reference the fileref in the FMTLIBXML= option each time you use the LASR procedure to load tables. This improves performance because the conversion to XML occurs once rather than each time LASR procedure transfers the data.

Formats are created with the FORMAT procedure. The following SAS statements show a simple example of creating a format and using the XML fileref in the LASR procedure.

### Program

```
proc format library=gendrfmt;
     value $gender   'M'='Male'        'F'='Female';
run;

options fmtsearch=(gendrfmt); 1

filename fmtxml 'genderfmt.xml';
libname  fmtxml XML92 xmltype=sasfmt tagset=tagsets.XMLsuv;

proc format library=gendrfmt cntlout=fmtxml.allfmts; 2
run;

proc lasr add data=sashelp.class fmtlibxml=fmtxml; 3
```

```
    format sex $gender.; 4

    performance host="grid001.example.com"
        install="/opt/TKGrid"
        nodes=ALL;
run;
```

### Program Description

1. The user-defined formats are available to the LASR procedure because they are added to the format catalog search order.
2. An XML stream for the formats in the file GenderFmt.xml is associated with the file reference FmtXml. The formats are converted to XML and stored in the file.
3. The file reference FmtXml is used with the FMTLIBXML= option in the PROC LASR statement. For subsequent uses of the LASR procedure, using the FMTLIBXML= option to reference the fileref is efficient because the formats are already converted to XML.
4. The $gender. format information is transferred to the server in an XML stream and associated with the variable that is named sex. However, the format must be available to the SAS session that runs the LASR procedure.

## Example 10: Saving a Table to HDFS

### Details

The server can save in-memory tables to HDFS. Use the SAVE statement to provide a table specification and the save options.

```
option set=GRIDHOST="grid001.example.com";
option set=GRIDINSTALLLOC="/opt/TKGrid";

proc lasr port=10010;
    save sales.sales2012 / path="/dept/grp1/" copies=1 blocksize=32m; 1
    save sales.avg2012 / fullpath path="/dept/grp1/avg/y2012" copies=1; 2
run;
```

### Program Description

1. The table that is named sales2012 is saved to HDFS as /dept/grp1/sales2012.sashdat.
2. The table that is named avg2012 is saved to HDFS as /dept/grp1/avg/y2012.sashdat. The FULLPATH option is used to rename the file.

*Chapter 4*

# IMSTAT Procedure (Analytics)

# Overview: IMSTAT Procedure (Analytics)

### *What Do the Analytic Statements for the IMSTAT Procedure Do?*

The analytic statements of the IMSTAT procedure are used to perform in-memory analytics with a SAS LASR Analytic Server. All analyses are performed on in-memory tables.

# Syntax: IMSTAT Procedure (Analytics)

**Tip:** For information about the data and server management statements, see Chapter 5, “IMSTAT Procedure (Data and Server Management),” on page 297.

**PROC IMSTAT** <*options*>;
  **AGGREGATE** *variable-name* (*aggregate-variable-options*) / ID=*variable-name*< *options*>;
  **ARM** ITEM=*item-variable* TRAN=*transaction-variable* </ *options*>;
  **ASSESS** <*variable-list*> / Y=*response-variable* <*options*>;
  **BOXPLOT** <*variable-list*> </ *options*>;
  **CLUSTER** <*variable-list*> </ *options*>;
  **CORR** <*variable-list*> </ *options*>;
  **CROSSTAB** *row*column* </ *options*>;
  **DECISIONTREE** *target-variable* </ *options*>;
  **DISTINCT** <*variable-list*> </ *options*>;
  **FORECAST** *timestamp-variable* </ *options*>;
  **FREQUENCY** *variable-list* </ *options*>;
  **GENMODEL** *dependent-variable* <(*class-variables*)> = *model-effects* </ *options*>;
  **GLM** *dependent-variable* <(*class-variables*)> = *model-effects* </ *options*>;
  **GROUPBY** <*variable-list*> </ *options*>;
  **HISTOGRAM** <*variable-list*> </ *options*>;

**HYPERGROUP** <*variable-list*> <*/ options*>;
**KDE** *variable-list* <*/ options*>;
**LOGISTIC** *dependent-variable* <(*class-variables*)> = *model-effects* <*/ options*>
**MDSUMMARY** *variable-list* <*/ <set-specification,…> options*>;
**NEURAL** <*target-variable*> <*/ options*>;
**OPTIMIZE** <*options*>;
**PERCENTILE** <*variable-list*> <*/ options*>;
**RANDOMWOODS** *target-variable* <*/ options*>;
**REGCORR** <*variable-list*> <*/ options*>;
**SUMMARY** <*variable-list*> <*/ options*>;
**TEXTPARSE** TXT=*text-variable* ID=*document-ID* <*options*>;
**TOPK** <*variable-list*> <*/ options*>;
**TRANSFORM** (*request1*) <(*request2*) …> <*/ options*>;
**QUIT**;

| Statement | Task | Example |
|---|---|---|
| AGGREGATE | Aggregate values of one or more variables | Ex. 14 |
| BOXPLOT | Retrieve box plot values | Ex. 2, Ex. 3 |
| CLUSTER | Perform a cluster analysis | Ex. 5 |
| CORR | Perform a pairwise correlation | Ex. 6 |
| CROSSTAB | Crosstabulate with measures of association and Chi-square tests | Ex. 7 |
| DECISIONTREE | Train and validate a decision tree, Store and score a decision tree | Ex. 8, Ex. 9 |
| FORECAST | Compute predicted values for observed and future (forecast) values of time series | Ex. 12, Ex. 13 |
| MDSUMMARY | Perform a multi-dimensional summary | Ex. 10 |
| PERCENTILE | Calculat percentiles and quartiles | Ex. 1 |
| REGCORR | Fit a regression model | Ex. 11 |
| TRANSFORM | Transform variables with imputation and binning | Ex. 17 |

## PROC IMSTAT (Analytics) Statement

Performs in-memory analytics with a SAS LASR Analytic Server.

### Syntax

**PROC IMSTAT** <*options*>;

### Summary of Optional Arguments

BATCHMODE
DATA=*libref.member-name*
FMTLIBXML=*file-reference*
IMMEDIATE
NODATE
NOPREPARSE
NOPRINT
NOTIMINGMSG
PGMMSG
SIGNER="*authorization-web-service-uri*"
TEMPTABLEINFO
TEMPTABLESQUEEZE
UCA

### Optional Arguments

**BATCHMODE**

By default, the IMSTAT procedure operates in interactive mode. If your program contains errors that prevent SAS from parsing or executing statements, the errors are reported in the SAS log, but they do not stop the procedure. If the errors are fatal errors such as running out of memory on the SAS client, the procedure stops.

In contrast, when the BATCHMODE option is specified in the PROC IMSTAT statement, the procedure behaves with respect to error handling as if it were not an interactive procedure. Whenever an error occurs, the procedure terminates and sets the SYSERR macro variable.

**Alias** BATCH

**DATA=*libref.member-name***

specifies the table to access from memory. The libref must be assigned from a SAS LASR Analytic Server engine LIBNAME statement.

**FMTLIBXML=*file-reference***

specifies the file reference for a format stream. For more information, see "FMTLIBXML" in the LASR procedure.

**IMMEDIATE**

specifies that the procedure executes one statement at a time rather than accumulating statements in RUN blocks.

**Alias** SINGLESTEP

**NODATE**

specifies to suppress the display of time and date-dependent information in results from the TABLEINFO statement.

**NOPREPARSE**

prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the

messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NOPRINT**
This option suppresses the generation of ODS tables and other printed output in the IMSTAT procedure. You can use this option to suppress printed output during execution of the actions, and then use the REPLAY statement to print the tables at a later point in the procedure execution.

**NOTIMINGMSG**
When an action completes successfully, the IMSTAT procedure generates a SAS log message that contains the execution time of the request. Specify this option to suppress the message.

**Alias** NOTIME

**PGMMSG**
specifies to capture messages associated with user-provided SAS statements on the server in a temporary table. Messages are produced when parsing errors occur, when code generation fails, or by PUT statements in a SAS program.

You can use this option as a debugging feature for SAS code that you submit through temporary column expressions. The macro variable _PGMMSG_ is used in the IMSTAT procedure to capture the name of the table. The _TEMPLAST_ macro variable is also updated in case this temporary table is the most recently created temporary table.

**Alias** PROGMST

**SIGNER="*authorization-web-service-uri*"**
specifies the URI for the SAS LASR Authorization web service. For more information, see *SAS Visual Analytics: Administration Guide*.

**Example** SIGNER="https://server.example.com/SASLASRAuthorization"

**TEMPTABLEINFO**
specifies to add additional information for temporary tables to the ODS table that is created on the SAS client. The information includes the time at which the temporary table was created in the server, the number of rows, and the number of columns.

**Alias** TEMPINFO

**TEMPTABLESQUEEZE**
requests that the temporary tables generated in the PROC IMSTAT session are automatically squeezed (compressed). You can use the INFO option in the COMPRESS statement to determine the compression ratio that was applied to the table.

**Alias** TEMPSQUEEZE

**UCA**
specifies that you want to use Unicode Collation Algorithms (UCA) to determine the ordering of character variables in the GROUPBY= operations and other operations that depend on the order of formatted values.

**Alias** UCACOLLATION

## AGGREGATE Statement

The AGGREGATE statement aggregates the values of one or more variables according to variable-specific options. The statement supports both numeric and class variables.

**Example:** "Example 14: Aggregating Time Series Data" on page 278

### Syntax

**AGGREGATE** *variable-name* <(*variable-options*)> / < *options*>;

#### *Required Argument*

***variable-name***
specifies the variable to aggregate.

#### *Optional Argument*

***variable-options***
specifies the options to apply to the *variable-name* that precedes it. The following options are available.

**AGGREGATOR=(*aggregate-method*)**
specifies the aggregate method to apply to the specified variable.

| Aggregate Method | Description |
|---|---|
| N | Number of nonmissing observations. |
| NMISS | Number of missing values. |
| NDISTINCT | Number of distinct values. |
| SUM, TOTAL | Sum of nonmissing values. |
| MEAN, AVERAGE, AVG | Arithmetic mean. |
| STD, STDDEV | Standard deviation. |
| STDERR | Standard error. |
| VAR | Sample variance. |

| Aggregate Method | Description |
|---|---|
| USS | Uncorrected sum of squares. |
| CSS | Corrected sum of squares. |
| Q1 | 25th percentile. |
| Q2, MEDIAN, MED | 50th percentile. |
| Q3 | 75th percentile. |
| TSTAT | *t*-statistic for H:mean=0. |
| PROBT | *P*-value for *t*-statistic. |
| MIN, MINIMUM | Minimum value. |
| MAX, MAXIMUM | Maximum value. |
| CV | Coefficient of variation. |
| LAST | Value at the last time. |
| LASTNOTEMPTY, LNE | Non-empty (nonmissing) value at the last time. |
| FIRST | Value at the first time. |
| FIRSTNOTEMPTY, FNE | Non-empty (nonmissing) value at the first time. |
| PERCENTILE | Specified list of percentiles. |
| SKEWNESS, SKEW | Sample skewness. |
| KURTOSIS, KURT | Sample kurtosis. |
| MODE | Most frequent value. |
| PGT | Percentage of cases greater than the specified value. |
| PLT | Percentage of cases less than the specified value. |
| PIN | Percentage of cases in the specified range. |

**Aliases** AGG=

ACCUMULATE=

**Default** SUM for numeric variables and N for character variables.

**Interaction** If you specify PGT or PLT as an aggregate method, then you are required to specify the N= or CHARN= option.

**CHARN="*value*"**
specifies the character value for AGGREGATE= methods PGT or PLT. If the associated variable is numeric and the FORMAT= option is not specified, then the CHARN= option is ignored. The value is case-sensitive.

In the following code example, the AGGREGATE statement computes the percentage of cars with a value for the Make variable that sorts alphabetically before CHARN='Honda' and then groups the results by the variable Origin.

**Example**

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

proc imstat data=example.cars;
    aggregate make (agg=plt charn="Honda") / groupby=(origin);
run;
```

**CHARRANGE <=> ("*lower-value*", "*upper-value*")**
specifies the inclusive character range of values to be considered in the aggregation. Enclose the values in quotation marks. If the associated variable is numeric and the FORMAT= option is not specified, then the CHARRANGE= option is ignored. The values are case-sensitive.

**Example**

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

proc imstat data=example.cars;
    aggregate make (agg=n charrange=("B", "V") format="$upcase1.")
                    / groupby=(origin type);
```

**FORMAT="*format-specification*"**
specifies the format to apply to the aggregated variable. If a format is not specified, then the unformatted values of the variable are used. Enclose the format specification in quotation marks.

**Example**

```
data example.letters;
    do id = 1 to 26;
        letter = substr("abcdefghijklmnopqrstuvwxyz", id, 1);
        output;
    end;
run;

proc imstat data=example.letters;
    aggregate letter (agg=plt charn="E" format="$UPCASE1.");
quit;
```

**N=*numeric-value***
specifies the numeric value for AGGREGATE= methods PGT or PLT. The following example calculates the percentage of cars in the Sashelp.Cars data set that have less than six cylinders.

**Example**

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

proc imstat data=example.cars;
    aggregate cylinders (agg=plt n=6);
quit;
```

**PERCENTILE=(*percentiles*)**
specifies one or more percentiles. For example, the following statements aggregate the 25th, 15th, and 40th percentiles of the Msrp variable and groups the results by Make.

**Example**

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

proc imstat data=example.cars;
    aggregate msrp (agg=percentile percentile=(25 15 40))
                   / groupby=(make);
quit;
```

**RANGE <=> (*lower-value*, *upper-value*)**
specifies the inclusive numeric range of values to be considered in the aggregation. For example, the following statements calculate the percentage of cars with an Mpg_City value within the range of 12 and 22 and groups the results by the Make variable.

**Example**

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

proc imstat data=example.cars;
    aggregate mpg_city (agg=pin range=(12, 22)) / groupby=(make);
quit;
```

## *AGGREGATE Statement Options*

**ALIGN= <BEGINNING | MIDDLE | ENDING>**
specifies the alignment of the representative value with respect to an interval or bin. The following example specifies a bin of width 5 years and the representing value of a bin is the bin's beginning value, by default. Given the bins [1985, 1990], [1990, 1995], …, [2000, 2005], and ALIGN=ENDING, the representing values of these bins are 1990, 1995,..., 2005.

**Alias** The values for the ALIGN= option have aliases of B, BEG, M, MID, and E, END, respectively.

**Default** BEGINNING

**Example**

```
data example.stocks(partition=(stock));
   set sashelp.stocks;
run;

proc imstat data=example.stocks;
    aggregate close (agg=mean) close (agg=std)
```

```
                / id=date idfmt="year."
                bin=('01jan1985'd, '01jan1990'd)
                align=m
                partition="IBM";
run;
    table example.&_templast_;
    fetch / format orderby=(date) to=20;
quit;
```

**BIN=(*lower-number*, *upper-number*)**
specifies the minimum and maximum values of one bin. The boundaries for additional bins are extrapolated from the range in the bin that you specify. For example, if the values of the ID= variable range from 0 to 100, and you specify BIN=(5, 15), then the server constructs 11 bins. The first bin is [–5, 5], the second bin is (5, 15], and the last bin is (95, 105]. The values on the upper boundary of a bin belong to this bin. The values on the lower boundary of a bin belong to the adjacent lower bin.

**Interaction** The BIN= option has no effect unless the ID= option is specified.

**Tip** If you are working with time series data, the INTERVAL= option is an alternative and offers a convenient syntax for binning time series values.

**EDGEID=*variable-name***
specifies the variable to use for identifying the order of the analysis variable. It is required when you specify one of the following aggregation methods:

- FIRST
- FIRSTNOTEMPTY, FNE
- LAST
- LASTNOTEMPTY, LNE

**Example**

```
data example.stocks;
    set sashelp.stocks;
 run;

proc imstat data=example.stocks;
    where stock="IBM" and date >= '01jan2003'd;

    aggregate close (agg=fne) close (agg=lne) close (agg=mean)
                / id=date idfmt="yyq6."
                  edgeid=date interval="quarter";
quit;
```

**GROUPBY=(*variable-list*)**
specifies a list of variable names, or a single variable name, to use as GROUPBY variables in the order of the grouping hierarchy. If you do not specify any GROUPBY variable names, then the calculation is performed across the entire table —possibly subject to a WHERE clause.

**GROUPBYLIMIT=*n***
specifies the maximum number of levels in a GROUPBY set. When the software determines that there are at least *n* levels in the GROUPBY set, it abandons the action, returns a message, and does not produce a result set. You can specify the

GROUPBYLIMIT= option if you want to avoid creating excessively large result sets in GROUPBY operations.

**GROUPFILTER=(*filter-options*)**
specifies a section of the group-by hierarchy to be included in the computation. With this option, you can request that the server performs the analysis for only a subset of all possible groupings. The subset is determined by applying the group filter to a temporary table that you generate with the GROUPBY statement.

You can specify the following suboptions in the GROUPFILTER option:

**DESCENDING**
specifies the top section or the bottom section of the groupings to be collected. If the DESCENDING option is specified, the top LIMIT=*n* (where $n > 0$) groupings are collected. Otherwise, the bottom LIMIT=*n* groupings are collected.

**Alias** DESC

**LIMIT=*n***
specifies the maximum number of distinct groupings to be collected, where integer $n >= 0$. If *n* is zero, then all distinct groupings (up to $2^{31}-1$) that satisfy the boundary constraints, such as LOWERSCORE=*f*, are collected.

**CAUTION** **High Cardinality Data Sets** Setting *n* to zero with high-cardinality data sets can significantly delay the response of the server.

**SCOREGT=*f***
specifies the exclusive lower bound for the numeric scores of the distinct groupings to collect.

**Alias** SGT=

**SCORELT=*f***
specifies the exclusive upper bound for the numeric scores of the distinct groupings to collect.

**Alias** SLT=

**VALUEGT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive lower bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VGT=

**VALUELT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive upper bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VLT=

**TABLE=*table-with-groupby-results***
specifies the in-memory table from which to load the group-by hierarchy. If the TABLE= option is not specified, then all other GROUPFILTER= options are ignored.

The following program request all the groupings of State, City, and then Trade_In_Model in the Cars_Program_All table. The groupings are ordered by the maximum value of New_Vehicle_Msrp for each grouping:

```
proc imstat;
```

```
    table example.cars_program_all;
    groupby state city trade_in_model / temptable
                weight=new_vehicle_msrp
                agg   =(max)
                order =weight;
run;
```

The TEMPTABLE option in the GROUPBY statement directs the server to save all the groupings in a temporary in-memory table. The following DISTINCT statement requests the count of the distinct unformatted values of Sales_Type for each of the selected groupings of State, City, and Trade_In_Model.

```
    table example.cars_program_all;
    distinct sales_type / groupfilter=(
                table  =mylasr.&_TEMPLAST_
                scoregt=40000
                valuelt=("FL","Ft Myers","")
                limit  =20
                descending);
run;
```

This example considers only groupings that have maximum values of the New_Vehicle_Msrp above 40,000 and with formatted values that are less than State="FL" and City="Ft Myers." The empty quotation marks result in no restriction on Trade_In_Model values. These groupings are ordered according to the maximum values of New_Vehicle_Msrp. Because of the DESCENDING option, this example collects the 20 top groupings within the specified group-by range for the DISTINCT analysis.

**Interaction** If you specify the GROUPFILTER= option, then the GROUPBY= and FORMATS= options have no effect.

**ID=*variable-name***
specifies a numeric variable that identifies the time associated with each observation in the input table. The values of the ID= variable are typically SAS date, time, or datetime values, but that is not a requirement.

**IDFORMAT=("*format-specification*")**
specifies the format for the ID= variable. If you do not specify this option, the default format for the variable is applied. If the ID= option is not specified, this option is ignored.

**Alias** IDFMT=

**IDEND=*numeric-value***
specifies the end value of the ID= variable to be included in the analysis. If the last ID= variable value is less than the specified IDEND= value, then the series is extended with missing values. If the last ID= variable value is greater than the specified IDEND= value, then the series is truncated.

The IDEND= value can be a date ('13SEP1998'd), a time ('12:34:56't), a datetime ('01MAY88:12:34:56'dt) or a numeric value. If the ID= option is not specified, then this option is ignored.

**IDSTART=*numeric-value***
specifies the beginning value of the ID= variable to be included in the analysis. If the first ID= variable value is greater than the specified IDSTART= value, then the series is prefixed with missing values. If the first ID= variable value is less than the

specified IDSTART= value, then the series is truncated. If the ID= option is not specified, then this option is ignored.

**INTERVAL="*interval*"**
specifies the time period for the accumulation of observations. For example, if you specify INTERVAL='MONTH', then the AGGREGATE statement summarizes the observations in monthly intervals. If the ID= option is not specified, then this option is ignored.

For information about specifying interval values, see "About Date and Time Intervals" in *SAS Language Reference: Concepts*.

**JUMPINGWINDOW**
specifies that during aggregation, a window considers data within a specified multiple of intervals. A jumping window resets the aggregation process when the specified range of time expires. If you do not specify the JUMPINGWINDOW option, a window always retains the same multiple of intervals.

**KEEP=(*variable-name1 <variable-name2…>* )**
specifies one or more variables to transfer from the active table to the ODS table output or temporary table. When multiple input observations contribute to an output observation (this is the most common situation), the minimum value is used.

**Alias** TABLEVARS=

**Interaction** This option is ignored unless a PARTITION, GROUPBY=, or ID= option is also specified.

**KEEPRECORD**
specifies to output an aggregated value for each input observation by aggregating input observations whose ID= values are specified by INTERVAL= and WINDOWINT= options. Be aware that using this option can increase the volume of the result set.

The following code sums the deposit amount and counts the number of deposits, grouped by member ID and date. The aggregation at each output record considers all observations with a date value that is within one year from each record's own date value.

**Example**

```
libname example sasiola host="grid001.example.com" port=10010 tag=hps;

data example.retaindata;
  input transaction dt memberid deposit;
  informat dt date9.;
  format   dt date9.;
  datalines;
    1      15Apr2014      2    5000
    2      01May2014      2    5000
    3     01May2014     3 8000
    4     02May2014     2    5000
    5     03May2014      1    4000
    6     03May2014     3    3000
    7     04May2014     2    4000
    8     04May2014     3    2000
    9     05May2014     1    3000
    10     06May2014     1    3000
    11     07May2014     1    2500
    12     08May2014     2    1000
```

```
    13     09May2014      2    2000
    14     10May2014     2    2000
    15     10May2014     4    12000
run;

proc imstat data=example.retaindata;
  aggregate
     deposit(agg=sum)
     deposit(agg=n)
     /
     groupby=memberid id=dt idfmt='DATE9.'
     interval='day' windowInt='year'
     noemptyinterval keeprecord;
run;
```

**NOEMPTYINTERVAL**
specifies that intervals that no ID= variable value belongs to are omitted from the output. By default, the empty intervals contain missing values.

**NOMISSING**
specifies that you do not want to include missing values in the determination of group-by values. This option also applies to analysis variables when you specify AGGREGATOR=NDISTINCT.

**Alias** NOMISS

**OFFSET=**
specifies the time series shift in order to match up with an existing time series. It can be used to match up with existing time series from previous year, for example.

**Alias** DIF=

**PARTIALTOINTERVAL=*numeric-value***
**PARTIALTODATE=*numeric-value***
specifies the time value when the aggregation within an interval or a bin is terminated. For example, if you specify INTERVAL='MONTH' and PARTIALTOINTERVAL='10FEB98'd, then the action aggregates records from the first 10 days of each month only.

**Aliases** PTD=

PTI=

**Example**
```
aggregate var1 (agg=n) / id=date interval='month' windowint='year'
     partialtointerval='10feb1998'd jumpingwindow;
```

**PARTIALTOWINDOW=*numeric-value***
specifies the time value when the aggregation within a window interval or a window bin is terminated. For example, if you specify WINDOWINT='YEAR', PARTIALTOWINDOW='10FEB98'd, and JUMPINGWINDOW, then the action aggregates records from the first 41 days of each year for each interval, except the interval contains all 31 days from January of each year. The PARTIALTOWINDOW= and PARTIALTOINTERVAL= options can be used together.

In addition to the above specification, if you specify INTERVAL='MONTH', PARTIALTOINTERVAL='08FEB98'd, then the action counts only the first 8 days

from JANUARY and the first 8 days from FEBURARY when it aggregates on intervals FEBURARY, … , DECEMBER for each year.

**Interaction** This option is ignored unless you specify the JUMPINGWINDOW option, because the starting time of a sliding window (a non-jumping window) is varying.

**Example**

```
aggregate var1 (agg=n) / id=date interval='month'
    windowint='year' partialtowindow='08feb98'd jumpingwindow;
```

**PARTITION <=*partition-key*>**
specifies that when the active table is partitioned, then the aggregation is performed separately for each specified *partition-key*. If you do not specify a *partition-key*, the analysis is performed for all partitions. You can use the PARTITIONINFO statement to retrieve the valid partition key values for a table.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SUBBINOFFSET=*n***
specifies the offset from the start of a bin. *n* must be positive and less or equal than the bin width. If the specified *n* is out of range, then this option is ignored.

**Interaction** You must specify ID= and BIN= to use this option.

**SUBBINWIDTH=*n***
specifies the width of the sub-bin within a bin. For example, if the values of the ID= variable range from 0 to 100, and you specify BIN=(5, 15), SUBBINOFFSET=2, and SUBBINWIDTH=5, then this action summarizes the observations with ID= variable values that fall into the ranges [–3, 2], (7, 12], (17, 22], …, (97, 102]. *n* must be positive and the sum of SUBBINOFFSET= and SUBBINWIDTH= must be less or equal to the bin width. If the specified *n* is out of range, then this option is ignored.

**Interaction** You must specify ID= and BIN= to use this option.

**Example**

```
aggregate var1 (agg=n) / id=id bin=(5, 15) subbinoffset=2 subbinwidth=5;
```

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the &_TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**Interaction** The TEMPTABLE option requires a group-by analysis or a partitioned analysis with this statement.

**WINDOWBIN=(*lower-number*, *upper-number*)**
specifies the minimum and maximum values of a time window for the aggregation of observations with respect to each bin. The construction of this option is similar to the BIN= option. If the width of the window is not a multiple of the width of the bin, then the action fails. If the value for *lower-number* is not equal to that of BIN= option, the action fails.

**WINDOWINT="*interval*"**
specifies the time window for the aggregation of observations with respect to each time interval. For example, if you specify INTERVAL='MONTH' and WINDOWINT='YEAR', then the AGGREGATE statement summarizes a year's worth of observations before the end of each monthly interval.

**WINDOWOFFSET=*n***
specifies the time series shift in terms of an integer multiple of the WINDOWINT= or WINDOWBIN= value. For example, if you specify WINDOWINT='YEAR' and WINDOWOFFSET=–3, then at each time interval, the aggregated records are from three years earlier.

**Alias** WINDOWDIF=

## Details

### *ODS Table Names*

The AGGREGATE statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| AggStatistics | Aggregation statistics for one column | Default |
| TempTable | Information about a temporary table | TEMPTABLE |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# ARM Statement

The ARM statement is used to perform associative rule mining (ARM). You can use it to derive frequent itemsets, perform association rule mining, and sequence mining.

## Syntax

**ARM** ITEM=*item-variable* TRAN=*transaction-variable* <*/ options*>;

### Required Arguments

**item-variable**
specifies the name of the variable in the active table that identifies items.

**transaction-variable**
specifies the name of the variable in the active table that identifies transactions.

### Optional Argument

***variable-list***
specifies one or more numeric variables. If you do not specify this option, then all numeric variables in the table are used.

### ARM Statement Options

**AGGREGATE=*aggregation-method***
lists the aggregator for which the score of an itemset at each occurrence in a data set is aggregated into a final score of such itemset. If the WEIGHT= variable is not specified, then the aggregator specification is ignored.

The available aggregation methods are as follows:

| | |
|---|---|
| MAX | maximum value |
| MEAN | arithmetic mean |
| MIN | minimum value |
| SUM | sum of the nonmissing values |

| | |
|---|---|
| **Alias** | AGG= |
| **Default** | SUM |
| **Interaction** | You must specify the WEIGHT= option to use this option. |

**FREQ=*variable***
specifies the numeric frequency variable to use for computing the score of each frequent itemset along with WEIGHT= option. When the FREQ= variable is not specified, the score of a frequent itemset equates the value of the WEIGHT= variable scaled by 1. Negative values for the specified variable are considered missing.

**ITEMAGG=*aggregation-method***
lists the aggregator for which the values of the WEIGHT= variable, and optionally the FREQ= variable, are rolled up into the score of an itemset at each occurrence in the data set, provided that a WEIGHT= variable is specified. If the WEIGHT= variable is not specified, then the aggregator specification is ignored. The aggregation methods are identical to the list in the AGGREGATE= option.

The ITEMAGG= and AGGREGATE= options work together to derive the final score of an itemset. Given an itemset, the score $S$ is first aggregated over the frequency, $f$, and weight, $w$, variables associated with each item at each occurrence among a transaction by item aggregator $\Phi_{item}$ (.). Then, the intermediate scores of such itemset among all the occurrences are then aggregated again by set aggregator $\Phi_{set}$ (.) See the following equation.

$$s = \Phi_{set}(\Phi_{item}(fw))$$

**ITEMFMT=("*format-specification*")**
specifies the formats for the ITEM= variable. If you do not specify the ITEMFMT= option, then the unformatted values of the ITEM= variable are used.

Enclose the format specification in quotation marks.

**ITEMSTBL**
specifies to save the derived frequent itemsets in a temporary table. By default, the frequent itemsets are not saved.

**MAXITEMS=*n***
specifies the maximal number of items to allow in a frequent itemset. The value must be greater than or equal to 1. If an invalid value is specified, then it is replaced with 1, the default value.

**Default** 1

**MINITEMS=*n***
specifies the minimal number of items to allow in a frequent itemset. The value must be greater than or equal to zero. If an invalid value is specified, then it is replaced with 0, the default value.

If you specify MAXITEMS= < MINITEMS=, the server swaps the values. If you specify MAXITEMS = MINITEMS, the server assigns MAXITEMS = MINITEMS + 1.

**Default** 0

**NOMISSING**
specifies that missing values of the ITEM= and TRAN= variables are excluded from analysis. By default, missing values of the ITEM= variable are considered a separate item. Missing values of the TRAN= variable are considered a separate transaction.

**Alias** NOMISS

**PARTITION <=*partition-key*>**
specifies to use partitioning variables. When only PARTITION is specified and the table is partitioned first by the TRAN= variable, and the TRANFMT= option is specified, the associative rule mining is performed separately for each value of the partition key. If a value for *partition-key* is specified, then the associative rule mining is performed on that partition only.

**RELSUPPORT**
specifies that the values for LOWER= and UPPER= in the SUPPORT option are relative to the most frequent itemset. For example, if 500 is the support of the most frequent itemset, then specifying RELSUPPORT SUPPORT(LOWER=0.1 UPPER=0.5) means the minimum and the maximum supports for the analysis are 50 and 250, respectively. When using this option, the values for LOWER= and UPPER= must be between 0 and 1. Otherwise, they are set to the default values 0.05 and 1.0, respectively.

**RULES( <*suboptions*> )**
specifies the requirements for how association rules are generated from frequent itemsets. The following suboptions are available:

**AGGREGATE=*aggregation-method***
lists the aggregator for which the score of a rule at each occurrence in a data set is aggregated into a final score of such rule. If the WEIGHT= variable is not specified, then the aggregator specification is ignored.

The available aggregation methods are as follows:

| | |
|---|---|
| MAX | maximum value |
| MEAN | arithmetic mean |
| MIN | minimum value |
| SUM | sum of the nonmissing values |

**Alias** AGG=

**Default** SUM

**Interaction** You must specify the WEIGHT= option to use this option.

**CONFIDENCE( <LOWER=*lower-value*> <UPPER=*upper-value*> )**
specifies the minimal and maximal confidence values of the association rules have to fulfill. The default value for LOWER= is 0.5.

If you specify UPPER= < LOWER=, the server swaps the values. If you specify the same value for LOWER= and UPPER=, the server adds ε (0.111022302462515 7e-12) to value and uses the result for UPPER=.

**Range** 0 to 1

**FREQ=*variable-name***
specifies the numeric frequency variable to use for computing the score of each association rule along with ORDER= option. When a FREQ= variable is not specified, the score of an association rule equates the value of the ORDER= variable scaled by 1. Negative values for the specified variable are considered missing.

**ITEMAGG=*aggregation-method***
lists the aggregator for which the values of the WEIGHT= variable, and optionally the FREQ= variable, are rolled up into the score of a rule at each occurrence in the data set, provided that a WEIGHT= variable is specified. If you do not specify a WEIGHT= variable, then the aggregator specification is ignored. The aggregation methods are identical to the list in the AGGREGATE= option.

**NUMLHS( <LOWER=*lower-value*> <UPPER=*upper-value*> )**
specifies the minimum and maximum number of items in the left-hand side (LHS) of a rule to allow. If you specify UPPER= < LOWER=, the server swaps the values.

**NUMRHS( <LOWER=*lower-value*> <UPPER=*upper-value*> )**
specifies the minimum and maximum number of items in the right-hand side (RHS) of a rule to allow. If you specify UPPER= < LOWER=, the server swaps the values.

**SCORE( <LOWER=*lower-value*> <UPPER=*upper-value*> )**
specifies the minimum and maximum scores of the association rules that are derived. If you specify UPPER= < LOWER=, the server swaps the values. If you specify the same value for LOWER= and UPPER=, the server adds ε (0.111022302462515 7e-12) to value and uses the result for UPPER=.

**WEIGHT=*variable-name***
specifies the numeric weight variable to use for computing the score of each association rule, along with FREQ= variable. If you do not specify a WEIGHT= variable, then the AGGREGATE=, FREQ=, and ITEMAGG= options are ignored.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SCORE( <LOWER=*lower-value*> <UPPER=*upper-value*> )**
specifies the minimum and maximum scores of the frequent itemsets that are derived. If you specify UPPER= < LOWER=, the server swaps the values. If you specify the same value for LOWER= and UPPER=, the server adds ε (0.1110223024625157e-12) to value and uses the result for UPPER=.

**SEQUENCES(TIME=*t* <*sub-options*> )**
specifies the requirements for how sequences are generated from the original table. The sequences do not necessarily depend on previously generated frequent itemsets. You can specify the following sub-options in SEQUENCES option:

**ADJACENT**
specifies that any two events of a sequence must be adjacent to each other in time in a transaction.

For example, in the following table, the transaction supports only two sequences with a chain length of 3. The first is $A \Rightarrow B \Rightarrow C$ and the second is $B \Rightarrow C \Rightarrow D$. The transaction does not support sequence $A \Rightarrow B \Rightarrow D$ because events B and D do not happen consecutively in this transaction. By default, ADJACENT option is not enabled so that the transaction would support the third sequence, $A \Rightarrow B \Rightarrow D$, when the chain length is 3.

| Transaction | Item | Time |
|---|---|---|
| 0 | A | 0 |
| 0 | B | 1 |
| 0 | C | 2 |
| 0 | D | 3 |

**AGGREGATE=*aggregation-method***
lists the aggregator for which the score of a sequence at each occurrence in a data set is aggregated into a final score of such sequence. If the WEIGHT= variable is not specified, then the aggregator specification is ignored.

The available aggregation methods are as follows:

| | |
|---|---|
| MAX | maximum value |
| MEAN | arithmetic mean |
| MIN | minimum value |
| SUM | sum of the nonmissing values |

| | |
|---|---|
| **Alias** | AGG= |
| **Default** | SUM |
| **Interaction** | You must specify the WEIGHT= option to use this option. |

**FREQ=*variable-name***
specifies the numeric frequency variable to use for computing the score of each sequence along with WEIGHT= option. When a FREQ= variable is not specified, the score of a sequence equates the value of the WEIGHT= variable scaled by 1. Negative values for the specified variable are considered missing.

**INCLUDEMISSTIME**
indicates that records with a missing value for the TIME= variable are considered for sequence analysis. If this option is specified, then the missing value for the TIME= variable is treated as the smallest value in a sequence.

**ITEMAGG=*aggregation-method***
lists the aggregator for which the values of the WEIGHT= variable, and optionally the FREQ= variable, are rolled up into the score of a rule at each occurrence in the data set, provided that a WEIGHT= variable is specified. If the WEIGHT= variable is not specified, then the aggregator specification is ignored. The aggregation methods are identical to the list in the AGGREGATE= option.

**ITEMSETFILTER=SINGLETONS | ALLITEMS | NONE**
specifies how the sequences are filtered by frequent itemsets. The SINGLETONS setting means that each item of any sequence has to be a frequent singleton. The ALLITEMS setting means the set of all distinct items of any sequence have to be a frequent itemset. The NONE setting indicates that sequences are not influenced by what frequent itemsets are derived.

The following IMSTAT statements specify that all frequent itemsets must have their support greater or equal than 150. The support ranges specified on sequences are LOWER=110 and UPPER=140. The support boundaries on frequent itemsets and sequences are disjoint. However, with FILTER=NONE, the sequences that are generated do not depend on the frequent itemsets.

```
proc imstat data=example.assocs;
  arm item=Product tran=Customer / maxItems=6 support(lower=150) itemsTbl
      sequs(time=Time minItems=4 maxItems=6 minWindow=0
            support(lower=110 upper=140) itemsetfilter=none);
```

| | |
|---|---|
| **Alias** | FILTER= |
| **Default** | SINGLETONS |

**LASRRULE=*table-name***
specifies an in-memory table that contains trained association rules. The rules are used to score the current active transaction table.

**LASRSEQU=*table-name***
specifies an in-memory table that contains trained sequences. The sequences are used to score the current active transaction table.

**MAXITEMS=*n***
specifies the maximal number of items to allow in any sequence. The value must be greater than or equal to 1. If an invalid value is specified, then it is replaced with 1, the default value.

specifies the maximum number of items to allow in any sequence. The value must be greater than or equal to 1. Otherwise, it is set to the default value, 1.

**MAXDURATION=*t***
specifies the maximum duration to allow between the onset time of the first item and the time of the last item in a sequence. If the difference is greater than *t*, then the sequence is excluded from the result set. The value must be greater than or equal to zero.

**MAXWINDOW=*t***
specifies the maximum difference to allow between the onset of any two adjacent items in a sequence. If the difference is greater than *t*, then the two items cannot be part of the same sequence. The value must be greater than or equal to zero.

**MINDURATION=*t***
specifies the minimum difference to allow between the onset time of the last item and the first item in a sequence. If the difference is less than *t*, then the sequence is excluded from the result set. The value must be greater than or equal to zero. If you specify a value for MAXDURATION= that is less than MINDURATION=, the server swaps the values.

**MINITEMS=*n***
specifies the minimal number of items to allow in a sequence. The value must be greater than or equal to 1. If an invalid value is specified, then it is replaced with 1, the default value.

If you specify MAXITEMS= < MINITEMS=, the server swaps the values. If you specify MAXITEMS = MINITEMS, the server assigns MAXITEMS = MINITEMS + 1.

**Default** 1

**MINWINDOW=*t***
specifies the minimum difference to allow between the onset of any two adjacent items in a sequence. If the difference is less than or equal to *t*, then the two items are treated as happening at the same time. The value must be greater than or equal to zero.

If you specify MAXWINDOW= < MINWINDOW=, the server swaps the values.

**NODUP**
specifies that duplicated items within a sequence are not allowed.

**NOMERGE**
specifies that a transaction supports only one sequence with the same number of events in that transaction. In the transaction table that is shown in the ADJACENT option, the transaction supports only one sequence, $A \Rightarrow B \Rightarrow C \Rightarrow D$. By default, the NOMERGE option is not enabled.

**Interaction** Specifying this option implies the ADJACENT option.

**SCORE( <LOWER=*lower-value*> <UPPER=*upper-value*> )**
specifies the minimum and maximum scores of the sequences that are derived. If you specify UPPER= < LOWER=, the server swaps the values. If you specify the same value for LOWER= and UPPER=, the server adds ε (0.1110223024625157e-12) to the value and uses the result for UPPER=.

**SUPPORT( <LOWER=*lower-value*> <UPPER=*upper-value*> )**
specifies the minimum and maximum support of one sequence allowed in the analysis. By default, LOWER=1 and UPPER= is not set. Valid values for LOWER= and UPPER= are integers greater than 0. If you specify an invalid

value for LOWER= or UPPER=, the server sets LOWER=1. The value for LOWER= must be less than or equal to the UPPER= value. If you specify UPPER= < LOWER=, the server swaps the values. Note that

**Default** LOWER=1

**Note** This option does not overwrite the SUPPORT option that is specified for deriving frequent itemsets.

**TIME=*t***
specifies the numeric variable to use for sorting the items in a sequence. This option is required for sequence analysis.

**TIMEAGG=*aggregation-method***
specifies how to aggregate the time values when two adjacent events are the same in a sequence.

The available aggregation methods are as follows:

MAX maximum value

MEAN arithmetic mean

MIN minimum value

For example, see the values in the following table:

| Transaction | Item | Time |
|---|---|---|
| 0 | A | 0 |
| 0 | B | 1 |
| 0 | B | 2 |
| 0 | D | 3 |

The aggregated timestamps for the events in the sequence with length three ($A \Rightarrow B \Rightarrow D$) are as follows:

| TIMEAGG= Value | Item A | Item B | Item D | Sequence |
|---|---|---|---|---|
| MAX | 0 | 1 | 3 | $A \Rightarrow B\&D$ |
| MIN | 0 | 1 | 3 | $A\&B \Rightarrow D$ |
| MEAN | 0 | 1.5 | 3 | $A \Rightarrow B \Rightarrow D$ |

If you also specify MINWINDOW=1, then the sequences will be different from the sequences shown in the previous table.

**WEIGHT=*variable-name***
specifies the numeric weight variable to use for computing the score of each sequence, along with FREQ= variable. If you do not specify a WEIGHT=

variable, then the AGGREGATE=, FREQ=, and ITEMAGG= options are ignored.

**WINDOWAGG=*aggregation-method***
is used with the MINWINDOW= and MAXWINDOW= options. It lists the aggregator for which the values of the TIME= variable to update the anchor time. The default value is MEAN.

The available aggregation methods are as follows:

MAX maximum value

MEAN arithmetic mean

MIN minimum value

For example, see the values in the following table:

| Transaction | Item | Time |
|---|---|---|
| 0 | A | 0 |
| 0 | B | 1 |
| 0 | C | 2 |
| 0 | D | 3 |

If MINWINDOW=0, then the following sequence is formed with a chain length of four because the time difference between two adjacent items is > 0.

$A \Rightarrow B \Rightarrow C \Rightarrow D$

If MINWINDOW=1 and WINDOWAGG=MIN, then the following sequence is formed with a chain length of two.

$A\&B \Rightarrow C\&D$

If MINWINDOW=1 and WINDOWAGG=MAX, then the following sequence is formed, with a chain length of one because after A and B are merged as concurrent items, the WINDOWAGG=MEAN setting defines the time value for A & B to be 0.5. Item C is then merged with A & B with a new merged time value of 1.0. Because the time value for item D is 3, then it is not merged with A & B &C.

$A\&B\&C \Rightarrow D$

If MAXWINDOW=1, then no two items in the transaction can form a sequence.

**SEQUSTBL**
specifies to save the derived sequences from frequent itemsets to a temporary table. By default, the ARM statement does not save sequences.

**Example** "Sequences Table" on page 92

**SUPPORT( <LOWER=*lower-value*> <UPPER=*upper-value*> )**
specifies the minimum and maximum frequencies to allow for derived frequent itemsets. If RELSUPPORT is not specified, then LOWER= and UPPER= are the minimum and maximum frequencies of frequent items that appeared in the transactions. If RELSUPPORT is specified, then specify the two values as the ratios

for the minimum and maximum frequencies of frequent itemsets to the frequency of the most frequent itemset.

By default, LOWER=1 when RELSUPPORT is not specified and LOWER=0.05 when RELSUPPORT is specified.

The values for LOWER= and UPPER= must be greater than or equal to zero. If you specify UPPER= < LOWER=, then the server swaps the values.

For example, the following statements derive and display frequent itemsets of sizes between MINITEMS=3 and MAXITEMS=4. The support for each frequent itemset must be between [LOWER=97, UPPER=100). LOWER= is inclusive and UPPER= is exclusive.

```
proc imstat data=example.assocs;
    arm item=Product tran=Customer / minItems=3 maxItems=4
        support(lower=97 upper100) itemsTbl;
run;
    table example.&_tempARMItems_;
    fetch / orderby=(_SetSize_ _Count) desc=_Count_;
run;
```

The ARM statement produces the following display, indicating that 14 frequent itemsets were derived.

| Association Rule Mining Summary | |
|---|---|
| **Descr** | **Value** |
| Data Source | WORK.ASSOCS |
| Number of Transactions | 1001 |
| Number of Frequent Itemsets | 14 |
| Frequent Itemset Table | _T_ACC833E3_7F8DF286E008 |

The FETCH statement displays the 14 frequent itemsets.

| Selected Records from Table _T_ACC833E3_7F8DF286E008 | | | | | | |
|---|---|---|---|---|---|---|
| _SetSize_ | _Count_ | _Support_ | PRODUCT1 | PRODUCT2 | PRODUCT3 | PRODUCT4 |
| 3.000000 | 99.000000 | 0.098901 | steak | hering | apples | |
| 3.000000 | 99.000000 | 0.098901 | steak | olives | apples | |
| 3.000000 | 99.000000 | 0.098901 | turkey | ice_crea | bourbon | |
| 3.000000 | 99.000000 | 0.098901 | peppers | cracker | chicken | |
| 3.000000 | 98.000000 | 0.097902 | corned_b | chicken | bourbon | |
| 3.000000 | 98.000000 | 0.097902 | turkey | coke | bourbon | |
| 3.000000 | 97.000000 | 0.096903 | peppers | baguette | apples | |
| 3.000000 | 97.000000 | 0.096903 | peppers | corned_b | bourbon | |
| 4.000000 | 99.000000 | 0.098901 | hering | baguette | avocado | artichok |
| 4.000000 | 99.000000 | 0.098901 | ham | cracker | avocado | artichok |
| 4.000000 | 97.000000 | 0.096903 | steak | hering | corned_b | apples |
| 4.000000 | 97.000000 | 0.096903 | steak | olives | corned_b | apples |
| 4.000000 | 97.000000 | 0.096903 | steak | olives | hering | apples |
| 4.000000 | 97.000000 | 0.096903 | olives | ice_crea | coke | bourbon |

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TRANFMT=("*format-specification*")**
specifies the formats for the TRAN= variable. If you do not specify the TRANFMT= option, then the unformatted values of the TRAN= variable are used.

Enclose the format specification in quotation marks.

**WEIGHT=*variable***
specifies the numeric weight variable to use for computing the score of each frequent itemset, along with FREQ= variable. If you do not specify a WEIGHT= variable, then the AGGREGATE=, FREQ=, and ITEMAGG= options are ignored.

## Details

### Overview

Frequent itemsets are the a priori information in order to mine association rules. These are widely used in market basket analysis, web usage mining, and bio-informatics. Association rules are popular for discovering relations among different values of a variable. Sequence mining aims to discover the causality relationship among items in transactions of customer purchasing habits or anti-money laundry, for example.

By specifying ITEM= , TRAN=, and optionally TIME=, the server derives either the frequent itemsets, the association rules, the sequences, or any combinations of them. If TIME= is not specified, the server does not generate sequence results. The frequent itemsets, the association rules, and the sequences are stored in separate temporary tables in the server.

### Frequent Itemsets Table

The frequent itemsets table is generated when you specify the ITEMSTBL option and it is accessed with the &_tempARMItems_ macro variable. See the following example:

```
data example.aggdata;
    input customer product $ time price amount product_id;
    datalines;
1 e 0    2.49 2 1
1 t 1 2999.00 1 2
1 e 2    2.49 2 1
1 t 3  499.00 1 2
1 e 4    3.49 3 1
1 t 5  199.00 1 2
2 t 0  199.00 1 2
2 e 1    3.49 2 1
2 h 2   50.00 1 3
2 e 3    3.49 1 1
2 t 4  499.00 1 2
2 e 5    3.49 1 1
;
run;

proc imstat data=example.aggdata;
    arm item=product tran=customer / maxitems=3 freq=amount
        weight=price itemagg=SUM agg=MIN itemstbl;
run;
    table example.&_tempARMItems_;
    fetch / orderby=(_SetSize_);
run;
```

The preceding statements generate the following output for the sample data:

| Selected Records from Table _T_AD916891_14CCA8A8 | | | | | | |
|---|---|---|---|---|---|---|
| _SetSize_ | _Count_ | _Support_ | _Score_ | product1 | product2 | product3 |
| 0 | 2.000000 | 1.000000 | . | | | |
| 1.000000 | 2.000000 | 1.000000 | 13.960000 | e | | |
| 1.000000 | 2.000000 | 1.000000 | 698.000000 | t | | |
| 1.000000 | 1.000000 | 0.500000 | 50.000000 | h | | |
| 2.000000 | 2.000000 | 1.000000 | 711.960000 | t | e | |
| 2.000000 | 1.000000 | 0.500000 | 63.960000 | h | e | |
| 2.000000 | 1.000000 | 0.500000 | 748.000000 | t | h | |
| 3.000000 | 1.000000 | 0.500000 | 761.960000 | t | h | e |

The columns in the frequent itemsets table are as follows:

_SetSize_
shows the number of items in the frequent itemset.

_Count_
shows the frequency for the frequent itemset in all the transactions.

_Support_
shows the ratio of the _Count_ value to the number of transactions.

_Score_
shows the aggregated values of the FREQ= and WEIGHT= values, when they are specified.

Consider the frequent itemset for product *t* (row 3 in the preceding table). It appears 3 times for customer 1 and 2 times for customer 2. First, the server performs item aggregation with each customer. Then, the server performs second stage aggregation to obtain the final score of a frequent itemset. In this case, the intermediate scores of itemset *t* are (1*2999.00 + 1*499.00 + 1*199.00) = 3697.00 and (1*199.00 + 1*499.00) = 698.00. The final score of this itemset is MIN(3697.00, 698.00) = 698.00.

**PRODUCTn**
shows the name of an item in the frequent itemset. The column name is variable. The name is based the column that is specified in the ITEM= option.

### *Association Rules Table*

The association rules table is generated when you specify the RULES and RULESTBL options. It is accessed with the &_tempARMRules_ macro variable. For example, the following statements derive association rules of sizes between MINITEMS=3 and MAXITEMS=4. The support range of each frequent itemset is set at LOWER=125 and UPPER=130. The minimal confidence value permitted is 0.8. Each association rule's score has to be greater or equal than 2.

```
proc imstat data=example.assocs;
    arm item=Product tran=Customer / minItems=3 maxItems=4 itemsTbl
        support(LOWER=125 UPPER=130) weight=TIME
```

```
        rules(confidence(LOWER=0.8) score(LOWER=1) weight=TIME) rulesTbl;
run;

    table example.&_tempARMRules_;
       fetch _SetSize_ -- _Rule_ / to=10;
run;
```

*Note:* The FETCH statement in the preceding example does not include the values from the ITEM= column in the display.

The preceding statements generate the following output for the Assocs data:

**Selected Records from Table _T_ADA529EF_149FA848**

| _SetSize_ | _SetCount_ | _SetSupport_ | _SetScore_ | _Score_ | _Confidence_ | _ExpConf_ | _Lift_ | _NumLHS_ | _NumRHS_ | _Rule_ |
|---|---|---|---|---|---|---|---|---|---|---|
| 3.000000 | 127.000000 | 0.126873 | 578.000000 | 578.000000 | 0.830065 | 0.487512 | 1.702655 | 2.000000 | 1.000000 | baguette, soda ==> cracker |
| 3.000000 | 129.000000 | 0.128871 | 466.000000 | 466.000000 | 0.843137 | 0.485514 | 1.736585 | 2.000000 | 1.000000 | baguette, soda ==> hering |
| 3.000000 | 127.000000 | 0.126873 | 722.000000 | 722.000000 | 0.907143 | 0.472527 | 1.919767 | 2.000000 | 1.000000 | bourbon , turkey ==> olives |
| 3.000000 | 126.000000 | 0.125874 | 1137.000000 | 1137.000000 | 0.840000 | 0.599401 | 1.401400 | 2.000000 | 1.000000 | cracker , ham ==> heineken |
| 4.000000 | 125.000000 | 0.124875 | 1165.000000 | 1165.000000 | 0.976563 | 0.599401 | 1.629232 | 3.000000 | 1.000000 | baguette, cracker , hering ==> heineken |
| 4.000000 | 125.000000 | 0.124875 | 1165.000000 | 1165.000000 | 0.862069 | 0.485514 | 1.775578 | 3.000000 | 1.000000 | baguette, cracker , heineken ==> hering |
| 4.000000 | 126.000000 | 0.125874 | 1068.000000 | 1068.000000 | 0.913043 | 0.487512 | 1.872862 | 3.000000 | 1.000000 | baguette, heineken, soda ==> cracker |
| 4.000000 | 126.000000 | 0.125874 | 1068.000000 | 1068.000000 | 0.992126 | 0.599401 | 1.655197 | 3.000000 | 1.000000 | baguette, cracker , soda ==> heineken |
| 4.000000 | 126.000000 | 0.125874 | 1068.000000 | 1068.000000 | 0.868966 | 0.317682 | 2.735329 | 3.000000 | 1.000000 | baguette, cracker , heineken ==> soda |
| 4.000000 | 126.000000 | 0.125874 | 1068.000000 | 1068.000000 | 0.823529 | 0.365634 | 2.252330 | 2.000000 | 2.000000 | baguette, soda ==> cracker , heineken |

The columns in the association rules table are as follows:

_SetSize_
shows the number of items in the frequent itemset.

_SetCount_
shows the frequency for the frequent itemset that contain the rule in all the transactions.

_SetSupport_
shows the ratio of the _SetCount_ value to the total number of transactions.

_SetScore_
shows the aggregated values WEIGHT= values of all the frequent itemsets. , when they are specified. The AGGREGATOR=SUM and ITEMAGG= option defaults to SUM.

_Score_
shows the aggregated values of the WEIGHT= value in the association rule suboptions. The AGGREGATOR=MAX and ITEMAGG= option defaults to SUM.

_Confidence_
shows the confidence for the association rule.

_ExpConf_
shows the expected confidence for the association rule.

_Lift_
shows the lift for the association rule.

_NumLHS_
shows the number of items in the left-hand-side of a rule.

_NumRHS_
shows the number of items in the right-hand-side of a rule.

_Rule_
shows the full string of the rule.

### Sequences Table

The sequences table is generated when you specify the SEQUENCES and SEQUSTBL options. For example, the following statements derive association rules of sizes between MINITEMS=3 and MAXITEMS=4. The support range of each frequent itemset is set at LOWER=125 and UPPER=130. The minimal confidence value permitted is 0.8. Each association rule's score has to be greater or equal than 2.

```
proc imstat data=example.assocs;
    arm item=Product tran=Customer / maxItems=3
    sequences(time=time minItems=3 maxItems=3 support(lower=110 upper=120))
        itemsetfilter=allitems sequstbl;
run;
  table example.&_tempARMSequs_;
  fetch _ChainLength_ _Count_ _Support_ _Probability_ _LiftProduct_
        PRODUCT: _Sequ_ / to=10;
run;
```

The preceding statements generate the following output for the Assocs data:

Selected Records from Table _T_A2AE6FA1_C949630

| _ChainLength_ | _Count_ | _Support_ | _Probability_ | _LiftProduct_ | PRODUCT1 | PRODUCT2 | PRODUCT3 | _Sequ_ |
|---|---|---|---|---|---|---|---|---|
| 3.000000 | 111.000000 | 0.110889 | 0.982301 | 1.992692 | artichok | heineken | ham | artichok==>heineken==>ham |
| 3.000000 | 111.000000 | 0.110889 | 0.982301 | 3.293706 | avocado | artichok | ham | avocado ==>artichok==>ham |
| 3.000000 | 112.000000 | 0.111888 | 0.957265 | 1.689384 | avocado | heineken | ham | avocado ==>heineken==>ham |
| 3.000000 | 114.000000 | 0.113886 | 0.786207 | 0.995213 | baguette | cracker | heineken | baguette==>cracker ==>heineken |
| 3.000000 | 113.000000 | 0.112887 | 0.974138 | 1.948608 | baguette | hering | artichok | baguette==>hering ==>artichok |
| 3.000000 | 118.000000 | 0.117882 | 0.921875 | 1.709705 | baguette | hering | avocado | baguette==>hering ==>avocado |
| 3.000000 | 113.000000 | 0.112887 | 0.889764 | 1.861288 | baguette | soda | cracker | baguette==>soda ==>cracker |
| 3.000000 | 113.000000 | 0.112887 | 0.818841 | 1.513848 | baguette | soda | heineken | baguette==>soda ==>heineken |
| 3.000000 | 113.000000 | 0.112887 | 0.875969 | 1.868948 | baguette | soda | hering | baguette==>soda ==>hering |
| 3.000000 | 111.000000 | 0.110889 | 0.982301 | 1.795377 | bourbon | cracker | chicken | bourbon ==>cracker ==>chicken |

The columns in the association rules table are as follows:

_ChainLength_
shows the number of items in the sequence.

_Count_
shows the frequency of transactions that contain the sequence.

_Support_
shows the ratio of the _Count_ value to the total number of transactions.

_Probability_
is defined as $\Pr\left(A \rightarrow B \rightarrow C \rightarrow D\right) = \frac{N(A \rightarrow B \rightarrow C \rightarrow D)}{\mathrm{MIN}(N(A),\ N(B),\ N(C),\ N(D))}$ where N() is the count function when ITEMSETFILTER=ALLITEMS.

_LiftProduct_
is defined as $Lift\left(A \rightarrow B \rightarrow C \rightarrow D\right) = \frac{\frac{N(A \rightarrow B \rightarrow C \rightarrow D)}{Ntrans}}{\left(\frac{N(A)}{Ntrans}\right) * \left(\frac{N(B)}{Ntrans}\right) * \left(\frac{N(C)}{Ntrans}\right) * \left(\frac{N(D)}{Ntrans}\right)}$
where Ntrans is the number of transactions.

_Separatorn_
shows the relationship of the items to the left and right. "==>" indicates that the item on the left occurs before the item on the right. "&" indicates that the two items are considered to happen at the same time.

### ODS Table Names

The ARM statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| ARMSummary | Association rule mining summary | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## ASSESS Statement

The ASSESS statement is used to assess one model or several models. For a set of classification models, the ASSESS statement returns three types of assessments: lift-related assessments, assessments related to a receiver operating characteristic (ROC), and concordance statistics. For a set of regression models, the ASSESS statement returns the summary statistics of the response variable for each bin of the predictions after a quantile binning of the predictions.

**Example:** "Example 16: Predicting Email Spam and Assessing the Model" on page 284

### Syntax

**ASSESS** <*variable-list*> / Y=*response-variable* <*options*>;

#### Required Argument

**Y=*response-variable***
specifies the response variable for model assessment.

**Alias** RESPONSE=

#### ASSESS Statement Options

**CUTSTEP=*n***
specifies a number between 0 and 1 that defines the step size in receiver operating characteristic (ROC) calculations.

**Alias** STEP=

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**EPSILON=*e***
specifies the tolerance that is used in determining the convergence of the iterative algorithm for the percentile calculation.

**Default** $1e^{-5}$

**EVENT="*quoted-strings*"**
specifies the formatted value of the response variable that represents the event. When this option is not specified, the ASSESS statement performs model assessment for a regression model and the response variable must be numeric.

**FORMATS=("*format-specification*",…)**
specifies the formats for the GROUPBY= variables. If you do not specify the FORMAT= option, or if you do not specify the GROUPBY= option, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**GROUPBY=(*variable-list*)**
specifies a list of variable names, or a single variable name, to use as GROUPBY variables in the order of the grouping hierarchy. If you do not specify any GROUPBY variable names, then the calculation is performed across the entire table —possibly subject to a WHERE clause.

**GROUPBYLIMIT=*n***
specifies the maximum number of levels in a GROUPBY set. When the software determines that there are at least *n* levels in the GROUPBY set, it abandons the action, returns a message, and does not produce a result set. You can specify the GROUPBYLIMIT= option if you want to avoid creating excessively large result sets in GROUPBY operations.

**GROUPFILTER=(*groupfilter-options*)**
specifies a section of the GROUPBY= hierarchy to include in the ASSESS computation.

**MAXITER=*i***
specifies a positive integer that determines the maximum number of iterations for the percentile algorithm.

**Default** 5 × the number of bins (NBINS= option).

**MERGEBINS=*b***
specifies the number of bins to create when a numeric GROUPBY variable exceeds the MERGELIMIT=*n* specification. If you specify a MERGELIMIT, but do not specify a value for the MERGEBINS= option, the server automatically calculates the number of bins.

**MERGELIMIT=*n***
specifies that when the number of unique values in a numeric GROUPBY variable exceeds *n*, the variable is automatically binned and the GROUPBY structure is determined based on the binned values of the variable, rather than the unique formatted values.

For example, if you specify MERGELIMIT=500, any numeric GROUPBY variable with more than 500 unique formatted values is binned. Instead of returning results for more than 500 groups, the results are returned for the bins. You can specify the number of bins with the MERGEBINS= option.

**NBINS=*n***
specifies the number of bins to use in the lift calculations.

**NOMISSING**
specifies that you do not want to include missing values in the determination of Group-By values. By default, levels with missing values are included.

**PARTITION <=*partition-key*>**

When you specify this option and the table is partitioned, the results are calculated separately for each value of the partition key. In other words, the partition variables function as automatic GROUPBY variables. This mode of executing calculations by partition is more efficient than using the GROUPBY= option. With a partitioned table, the server takes advantage of knowing that observations for a partition cannot be located on more than one worker node.

If you do not specify a *partition-key*, the analysis is performed for all partitions. If you do specify a *partition-key*, the analysis is carried out for the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid partition key values for a table.

You can specify a *partition-key* in two ways. You can supply a single quoted string that is passed to the server, or you can specify the elements of a composite key separated by commas. For example, if you partition a table by variables GENDER and AGE, with formats $1 and BEST12, respectively, then the composite partition key has a length of 13. You can specify the partition for the 11-year-old females as follows:

```
statement / partition="F          11"; /* passed directly to the server */
statement / partition="F","11";        /* composed by the procedure */
```

If you choose the second format, the procedure composes a key based on formatting information from the server.

**Alias** PART=

**RAWORDER**

specifies that the ordering of the GROUPBY variables is based on the raw values of the variables instead of the formatted values.

**SAVE=*table-name***

saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**

requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the &_TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**YFORMAT="*quoted-string*"**
specifies the format for the response variable. This format produces the event specified in the EVENT= option.

**Alias** YFMT=

## Details

### Overview

You can compare multiple models by specifying predicted values from those models in the *variable-list*. You can compare models in different data segments with the GROUPBY= option. Note that you must specify the response variable for the ASSESS statement.

When *variable-list* is not provided, assessment statistics are computed for all numerical variables in the active table.

### ODS Table Names

The ASSESS statement generates the following ODS tables.

| ODS Table Name | Description | Option |
|---|---|---|
| LIFTInfo | Lift information | Default |
| LIFTRegInfo | Lift Information | When EVENT= is not specified. |
| ROCInfo | Receiver operating characteristic information | Default |

| ODS Table Name | Description | Option |
|---|---|---|
| TempTable | Information about a temporary table | TEMPTABLE |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## BOXPLOT Statement

The BOXPLOT statement generates a table with the information that can be used to generate a box plot. It does not generate the plot.

**Examples:** "Example 2: Retrieving Box Values" on page 257

"Example 3: Retrieving Box Plot Values with the NOUTLIERLIMIT= Option" on page 258

### Syntax

**BOXPLOT** <*variable-list*> <*/ options*>;

#### *Optional Argument*

***variable-list***
specifies one or more numeric variables. If you do not specify this option, then all numeric variables in the table are used.

#### *BOXPLOT Statement Options*

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**FORMATS=("*format-specification*",…)**
specifies the formats for the GROUPBY= variables. If you do not specify the FORMATS= option, or if you omit the entry for a GROUPBY variable, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
   boxplot x / groupby=(a b) formats=("8.3", "$10");
quit;
```

**GROUPBY=(*variable-list*)**
specifies a list of variable names, or a single variable name, to use as GROUPBY variables in the order of the grouping hierarchy. If you do not specify any GROUPBY variable names, then the calculation is performed across the entire table —possibly subject to a WHERE clause.

**GROUPBYLIMIT=*n***
specifies the maximum number of levels in a GROUPBY set. When the software determines that there are at least *n* levels in the GROUPBY set, it abandons the action, returns a message, and does not produce a result set. You can specify the GROUPBYLIMIT= option if you want to avoid creating excessively large result sets in GROUPBY operations.

**MERGEBINS=*b***
specifies the number of bins to create when a numeric GROUPBY variable exceeds the MERGELIMIT=*n* specification. If you specify a MERGELIMIT, but do not specify a value for the MERGEBINS= option, the server automatically calculates the number of bins.

**MERGELIMIT=*n***
specifies that when the number of unique values in a numeric GROUPBY variable exceeds *n*, the variable is automatically binned and the GROUPBY structure is determined based on the binned values of the variable, rather than the unique formatted values.

For example, if you specify MERGELIMIT=500, any numeric GROUPBY variable with more than 500 unique formatted values is binned. Instead of returning results for more than 500 groups, the results are returned for the bins. You can specify the number of bins with the MERGEBINS= option.

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**NOUTLIERBINS=*b***
specifies the number of bins for reporting outliers. The default number of bins is 10 if you do not specify an NOUTLIERBINS= value, but do specify the OUTLIERS option. Specifying a nonzero value for NOUTLIERBINS= implies the specification of the OUTLIERS option.

**Alias** NOUTBINS=

**Default** 10

**NOUTLIERLIMIT=*k***
specifies the largest number of outliers to be returned. If you request outliers with the OUTLIERS option, and you specify a NOUTLIERLIMIT= value, the actual outliers are being returned rather than the binned values. Specifying a nonzero value for NOUTLIERLIMIT= implies the specification of the OUTLIERS option.

**Alias** NOUTLIMIT=

**OUTLIERS**
specifies to include outliers in computations and results. If the NOUTLIMIT=*n* option is specified, then the server returns up to *n* outliers on the high and low ends of the distribution. Otherwise, outliers are binned into NOUTLIERBINS=*b* bins.

**PARTITION <=*partition-key*>**
When you specify this option and the table is partitioned, the results are calculated separately for each value of the partition key. In other words, the partition variables function as automatic GROUPBY variables. This mode of executing calculations by partition is more efficient than using the GROUPBY= option. With a partitioned

table, the server takes advantage of knowing that observations for a partition cannot be located on more than one worker node.

If you do not specify a *partition-key*, the analysis is performed for all partitions. If you do specify a *partition-key*, the analysis is carried out for the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid partition key values for a table.

You can specify a *partition-key* in two ways. You can supply a single quoted string that is passed to the server, or you can specify the elements of a composite key separated by commas. For example, if you partition a table by variables GENDER and AGE, with formats $1 and BEST12, respectively, then the composite partition key has a length of 13. You can specify the partition for the 11-year-old females as follows:

```
statement / partition="F          11"; /* passed directly to the server */
statement / partition="F","11";        /* composed by the procedure */
```

If you choose the second format, the procedure composes a key based on formatting information from the server.

**Alias** PART=

**RAWORDER**
specifies that the ordering of the GROUPBY variables is based on the raw values of the variables instead of the formatted values.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### *ODS Table Names*

The BOXPLOT statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| BoxPlot | Input for a box plot display | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# CLUSTER Statement

The CLUSTER statement can be used to perform a *k*-means cluster analysis that uses the Euclidean distance between values or it can use a density-based algorithm—DBSCAN—that was originally developed to discover clusters in large spatial databases with noise.

**Example:** "Example 5: Performing a Cluster Analysis" on page 261

## Syntax

**CLUSTER** <*variable-list*> <*/ options*>;

### *Optional Argument*

***variable-list***
specifies a list of variables. If you do not specify this option, then all the variables in the table are used.

For the *k*-means clustering, the variable list defines a vector for each observation that is used to compute the Euclidean distance between the observations. By minimizing the within-cluster sum of squares (WCSS) of this distance, a set of clusters and their centers are determined.

### *CLUSTER Statement Options*

**BUBMAXPTS=*n***
specifies the maximum number of points in each bubble and must exceed the value specified in the BUBMINPTS= option.

**Applies to** METHOD=DBSCAN

**BUBMINPTS=*n***
specifies the minimum number of points in each bubble.

**Default** 1

**Applies to** METHOD=DBSCAN

**CLUSTINFO**
specifies to save the cluster center that each observation belongs to, and the distance between them, into the temporary table.

**Interaction** You must specify the TEMPTABLE option along with this option.

**CODE <(*code-generation-options*)>**
requests that the server produce SAS scoring code based on the actions that it performed during the analysis. This option is available when the METHOD=KMEANS only.

The server generates DATA step code. By default, the code is replayed as an ODS table by the procedure as part of the output of the statement. More frequently, you might want to write the scoring code to an external file by specifying the FILENAME= option.

The scoring code computes a value for a cluster identifier and includes the value in the ClusterID variable. When you run the scoring code, if any variables in an observation have missing values, then the row is given a ClusterID of –1. In addition, if you compare the results from running the statement interactively to the results from running the scoring code, you can find differences. Observations can be assigned to different clusters because the generated scoring code has less accuracy compared to the in-memory cluster centers.

**COMMENT**
specifies to add comments to the code in addition to the header block. The header block is added by default.

**FILENAME='*path*'**
specifies the name of the external file to which the scoring code is written. This suboptions applies only to the scoring code itself.

**Alias** FILE=

**FORMATWIDTH=*k***
specifies the width to use in formatting derived numbers such as parameter estimates in the scoring code. The server applies the BEST format, and the default format for code generation is BEST20.

**Alias** FMTW=

**Range** 4 to 32

**LABELID=*id***
specifies a group identifier for group processing. The identifier is an integer and is used to create array names and statement labels in the generated code.

**LINESIZE=*n***
specifies the line size for the generated code.

**Alias** LS=

**Default** 72

**Range** 64 to 256

**CONV=*c***
specifies the convergence criterion *c* for the *k*-means analysis. When the relative change in within-cluster sum of squares (WCSS) between successive iterations is less than *c*, the analysis is presumed to have converged.

**Default** $1e>^{-5}$

**Applies to** METHOD=KMEANS

**DISP=(*variable-list*)**
**DISP=*variable-name***
specifies the variable or variables to include in the clustering results. Use this option with the NSAMP= option to generate output that is suitable for graphing.

**DIST= EUC | SQUAREDEUC | MANHATTAN | MAXIMUM | COSINE | JACCARD | HAMMING**
specifies the distance measure used in the DBSCAN method. The *k*-means method uses DIST=EUC.

**DMAX=*v***
specifies the maximum diameter of bubbles with the specified DIST= distance measure.

**Default** 0

**EPS=*r***
specifies the distance value for neighborhood querying in the DBSCAN method. You must specify a value for *r* when METHOD=DBSCAN. There is no default value since *r* is a distance measurement and depends on the range of the data.

The values of EPS= and MINPTS= are important for the number of clusters that DBSCAN can find. The EPS= value determines the minimal radius of the clusters in terms of the distance measurement. (See the DIST= on page 102 option.) The value of MINPNTS=*n* suggests that data points in a cluster with less than *n* observations are noise.

**FREQ=(*variable-list*)**
**FREQ=*variable-name***
specifies the variable or variables that are used to perform the frequency analysis for each cluster. The procedure generates separate output tables for each variable.

**INITMETHOD=FORGY | RAND | AVG**
specifies the method for obtaining the initial estimate of cluster assignment. For the INIT=FORGY partition method, the initial mean centers are computed from NUMCLUS × $n_t$ × $n_n$ observations where $n_t$ and $n_n$ are the number of threads and number of nodes used by the server.

When you specify INIT=RAND, all methods are assigned randomly to one of the NUMCLUS clusters. When you specify INIT=AVG, the initial centers are formed by averaging the observations on a thread-by-thread basis.

**Alias** INIT=

**Default** FORGY

**MAXITER=*i***
specifies a positive integer that determines the maximum number of iterations for clustering.

**Default** 10

**METHOD=KMEANS | DBSCAN**
specifies the clustering method for the analysis.

**Default** KMEANS

**MINPTS=*n***
specifies the minimum number of points required in one cluster for the DBSCAN method. The EPS= and MINPTS= options can have a dramatic effect on the clusters that are generated with the DBSCAN method. You should exercise care in specifying the values for these options.

**Default** 1, which means that any cluster should contain at least one data point.

**NOCASE**
specifies that the comparison between terms and the values of character variables is case insensitive. By default, comparisons are case sensitive.

**NOIDF**
specifies that only the term frequency is used to construct the vectors, and not the inverse document frequency.

**NONORM**
specifies that the term frequency-inverse document frequency (TF-IDF) vectors are not normalized.

**NOPREPARSE**
prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NREP=*k***
specifies the number of representative points for each bubble.

**Default** 1

**NSAMP=*k***
specifies the number of sample points that are returned for each cluster. Note that this returns the *k* nearest and *k* farthest points from the cluster centers including their distances.

**Default** 0

**NUMCLUSTERS=*number***
specifies the number of clusters for the *k*-means analysis.

**Alias** NUMCLUS=

**Default** 2

**NSAMP=*k***
specifies the number of sample points to return for each cluster. This option returns the *k* nearest and *k* farthest points from the cluster centers, including their Euclidean distances.

**Default** 0

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SAVETERMS**
specifies to save the TF-IDF vectors into the temporary table along with clustering results.

**Interaction** You must specify the TEMPTABLE option along with this option.

**SAVEVECTORS**
specifies to save the distance vectors to the temporary table along with clustering results. When the VARS= option on page 106 already includes the variables used to obtain the distance vectors, these variables are not saved again

**Alias** SAVEVEC

**Interaction** You must specify the TEMPTABLE option along with this option.

**SEED=*number***
specifies the random number seed to use when the initialization methods INIT=FORGY or INIT=RAND are also specified. Specifying a nonzero number results in a reproducible random number stream for the specific combination of number of threads and number of worker nodes in the server.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the

estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the &_TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**TERMS=("*term1*" <, "*term2*"…> )**
specifies the terms used in computing term frequency. Each string represents one term. Specifying the TERMS= option triggers the use of TF-IDF to compute distance vectors for character variables. The TERMS= option is useful if you have a fairly small number of terms to pass to the server. If the number of terms is large, you might want to use the TERMDATA= option instead.

**TERMDATA=*table-name***
specifies an in-memory table in the server that contains the term list. Specifying the TERMDATA= option triggers the use of TF-IDF to compute distance vectors for character variables. If you specify both the TERMS= and TERMDATA= options, the server uses the union of the two sets in the TF-IDF calculation. Note that the IMSTAT procedure assumes that the TERMDATA= table already exists in the server.

**TIMEOUT=*seconds***
specifies the maximum number of seconds that the server should run the statement. If the time-out is reached, the server terminates the request and generates an error and error message. By default, there is no time-out.

**TOKENS=("*token1*" <, "*token2*"…> )**
specifies the tokens that separate terms when scanning character variables. If you do not specify the tokens (term delimiters), then the terms are compared against the full raw length of the character variable.

For example, if the term list is "better", "worse", and the variable Opinion contains "Recent news is better than last week," then without a token list that contains the " " (blank) delimiter, the term "better" is not counted. In other words, the absence of the blank token prevents the TF-IDF from scanning "better". If your token list is large, you might want to use the TOKENDATA= option instead.

**TOKENDATA=*table-name***
specifies an in-memory table in the server that contains the tokens list.

**VARS=*variable-name***
**VARS=(*variable-name1* <, *variable-name2*, …> )**
specifies the names of the variables to transfer to a temporary table in the server. This option is ignored unless you score an in-memory table and the TEMPTABLE option is specified. The observations with these variables are copied to the generated temporary table.

## Details

### *Clustering Methods*

Two clustering methods are implemented in the CLUSTER statement:

- The default clustering method is *k*-means clustering. For this method, the optional list of variables defines a vector for each observation, which is used to compute the Euclidean distance between the observations. By minimizing the within-cluster sum of squares (WCSS) of this distance, a set of clusters and their centers can be determined.
- You can also use a density-based algorithm, DBSCAN. It was originally developed to discover clusters in large spatial databases with noise, and was published in the proceedings of KDD 1996. You do not need to supply the number of clusters for the DBSCAN method. The bubbling scheme is designed specifically to improve the DBSCAN performance for large data sets. Bubbling is disabled by default, because the default value of the DMAX= option is 0.

The statement also supports TF-IDF (term frequency-inverse document frequency) to compute distance vectors for character variables. For any observation with TF-IDF, the total length of the distance vector is given by the number of numerical variables plus the number of terms. The following options relate to TF-IDF calculations: TERMS=, TERMDATA=, TOKENS=, TOKENDATA=, NOCASE, NOIDF, NONORM, and SAVETERMS. You trigger TF-IDF calculations by specifying the TERMS= or TERMDATA= options in the CLUSTER statement.

### *ODS Table Names*

The CLUSTER statement generates the following ODS tables.

| ODS Table Name | Description | Option |
|---|---|---|
| CLUSTKMEANS | k-means clustering analysis summary | Default |
| CLUSTFREQ | Frequency analysis summary of clusters | FREQ= |
| TempTable | Information about a temporary table | TEMPTABLE |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# CORR Statement

The CORR statement is used to calculate a matrix of pairwise correlations of numeric variables in an in-memory table.

**Example:** "Example 6: Performing a Pairwise Correlation" on page 262

## Syntax

**CORR** <*variable-list*> <*/ options*>;

### Optional Argument

***variable-list***
specifies one or more numeric variables. If you do not specify this option, then all numeric variables in the table are used.

### CORR Statement Options

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### ODS Table Names

The CORR statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| Correlation | Matrix of pairwise correlations | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## CROSSTAB Statement

The CROSSTAB statement is used to calculate one and two dimensional tables. You can use GROUPBY= variables, partitioned tables, a WEIGHT= variable to calculate multiple statistics for each table cell, and calculate marginal versions of the statistics as well.

**Example:** "Example 7: Crosstabulation with Measures of Association and Chi-Square Tests" on page 263

### Syntax

**CROSSTAB** *row-variable* </ *options*>;

**CROSSTAB** *row***column* </ *options*>;

#### *Required Argument*

**row-variable**
specifies to create a one-dimensional table using the specified variable.

#### *Optional Argument*

**row**
**column**
specifies to create a two-dimensional table based on the two variables.

#### *CROSSTAB Statement Options*

**ACROSSBY**
specifies that the levels of row and column variables are the same across the GROUPBY= variables. If you specify this option, then the tables in each group are shown with the same row and column layout. If the ACROSSBY option is not specified, then the particular row and column levels might be different among the groups based on which values of the row and column variables occur in each group.

**Alias** ACROSS

**AGGREGATE=(*statistic(s)*)**
specifies the statistics to use as aggregation methods for which crosstabulations are computed when a WEIGHT variable is also specified. If no WEIGHT variable is specified, then the N aggregator is applied. In other words, the crosstabulation shows the frequency with which the values occur when no WEIGHT variable is specified. If you specify multiple aggregation methods, then the server computes a crosstabulation for each method.

The available aggregation methods are as follows:

| | |
|---|---|
| CSS | corrected sum of squares |
| CV | coefficient of variation |
| MAX | maximum value |
| MEAN | arithmetic mean |
| MIN | minimum value |

| | |
|---|---|
| N | number of observations |
| PROBT | *p*-value for the *t*-statistic |
| STD | standard deviation |
| STDERR | standard error |
| SUM | sum of the nonmissing values |
| TSTAT | t-statistic for the null hypothesis that the mean equals zero |
| USS | uncorrected sum of squares |
| VAR | sample variance |

**Alias** AGG=

**ASSOCIATION**
specifies to calculate the measures of association between the row and column variable of the cross tabulation. The option generates the following measures: the Gamma statistic, Kendall's Tau-b, Stuart's Tau-c, Somers' measures, Lambda measures, and uncertainty measures.

**Alias** MEASURES

**CHISQ**
computes Chi-square statistics for the test of independence of the row and column variables and their asymptotic *p*-values. The option calculates the Pearson Chi-square statistic as well as the likelihood-ratio test.

**COLBINS=*b***
specifies the number of bins to use for binning the column variable. If the values in the data are such that there are no contributing observations for a bin, then the server does not return the bin. As a result, the server can return fewer bins than you specified.

**Alias** XBINS=

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**FORMATS=("*format-specification*",…)**
specifies the formats for the GROUPBY= variables. If you do not specify the FORMATS= option, or if you omit the entry for a GROUPBY variable, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
   crosstab x*y / groupby=(a b) formats=("8.3", "$10");
quit;
```

**GROUPBY=(*variable-list*)**
specifies a list of variable names, or a single variable name, to use as GROUPBY variables in the order of the grouping hierarchy. If you do not specify any

GROUPBY variable names, then the calculation is performed across the entire table —possibly subject to a WHERE clause.

**GROUPBYLIMIT=*n***
specifies the maximum number of levels in a GROUPBY set. When the software determines that there are at least *n* levels in the GROUPBY set, it abandons the action, returns a message, and does not produce a result set. You can specify the GROUPBYLIMIT= option if you want to avoid creating excessively large result sets in GROUPBY operations.

**MARGINS= ROW | COL | ALL | NONE**
specifies whether to calculate marginal values in addition to the crosstabulation. The default is MARGINS=NONE, which specifies that no marginal values are calculated. The MARGINS=ROW setting calculates margins for the row variable only. The MARGINS=COL setting calculates margins for the column variable only. The MARGINS=ALL setting calculates row margins, column margins, and the overall margin.

These calculations are repeated for each aggregate method in the CROSSTAB request.

**Default** NONE

**MERGEBINS=*b***
specifies the number of bins to create when a numeric GROUPBY variable exceeds the MERGELIMIT=*n* specification. If you specify a MERGELIMIT, but do not specify a value for the MERGEBINS= option, the server automatically calculates the number of bins.

**MERGELIMIT=*n***
specifies that when the number of unique values in a numeric GROUPBY variable exceeds *n*, the variable is automatically binned and the GROUPBY structure is determined based on the binned values of the variable, rather than the unique formatted values.

For example, if you specify MERGELIMIT=500, any numeric GROUPBY variable with more than 500 unique formatted values is binned. Instead of returning results for more than 500 groups, the results are returned for the bins. You can specify the number of bins with the MERGEBINS= option.

**NOEMPTY**
specifies that empty cells are not returned to the SAS session (only full cells are returned). When this option is specified, the server arranges the results into a vector of nonzero values with row and column indices. This sparse storage is memory efficient when the table has many empty cells.

**Alias** FULLCELL

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**NOPREPARSE**
prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects

problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NOTEMPPART**
specifies that the temporary table generated by the TEMPTABLE option is not partitioned by the GROUPBY= variables. When you request a temporary table with the CROSSTAB statement, by default, the server creates a partitioned table. When the number of groups is large, this can result in many small partitions, and requires extra memory resources to store the partition information for the temporary table. By specifying this option, the temporary table is organized similarly to the default table, but is not a partitioned table.

**Alias** NOTP

**PARTITION <=*partition-key*>**
When you specify this option and the table is partitioned, the results are calculated separately for each value of the partition key. In other words, the partition variables function as automatic GROUPBY variables. This mode of executing calculations by partition is more efficient than using the GROUPBY= option. With a partitioned table, the server takes advantage of knowing that observations for a partition cannot be located on more than one worker node.

If you do not specify a *partition-key*, the analysis is performed for all partitions. If you do specify a *partition-key*, the analysis is carried out for the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid partition key values for a table.

You can specify a *partition-key* in two ways. You can supply a single quoted string that is passed to the server, or you can specify the elements of a composite key separated by commas. For example, if you partition a table by variables GENDER and AGE, with formats $1 and BEST12, respectively, then the composite partition key has a length of 13. You can specify the partition for the 11-year-old females as follows:

```
statement / partition="F          11"; /* passed directly to the server */
statement / partition="F","11";        /* composed by the procedure */
```

If you choose the second format, the procedure composes a key based on formatting information from the server.

**Alias** PART=

**RAWORDER**
specifies that the ordering of the GROUPBY variables is based on the raw values of the variables instead of the formatted values.

**ROWBINS=*n***
specifies the number of bins to use for binning the row variable. If the values in the data are such that there are no contributing observations for a bin, then the server

does not return the bin. As a result, the server can return fewer bins than you specified.

**Alias** YBINS=

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the &_TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**Interaction** The TEMPTABLE option requires a group-by analysis or a partitioned analysis with this statement.

**WEIGHT=*variable-name***
specifies the numeric weight variable to use for calculating the statistics in the table cell and the margins of the table. If no WEIGHT variable is specified, then the only aggregate method that is available to the CROSSTAB statement is N. In this case, then number of observations (frequency) is reported in each table cell.

## Details

### *ODS Table Names*

The CROSSTAB statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| Association | Measures of association in a crosstabulation | ASSOCIATION |
| ChiSq | Chi-Square statistic in a crosstabulation | CHISQ |
| CrossTab | Crosstabulation from a LASR Analytic Server table | Default |
| TempTable | Information about a temporary table | TEMPTABLE |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# DECISIONTREE Statement

The DECISIONTREE statement provides an implementation of a C4.5 decision tree method for classification. You specify a single column as the target variable when you generate the decision tree. You can also score against the generated tree.

**Examples:** "Example 8: Training and Validating a Decision Tree" on page 265
"Example 9: Storing and Scoring a Decision Tree" on page 267

## Syntax

**DECISIONTREE** *target-variable* <*/ options*>;

### *DECISIONTREE Statement Options*

**ASSESS**
specifies that predicted probabilities are added to the temporary result table for the event levels. You can use these predicted probabilities in an ASSESS statement.

**CODE <(*code-generation-options*)>**
requests that the server produce SAS scoring code based on the actions that it performed during the analysis. The server generates DATA step code. By default, the code is replayed as an ODS table by the procedure as part of the output of the

statement. More frequently, you might want to write the scoring code to an external file by specifying options.

The scoring code computes the predicted value of the response variable and prefixes the name with "DT_". For example, if the response variable is **Y**, the generated code stores the predicted value as **DT_Y**. The name of the variable is truncated to fit within the SAS name length requirements.

**COMMENT**
specifies to add comments to the code in addition to the header block. The header block is added by default.

**FILENAME='*path*'**
specifies the name of the external file to which the scoring code is written. This suboption applies only to the scoring code itself.

| | |
|---|---|
| **Alias** | FILE= |

**FORMATWIDTH=*k***
specifies the width to use in formatting derived numbers such as parameter estimates in the scoring code. The server applies the BEST format, and the default format for code generation is BEST20.

| | |
|---|---|
| **Alias** | FMTW= |
| **Range** | 4 to 32 |

**LABELID=*id***
specifies a group identifier for group processing. The identifier is an integer and is used to create array names and statement labels in the generated code.

**LINESIZE=*n***
specifies the line size for the generated code.

| | |
|---|---|
| **Alias** | LS= |
| **Default** | 72 |
| **Range** | 64 to 256 |

**NOTRIM**
specifies to format the variables using the full format width with padding. By default, leading and trailing blanks are removed from the formatted values.

**REPLACE**
specifies to overwrite the external file if a file with the specified name already exists. The option has no effect unless you specify the FILENAME= option.

**CFLEV=*number***
specifies a value between 0 and 1 that controls the aggressiveness of tree pruning according to the C4.5 algorithm. Smaller numbers indicate to more aggressive pruning. See the "PRUNE" on page 117 option.

**DETAIL**
requests detailed information about the classification results when scoring a table against a previously calculated tree.

**FORMATS=("*format-specification*",…)**
specifies the formats for the input variables. If you do not specify the FORMATS= option, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
   decisiontree x / input=(a b) formats=("8.3", "$10");
quit;
```

**GAIN**
specifies that the splitting criterion is changed to information gain. Typically, this criterion intends to generate trees with more nodes than information gain ratio.

**GREEDY**
specifies how to perform splitting under specific circumstances.

Assuming that one variable has $q$ levels, when binary splitting is performed and $q$ is less than 15, or option MAXBRANCH > 2 and $q < 12$, all possible binary splits are enumerated and the split with the largest gain or gain ratio is chosen for the variable.

When $q$ is less than 1024 and splitting is not just binary, local greedy searches are applied to determine the optimum local split. Specifically, when the variable is numeric, $q$ levels (similar to $q$ bins) are sorted by value.

When the variable is nominal, the $q$ levels are ordered by random weights. The best binary splitting is applied until the desired number of branches is reached. Only a local optimum can be found with this technique.

For values of $q \geq 1024$, the default $k$-means clustering algorithm is applied to determine the splits.

**IMPUTE**
specifies how to treat observations with nonmissing values for the target variable during scoring. When this option is specified, the observed values are used as the predicted values. That is, the observed value is assumed to be known without error. Only the observations with missing values for the target variable are then scored against the decision tree based on their values for the input variables.

The IMPUTE option is useful if you want to replace missing values of a target variable with classified values based on the decision tree.

**INPUT=*variable-name***
**INPUT=(*variable-name1 <variable-name2, …>* )**
specifies the variables to use for building the tree. You can add the target variable to the input list if you want to assign a format to the target variable by using the FORMATS= option. Any numeric variable that is not specified in the NOMINAL= option is binned according to the NBINS= specification.

**LEAFSIZE=*m***
specifies the minimal number of observations on each node. When the number of observations on a tree node is less than $m$, the node is changed to a leaf during the building of the decision tree.

**Interaction** Specifying the LEAFSIZE option affects the pruning of the tree.

**MAXBRANCH=*n***
specifies the maximum number of children (branches) to allow for each level of the tree.

**Default** 2

**MAXLEVEL=*n***
specifies the maximum number of the tree level.

**Default** 6

**MULTVAR**
specifies to allow a variable to appear multiple times when traversing the tree from top to bottom.

**NBINS=*k***
specifies the number of bins to use in the calculation of the decision tree. The number of bins affects the accuracy of the tree. The accuracy increases as values of *k* increase. However, computing time and memory consumption also increase as values of *k* increase.

**Default** 2

**NBINSTARGET=*k***
specifies the number of bins to use for a numeric target variable. The number of bins affects the accuracy of the tree. The accuracy increases as values of *k* increase. However, computing time and memory consumption also increase as values of *k* increase. When *k* is greater than zero, the numeric target variable is binned into equally sized bins first and then the bins are used to perform the classification.

**Default** 0

**NOMINAL=*variable-name***
**NOMINAL=(*variable-list*)**
specifies the numeric variables to use as nominal variables. Binning is not applied to the specified variables. The target variable is always treated as a nominal variable and does not need to be listed.

**NOMISSOBS**
specifies to ignore observations that have missing values in the analysis variables when building a decision tree. When scoring a data set, any observations with missing values in the analysis variables for the decision tree are ignored when this option is specified.

When this option is not specified, the DECISIONTREE statement builds a tree by applying the following policy for missing values:

- for an interval variable, the smallest machine value is assigned
- for a nominal variable, missing values are represented by a separate level

**NOPREPARSE**
prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NOPRUNEOBS**
specifies not to prune any observations when building a decision tree.

**NOSCORE**
suppresses the generation of the scoring temporary table when the TEMPTABLE option is specified. In this case, the server generates only one temporary table and the table contains the decision tree.

**PRUNE**
requests to prune the tree according to the C4.5 algorithm. Pruning can increase the error of misclassification. You can control the aggressiveness of pruning with the CFLEV= option. Smaller values for the CFLEV= option result in more aggressive pruning.

**PRUNEGROW**
specifies to enable C4.5 pruning when building a classification decision tree. The tree could have large a misclassification rate but the building process is performed quickly.

**REG**
specifies to build a regression tree. Minimal cost-complexity pruning is applied to prune the tree.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SCOREDATA=*table-name***
specifies the in-memory table that contains the scoring data. The table must exist in-memory on the server. The DECISIONTREE statement in the IMSTAT procedure does not transfer a local data set to the server.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**STAT**
specifies to generate two additional tables that contain statistical information about the variables that are used in the decision tree. One table contains the variable

importance information, which is determined by the total Gini reduction. The second table contains the variable splitting information for each node in the decision tree.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
specifies to store the results in a temporary table. The type of information that is stored depends on whether you are building a decision tree or scoring a table with a decision tree.

When you are building a decision tree, the generated decision tree is stored in the server and the input table is automatically scored using this tree. The scoring details are saved in a temporary table. The _TEMPTREE_ macro variable stores the name of the temporary table for the tree. The _TEMPSCORE_ macro variable stores the name of the temporary table that has the scoring results of traversing the decision tree. You can suppress the generation of the scoring temporary table (_TEMPSCORE_) during the tree building phase by specifying the NOSCORE option.

When you are scoring a table using a decision tree, the TEMPTABLE option requests to store the scoring details in a temporary table in the server. The IMSTAT procedure displays the name of the table and stores it in the _TEMPSCORE_ macro variable. Be aware that the DETAIL option can generate a very large amount of scoring results when the in-memory table that is specified in the SCOREDATA= option is large. Observations from the scored data set can be transferred to the temporary table using the VARS= option.

**TIMEOUT=*s***
specifies the maximum number of seconds that the server should run the statement. If the time-out is reached, the server terminates the request and generates an error and error message. By default, there is no time-out.<

**TREEDATA=*libref.member-name***
**TREETAB=*saved-table***
**TREELASRTAB=*table-name***
specifies the saved table that contains the generated tree. In order to score a (validation) data set against the generated tree, you need the validation data and a representation of the tree. Specify these options as follows:

- The TREEDATA= option is used to specify the name of a SAS data set that stores the generated tree. The data set is local to the SAS client.
- The TREETAB= option is used to specify a table on the SAS client that stores the generated tree.
- The TREELASRTAB= option is used to specify a valid decision tree that is stored in an in-memory table.

The data set with the observations to score is specified in the SCOREDATA= option

**Alias** SCORETAB=

**VARS=*variable-name***
**VARS=(*variable-name1 <variable-name2, …>* )**
specifies the variables to transfer from the input table to the temporary table in the server that contains the results of scoring a decision tree. This option has no effect unless you specify the TEMPTABLE option and you score a decision tree.

## Details

### *ODS Table Names*

The DECISIONTREE statement generates ODS tables that are specified in the following table.

| ODS Table Name | Description | Option |
|---|---|---|
| DTREE | Classification decision tree | Default |
| DTreeVarImpInfo | Variable importance in a decision tree | STAT |
| DTreeVarStatInfo | Variable information for decision tree | STAT |
| DTREESCORE | Classification decision tree scoring summary | SCOREDATA= |
| GeneratedCode | Generated SAS code from modeling task | CODE |
| TempTable | Information about a temporary table | TEMPTABLE |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# DISTINCT Statement

The DISTINCT statement calculates the count of unique raw values of variables. You can specify the variables to calculate in the variable list. If no list is specified, the count of unique raw values is calculated for all variables.

## Syntax

**DISTINCT** <*variable-list*> <*/ options*>;

### DISTINCT Statement Options

**FORMATS=("*format-specification*",…)**
specifies the formats for the GROUPBY= variables. If you do not specify the FORMAT= option, or if you do not specify the GROUPBY= option, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
   DISTINCT x / groupby=(a b) formats=("8.3", "$10");
quit;
```

**GROUPBY=(*variable-list*)**
specifies a list of variable names, or a single variable name, to use as GROUPBY variables in the order of the grouping hierarchy. If you do not specify any GROUPBY variable names, then the calculation is performed across the entire table —possibly subject to a WHERE clause.

**GROUPBYLIMIT=*n***
specifies the maximum number of levels in a GROUPBY set. When the software determines that there are at least *n* levels in the GROUPBY set, it abandons the action, returns a message, and does not produce a result set. You can specify the GROUPBYLIMIT= option if you want to avoid creating excessively large result sets in GROUPBY operations.

**GROUPFILTER=(*filter-options*)**
specifies a section of the group-by hierarchy to be included in the computation. With this option, you can request that the server performs the analysis for only a subset of all possible groupings. The subset is determined by applying the group filter to a temporary table that you generate with the GROUPBY statement.

You can specify the following suboptions in the GROUPFILTER option:

**DESCENDING**
specifies the top section or the bottom section of the groupings to be collected. If the DESCENDING option is specified, the top LIMIT=*n* (where $n > 0$) groupings are collected. Otherwise, the bottom LIMIT=*n* groupings are collected.

**Alias** DESC

**LIMIT=*n***
specifies the maximum number of distinct groupings to be collected, where integer $n >= 0$. If *n* is zero, then all distinct groupings (up to $2^{31}-1$) that satisfy the boundary constraints, such as LOWERSCORE=*f*, are collected.

**CAUTION** **High Cardinality Data Sets** Setting *n* to zero with high-cardinality data sets can significantly delay the response of the server.

**SCOREGT=*f***
specifies the exclusive lower bound for the numeric scores of the distinct groupings to collect.

**Alias** SGT=

**SCORELT=*f***
specifies the exclusive upper bound for the numeric scores of the distinct groupings to collect.

**Alias** SLT=

**VALUEGT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive lower bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VGT=

**VALUELT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive upper bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VLT=

**TABLE=*table-with-groupby-results***
specifies the in-memory table from which to load the group-by hierarchy. If the TABLE= option is not specified, then all other GROUPFILTER= options are ignored.

The following program request all the groupings of State, City, and then Trade_In_Model in the Cars_Program_All table. The groupings are ordered by the maximum value of New_Vehicle_Msrp for each grouping:

```
proc imstat;
    table example.cars_program_all;
    groupby state city trade_in_model / temptable
                weight=new_vehicle_msrp
                agg   =(max)
                order =weight;
run;
```

The TEMPTABLE option in the GROUPBY statement directs the server to save all the groupings in a temporary in-memory table. The following DISTINCT statement requests the count of the distinct unformatted values of Sales_Type for each of the selected groupings of State, City, and Trade_In_Model.

```
    table example.cars_program_all;
    distinct sales_type / groupfilter=(
                table  =mylasr.&_TEMPLAST_
                scoregt=40000
                valuelt=("FL","Ft Myers","")
                limit  =20
                descending);
run;
```

This example considers only groupings that have maximum values of the New_Vehicle_Msrp above 40,000 and with formatted values that are less than State="FL" and City="Ft Myers." The empty quotation marks result in no restriction on Trade_In_Model values. These groupings are ordered according to the maximum values of New_Vehicle_Msrp. Because of the DESCENDING option, this example collects the 20 top groupings within the specified group-by range for the DISTINCT analysis.

**Interaction** If you specify the GROUPFILTER= option, then the GROUPBY= and FORMATS= options have no effect.

**MAXNVALS=*n***
specifies the maximum size that trees are allowed to consume during the calculation of distinct counts. If you execute a DISTINCT statement with a GROUPBY= or

PARTITION= option, then the MAXNVALS limit applies within the groups or partitions.

**Default** 6

**NOMISSING**
specifies that you do not want to include missing values in the determination of the distinct count.

**Alias** NOMISS

**NOPREPARSE**
prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NOTEMPPART**
specifies that the temporary table generated by the TEMPTABLE option is not partitioned by the GROUPBY= variables. When you request a temporary table with the DISTINCT statement, by default, the server partitions the table and the size of a partition is equal to the number of analysis variables in the *variable-list* of the DISTINCT statement. When the number of groups is large, this can result in many small partitions, and requires extra memory resources to store the partition information for the temporary table. By specifying this option, the temporary table is organized similarly to the default table, but is not a partitioned table.

**Alias** NOTP

**ORDERBY=(*variable-list*)**
specifies one or more variables by which to order the result set. The variables specified in *variable-list* are either one or more of the GROUPBY= variables or one or more of the analysis variables. If you specify an incorrect variable, the server returns an error and no result set. Separate multiple variables with a space.

When there are ties given the ordering of the ORDERBY= variable values, the server sorts the tied items by the GROUPBY= or PARTITION= variable values (unless neither the GROUPBY option or the PARTITION option are specified). If the ORDERBY= option is not specified, the result set is ordered by the formatted values of GROUPBY= variables.

The following DISTINCT statement requests the number of the distinct raw values of Invoice, grouped by Type in the Cars table:

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

```
proc imstat data=example.CARS;
    distinct Invoice / GROUPBY=Type ORDERBY=Invoice MAXNVALS=32;
run;
```

The result set is ordered by Invoice values. The statement produces the following output:

| Number of Distinct Values in Table EXAMPLE.CARS | | | | |
|---|---|---|---|---|
| Type | Column | Number of Distinct Values | Number of Missing Values | Truncated |
| Hybrid | Invoice | 3 | 0 | No |
| Truck | Invoice | 24 | 0 | No |
| Wagon | Invoice | 30 | 0 | No |
| SUV | Invoice | 32 | 0 | Yes |
| Sedan | Invoice | 32 | 0 | Yes |
| Sports | Invoice | 32 | 0 | Yes |

There are three items tied with the same distinct Invoice raw value, 32. These items are then ordered by the formatted values of Type.

**ORDERBYDESC**

specifies the sort order for the result set. The default is ascending order. Specifying this option arranges the results in descending order. This option has no effect unless you specify the ORDERBY= option.

**PARTITION <=*partition-key*>**

When you specify this option and the table is partitioned, the results are calculated separately for each value of the partition key. In other words, the partition variables function as automatic GROUPBY variables. This mode of executing calculations by partition is more efficient than using the GROUPBY= option. With a partitioned table, the server takes advantage of knowing that observations for a partition cannot be located on more than one worker node.

If you do not specify a *partition-key*, the analysis is performed for all partitions. If you do specify a *partition-key*, the analysis is carried out for the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid partition key values for a table.

You can specify a *partition-key* in two ways. You can supply a single quoted string that is passed to the server, or you can specify the elements of a composite key separated by commas. For example, if you partition a table by variables GENDER and AGE, with formats $1 and BEST12, respectively, then the composite partition key has a length of 13. You can specify the partition for the 11-year-old females as follows:

```
statement / partition="F          11"; /* passed directly to the server */
statement / partition="F","11";        /* composed by the procedure */
```

If you choose the second format, the procedure composes a key based on formatting information from the server.

**Alias** PART=

**RESULTLIMIT=*k***

specifies that the number of items that are returned to the client is limited to *k* times the number of analysis variables if you also specify the GROUPBY= or ORDERBY= option.

The following DISTINCT statement requests the numbers of the distinct raw values of Invoice grouped by Type in the Cars table:

The following DISTINCT statement requests the numbers of the distinct raw values of Invoice and Cylinders grouped by Type, and limits the results to (2 variables × 4) = 8 rows.

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

proc imstat data=example.CARS;
    distinct Invoice Cylinder / resultlimit=4;
run;
```

The statement produces the following output:

| Number of Distinct Values in Table EXAMPLE.CARS | | | |
|---|---|---|---|
| Type | Column | Number of Distinct Values | Number of Missing Values |
| Hybrid | Invoice | 3 | 0 |
| Hybrid | Cylinders | 2 | 0 |
| SUV | Invoice | 60 | 0 |
| SUV | Cylinders | 4 | 0 |
| Sedan | Invoice | 260 | 0 |
| Sedan | Cylinders | 5 | 0 |
| Sports | Invoice | 49 | 0 |
| Sports | Cylinders | 6 | 2 |

Four groups of two rows each are displayed. Without the RESULTLIMIT= option, six groups of two rows are displayed.

**SAVE=*table-name***

saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**

requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**SORTAGG=*aggregation-method***
specifies the aggregator for which the ordering of the result set is based, if the ORDERBY= option is specified.

The available aggregation methods are as follows:

N number of observations

NMISS number of missing observations

**Interaction** You must specify the ORDERBY= option to use this option.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the &_TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**Interaction** The TEMPTABLE option requires a group-by analysis or a partitioned analysis with this statement.

**VARFORMATS=("*format-specification*",…)**
specifies the formats for the analysis variables. If you do not specify this option, the distinct count is based on the number of distinct unformatted values of a variable. Note that the FORMATS= option controls the formatting of the GROUPBY= variables and the VARFORMATS= option controls the formatting of the analysis variables. It is possible to specify a different format for a variable if it appears as a GROUPBY variable and as an analysis variable.

You can specify a combination of formatted and unformatted value counts by submitting an empty string as the format for variables that you do not wish to format.

For example, in the following code the distinct count of variable Invoice is based on the formatted values according to the user-defined format PRICE20. The distinct count of variable Msrp is based on its unformatted values.

**Example**
```
proc imstat data=example.cars;
   distinct msrp invoice / varformats=("", "PRICE20");
run;
```

## Details

### *ODS Table Names*

The DISTINCT statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| DistinctCount | Distinct counts for one or more columns | Default |
| TempTable | Information about a temporary table | TEMPTABLE |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# FORECAST Statement

The FORECAST statement computes predicted values, measures of precision, and confidence limits for observed and future (forecast) values of a time series. The models generated by the FORECAST statement belong to the exponential smoothing method (ESM) and autoregressive integrated moving average (ARIMA) families.

**Examples:** "Example 12: Forecasting and Automatic Modeling" on page 272
"Example 13: Forecasting with Goal Seeking" on page 275

## Syntax

**FORECAST** *timestamp-variable* <*/ options*>;

**FORECAST** DATA=*libref.member-name timestamp-variable* <*/ options*>;

### *Required Arguments*

***timestamp-variable***
specifies the name of the SAS datetime variable to use.

**DATA=*libref.member-name***
specifies the libref and table name of a SAS data set when you specify the DATA= option. The data set must contain the timestamp variable and one or more of the analysis variables. The procedure then sends these values to the server to request the forecast calculation. With this option, there is no aggregation, as the values read from the data set are assumed to constitute the series of interest.

You can produce forecasts for multiple variables when you specify DATA=, but you cannot specify an aggregation method for the variables or specify the TAIL= and HEAD= options in the FORECAST statement.

When you specify the DATA= option, and a data set is sent to the server, you can also request a goal seeking analysis. The data set then must contain variables that identify the goal variable, the control variable, possibly bounds for the control variables, and a weight variable.

## *FORECAST Statement Options*

**AGGREGATE=(*list-of-aggregators*)**
specifies the aggregate method on which the ordering of the result set is based. The following methods are valid:

The available aggregation methods are as follows:

| | |
|---|---|
| CSS | corrected sum of squares |
| CV | coefficient of variation |
| MAX | maximum value |
| MEAN | arithmetic mean |
| MIN | minimum value |
| N | number of observations |
| PROBT | *p*-value for the *t*-statistic |
| STD | standard deviation |
| STDERR | standard error |
| SUM | sum of the nonmissing values |
| TSTAT | *t*-statistic for the null hypothesis that the mean equals zero |
| USS | uncorrected sum of squares |
| VAR | sample variance |

Each analysis variable can be associated with a different aggregate method. For example, the following statement forecasts the sum of expenses and the mean of revenue:

```
forecast ts / vars     =(expenses revenue)
                     aggregate=(sum mean);
```

The default aggregation is the mean of the analysis variables within unique values of the timestamp variable.

**Interaction** This option has no effect if you specify a data set with the DATA= option.

**CONTROLVARS=(*variable1-name <variable2-name…>* )**
specifies the controllable variables used in goal seeking. Control variables act like independent variables in the automatic modeling step. Only control variables are passed to the optimization step in goal seeking. The optimization determines the best values for the control variables that meet the values of the GOAL= variable.

Variables listed as control variables cannot appear in the list of independent variables.

When you also specify INDEP= variables, the goal-seeking analysis gives precedence to controllable variables over non-controllable (specified with the INDEP= option) for its variable selection. Relative precedence of controllable variables is maintained, as is relative precedence of non-controllable variables.

**Alias** CONTROL=

**FORMATS=("*format-specification*")**
specifies the format for the time stamp variable. The observations are grouped by the formatted values of the time stamp variable. If multiple values map to the same formatted value, the smallest is kept as the representative value. These values form the time stamps for the forecast.

If you do not specify the FORMATS= option, the default format is applied for the time stamp variable.

**Interaction** This option has no effect if you specify a data set with the DATA= option.

**FRAME=LEAD | HORIZON**
**FRAME=TAIL | HISTORY**
**FRAME=BOTH**
specifies how to compose the main result table. The default is FRAME=BOTH and the result set contains the observed series (the history) as well as the forecast (the horizon). If you specify FRAME=LEAD (or FRAME=HORIZON), then only the future values are returned. You can control the length of the horizon with the LEAD= option.

If you specify FRAME=TAIL (or FRAME=HISTORY), then only the results for the historic values are returned. The returned values are the aggregated values, their predicted values, residuals, prediction standard errors, and confidence limits. You can control the number of the historical records with the TAIL= option.

**Alias** WINDOW=

**Default** BOTH

**GOALVAR=*variable-name***
specifies the variable in the active table that contains the goal (the desired forecast) for goal seeking.

**Alias** GOAL=

**Interaction** You must use the DATA= option to perform forecasting with goal seeking.

**Example** "Example 13: Forecasting with Goal Seeking" on page 275

**HOST="*host-name*"**
specifies the machine to which you want to connect to produce the forecast when you specify the DATA= option in the FORECAST statement. If you do not specify the host information, it is determined from the active table.

**INDEP=*variable-name***
**INDEP=(*variable-list*)**
specifies the independent variables used in automatic modeling. When you specify one or more independent variables, the server performs model selection automatically and determines the best-fitting time series model and the important independent variables. If any variables are selected, a table is generated to show the

actual and predicted values for each variable. Specify the INFO option to view the Forecast Information table that displays the selected time series model.

Variables that are listed as independent variables cannot appear in the list of control variables.

**Alias** INDEPVARS=

**INFO**

specifies to display a forecast information table for each analysis variable. Each table provides informational details about the forecast. For example, you can learn from this table what time units were applied and which method was used to compute the forecast.

The server performs automatic model selection. The available methods and the associated ARIMA models are as follows:

| | |
|---|---|
| Damped-trend exponential smoothing | ARIMA(1, 1, 2) |
| Linear exponential smoothing | ARIMA(0, 2, 2) |
| Seasonal exponential smoothing | ARIMA( 0, 1, $p$ + 1)(0, 1, 0)$_p$ |
| Simple exponential smoothing | ARIMA(0, 1, 1) |
| Winters method (additive) | ARIMA( 0, 1, $p$ + 1)(0, 1, 0)$_p$ |
| Winters method (multiplicative) | There is no ARIMA equivalent. |

**LEAD=*n***

specifies the forecast horizon (in number of time intervals).

**Default** 12

**Interaction** This option has no effect if you specify a data set with the DATA= option.

**LOWERBOUNDS <=> (*boundary-specification1* <, *boundary-specification2*…>**

specifies lower boundary variables for the control variables. A *boundary-specification* is specified with the following form:

```
control-variable = boundary-variable
```

For example, in the following FORECAST statement the variable Pricelb in data set Merged2 contains the lower boundary values for the control variable Price, and the variable Priceub contains the upper boundary values for the control variable Price.

**Alias** LOWER=

**Example**

```
forecast data=merged2 date / dep    =sale
                             control=price
                             lower(price=pricelb)
                             upper(price=priceub)
                             goal   =gsale
                             lead   =12;
```

**NOPREPARSE**

specifies to prevent the procedure from pre-parsing and pre-generating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as-is" and then the server attempts to generate code from it. If the server detects

problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to pre-parse and pre-generate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

**Alias** NOPREP

**Interaction** This option has no effect if you specify a data set with the DATA= option.

**PORT=*number***
specifies to use the server that is listening on that port to produce the forecast when you specify the DATA= option in the FORECAST statement. You can use this option with the HOST= option to use a specific server. If you do not specify a PORT= value, the behavior of the FORECAST statement depends on whether a table is active. If there is no active table, then the IMSTAT procedure tries to connect to the server using the LASRPORT macro variable. If a table is active, then a connection is made to the server that has the active table.

**STAMPLIMIT=*m***
specifies a hard limit for the number of time stamps. If that number reaches *m*, then execution stops and the server generates an error message. This option is useful to protect against the generation of very large result sets. You can also limit the number of time stamps used in the forecast with the TAIL= option. Using the TAIL= option also reduces the size of the result set.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**TAIL=*k***
specifies the number of most recent time intervals on which to base the estimation of the predicted and forecasted values. The TAIL= option enables you to restrict the length of the series that is used in the forecast.

For example, if the aggregation results in 500 unique values of the time stamp, then specifying TAIL=30 uses only the thirty most recent values in the estimation procedure. If you do not specify the TAIL= option, then all the aggregated time stamps are used in the estimation procedure. This option can also limit the size of the result set since at most *k* observations are used in the computation of the forecast.

**Interaction** This option has no effect if you specify a data set with the DATA= option.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**UPPERBOUNDS <=> (*boundary-specification1* <, *boundary-specification2*…>**
specifies upper boundary variables for the control variables. A *boundary-specification* is specified with the following form:

```
control-variable = boundary-variable
```

asdfThe boundary specification is identical to the LOWERBOUNDS= on page 129 option.

**Alias** UPPER=

**VARS=*variable-name***
**VARS=(*variable-list*)**
specifies one or more numeric analysis variables to forecast. If you do not specify the VARS= option, a forecast is produced for all numeric variables in the active table. If you specify a data set with the DATA= option, you must specify the analysis variables in the VARS= option. If you do not, the server generates an error.

**Alias** DEPVARS=

**WEIGHTVAR=*variable-name***
specifies the optional weight variable for goal-seeking analysis.

**Alias** WEIGHT=

## Details

### *Accessing Data with the FORECAST Statement*

There are two ways to use the FORECAST statement. You can use the active table or you can specify a data set with the DATA= option. The following paragraphs provide more information about these choices. In either case, the table does not need to be sorted by values of the time stamp variable.

When you use the active table, the server forms a time series by aggregating the values of the analysis variables according to the unique (formatted) values of a numeric time stamp variable. The time stamp variable must be a SAS datetime type. The aggregate series (one for each analysis variable) are then used to compute predicted values of the series. The predicted values can cover the observed time interval or can apply to future observations. Measures of precision (standard errors of prediction and confidence limits) are also available. You can produce forecasts for multiple variables and you can vary the method for aggregating values on a variable-by-variable basis.

Alternatively, you can specify a SAS data set with the DATA= option. The data set must have a time stamp variable and one or more of the analysis variables. In this case, the data are sent to the server for the forecast calculation. In this case, there is no aggregation because the values read from the data set are assumed to constitute the series of interest. You can produce forecasts for multiple variables when you use the DATA= option, but you cannot specify the aggregation technique for the variables or specify the TAIL= and HEAD= options in the FORECAST statement.

### ODS Table Names

The FORECAST statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| Forecast | Results from a series forecast | Default |
| ForecastInfo | Information about a series forecast | INFO |
| ForecastSelectedVars | Selected independent variables from a series forecast | INDEP= |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## FREQUENCY Statement

The FREQUENCY statement is used to calculate a frequency distribution for one or more variables.

### Syntax

**FREQUENCY** *variable-list* </ *options*>;

#### Required Argument

***variable-list***
specifies the numeric and character variables to use for calculating the frequency distribution. The distribution is calculated for the unique formatted values of the variables.

#### FREQUENCY Statement Options

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**FORMATS=("*format-specification*", …)**
specifies the format to apply to each variable. Specify the list as a comma-separated list and enclose each format specification in quotation marks. If you do not specify a format, then the default format for the variable is used.

**MERGEBINS=*b***
specifies the number of bins to create when a numeric GROUPBY variable exceeds the MERGELIMIT=*n* specification. If you specify a MERGELIMIT, but do not specify a value for the MERGEBINS= option, the server automatically calculates the number of bins.

**MERGELIMIT=*n***
specifies that when the number of unique values in a numeric GROUPBY variable exceeds *n*, the variable is automatically binned and the GROUPBY structure is determined based on the binned values of the variable, rather than the unique formatted values.

For example, if you specify MERGELIMIT=500, any numeric GROUPBY variable with more than 500 unique formatted values is binned. Instead of returning results for more than 500 groups, the results are returned for the bins. You can specify the number of bins with the MERGEBINS= option.

**NOEMPTY**
specifies that empty cells are not returned to the SAS session (only full cells are returned). When this option is specified, the server eliminates all levels with zero frequency from the result set.

**Alias** FULLCELL

**NOMISS**
specifies that missing values are excluded in the calculation of formatted values. By default, levels with missing values are included.

**Alias** NOMISSING

**RAWORDER**
specifies that the ordering of the GROUP BY value is based on the raw values of the variables instead of the formatted values.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### *ODS Table Names*

The FREQUENCY statement generates the following ODS table for each analysis variable.

| ODS Table Name | Description | Option |
|---|---|---|
| Frequency | Frequency information for one column | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# GENMODEL Statement

The GENMODEL statement is used to fit statistical models from the class of generalized linear models and some related models.

## Syntax

**GENMODEL** *dependent-variable* <(*class-variables*)> = *model-effects* </ *options*>;

### *Required Arguments*

***dependent-variable***
specifies the variable to model. This variable is also referred to as the response variable.

***model-effects***
specifies a list of variables to use for modeling the dependent variable.

### *Optional Argument*

***class-variables***
specifies a list of variables to use as classification variables. The variables in this list take the place of the CLASS statement in traditional SAS procedures.

## GENMODEL Statement Options

**ALLIDVARS**

requests that all variables in the input table are treated as ID variables when a scoring table is produced. In other words, if this option is specified, all variables from the input table, including computed columns, are transferred to the scoring table. This option has no effect unless you specify the SCORE option.

**ALPHA=*number***

specifies a number between 0 and 1 from which to determine the confidence level for approximate confidence intervals of the parameter estimates. The default is α = 0.05, which leads to 100 x (1- α)% = 95% confidence limits for the parameter estimates.

**Default** 0.05

**CI**

specifies to add confidence intervals to the table of parameter estimates. The confidence level is 100*(1-α)% where α is determined by the ALPHA= option. The default value is α = 0.05. This value is equivalent to a 95% confidence limit.

**Default** 0.05

**CLASSFORMATS=("*format-name1*" <, "*format-name2*" …> )**

specifies the formats for the classification variables in the model. If you do not specify the CLASSFORMATS= option, the default format is applied for the classification variable. That default format was determined when the table was originally loaded into the server. In the following example, the CLASSFORMAT= values apply to variables x1 and x2.

**Alias** CLASSFMT=

**Example**

```
genmodel y (x1 x2) = x3-x7 / classformats=("YN.", "F8.");
```

**CODE <(*code-generation-options*)>**

requests that the server produce SAS scoring code based on the actions that it performed during the analysis. The server generates DATA step code. By default, the code is replayed as an ODS table by the procedure as part of the output of the statement. More frequently, you might want to write the scoring code to an external file by specifying options.

The scoring code computes the predicted value of the response variable on the data scale (the inverse link scale) and prefixes the name with "P_". For example, if the response variable is **Y**, the generated code stores the predicted value as **P_Y**. The name of the variable is truncated to fit within the SAS name length requirements.

**COMMENT**

specifies to add comments to the code in addition to the header block. The header block is added by default.

**FILENAME='*path*'**

specifies the name of the external file to which the scoring code is written. This suboption applies only to the scoring code itself. If you request that the server generate IMSTAT programming statements with the IMSTAT suboption, then these statements are saved as an ODS table.

**Alias** FILE=

**FORMATWIDTH=*k***
specifies the width to use in formatting derived numbers such as parameter estimates in the scoring code. The server applies the BEST format, and the default format for code generation is BEST20.

| | |
|---|---|
| **Alias** | FMTW= |
| **Range** | 4 to 32 |

**IMSTAT**
specifies to generate IMSTAT programming statements that reproduce the analysis in addition to the scoring code. For example, this option is helpful when you perform variable selection and you want to capture the modeling code that reflects only the selected variables.

**IMSTATONLY**
specifies to generate the IMSTAT programming statements only. No scoring code is produced.

**LABELID=*id***
specifies a group identifier for group processing. The identifier is an integer and is used to create array names and statement labels in the generated code.

**LINESIZE=*n***
specifies the line size for the generated code.

| | |
|---|---|
| **Alias** | LS= |
| **Default** | 72 |
| **Range** | 64 to 256 |

**NOTRIM**
requests that the comparison of the formatted values for class variables and group-by variables is based on the full format width with padding. By default, the leading and trailing blanks are removed from the formatted values.

**REPLACE**
specifies to overwrite the external file with the new contents if the file already exists. This option has no effect unless you specify the FILENAME= option.

**DIST=*distribution***
specifies the distribution of the response variable. See the following list for the available values:

- BETA
- BINARY | BERNOULLI
- EXPONENTIAL | EXPO
- GAMMA | GAM
- GENPOISSON
- GEOMETRIC | GEOM
- INVGAUSS | IGAUSSIAN | IG
- NEGBINOMIAL | NEGBIN | NB
- NORMAL | GAUSSIAN | GAUSS
- POISSON | POI

- T | STUDENT
- WEIBULL

**EXCLUDE=(*list-of-ODS-tables*)**
specifies the result tables that you want to exclude from being generated on the server and from being sent to the client. The GENMODEL statement can generate the following tables:

| Table Name | Table Alias | Description | Condition |
|---|---|---|---|
| ModelInfo | | Information about the model—constant across groups or partitions. | This table is shown by default. |
| ClassLevels | Class | Information about the classification variables, such as the number of levels and their values. | This table is shown when classification variables are present in the model. |
| ConvergenceStatus | Convergence | Convergence status of optimization | This table is shown by default. |
| Dimensions | Dim | Model dimensions | This table is shown by default. |
| FitStatistics | Fit | Fit statistics customary for generalized linear models | This table is shown when it is requested with the SELECT= option. |
| OptIterHistory | IterHist | Iteration history | This table is shown when the ITDETAILS option is used or when the table is requested with the SELECT= option. |
| ParameterEstimates | ParmEstimates<br>Pest | The solutions for the linear model coefficients | This table is shown by default. |
| Tests3 | | Type III tests of model effects | This table is shown when the distribution is in the exponential family, the effects contain classification variables, and the NOSTDERR option is not specified. |

Whether a table is shown by default or not, you can request any table with the SELECT= option in the GENMODEL statement. The Condition column in the table identifies when a table is produced by default.

**FORMATS=("*format-specification*" <,…> )**
specifies the formats for the GROUPBY variables. If you do not specify the FORMATS= option, or if you omit the entry for a GROUPBY variable, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
   statement / groupby=(a b) formats=("8.3", "$10");
quit;
```

**FCONV=*r***
specifies a relative function convergence criterion. For all techniques except NMSIMP, termination requires a small relative change of the function value in successive iterations. Suppose that $\Psi$ is the $p \times 1$ vector of parameter estimates in the optimization, and the objective function at the $k$th iteration is denoted as $f(\Psi)^k$. Then, the FCONV criterion is met if

$$\frac{\left(f\left(\Psi^{(k)}\right) - f\left(\Psi^{(k-1)}\right)\right)}{f\left(\Psi^{(k-1)}\right)} \leq r$$

**Default** $r=10^{-\text{FDIGITS}}$ where FDIGITS is $-\log_{10}(e)$ and e is the machine precision.

**FREQ=*variable-name***
specifies the numeric variable that provides frequencies for the analysis. For example, if the FREQ= variable has the value 5, then it implies that the record represents five such observations with identical values for the modeling variables. If you specify a FREQ= variable, then only the observations with a value that is not missing and greater than zero for the variable are used in the analysis.

**GCONV=*r***
specifies a relative gradient convergence criterion. For all optimization techniques except CONGRA and NMSIMP, termination requires that the normalized predicted function reduction is small. The default value is $r$ = 1e-8. Suppose that $\Psi$ is the $p \times 1$ vector of parameter estimates in the optimization with $i$th element $\Psi_i$. The objective function, its $p \times 1$ gradient vector, and its $p \times p$ Hessian matrix are denoted, $f(\Psi)$, $g(\Psi)$, and $H(\Psi)$, respectively. Then, if superscripts denote the iteration count, the normalized predicted function reduction at iteration $k$ is

$$\frac{g\left(\Psi^{(k)}\right)' H\left(\Psi^{(k)}\right)^{-1} g\left(\Psi^{(k)}\right)}{f\left(\Psi^{(k)}\right)}$$

The GCONV convergence criterion is assumed to be met if that value is less than or equal to $r$.

Note that it is possible that the relative gradient reduction is small, even if one or more gradients is still substantial in absolute value. If this situation occurs, you can disable the GCONV criterion by setting $r$=0. If the optimization would have stopped early due to meeting the GCONV criterion, the iterative process usually takes one more step until the gradients are small in absolute value.

**GROUPBY=(*variable-list*)**
specifies a list of variable names, or a single variable name, to use as GROUPBY variables in the order of the grouping hierarchy. If you do not specify any GROUPBY variable names, then the calculation is performed across the entire table —possibly subject to a WHERE clause.

**GROUPFILTER=(*filter-options*)**
specifies a section of the group-by hierarchy to be included in the computation. With this option, you can request that the server performs the analysis for only a subset of all possible groupings. The subset is determined by applying the group filter to a temporary table that you generate with the GROUPBY statement.

You can specify the following suboptions in the GROUPFILTER option:

**DESCENDING**
specifies the top section or the bottom section of the groupings to be collected. If the DESCENDING option is specified, the top LIMIT=*n* (where $n > 0$) groupings are collected. Otherwise, the bottom LIMIT=*n* groupings are collected.

**Alias** DESC

**LIMIT=*n***
specifies the maximum number of distinct groupings to be collected, where integer $n >= 0$. If *n* is zero, then all distinct groupings (up to $2^{31}-1$) that satisfy the boundary constraints, such as LOWERSCORE=*f*, are collected.

**CAUTION** **High Cardinality Data Sets** Setting *n* to zero with high-cardinality data sets can significantly delay the response of the server.

**SCOREGT=*f***
specifies the exclusive lower bound for the numeric scores of the distinct groupings to collect.

**Alias** SGT=

**SCORELT=*f***
specifies the exclusive upper bound for the numeric scores of the distinct groupings to collect.

**Alias** SLT=

**VALUEGT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive lower bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VGT=

**VALUELT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive upper bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VLT=

**TABLE=*table-with-groupby-results***
specifies the in-memory table from which to load the group-by hierarchy. If the TABLE= option is not specified, then all other GROUPFILTER= options are ignored.

The following program request all the groupings of State, City, and then Trade_In_Model in the Cars_Program_All table. The groupings are ordered by the maximum value of New_Vehicle_Msrp for each grouping:

```
proc imstat;
    table example.cars_program_all;
    groupby state city trade_in_model / temptable
                weight=new_vehicle_msrp
                agg   =(max)
                order =weight;
run;
```

The TEMPTABLE option in the GROUPBY statement directs the server to save all the groupings in a temporary in-memory table. The following DISTINCT statement requests the count of the distinct unformatted values of Sales_Type for each of the selected groupings of State, City, and Trade_In_Model.

```
    table example.cars_program_all;
    distinct sales_type / groupfilter=(
                table  =mylasr.&_TEMPLAST_
                scoregt=40000
                valuelt=("FL","Ft Myers","")
                limit  =20
                descending);
run;
```

This example considers only groupings that have maximum values of the New_Vehicle_Msrp above 40,000 and with formatted values that are less than State="FL" and City="Ft Myers." The empty quotation marks result in no restriction on Trade_In_Model values. These groupings are ordered according to the maximum values of New_Vehicle_Msrp. Because of the DESCENDING option, this example collects the 20 top groupings within the specified group-by range for the DISTINCT analysis.

**Interaction** If you specify the GROUPFILTER= option, then the GROUPBY= and FORMATS= options have no effect.

**IDVARS=(*variable-list*)**
**IDVARS=*variable-name***
specifies the variables from the active table to transfer to the temporary table that is created by scoring the input table. This option has no effect unless the SCORE option is also specified. (See the SCORE option for details about which variables are added to the temporary table by default.) The IDVARS= option should be used to transfer additional columns from the input table to the scoring table.

**Alias** ID=

**Tip** Instead of this option, you can specify the ALLIDVARS option to transfer all variables from the input table to the scoring table.

**ITDETAILS**
requests to add details about the iterative model fitting process (an iteration history) to the ODS output tables.

**Alias** ITDETAIL

**KEYORDER**
requests that the results for a partitioned analysis are displayed in the order of the partition keys. If this option is not specified, then results are displayed by using the

partitions on the first worker node followed by the partitions on the second node, and so on. Without this option, the results are likely to have random ordering of the partitions. The KEYORDER option makes result collection less efficient but produces a natural, predictable order.

**LINK=*function***
specifies the link function to use for the model fitting process. If you do not specify a link function, the server selects the most appropriate function for the distribution of the data. See the following list for the available functions:

- IDENTITY | IDENT
- LOGIT
- PROBIT
- LOG
- LOGLOG
- CLOGLOG | CLL
- RECIP
- POWMINUS2
- POWER(v) | POW(v) | POM(v)

**MAXFUNC=*n***
specifies the maximum number *n* of function calls in the iterative model fitting process. The default value depends on the optimization technique as follows:

| Optimization Technique | Default Number of Function Calls |
|---|---|
| TRUREG, NRRIDG, and NEWRAP | 125 |
| QUANEW and DBLDOG | 500 |
| CONGRA | 1000 |
| NMSIMP | 3000 |

**Alias** MAXFU=

**MAXITER=*i***
specifies the maximum number of iterations in the iterative model fitting process. The default value depends on the optimization technique as follows:

| Optimization Technique | Default Number of Iterations |
|---|---|
| TRUREG, NRRIDG, and NEWRAP | 50 |
| QUANEW and DBLDOG | 200 |
| CONGRA | 400 |
| NMSIMP | 1000 |

**Alias** MAXIT=

**MAXTESTLEV=*n***
specifies the maximum number of levels in an effect for which the server generates Type III tests. The idea behind the MAXTESTLEV= option is that testing effects for significance that have a large number of levels is typically not meaningful. The effects tend to be highly significant anyway, but determining the exact significance level is computationally intensive. The default value is 300 and implies that no test statistics are produced for any effect that has more than 300 levels.

**Default** 300

**NOCLPRINT <=*n*>**
specifies the number of levels for each classification variables to show in the Class Level Information ODS table. If you do not specify the NOCLPRINT option, all unique values are shown in the order of the class variable levelization. If you specify NOCLPRINT=*n*, then the values are shown for those classification variables that have less than *n* levels only. The value for *n* must be at least 1.

If you specify the NOCLPRINT option without specifying a value for *n*, then *n* = 0 is assumed. This enables you to get a listing of the classification variables in the model. This might be useful if you did not identify classification variables explicitly—without listing their (possibly many) levels.

For example, the following Class Level Information table is displayed with NOCLPRINT=4. Because the number of levels for variable Smoking_Status exceeds 4, the values are not displayed.

| Class Level Information for Table HPS.HEART | | |
|---|---|---|
| **Class** | **Levels** | **Values** |
| **Weight_Status** | 3 | Normal Overweight Underweight |
| **Smoking_Status** | 5 | not printed |

**NOINT**
suppresses the inclusion of an intercept in the model. By default, all models contain an intercept term.

**NOPREPARSE**
prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NOSTDERR**
prevents the computation of the covariance matrix and the standard errors of the parameter estimates. When you specify this option, the Type III tests for the model effects are also not available.

**Alias** NOSTD

**OFFSET=*variable-name***
specifies the offset variable for the analysis. An offset variable can be thought of as a regressor variable whose regression coefficient is known to be 1. Offsets are used to shift the linear predictors by a certain amount. For example, an offset can be used to accommodate constants in the underlying model. In generalized linear models, offsets arise frequently when the data represents a value relative to some measure of size. For example, if you model the number of stops ($Y$) for each trip and the trips are of different length ($t$), then you are really interested in the random variable $Y/t$. The generalized linear model becomes as follows:

$$g\left(\frac{\mu}{t}\right) = \eta$$

If you choose the log link function, this can be written as follows:

$$\log \mu - \log t = \eta$$
$$\log \mu = \eta + \log t$$

The value $\log(t)$ is the offset of the model.

**PARTITION <=*partition-key*>**
specifies to fit the model separately for each value of the partition key. In other words, the partition variables function as automatic group-by variables for the request.

If you do not specify a value for *partition-key*, then the analysis is performed for all partitions. If you do specify a value, then the analysis is performed for the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid *partition-key* values for a table.

**Alias** PART=

**SELECT=(*list-of-ODS-tables*)**
specifies the list of ODS tables that you want to display for the analysis. The specified list replaces the default tables that are generated by the server and displayed. See the EXCLUDE= option for the list of default tables and the table names that you can display.

**SHOWSELECTED**
requests that the server perform variable selection for the model. A backward selection method is used, where the significance level for an effect to remain in the model is determined by the SLSTAY= option. This option performs variable selection like the VARSEL option, but in contrast to the latter option, it displays output only for the selected effects.

**Alias** SHOWSEL

**SLSTAY=*α***
specifies the significance level used in determining whether effects should stay in the model during variable selection.

**Default** 0.1

**Range** 0 to 1

**TECHNIQUE=*value***
specifies the optimization technique for the iterative model-fitting process. The valid values are as follows:

- CONGRA (CG): performs a conjugate-gradient optimization
- DBLDOG (DD): performs a version of the double-dogleg optimization
- NMSIMP (NS): performs a Nelder-Mead simplex optimization
- NONE: does not perform any optimization. This value can be useful to perform a grid search without optimization.
- NEWRAP (NRA): performs a (modified) Newton-Raphson optimization that combines a line-search algorithm with ridging
- NRRIDG (NRR): performs a (modified) Newton-Raphson optimization with ridging
- QUANEW (QN): performs a quasi-Newton optimization
- TRUREG (TR): performs a trust-region optimization

The factors that go into choosing a particular optimization technique for a particular problem are complex. Trial and error can be involved.

**Default** NRRIDG

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**Interaction** For information about the interaction between the TEMPTABLE, CODE, and SCORE options, see "Temporary Tables, Generated Code, and Scoring" on page 253.

**VALIDATE=*f***
specifies the proportion *f* in the validation data set.

| | |
|---|---|
| **Alias** | VALPROP= |
| **Range** | 0 to 1 |
| **Interaction** | If you specify both the ROLEVAR= option and the VALIDATE= option, then the ROLEVAR= setting supersedes the VALIDATE= option. |

**VARSELECTION**
specifies that the server perform variable selection for the model. A backward selection method is used, where the significance level for an effect to remain in the model is determined by the SLSTAY= option. In contrast to the SHOWSEL option, all effects are reported in the IMSTAT output.

**Alias** VARSEL

**WEIGHT=*variable-name***
specifies the numeric variable to use as a weighing variable in solving the linear model.

## Details

### *About the GENMODEL Statement*

The GENMODEL statement offers the options that are necessary to formulate a model in the generalized linear model family, and to specify some other, closely related, models. For example, you can fit data to the generalized Poisson distribution for overdispersed Poisson counts. Or, you can use the shifted T distribution for symmetric data that is heavier in the tails than the normal distribution.

### *ODS Table Names*

The ODS tables that can be generated with the GENMODEL statement are described in the EXCLUDE= option on page 137.

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# GLM Statement

The GLM statement is used to fit models that are similar to those handled by the GLM procedure. There are some important differences in syntax and functionality between the GLM procedure and the GLM statement in IMSTAT.

## Syntax

**GLM** *dependent-variable* <(*class-variables*)> = *model-effects* <| *options*>;

### *Required Arguments*

***dependent-variable***
specifies the variable to model. This variable is also referred to as the response variable.

***model-effects***
specifies a list of variables to use for modeling the dependent variable.

## Optional Argument

***class-variables***
specifies a list of variables to use as classification variables. The variables in this list take the place of the CLASS statement in traditional SAS procedures.

## GLM Statement Options

**ALLIDVARS**
requests that all variables in the input table are treated as ID variables when a scoring table is produced. In other words, if this option is specified, all variables from the input table, including computed columns, are transferred to the scoring table. This option has no effect unless you specify the SCORE option.

**ALPHA=*number***
specifies a number between 0 and 1 from which to determine the confidence level for approximate confidence intervals of the parameter estimates. The default is α = 0.05, which leads to 100 x (1- α)% = 95% confidence limits for the parameter estimates.

**Default** 0.05

**CHISQ**
requests that *p*-values in the table of parameter estimates and Type III tests are determined as probabilities under a $x^2$ distribution. This means that instead of two-sided *p*-values based on the *t* distribution, the *p*-values are computed as two-sided probabilities under a standard normal distribution. Similarly, the assumption of *F* distributions with finite denominator degrees of freedom is ignored in lieu of assuming infinite degrees of freedom.

**CI**
specifies to add confidence intervals to the table of parameter estimates. The confidence level is 100*(1-α)% where α is determined by the ALPHA= option. The default value is α = 0.05. This value is equivalent to a 95% confidence limit.

**Default** 0.05

**CLASSFORMATS=("*format-name1*" <, "*format-name2*" …> )**
specifies the formats for the classification variables in the model. If you do not specify the CLASSFORMATS= option, the default format is applied for the classification variable. That default format was determined when the table was originally loaded into the server. In the following example, the CLASSFORMAT= values apply to variables x1 and x2.

**Alias** CLASSFMT=

**Example**
```
glm y (x1 x2) = x3-x7 / classformats=("YN.", "F8.");
```

**CODE <(*code-generation-options*)>**
requests that the server produce SAS scoring code based on the actions that it performed during the analysis. The server generates DATA step code. By default, the code is replayed as an ODS table by the procedure as part of the output of the statement. More frequently, you might want to write the scoring code to an external file by specifying options.

The scoring code computes the predicted value of the response variable on the data scale (the inverse link scale) and prefixes the name with "P_". For example, if the response variable is **Y**, the generated code stores the predicted value as **P_Y**. The name of the variable is truncated to fit within the SAS name length requirements.

**COMMENT**
specifies to add comments to the code in addition to the header block. The header block is added by default.

**FILENAME='*path*'**
specifies the name of the external file to which the scoring code is written. This suboption applies only to the scoring code itself. If you request that the server generate IMSTAT programming statements with the IMSTAT suboption, then these statements are saved as an ODS table.

**Alias** FILE=

**FORMATWIDTH=*k***
specifies the width to use in formatting derived numbers such as parameter estimates in the scoring code. The server applies the BEST format, and the default format for code generation is BEST20.

**Alias** FMTW=

**Range** 4 to 32

**IMSTAT**
specifies to generate IMSTAT programming statements that reproduce the analysis in addition to the scoring code. For example, this option is helpful when you perform variable selection and you want to capture the modeling code that reflects only the selected variables.

**IMSTATONLY**
specifies to generate the IMSTAT programming statements only. No scoring code is produced.

**LABELID=*id***
specifies a group identifier for group processing. The identifier is an integer and is used to create array names and statement labels in the generated code.

**LINESIZE=*n***
specifies the line size for the generated code.

**Alias** LS=

**Default** 72

**Range** 64 to 256

**NOTRIM**
requests that the comparison of the formatted values for class variables and group-by variables is based on the full format width with padding. By default, the leading and trailing blanks are removed from the formatted values.

**REPLACE**
specifies to overwrite the external file with the new contents if the file already exists. This option has no effect unless you specify the FILENAME= option.

**EXCLUDE=(*list-of-ODS-tables*)**
specifies the result tables that you want to exclude from being generated on the server and from being sent to the SAS session. The GLM statement can generate the following tables:

| Table Name | Table Alias | Description | Condition |
|---|---|---|---|
| ModelInfo | | Information about the model—constant across groups or partitions. | This table is shown by default. |
| ClassLevels | Class | Information about the classification variables, such as the number of levels and their values. | This table is shown when classification variables are present in the model. |
| Dimensions | Dim | Model dimensions | This table is shown by default. |
| FitStatistics | Fit | Fit statistics customary for generalized linear models | This table is shown when it is requested with the SELECT= option. |
| OverallAnova | GlobalAnova | Model, source, and error decomposition of variation | This table is shown when classification variables are present in the model. |
| ModelAnova | ANOVA | Variance decomposition with significance tests for all model effects | This table is shown when classification variables are present in the model. |
| ParameterEstimates | ParmEstimates<br>Pest | The solutions for the linear model coefficients | This table is shown when there are no classification variables in the model. |
| Tests3 | | Type III tests of model effects | This table is shown when it is requested with the SELECT= option. |

Whether a table is shown by default or not, you can request any table with the SELECT= option in the GLM statement. The Condition column in the table identifies when a table is produced by default. For example, if the model contains classification variables, the statement shows an OverallAnova table and a ModelAnova table. If there are no classification variables, the statement shows a table of parameter estimates and no ANOVA tables.

**FORMATS=("*format-specification*" <,…> )**
specifies the formats for the GROUPBY variables. If you do not specify the FORMATS= option, or if you omit the entry for a GROUPBY variable, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
   statement / groupby=(a b) formats=("8.3", "$10");
quit;
```

**FREQ=*variable-name***
specifies the numeric variable that provides frequencies for the analysis. For example, if the FREQ= variable has the value 5, then it implies that the record represents five such observations with identical values for the modeling variables. If you specify a FREQ= variable, then only the observations with a value that is not missing and greater than zero for the variable are used in the analysis.

**GROUPBY=(*variable-list*)**
specifies a list of variable names, or a single variable name, to use as GROUPBY variables in the order of the grouping hierarchy. If you do not specify any GROUPBY variable names, then the calculation is performed across the entire table —possibly subject to a WHERE clause.

**GROUPBYMODE= DATA | MODEL | LASR**
specifies the parallelization technique for group-by processing. The default is GROUPBYMODE=MODEL in which threads solve separate models following a lateral reconciliation of cross-product matrices. This mode is appropriate in situations with many groups and relatively small cross-product matrices. Model-parallel processing minimizes passes through the data.

Specify GROUPBYMODE=DATA to form the cross-product matrices in parallel across the data and one group at a time. This data-parallel technique is appropriate in situations with few groups and many observations per group or in applications with large cross-product matrices. Data-parallel processing consumes fewer resources than model-parallel processing but passes through the data more often.

If you specify GROUPBYMODE=LASR, then the server examines the data structure of the groups to select the parallelization mode.

**Default** MODEL

**GROUPFILTER=(*filter-options*)**
specifies a section of the group-by hierarchy to be included in the computation. With this option, you can request that the server performs the analysis for only a subset of all possible groupings. The subset is determined by applying the group filter to a temporary table that you generate with the GROUPBY statement.

You can specify the following suboptions in the GROUPFILTER option:

**DESCENDING**
specifies the top section or the bottom section of the groupings to be collected. If the DESCENDING option is specified, the top LIMIT=*n* (where $n > 0$) groupings are collected. Otherwise, the bottom LIMIT=*n* groupings are collected.

**Alias** DESC

**LIMIT=*n***
specifies the maximum number of distinct groupings to be collected, where integer $n >= 0$. If $n$ is zero, then all distinct groupings (up to $2^{31}-1$) that satisfy the boundary constraints, such as LOWERSCORE=*f*, are collected.

**CAUTION** **High Cardinality Data Sets** Setting *n* to zero with high-cardinality data sets can significantly delay the response of the server.

**SCOREGT=*f***
specifies the exclusive lower bound for the numeric scores of the distinct groupings to collect.

**Alias** SGT=

**SCORELT=*f***
specifies the exclusive upper bound for the numeric scores of the distinct groupings to collect.

**Alias** SLT=

**VALUEGT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive lower bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VGT=

**VALUELT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive upper bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VLT=

**TABLE=*table-with-groupby-results***
specifies the in-memory table from which to load the group-by hierarchy. If the TABLE= option is not specified, then all other GROUPFILTER= options are ignored.

The following program request all the groupings of State, City, and then Trade_In_Model in the Cars_Program_All table. The groupings are ordered by the maximum value of New_Vehicle_Msrp for each grouping:

```
proc imstat;
    table example.cars_program_all;
    groupby state city trade_in_model / temptable
                weight=new_vehicle_msrp
                agg   =(max)
                order =weight;
run;
```

The TEMPTABLE option in the GROUPBY statement directs the server to save all the groupings in a temporary in-memory table. The following DISTINCT statement requests the count of the distinct unformatted values of Sales_Type for each of the selected groupings of State, City, and Trade_In_Model.

```
    table example.cars_program_all;
    distinct sales_type / groupfilter=(
                table  =mylasr.&_TEMPLAST_
                scoregt=40000
                valuelt=("FL","Ft Myers","")
```

```
                    limit  =20
                    descending);
run;
```

This example considers only groupings that have maximum values of the New_Vehicle_Msrp above 40,000 and with formatted values that are less than State="FL" and City="Ft Myers." The empty quotation marks result in no restriction on Trade_In_Model values. These groupings are ordered according to the maximum values of New_Vehicle_Msrp. Because of the DESCENDING option, this example collects the 20 top groupings within the specified group-by range for the DISTINCT analysis.

**Interaction** If you specify the GROUPFILTER= option, then the GROUPBY= and FORMATS= options have no effect.

**IDVARS=(*variable-list*)**
**IDVARS=*variable-name***
specifies the variables from the active table to transfer to the temporary table that is created by scoring the input table. This option has no effect unless the SCORE option is also specified. (See the SCORE option for details about which variables are added to the temporary table by default.) The IDVARS= option should be used to transfer additional columns from the input table to the scoring table.

**Alias** ID=

**Tip** Instead of this option, you can specify the ALLIDVARS option to transfer all variables from the input table to the scoring table.

**INCLUDEMISS**
specifies to treat missing values for classification variables as valid levels. If the INCLUDEMISS option is not specified, observations with missing values in the classification variables are not used in the analysis.

**INFORMATIVE**
requests that missing values are handled by modeling them through extra model effects. These effects consist of dummy variables that take on the value 1 when the value of a continuous model variable that is involved in the effect is missing. Otherwise, they are assigned the value 0. The missing value in the original model effect is replaced with the average value for the effect for the nonmissing values.

For continuous-by-class effects, such as A**x*, where A is a classification variable and *x* is a continuous variable, informative missingness creates multiple dummy columns and substitutes the effect mean of *x* that corresponds to the respective level of A.

Specifying the INFORMATIVE option implies the INCLUDEMISS option. That is, when you choose to model informative missingness, then missing values for classification variables are treated as valid levels. For more information, see "Informative Missingness" on page 159.

**Alias** INFORMMISS

**KEYORDER**
requests that the results for a partitioned analysis are displayed in the order of the partition keys. If this option is not specified, then results are displayed by using the partitions on the first worker node followed by the partitions on the second node, and so on. Without this option, the results are likely to have random ordering of the partitions. The KEYORDER option makes result collection less efficient but produces a natural, predictable order.

**MAXTESTLEV=*n***

specifies the maximum number of levels in an effect for which the server generates Type III tests. The idea behind the MAXTESTLEV= option is that testing effects for significance that have a large number of levels is typically not meaningful. The effects tend to be highly significant anyway, but determining the exact significance level is computationally intensive. The default value is 300 and implies that no test statistics are produced for any effect that has more than 300 levels.

**Default** 300

**NAME=*SAS-name***

specifies the name to use for identifying the model in the server output and in the temporary table of results generated by the TEMPTABLE option. SAS name rules apply. For example, the following statements add the 'Model' entry to the ModelInformation table.

```
proc imstat;
   table hps.iris;
   glm sepalwidth = sepallength / name = FirstModel;
run;
```

| Model Information | |
|---|---|
| Model | FirstModel |
| Data Source | HPS.IRIS |
| Response Variable | SepalWidth |

**NOCLPRINT <=*n*>**

specifies the number of levels for each classification variables to show in the Class Level Information ODS table. If you do not specify the NOCLPRINT option, all unique values are shown in the order of the class variable levelization. If you specify NOCLPRINT=*n*, then the values are shown for those classification variables that have less than *n* levels only. The value for *n* must be at least 1.

If you specify the NOCLPRINT option without specifying a value for *n*, then *n* = 0 is assumed. This enables you to get a listing of the classification variables in the model. This might be useful if you did not identify classification variables explicitly—without listing their (possibly many) levels.

For example, the following Class Level Information table is displayed with NOCLPRINT=4. Because the number of levels for variable Smoking_Status exceeds 4, the values are not displayed.

| Class Level Information for Table HPS.HEART | | |
|---|---|---|
| Class | Levels | Values |
| Weight_Status | 3 | Normal Overweight Underweight |
| Smoking_Status | 5 | not printed |

**NOINT**

suppresses the inclusion of an intercept in the model. By default, the GLM statement adds an intercept as the first model effect to the model. Exclusion of the intercept is useful in certain models to achieve a desired interpretation of the model effects.

For example, the following code sample shows a cell-means model where the coefficients in the β vector estimate the means of Y in the groups associated with levels of A.

```
glm y (A) = A / noint;
```

**NOPREPARSE**

prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**PARTITION <=*partition-key*>**

specifies to fit the model separately for each value of the partition key. In other words, the partition variables function as automatic group-by variables for the request.

If you do not specify a value for *partition-key*, then the analysis is performed for all partitions. If you do specify a value, then the analysis is performed for the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid *partition-key* values for a table.

**Alias** PART=

**ROLEVAR=*variable-name***

specifies a variable in the in-memory table that defines whether an observation belongs to the training set, the validation set, or is to be excluded from the analysis. The role variable can have a numeric or character type, and it can be a temporary computed variable.

If the role variable data type is numeric, the values of *variable-name* are interpreted as follows:

- value = 1: this observation is in the training set
- value = 2: this observation is in the validation set
- any other value: this observation is to be excluded from the analysis

If the role variable data type is character, the values of *variable-name* are interpreted as follows:

- If the first non-blank character is 't' or 'T', then the observation is in the training set.
- If the first non-blank character is 'v' or 'V', then he observation is in the validation set.

- Any other value for the first non-blank character, including an all blank entry, leads to the exclusion of the observation from the analysis.

**Alias** ROLE=

**Interactions** You can divide the data at random into training and validation sets by providing the VALIDATE= and SEED= options.

If you specify both the ROLEVAR= option and the VALIDATE= options, then the ROLEVAR= setting supersedes the VALIDATE= option.

**SCORE <(*score-statistic1score-statistic2…*)>**
requests that the active table be scored after the model is fit and the results be stored in a temporary table. The server automatically adds all model variables to the temporary table with the score results. These results include the response variable, the class variables, all explanatory variables from which effects are formed, and the WEIGHT=, and FREQ= variables.

In addition, if the active table is partitioned or ordered, the partition variables and order-by variables are transferred from the input table to the temporary table. The temporary table is partitioned and ordered in the same way as the active table.

If the analysis uses the GROUPBY= option, the variables in the group-by list are also transferred to the scoring table. If you want to transfer additional variables, you can specify them with the IDVARS= option.

If you do not specify the list of score statistics, default statistics are computed. These statistics are identified with Yes in the Default column in the table below. You can request that the following statistics be computed for each observation:

| Keyword and Aliases | Column Name | Description | Default |
|---|---|---|---|
| PRED, PREDICTED, MEAN | _PRED_ | Predicted value | Yes |
| RESID, RESIDUAL, R | _RESID_ | Raw residual (observed - predicted) | Yes |
| STUDENT | _STUDENT_ | Studentized residual | Yes |
| RSTUDENT | _RSTUDENT_ | Studentized residual with the current observation removed | Yes |
| LEVERAGE, H | _LEVERAGE_ | Leverage value of the observation | Yes |
| STDP | _STDP_ | Standard error of the mean predicted value | No |

| Keyword and Aliases | Column Name | Description | Default |
|---|---|---|---|
| STDR | _STDR_ | Standard error of the residual | No |
| STDI | _STDI_ | Standard error of the (individual) predicted value | No |
| LCLM, LOWERMEAN | _LCLM_ | Lower confidence limit for the mean of the predicted value | No |
| UCLM, UPPERMEAN | _UCLM_ | Upper confidence limit for the mean of the predicted value | No |
| LCL, LOWERPRED | _LCL_ | Lower confidence limit for the predicted value | No |
| UCL, UPPERPRED | _UCL_ | Upper confidence limit for the predicted value | No |
| COOKD, COOKSD | _COOKD_ | Cook's *D* influence measure | No |
| DFFITS | _DFFITS_ | Standardized influence of the observation on predicted value | No |
| COVRATIO | _COVRATIO_ | Standardized influence of the observation on the covariance matrix of the parameter estimates | No |
| LIKEDIST, LD | _LIKEDIST_ | Displacement (distance) of log-likelihood when the observation is removed (assuming normal distribution) | No |

If you specify SCORE(_ALL_), then the server calculates and adds all the possible output statistics to the temporary table. The confidence levels for the LCLM, LCL, UCLM, and UCL confidence bounds are determined from the significance level specified in the ALPHA= option as (100 (1-α)%). The default is $\alpha = 0.05$.

The server determines the column names for the output statistics. This differs from many SAS procedures where you can specify the name for the statistic.

**SEED=*number***
specifies the random number seed for generating random numbers. The random number is used to determine whether an observation belongs to the training or validation data set. The SEED= option has no effect unless you specify the VALPROP= option. If the specified number is negative or zero, the random number generation is based on the computer clock of the server—this generates a non-reproducible random number sequence.

**SELECT=(*list-of-ODS-tables*)**
specifies the list of ODS tables that you want to display for the analysis. The specified list replaces the default tables that are generated by the server and displayed. See the EXCLUDE= option for the list of default tables and the table names that you can display.

**SHOWSELECTED**
requests that the server perform variable selection for the model. A backward selection method is used, where the significance level for an effect to remain in the model is determined by the SLSTAY= option. This option performs variable selection like the VARSEL option, but in contrast to the latter option, it displays output only for the selected effects.

**Alias** SHOWSEL

**SLSTAY=*α***
specifies the significance level used in determining whether effects should stay in the model during variable selection.

**Default** 0.1

**Range** 0 to 1

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**Interaction** For information about the interaction between the TEMPTABLE, CODE, and SCORE options, see "Temporary Tables, Generated Code, and Scoring" on page 253.

**VALIDATE=*f***
specifies the proportion *f* in the validation data set.

| | |
|---|---|
| **Alias** | VALPROP= |
| **Range** | 0 to 1 |
| **Interaction** | If you specify both the ROLEVAR= option and the VALIDATE= option, then the ROLEVAR= setting supersedes the VALIDATE= option. |

**VARSELECTION**
specifies that the server perform variable selection for the model. A backward selection method is used, where the significance level for an effect to remain in the model is determined by the SLSTAY= option. In contrast to the SHOWSEL option, all effects are reported in the IMSTAT output.

| | |
|---|---|
| **Alias** | VARSEL |

**VIF**
produces variance inflation factors and tolerances, the reciprocal of the VIF, for the parameter estimates.

**WEIGHT=*variable-name***
specifies the numeric variable to use as a weighing variable in solving the linear model.

When you specify a WEIGHT= variable, the normal equations

$$X'Xb = X'Y$$

are replaced by

$$X'WXb = X'WY$$

where $W$ is a diagonal matrix with the values of the variable specified in the WEIGHT= option on the diagonal. Only the observations with a weight value that is not missing and greater than zero are used in the analysis.

## Details

### *Basic Syntax*

The basic syntax of the GLM statement requires that you specify the response variable (the dependent variable), an equal sign (=) and then the model effects. The dependent variable must be numeric. In contrast to the GLM procedure, you can specify only one dependent variable in the GLM statement of the IMSTAT procedure.

The underlying statistical model of the GLM statement is as follows:

$$Y = X\mathrm{B} + e$$

where $Y$ is an $(n \times 1)$ random vector of the dependent variable, $X$ is an $(n \times p)$ design matrix, $B$ is a $(p \times 1)$ vector of coefficients, and $e$ is an $(n \times 1)$ vector of random disturbances (errors). The key assumptions of a GLM type of model are that the errors $e$ are uncorrelated, homoscedastic (have the same variance $\sigma^2$), and have zero mean. If these assumptions are met, then the model is correct and the elements of the vector $Y - XB$ are stochastically unrelated. The goals of the GLM analysis are as follows:

- to estimate the unknowns $\beta$ and $\sigma^2$

- to diagnose the appropriateness of the specified model
- to select appropriate variables and terms for the *X* matrix
- to predict the average response and unobserved values of the response variable with confidence
- to test hypotheses about the elements of β provided that the model is acceptable

A model effect is a syntactic expression of how one or more variables act together to define columns in the design matrix *X* of the linear statistical model. In other words, how you specify the *model-effects* on the right-hand side of the GLM statement affects how the server constructs the *X* matrix and how you interpret the results of the analysis pertaining to the contributing variables.

There are a few basic types of effects:

- the intercept is included by default in every model. It is then the leading effect in *X* and simply adds a column of ones to this matrix. The intercept can be approximately interpreted as an adjustment for the mean of the response variable.
- a continuous effect consists of only numeric non-class variables. The simplest continuous effect contains only one variable. For example, if you add the numeric variable Age to the model, you are adding a continuous effect. If the variable Height is not a classification variable, and you add the term Age*Height to the model, you are adding a continuous interaction effect.
- a classification variable is a variable that is used in the model not through its raw values, but through an encoding of its unique (formatted) values. For example, if variable Gender is used as a classification variable in the model, then it represents two levels.
- a classification effect is a model effect that contains one or more classification variables. A "pure" classification effect comprises only classification variables, a continuous classification effect also involves some continuous variables.
    - If A and B are classification variables, and X and Z are non-classification variables, then the effects A, B, and A*B are pure classification effects, termed the A and B main effect and the interaction of A and B, respectively. The effect A*Z would be a continuous classification effect.
    - Effect A is said to be nested within effect B if levels of A within one level of B do not mean the same thing for other levels of B. Nested effects are expressed with parenthetical notation. For example, if City and State are classification variables, then City(State) represents the nested effect of cities within states. One example of appropriate nesting is when city #1 in Alaska refers to a different city than city #1 in Colorado.
    - Two effects are said to be crossed if the levels of one effect retain their interpretation across the levels of the other effect. If Married is a classification variable that groups individuals into married and unmarried status, and Gender is a two-level variable, the Gender*Married effect is crossed, because a man in the unmarried group is also a man in the married group.

Deciding which variables to involve in a statistical model and how the variables should act and interact is key in modeling. The following rules apply for the GLM statement in the IMSTAT procedure:

- A character variable that is used in a model effect is treated as a classification variable.
- A numerical variable that is used in a model effect is treated as a non-classification variable.

- All variables explicitly listed in the optional variable list that follows the specification of the dependent variable in the GLM statement are classification variables.
- The role of temporary computed variables is determined by the data type.
  - If the computed column is of character type, then it is automatically added to the model as a classification variable if it appears in a model effect.
  - If the computed column is of numeric type, then it is treated as a classification variable only if it is specified in the list of *class-variables*.

The following example of modeling the Sashelp.Class data set shows how variables act and interact. The following GLM statement models a student's height as a function of his or her weight and gender:

```
glm height = weight sex;
```

The Sex variable, because it has character type, is treated as a classification variable. The following GLM statement is equivalent, but it expresses the classification variables explicitly:

```
glm height(sex) = weight sex;
```

Computed columns can also be used. The following example uses the same variables that were used in the previous examples, but specifies them as computed columns to demonstrate the syntax:

```
table lasr.class(tempnames=(t1 t2));
glm height = t1 t2 / tn=(t1 t2) te="t1=weight; t2=sex;";
```

The analysis that uses the computed columns (T1 and T2) is identical to the previous GLM statement. This is because T2 would be discovered by the server to be of character type and would be added automatically to the list of classification variables.

### *Informative Missingness*

The concept of informative missingness is one way to account for missing values in statistical analyses and, in particular, statistical modeling. Missing values are a problem because they reduce the amount of available data. When working with classification variables (factors, which are levelized variables), a missing value can be treated as an actual level of the variable and can participate in the analysis.

When continuous variables have missing values, however, the observation is removed from the analysis. In data with many missing values, this can reduce the amount of available data greatly, and the sets of observations used in one model versus another model can vary based on which variables are included in the model.

Of course, there are many reasons for missing values and substituting values for missing values has to be done with caution. For example, the famous Framingham Heart study data set contains 5,209 observations on subjects in a longitudinal study that helped understand the relationship between smoking, cholesterol, and coronary heart disease. One of the variables in the data set is AgeCHDdiag. This variable represents the age at which a patient was diagnosed with coronary heart disease (CHD). If you include this variable in a statistical model, only 1,449 observations are available, since the value cannot be observed unless a patient has experienced CHD. Including this variable acts as a filter that reduces the analysis set to the subjects with CHD. We cannot impute the value for subjects where the variable has a missing value, because we cannot impute an age at which someone who has not had CHD would have contracted coronary heart disease.

With informative missingness, we are not as much substituting imputed values for the missing values, as we are modeling the missingness. Consider a simple linear regression model:

$$y = \beta_0 + \beta_1 + \varepsilon$$

Suppose that some of the values for the regressor variable $x$ are missing. The fitted model uses only observations for which $y$ and $x$ have been observed.

In order to predict the outcome $y$ for an observation with missing $x$, we either assume that $y$ is missing or substitute a value for the missing $x$, such as the average value, $\overline{x}$. Because the estimate for the intercept is $\widehat{\beta}_0 = \overline{y} - \widehat{\beta}_1\overline{x}$ in the simple linear regression model, the predicted value would be the average response of the nonmissing values, $\overline{y}$.

With informative missingness, we extend the model by adding extra effects for each effect that contains at least one continuous variable. In the simple linear regression model, we add one column to the model and slightly change the content of the $x$ variable:

$$y = \beta_0 + \beta_1 x^* + \beta_2 x^{**} + \varepsilon_1$$

The variable $x^*$ contains the original values of $x$ if these are not missing, and the average of $x$ otherwise:

$$x^* = \begin{cases} x & \text{if x is not missing} \\ \overline{x} & \text{otherwise} \end{cases}$$

The variable $x^{**}$ is a dummy variable with value 1 when $x$ is missing, and zero otherwise:

$$x^{**} = \begin{cases} 1 & \text{if x is missing} \\ 0 & \text{otherwise} \end{cases}$$

The fitted model is not the same model that results from substituting $\overline{x}$ for the missing values during training. This can be seen, since the model that simply substitutes $\overline{x}$ for the missing values is as follows:

$$y = \beta_0 + \beta_1 x^* + \varepsilon_2$$

The informative missing model has an extra parameter, and unless all values of $x^{**}$ are zero—in which case there are no missing values—the informative missing model has a higher $R^2$ value, because it picks up more variation.

The parameter estimate for $\beta_2$ measures the amount by which the predicted value differs from a predicted value at $\overline{x}$.

### ODS Table Names

The ODS tables that can be generated with the GLM statement are described in the EXCLUDE= option on page 148.

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## GROUPBY Statement

The GROUPBY statement derives the grouping hierarchy of the distinct formatted values for the specified variables. If no list of variable names is specified, the grouping hierarchy is computed for all variables in the

active table. The statement can return a section of all distinct groupings to the client or save the entire grouping set as a temporary table in the server.

## Syntax

**GROUPBY** <*variable-list*> <*/ options*>;

### GROUPBY Statement Options

**AGGREGATE=(*aggregation-method*)**

lists the aggregator on which the ordering of the result set is based.

lists the aggregator for which the values of the WEIGHT variable are rolled up into a rank order score, provided that a WEIGHT= variable is specified. If no WEIGHT= variable is specified, then the aggregator specification is ignored.

The available aggregation methods for the GROUPBY statement are as follows:

| | |
|---|---|
| MAX | maximum value |
| MEAN | arithmetic mean |
| MIN | minimum value |
| N | counts the nonmissing values of the weight variable |
| SUM | sum of the weight values |

**Alias** AGG=

**Default** SUM

**DESCENDING**

specifies to arrange the returned grouping hierarchy of the variables in descending order of the item rankings. If this option is not specified, the returned items are arranged in ascending order. When combined with the LIMIT=*n* option, the GROUPBY statement can either return the top *n* or the bottom *n* distinct groupings.

**Interaction** The DESCENDING option is ignored if the TEMPTABLE option is specified.

**FREQ=*variable-name***

specifies the numeric frequency variable that is used to compute the ranking of a distinct grouping. When this option is specified, the AGGREGATE= and WEIGHT= options are ignored. The following GROUPBY statement requests the top 5 groupings of Region and then Product from the Prdsale table. The groupings are rank ordered by the sum of the Actual column:

**Example**

```
proc imstat data = mylasr.prdsale;
    groupby region product / freq = actual limit = 5;
run;
```

**LIMIT=*n***

specifies the maximum number of distinct groupings to be returned. When combined with the DESCENDING option, the GROUPBY statement can either return the top *n* or the bottom *n* distinct groupings. The value for *n* must be a positive integer. For example, the commands below return the bottom 5 groupings according to their Score values:

**Default** 0

**Interaction** This option is ignored if the TEMPTABLE option is specified.

**Tip** If *n* is zero, then all distinct groupings are returned (up to $2^{31}$–1). With high-cardinality data sets, setting *n* to zero can significantly delay the response of the server.

**Example**

```
proc imstat data = mylasr.prdsale;
    groupby region product / weight  = actual
                             agg     = max
                             valuegt = ("West", "Chair")
                             limit=5;
run;
```

**NOMISSING**
specifies that missing values are excluded in the determination of GROUPBY values. By default, levels with missing values are included.

**Alias** NOMISS

**NOTEMPPART**
specifies that the temporary table that is generated by the TEMPTABLE option is not partitioned by the group-by variables. When you create a temporary table with the GROUPBY statement, by default, the server partitions the table and each partition has a single row. When the number of groups is large, this results in many tiny partitions and requires additional memory resources to store the partition information for the temporary table.

By specifying this option, the temporary table is organized similarly to the default table, but it is not partitioned. This also enables more efficient processing of the table in threaded computations. For example, it is more efficient if you were to add computed columns to the table that you want to use as dimension keys in subsequent SCHEMA statements.

**ORDER=*rank-order-type***
specifies the rank ordering to use for sorting the distinct groupings. The following rank-order types are valid in the GROUPBY statement:

| | |
|---|---|
| FREQ | frequency count of the variables |
| VALUE | formatted values of the variables |
| WEIGHT | aggregate values of the WEIGHT= variables |

**Default** FREQ

**PARTITION <=*partition-key*>**
specifies that when the active table is partitioned, then the *variable-list* that you specify for the GROUPBY statement is expanded to include the partition variables. The partition variables are added to the beginning of the *variable-list*.

If you do not specify a *partition-key*, the analysis is performed for all partitions. If you do specify a *partition-key*, the analysis is carried out for the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid partition key values for a table.

The following GROUPBY statement works on the partitioned table Cars. The grouping hierarchy becomes **Type** ⇨ **Origin** ⇨ **Make**:

**Example**

```
data mylasr.cars(partition=(type));
    set sashelp.cars;
run;

proc imstat data=mylasr.cars;
  groupby origin make / partition weight=invoice agg=max
                        descending limit=5 ;
run;
```

**SAVE=*table-name***

saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**

requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**SCOREGT=*f***

specifies the exclusive lower bound of the numeric rank order scores of the distinct groupings to return. All distinct groupings with numeric rank order scores that are greater than *f* are returned.

**Alias** SGT=

**Interaction** This option is ignored if the TEMPTABLE option is specified.

**SCORELT=*f***

specifies the exclusive upper bound of the numeric rank order scores of the distinct groupings to return. All distinct groupings with numeric rank order scores that are less than *f* are returned.

**Alias** SLT=

**Interaction** This option is ignored if the TEMPTABLE option is specified.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the &_TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**VALUEGT=("*format-specification*", …)**
specifies the exclusive lower bound of the variable's formatted values of the distinct groupings to return. All distinct groupings with formatted values for the variable that are lexicographically greater than the specified bound are returned.

**Alias** VGT=

**Interaction** This option is ignored if the TEMPTABLE option is specified.

**VALUELT=("*format-specification*", …)**
specifies the exclusive upper bound of the variable's formatted values of the distinct groupings to return. All distinct groupings with formatted values for the variable that are lexicographically less than the specified bound are returned.

**Alias** VGT=

**Interaction** This option is ignored if the TEMPTABLE option is specified.

**VARFORMATS=("*format-specification*", …)**
specifies the formats for the variables. If you do not specify the VARFORMATS= option, the default formats are applied for the variables.

**WEIGHT=*variable-name***
specifies the numeric weight variable to use for computing the rank order score of a distinct grouping.

**Interaction** When the WEIGHT= option is specified, the server sets the ORDER= option to ORDER=WEIGHT.

## Details

### *ODS Table Names*

The GROUPBY statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| GroupBy | Top K groupby values | Default |
| TempTable | Information about a temporary table | TEMPTABLE |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## HISTOGRAM Statement

The HISTOGRAM statement calculates a histogram table for numeric variables.

### Syntax

**HISTOGRAM** <*variable-list*> <*/ options*>;

#### *Optional Argument*

**variable-list**
specifies a single variable or a list of numeric variables. Separate each variable name by at least one space. If you do not specify this option, a histogram table is calculated for each numeric variable.

#### *HISTOGRAM Statement Options*

**BINVALS=*list-of-values***
specifies an array of NBINS lower bin boundaries as *list-of-values*. The histogram binning then uses those values strictly and does not alter them so that they are equally spaced (or “nice”). This option is useful to compute a histogram with bins that are the same as those of another histogram so that the values can be compared or overlaid. The bins do not need to be equally spaced.

**EQUALFREQ**
specifies to create bins such that each bin contains the same fraction of the data.

**Alias** EQUAL

**MAX=*number***
specifies the upper end of the range to determine the histogram bins. By default, the maximum value is determined from the data (subject to the WHERE clause). The bins of the histogram can extend beyond the extreme values when the "nice-ing" algorithm places bin boundaries on numbers that are convenient to label on axes.

**MIN=*number***
specifies the lower end of the range to determine the histogram bins. By default, the minimum value is determined from the data (subject to the WHERE clause). The bins of the histogram can extend beyond the extreme values when the "nice-ing" algorithm places bin boundaries on numbers that are convenient to label on axes.

**NBINS=*k***
specifies the number of bins to use for calculating the histogram.

**NOEMPTYBIN**
prevents bins without observations from being displayed. The leading and trailing empty bins are trimmed. Any internal empty bins are combined into the first non-empty bin to the immediate right. The mid-value of the bin into which the empty bins are combined is not adjusted. If the mid-value is not missing, then you can use the asymmetry of a bin as an indicator that it was combined with empty bins.

**NONICE**
specifies that the "nice-ing" algorithm is suspended. The boundaries of the histogram are based on the actual range of the data (subject to the WHERE clause) or on the MIN= and MAX= values that you specify. The bin boundaries are not guaranteed to fall on "nice" values.

**OUTLIERBIN**
specifies that outliers are placed in special bins in the two tails. Outliers with values that fall below Q1 – 1.5*IQR are placed in the left-most bin. Outliers with a value that is above Q3 + 1.5*IQR are placed in the right-most bin. IQR is the inter-quartile range, which covers the central 50% of the distribution of the variable. The mid-value reported by the IMSTAT procedure can be used as an indicator whether a bin is an outlier bin. The mid-value is set to 1 for an outlier bin and set to missing otherwise.

**Interaction** This option is ignored if you specify the EQUALFREQ option.

**ROUNDINGDIRECTON=*direction***
specifies the direction to round numbers when a rounding factor is specified. For example, if you specify ROUNDINGFACTOR=5, a bin boundary of 6.2 is rounded up to 10, down to 5, and nearest to 5.

The following directions are valid in the HISTOGRAM statement:

| | |
|---|---|
| UP | Round up to a multiple of the ROUNDINGFACTOR= value. |
| DOWN | Round down to a multiple of the ROUNDINGFACTOR= value. |
| NEAREST | Round to the nearest multiple of the ROUNDINGFACTOR= value. |

**Default** UP

**ROUNDINGFACTOR=*value***
specifies the factor to use for rounding up internal bin boundaries. The lower bound of the left-most bin and the upper bound of the right-most bin are not rounded. For example, when you work with prices in dollars, specifying ROUNDINGFACTOR=0.01 rounds the bin boundaries to cents. In the event that the specified rounding factor is greater than the bin width and multiple bins round up to the same number, the bins are collapsed into a single bin.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### ODS Table Names

The HISTOGRAM statement generates the following ODS table for each analysis variable.

| ODS Table Name | Description | Option |
|---|---|---|
| Histogram | Histogram data | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# HYPERGROUP Statement

The HYPERGROUP statement analyzes a graph whose vertices are identified by values of the analysis variables and whose edges are named by those variables within the same observation. The analysis that can be performed falls into three general areas: structural analysis of the overall graph, centrality calculation for individual vertices, and layout of the graph in either 2-D or 3-D space.

## Syntax

**HYPERGROUP** <*variable-list*> <*/ options*>;

### Optional Argument

***variable-list***
specifies the variables to include in the analysis. The variables must have a character data type. Separate variable names with a space. If you do not specify any variables, then all character variables in the active table are used in the analysis.

More information, including the namespace specification is available in "Specifying Analysis Variables" on page 176.

### HYPERGROUP Statement Options

**C=*relative-strength***
specifies the relative strength of local forces to global forces with regard to laying out the positions of vertices and edges. The Walshaw layout is a force-directed algorithm that finds positions of vertices so that no vertices are too close together and so that (usually) edges are about the same length. The force term in a force-directed layout algorithm is related to springs. Imagine each vertex is a ring and each edge is a spring whose ends are hooked around the rings of the vertices to which the edge connects.

Each spring is equally springy (has the same spring constant). If a spring is too compressed, it wants to push apart the vertices at its ends. If the spring is too extended, the spring wants to pull the vertices closer together. The forces that are exerted by these springs, for which there is a corresponding edge, are local forces.

In addition, vertices that are near each other that might or might not be connected by an edge are modeled as if there is a temporary spring between them that is capable of repulsion only. This is done to keep vertices from being located too close to each other. If vertices are very close, the repulsion is very great. These forces are known as global forces. There is not necessarily an edge between two vertices that exert global forces against each other.

This option controls the relative strength of local forces to global forces. In general, larger values for C= result in graphs with more space between vertices. The effect is that the repulsion of vertices that are strongly connected repel other vertices in other strongly connected subgraphs—the effect is to lengthen weak edges. Good values for C= begin at 0.01. Values above 0.1 typically cause global forces to become too strong relative to local forces.

**Default** 0.01

**Applies to** LAYOUT=WALSHAW

**CENTRALITY**

specifies to quantify the importance of each vertex among its peers. Many types of centrality have been defined. The HYPERGROUP statement supports five that are commonly used. Four of these are based on shortest paths (the smallest number of edges in a path from one vertex to the other). The fifth is a geometric measure that can be calculated when graph layout is performed. For more information, see "Centrality Measures" on page 177.

**CLOSITERS=*n***

specifies the number of layout iterations that are performed before a sub-algorithm determines the vertices that are close to each other. Increasing the value can improve performance because more iterations are performed between attempts to evaluate which nodes are too close together.

**Default** 3

**COMMALGORITHM= ASYNCHRONOUS |**
**COMMALGORITHM= SYNCHRONOUS | SEMISYNCHRONOUS**
**COMMALGORITHM= LLSYNCHRONOUS | LLSEMISYNCHRONOUS**

specifies a particular label propagation algorithm when STRUCTURAL=COMMUNITY analysis is specified. The LL prefix indicates to use a parallel version of the algorithm.

**Alias** COMMALG

**COMMITERS=*n***

specifies the number of iterations to perform while determining communities. Communities are determined by a variant of the label propagation algorithm described in Raghavan, Reka, and Kumara (2007). The algorithm is iterative, and stops when COMMITERS= iterations have been performed.

The algorithm might perform fewer iterations than are specified if all vertices have this property: a community *c* is formed by a set of vertices so that for any vertex *v* in *c*, the number of edges directed from *v* to other vertices in *c* outnumber or tie the number of edges directed from *v* to vertices outside *c*.

**Default** 20

**Tip** The synchronous algorithms that are available with the COMMALG= option can require larger values for COMMITERS= for convergence to occur.

**COMMLAYOUTS**
specifies to lay out coordinates for the community graph that is produced with the STRUCTURE=COMMUNITY (or BOTH) option. The coordinates shown are returned in the _TEMPHYPGRP3_ and TEMPEDGES3_ tables.

**COMMMAX=*n***
specifies the maximum number of iterations to perform to determine labeling for communities. For the label propagation algorithm used when you specify STRUCTURAL=COMMUNITY, this option, together with COMMPRECENDENCE, alters tie-breaking schemes when there is a choice as to what value should be assigned to a vertex label.

Label propagation is an epidemic algorithm. Avoid setting COMMMAX= too low, because low values tend to infect vertices with the wrong label in early iterations. Different combinations of this option with COMMPRECEDENCE can affect the number and quality of the communities found. Refer to Cardasco and Gargano (2011 and 2012) for a description the algorithm and these options.

**COMMPRECEDENCE**
An option for tuning the label propagation algorithm used with STRUCTUAL=COMMUNITIES analysis. See the explanation of the COMMMAX= option for details.

**Alias** COMMPRE

**CREATETEMPLAST= NEVER | ALWAYS | MULTIPLE**
specifies when to create the _TEMPLAST_ temporary table that identifies the hypergroups and analysis variables. If you use a large active table with the HYPERGROUP statement, then the _TEMPLAST_ temporary table can be large as well.

**NEVER**
specifies to never create the _TEMPLAST_ temporary table. Be aware that the other in-memory temporary tables like _TEMPHYPGRP_ and _TEMPEDGES_ are created. These have summarized information about the hypergroups and are smaller than the _TEMPLAST_ table.

**ALWAYS**
specifies to create the _TEMPLAST_ temporary table.

**MULTIPLE**
specifies to create the _TEMPLAST_ temporary table when the analysis results in more than one hypergroup.

**Default** ALWAYS

**FAR_AWAY=*d***
specifies how to tune the layouts when LAYOUY=WALSHAW is specified.

When FAR_AWAY=1, the default value, the Walshaw algorithm models global forces between vertices if these vertices are not far away from each other. How far away is determined is complicated, depends on the size of the graph, and depends on what stage of graph partitioning is being performed. The FAR_AWAY= option expresses a multiple of the usual value that the algorithm would calculate.

In other words, if *d* is usually the greatest distance between two vertices that are allowed to exert a global force on each another, then specifying FAR_AWAY=2 indicates that vertices that are twice *d* away from each other are allowed to exert global forces. Of course, the farther away vertices are from each other, the weaker are the global forces, but even vertices farther away (but not excluded), can be influential if there are enough vertices to include. The FAR_AWAY= option controls how many of the distant vertices can exert pull.

The result of using larger values for FAR_AWAY= is similar to using larger values for C=. In both cases, the larger values for the options makes the layouts more spacious, at the expense of laying out all edges to have similar lengths. A distinctive feature of larger FAR_AWAY= values is that it causes vertices to be positioned farther from the center toward the nearest pane border.

**Default** 1

**FORMATS=("*format-specification*",…)**
specifies the formats for the GROUPBY= variables. If you do not specify the FORMAT= option, or if you do not specify the GROUPBY= option, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**GRAPHPARTITION**
specifies to tune the layout to improve the separation of vertices. This option can increase the processing duration.

**Applies to** LAYOUT=WALSHAW or LAYOUT=FRUCHGOLD

**GROUPBY=(*variable-list*)**
specifies a list of variable names, or a single variable name, to use as GROUPBY variables in the order of the grouping hierarchy. If you do not specify any GROUPBY variable names, then the calculation is performed across the entire table —possibly subject to a WHERE clause.

**GROUPBYLIMIT=*n***
specifies the maximum number of levels in a GROUPBY set. When the software determines that there are at least *n* levels in the GROUPBY set, it abandons the action, returns a message, and does not produce a result set. You can specify the GROUPBYLIMIT= option if you want to avoid creating excessively large result sets in GROUPBY operations.

**GROUPFILTER=(*groupfilter-options*)**
specifies a section of the GROUPBY= hierarchy to include in the HYPERGROUP computation.

**HEIGHT=*z***
specifies the maximum value for the frame's coordinate space in the Z-axis.

Be aware that the MARGIN= value is subtracted from the HEIGHT= value.

**Default** 100 units

**Interaction** This option is used only when you specify the THREED option.

**HIGHDEGREE= 0 | 1**
specifies to enable a heuristic that begins partitioning by eliminating vertices of unusually high degree. Some graphs have many vertices with low degree. The degree of a vertex is the number of edges that originate from or are directed toward the

vertex. However, some graphs might have some vertices with very high degree. It is often beneficial to treat these high degree vertices as partitions early in the partitioning algorithm, even if they do not strictly split a graph. This simplifies the processing so that from what is left of the remaining graphs are less dense and faster to process.

**Default** 0 (disabled)

**Range** 0 or 1

**LAYOUT= WALSHAW | FRUCHGOLD | OTHER**
specifies one of three force-directed algorithms to use for graph layout.

If you specify either LAYOUT=WALSHAW or LAYOUT=FRUCHGOLD, you can also specify the GRAPHPARTITION option, so that graph partitioning is used.

The WALSHAW option performs the algorithm described by C. Walshaw (2000). The FRUCHGOLD option performs the algorithm described by T.M.J. Fruchterman and E.M. Reingold (1991). Specifying OTHER performs an algorithm that is proprietary to SAS.

The force term in force-directed layout algorithm is related to springs. Imagine each vertex is a ring and each edge is a spring whose ends are hooked around the rings of the vertices the edge connects. Each spring is equally springy. If a spring is too compressed, it wants to push apart the vertices at its ends. If the spring is too extended, the spring wants to pull the vertices closer together. In addition, vertices that are near each other but are not connected by an edge are modeled as if there is a temporary spring between them that is capable of only repulsion. This method for modeling is done to prevent laying out vertices too closely to each other. Of course, if vertices are very close, the repulsion is very great.

**Default** LAYOUT=WALSHAW

**LENGTH=*y***
specifies the maximum value for the frame's coordinate space in the Y-axis.

Be aware that the MARGIN= value is subtracted from the LENGTH= value.

**Default** 100 units

**MARGIN=*n***
specifies the size of the border around the frame's coordinate space to remain free of vertices. For example, if you specify LENGTH=100, WIDTH=100, and MARGIN=12, then the frame coordinate space is 100 × 100 units and vertices have coordinates within the corners (12, 12), (12, 88), (88, 12), and (88, 88).

**MAXNODES=*n***
specifies to tune graph partitioning by specifying the maximum number of nodes to permit in a partition. Each time a partitioning is performed, the resulting set of partitioned subgraphs is examined. If any exceed the maximum number of nodes specified in this option, then the partitioning is repeated on those partitions.

**Default** 4

**MAXNVALS=*i***
specifies a positive integer that determines the maximum number of iterations for the percentile algorithm.

**Default** 1000

**NITERATIONS=*i***

specifies a positive integer that determines the maximum number of iterations to execute for the forced-directed layout algorithm. A value between 200 and 5000 produces good results with most data sets. The LAYOUT=WALSHAW layout algorithm might stop before completing all NITER= iterations if the algorithm detects that convergence has occurred.

If you specify NITERS=0, then it is the same as specifying the NOCOORD option.

**Alias** NITERS

**Default** 1000

**NOCOLOR**

specifies not to run the graph partitioning algorithm to assign colors to strongly connected communities. The algorithm is run by default. This option is useful if you do not use the color values. You can avoid the processing that is performed to assign color categories.

**Alias** NOCOLOUR

**NOCOORD**

specifies not to perform graph layout of vertices and edges. Graph layout is the most time-consuming calculation that the HYPERGROUP statement performs. This option is useful if you do not need a visual or geometric layout, or calculation of centroid centrality. This option can improve the response time and conserve machine resources.

**NOPENDANTS**

specifies to simplify the graph layout by removing pendants (nodes of degree one). This option is performed repeatedly until no pendants remain in the graph.

**NOVARS**

specifies not to transfer additional variables to the _TEMPLAST_ table. See also VARS=.

**PARTITION <=*partition-key*>**

When you specify this option and the table is partitioned, the results are calculated separately for each value of the partition key. In other words, the partition variables function as automatic GROUPBY variables. This mode of executing calculations by partition is more efficient than using the GROUPBY= option. With a partitioned table, the server takes advantage of knowing that observations for a partition cannot be located on more than one worker node.

If you do not specify a *partition-key*, the analysis is performed for all partitions. If you do specify a *partition-key*, the analysis is carried out for the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid partition key values for a table.

You can specify a *partition-key* in two ways. You can supply a single quoted string that is passed to the server, or you can specify the elements of a composite key separated by commas. For example, if you partition a table by variables GENDER and AGE, with formats $1 and BEST12, respectively, then the composite partition key has a length of 13. You can specify the partition for the 11-year-old females as follows:

```
statement / partition="F          11"; /* passed directly to the server */
statement / partition="F","11";        /* composed by the procedure */
```

If you choose the second format, the procedure composes a key based on formatting information from the server.

**Alias** PART=

**RADIANS**
specifies to return the centroid centrality angles in radians rather than degrees.

**Applies to** CENTRALITY option

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SCALECOORDS**
specifies to scale vertex coordinate values so that they are within the boundaries specified with the LENGTH=, WIDTH=, and HEIGHT= options. This option is useful when you specify LAYOUT=FRUCHGOLD or LAYOUT=OTHER algorithm and GRAPHPARTITION is not specified.

**SEPARATOR= NODES | VERTICES**
**SEPARATOR= ARCS | EDGES**
**SEPARATOR= HYBRID**
specifies how to tune the graph partitioning algorithm by indicating how to choose partition separators.

Graph partitioning works to find separators that are small and partitions that are large. A vertex separator is a set of vertices that, if removed from the graph, results in two or more separate sub-graphs that correspond to partitions. An edge separator is a set of edges that, if removed, results in two or more separate sub-graphs that correspond to partitions. There is never an edge between vertices in separate partitions.

By default, SEPARATOR=HYBRID. In this case, a vertex separator is ultimately determined, but in the initial stages of graph partitioning, edge separators are determined. As graph partitioning continues, vertex separation is used.

**Default** HYBRID

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**STRUCTURAL= NONE | COLOR | COLOUR | COMMUNITY | BOTH**

Hypergroups (completely disconnected subsets) are always identified within the graph. Specify this option to request additional structural analyses that identify strongly connected components within each hypergroup. This option enables you to find subsets of the graph whose vertices have many interrelationships internally, but fewer between the subset. Unlike hypergroups, these subsets are not disconnected from each other.

When you specify this option, two additional temporary tables are created for vertices and edges. These tables depict the strongly connected components as a graph. Using these tables, it is possible to zoom out from the detailed graph—to depict the mesostructure or macrostructure of the graph. In a sense, this is the graph theory equivalent of aggregation on numeric quantities.

**BOTH**

specifies to perform COLOR and COMMUNITY analysis.

**COLOR | COLOUR**

specifies to identify the strongly connected components with the graph partition algorithm and assigns a color value to each component. A color value is assigned to each vertex and edge. The following table identifies each component, table, and column name that includes a color value.

| Component | Table Name | Column Name |
|---|---|---|
| Vertices | _TEMPHYPGRP_ | _COLOR_ |
| Edges | _TEMPEDGES_ | _SCOLOR_ and _TCOLOR_ |

In addition, the graph of the derived components is described in the _TEMPHYPGRP2_ and _TEMPEDGES2_ temporary tables.

**COMMUNITY**

specifies to identify the strongly connected components with the label propagation algorithm and assign each component with a community value. A community value is assigned to each vertex and edge. The following table identifies each component, table, and column name that includes a community value.

| Component | Table Name | Column Name |
|---|---|---|
| Vertices | _TEMPHYPGRP_ | _COMMUNITY_ |
| Edges | _TEMPEDGES_ | _SCOMMUNITY_ and _TCOMMUNITY_ |

In addition, the graph of the derived communities is described in the _TEMPHYPGRP3_ and _TEMPEDGES3_ temporary tables.

Sometimes communities are better at indicating the components that are strongly connected. Sometimes colors do better—particularly when the graph is less structured but still can be usefully divided by separators. For many graphs that have structure, vertices that have the same color also have the same community value, although the color value and the community values can be different.

**Default** NONE

**TEMPTABLE**
specifies to store the results of the analysis in in-memory tables on the server. You do not need to specify this option because the HYPERGROUP statement always generates in-memory tables for the result sets.

**THREED**
specifies to graph the layout in three dimensions instead of two dimensions. The HEIGHT= option controls the maximum values for the Z-axis.

**Alias** D3

**TOPLEFT**
specifies to produce the graph layout coordinates and centroid centrality angles based on an origin at the top left corner of the drawing window. By default, the HYPERGROUP statement generates coordinates based on an origin at bottom left corner of the drawing window.

**VARFORMATS=("*format-specification*",…)**
specifies the formats to apply to the variables. If you do not specify the VARFORMATS= option, the default formats are applied to the variables.

**VARIABLES=(*variable-1* … *variable-n*)**
specifies the variables from the active table to transfer to the generated _TEMPLAST_ table as additional ID variables. The variables that are specified after the HYPERGROUP statement are always transferred. By default, all variables are transferred.

**Alias** VARS=

**See** "NOVARS" on page 172

**WIDTH=*x***
specifies the maximum value for the frame's coordinate space in the X-axis.

Be aware that the MARGIN= value is subtracted from the WIDTH= value.

**Default** 100 units

## Details

### *Introduction to the HYPERGROUP Statement*

Hypergroups extend how SAS LASR Analytic Server can identify data that are connected, by variables having disjoint sets of values. Hypergroup technology is used to perform analytics after data has been split up in meaningful ways. Most IMSTAT procedure statements enable analysis to be split into independent parts by values of GROUPBY variables in individual records. In contrast, hypergroups enable analysis based on data that are spread across more than one record.

The algorithms used to determine hypergroups are based on graph theory. Vertices have names that are values of hypergroup variables, and edges that connect vertices (v1, v2) are generated if there is any record in data that has v1 and v2 as values in adjacent hypergroup variables.

Besides generating the _TEMPLAST_ table (similar to the active table, but with an additional _HypGrp_ variable) , the HYPERGROUP statement generates the _TEMPHYPGRP_ that includes information about the vertices and the _TEMPEDGES_

table that includes information about the edges. Both these tables have indices with respect to all the data, and other indices with respect to vertices in subgraphs that correspond to separate hypergroups.

The _TEMPHYPGRP_ and _TEMPEDGES_ tables also have vertex coordinates and suggested color and community values. These values can assist with plotting the hypergroups. If you use the HYPERGROUP statement to identify groups only and do not plan to use the plotting information, then specify NOCOORD and NOCOLOR to improve processing times.

## *Specifying Analysis Variables*

### Simple Syntax

The analysis variables for HYPERGROUP are interpreted to form the graph. At least two variables must be listed. In a given observation, each analysis variable value is interpreted as the name of a vertex. Consecutive analysis variables in the list define an edge between the vertices they name. Thus, in this example:

```
hypergroup a b;   /* One edge for each observation  */
hypergroup a b c; /* Two edges for each observation */
```

The first statement defines one edge for each observation—between the vertex identified by the value of variable A and the vertex identified by the value of variable B. The second statement defines two edges per observation—the first between the A and B vertices and the second between the B and C vertices.

*Note:* When an analysis variable is a missing value, no edge is produced.

A simple example follows:

```
libname example sasiola host="grid001.example.com" port=10010 tag=hps;

data example.sales;
  input @1 Customer $ @7 Dealer $12. @21 Model $;
  cards;
Tina  AutoEmporium    Focus
Tony  AutoEmporium    Sonic
Tom   Prestige        TT
Tom   AutoEmporium    ATS
Blake AutoMall        Accord
Bob   AutoMall        RAV4
Bart  AutoMall        Civic
Beth  AutoMall        Accord

;;;

proc imstat data=example.sales;
  hypergroup customer dealer / vars=(model);
run;

  table example.&_TEMPLAST_;
  fetch / format;
run;
  table example.&_TEMPHYPGRP_;
  fetch / format;
run;
  table example.&_TEMPEDGES_
```

```
  fetch / format;
run;
```

By default, HYPERGROUP considers all vertex identifiers the equivalent, regardless of what analysis variables they come from. In the previous simple example, "Tom" is just a vertex, neither a customer or a dealer. Furthermore, there is nothing preventing you from linking "Bob" and "Tom." All analysis is based on edge relationships, not on attributes of the vertices. Do not misinterpret the syntax as indicating that a link is specifically from "Bob" to "AutoMall." This is not a directed graph. The presence of an edge indicates an undirected connection between "Bob" and "AutoMall," with no more significance in one direction than the other.

### Namespace Syntax

The following HYPERGROUP example is similar to the previous example, but demonstrates the syntax for using namespaces.

```
proc imstat data=example.sales;
  hypergroup (customer, dealer) / vars=(model);
run;
```

By enclosing variables in parentheses (and separating them with commas), you can indicate that the values from one or more columns form their own vertex namespace. Each comma between variables specifies a different namespace. You can avoid interpreting a customer name and a dealership name as the same vertex by specifying the preceding syntax to use different namespaces.

By default, without the namespace syntax, the vertex identifiers in different analysis variables are equivalent. A larger data set might include a customer named "Don Smith" and a dealership that is also named "Don Smith." In this case, they would be considered equivalent and identify the same vertex unless namespace syntax is used.

When you specify the namespace syntax, an additional ODS table is generated. The table identifies each analysis variable and a namespace value.

***Figure 4.1*** *ODS Table for Namespaces*

| Hypergroup Namespaces for Table HPS.SALES | |
|---|---|
| _Variable_ | _Namespace_ |
| customer | 0 |
| dealer | 1 |

A corresponding _Namespace_ column is also added to the _TEMPHYPGRP_ table. Two additional columns, _Snspace_ and _Tnspace_, are added to the _TEMPEDGES_ table.

## *Centrality Measures*

### Overview

When you specify the CENTRALITY option, additional columns are added to the _TEMPHYPGRP_ and _TEMPEDGES_ temporary tables that the HYPERGROUP statement generates. Information about the measures of centrality and the additional columns is included in the following sections.

If you specify COMMCENTRALITY, then similar tables and columns are generated, but they include COMM in the column names. When COMMCENTRALITY is specified, the information is related to centralities, but with respect to each community.

### Graph Centrality

Let $L_{v,w}$ be the length of the shortest path from vertex $v$ to vertex $w$, when $w$ is reachable from $v$. Graph centrality for a vertex $v$ is the greatest $L_{v,w}$ for any reachable $w$. For graphs that are wider than they are taller, or vice versa, the vertices in the middle have smaller graph centrality than the vertices at the pointy ends.

Graph centrality measures are included in the temporary tables that are generated by the HYPERGROUP statement. Although graph centrality is the name of this centrality, the column names with the measures use the word reach. This is because the quantification is a measure of how long it is to reach the vertex that is further out than any other.

### Closeness Centrality

Let $S_v$ = sum of $L_{v,w}$, in other words, the sum of the lengths of the shortest paths from vertex $v$ to vertex $w$ for all other reachable vertices. Let $S_{max}$ = the greatest $S_v$ where the closeness centrality for vertex $v = S_v / S_{max}$.

Closeness centrality measures are included in the temporary tables that are generated by the HYPERGROUP statement. Closeness centrality values can be found in the _CloseComm_ column in _TEMPHYPGRP_ for each vertex, and in the _SourceCloseComm_ and _TargetCloseComm_ columns in _TEMPEDGES_ for the source and target vertices for each edge.

### Stress Centrality

Stress centrality is another centrality that requires the shortest paths to be determined between reachable vertices. In addition, if some vertex $v$ can reach another vertex $w$, there might be more than one shortest path between them. Such shortest paths are multiple optima. For a vertex $v$, let $N_v$ = the number of times that $v$ is crossed when traversing all shortest paths, even those that are multiple optima. Let $N_{max}$ = the greatest $N_v$ . The stress centrality for vertex $v = N_v / N_{max}$.

Stress centrality measures are included in the temporary tables that are generated by the HYPERGROUP statement.

### Betweenness Centrality

Betweenness centrality quantifies the number of times a vertex is crossed along the shortest path, or paths, between two other vertices.

Let $T_{x,y}$ = the total number of shortest paths from a vertex $x$ to reachable vertex $y$. Let $T_{x,y}(v)$ be the number of the paths that cross vertex $v$. Therefore, the fraction of shortest paths that cross vertex $v = T_{x,y}(v) / T_{x,y}$. Let $B_v$ = the sum of $T_{x,y}(v) / T_{x,y}$ for all pairs of reachable vertices $x$ and $y$. Let $B_{max}$ = the greatest $B_v$. The betweenness centrality for vertex $v = B_v / B_{max}$.

Betweenness centrality measures are included in the temporary tables that are generated by the HYPERGROUP statement.

### Centroid Centrality

Centroid centrality is different from the other measures of centralities because it uses information of the layout. The X coordinate of the centroid is the sum of X coordinates of other vertices, divided by the number of vertices. The Y coordinate is calculated the same way. The centroid centrality for each vertex is represented as the polar coordinates from the centroid to the vertex and consists of an angle and a magnitude.

Centroid centrality measures are included in the temporary tables that are generated by the HYPERGROUP statement.

## Result Tables

### Overview

The HYPERGROUP statement generates temporary tables that described the results of the analysis. Because they are temporary tables, they exist in memory until your program crosses a RUN boundary like the QUIT statement, a new PROC statement, or a DATA statement.

The following code example uses the same data set that is shown in "Simple Syntax" on page 176.

```
proc imstat data=example.sales;
  hypergroup customer dealer / vars=(model);  1
run;

  table example.&_TEMPLAST_;  2
  fetch / format;
run;

  table example.&_TEMPHYPGRP_;  3
  fetch / format;
run;

  table example.&_TEMPEDGES_;  4
  fetch / format;
run;
```

1 Because the analysis variables, customer and dealer, are specified with the simple syntax, the HYPERGROUP statement considers them both as vertices. The statement does not differentiate between customers and dealers that might have the same value. The VARS= option copies the model variable to the _TEMPLAST_ table.

2 The _TEMPLAST_ temporary table is set as the active table. The FETCH statement prints the first 20 rows from the table and formats the variables.The _TEMPLAST_ on page 179 temporary table is set as the active table.

3 This is identical to the previous description except that the _TEMPHYPGRP_ temporary table is set as the active table.

4 This is identical to the previous description except that the _TEMPEDGES_ temporary table is set as the active table.

*Note:* The example shows the FETCH statement with the FORMAT option. This improves the readability of the results but converts numeric variables to characters.

If you want to save the tables to the SAS client and preserve the numeric columns, you can use syntax that is similar to **FETCH / OUT=*libref*.HYPGROUPS;**. If you want to analyze the tables with clients like SAS Visual Analytics, then use the PROMOTE statement to make it a permanent table. For metadata-aware applications like SAS Visual Analytics, you also need to register the table in SAS metadata.

### The _TEMPLAST_ Table

The HYPERGROUP statement generates this table by default. The key features of this table are as follows:

- For each row in the active table that is analyzed, subject to a WHERE clause, there is a row in the _TEMPLAST_ table.
- The _HypGrp_ variable identifies the hypergroup number (0, 1, 2, and so on) for analysis variables.
- The analysis variables are included in the table as well as any variables that you specify in the VARS= option.

The following example output shows how the _HypGrp_ variable identifies the data as two disjoint groups.

***Figure 4.2*** *Sample _TEMPLAST_ Table*

| Selected Records from Table _T_BDDBBAB1_10DE6460 | | | |
|---|---|---|---|
| _HypGrp_ | Customer | Dealer | Model |
| 0 | Tina | AutoEmporium | Focus |
| 0 | Tony | AutoEmporium | Sonic |
| 0 | Tom | Prestige | TT |
| 0 | Tom | AutoEmporium | ATS |
| 1 | Blake | AutoMall | Accord |
| 1 | Bob | AutoMall | RAV4 |
| 1 | Bart | AutoMall | Civic |
| 1 | Beth | AutoMall | Accord |

### The _TEMPHYPGRP_ Table

The HYPERGROUP statement generates this table by default. The table includes records that are related to the values of the hypergroup variables. The key features of this table are as follows:

- The _Value_ variable identifies the values of the hypergroup variables. These are the graph vertices names.
- The _Index_ variable identifies each vertex index.
- The _HypGrp_ variable identifies the hypergroup number for the vertex.
- The _IndexH_ variable identifies a vertex index within a hypergroup subgraph.
- The _XCoord_ and _YCoord_ variables identify the coordinates of the vertex.
- The _Color_ variable is the index of a strongly connected component found by the graph partitioning algorithm.

The following display shows the output for the sample data set. Notice that "AutoMall" and "Bart" are both vertices and that they are in the same hypergroup.

***Figure 4.3*** *Sample _TEMPHYPGRP_ Table*

| Selected Records from Table _T_BDDBBADA_10DE5DA0 | | | | | | |
|---|---|---|---|---|---|---|
| _Value_ | _Index_ | _HypGrp_ | _IndexH_ | _XCoord_ | _YCoord_ | _Color_ |
| AutoEmporium | 0 | 0 | 0 | 44.551263029 | 43.477576325 | 0 |
| AutoMall | 1 | 1 | 0 | 50.390746059 | 44.640587013 | 0 |
| Bart | 2 | 1 | 2 | 16.770733384 | 56.903975263 | 0 |
| Beth | 3 | 1 | 1 | 75.929412381 | 19.596621572 | 0 |
| Blake | 4 | 1 | 4 | 49.284334819 | 80.403378428 | 0 |
| Bob | 5 | 1 | 3 | 83.229266616 | 58.819111574 | 0 |
| Prestige | 6 | 0 | 2 | 60.703527865 | 59.372245511 | 0 |
| Tina | 7 | 0 | 4 | 22.964985888 | 72.419689977 | 0 |
| Tom | 8 | 0 | 1 | 77.035014112 | 27.580310023 | 0 |
| Tony | 9 | 0 | 3 | 72.608918148 | 67.309276841 | 0 |

This data contains two hypergroups, one for customers of AutoEmporium and Prestige. The other hypergroup is for customers of AutoMall. AutoEmporium and Prestige are in the same hypergroup because Tom shopped at both dealers. But, apart from Tom, there was no cross-shopping. If, however, even one customer of AutoMall had also purchased from one of the other two dealers, then all vertices would have been connected. This would result in identifying only one hypergroup. In real world data for car buying, there would likely be enough cross-shopping with a region such that only one hypergroup is created. The HYPERGROUP statement provides more sophisticated structural analysis for such scenarios.

### The _TEMPEDGES_ Table

The HYPERGROUP statement generates this table by default. The _TEMPEDGES_ table includes the same variables as the _TEMPLAST_ table, except that instead of having the hypergroup variables (in this example, Customer and Dealer), they are replaced with _Source_ and _Target_ variables. These two columns have values for the origin and destination of edges. The table also includes index and coordinate variables that are associated with the source and target vertices.

The columns in this table have a similar interpretation to those in _TEMPHYPGRP_ table, but there is a column for each of source and target vertices of the edge. (This terminology is a misnomer because the edges are undirected). Source and target do not need to be considered for the hypergroup assignment. Because hypergroups are completely disconnected from each other, edges cannot connect vertices in different hypergroups.

The following display shows the results of the COLUMNINFO statement for the _TEMPEDGES_ table. All of the columns are always included, except that the model

column is included because it was included in the VARS= option to the HYPERGROUP statement.

***Figure 4.4*** *Column Information for the _TEMPEDGES_ Table*

| Column Information for Table _T_90A93D7A_1389A698 | | | | |
|---|---|---|---|---|
| Id | Column | Type | Length | Format |
| 1 | _HypGrp_ | Num | 8 | BEST12. |
| 2 | _Source_ | Char | 12 | $12. |
| 3 | _Target_ | Char | 12 | $12. |
| 4 | _Sindex_ | Num | 8 | BEST12. |
| 5 | _Tindex_ | Num | 8 | BEST12. |
| 6 | _SindexH_ | Num | 8 | BEST12. |
| 7 | _TindexH_ | Num | 8 | BEST12. |
| 8 | _SourceX_ | Num | 8 | BEST12. |
| 9 | _SourceY_ | Num | 8 | BEST12. |
| 10 | _TargetX_ | Num | 8 | BEST12. |
| 11 | _TargetY_ | Num | 8 | BEST12. |
| 12 | _SColor_ | Num | 8 | BEST12. |
| 13 | _TColor_ | Num | 8 | BEST12. |
| 14 | model | Char | 8 | $8. |

The table includes records that are related to the edges between vertices. The key features of this table are as follows:

- The _HypGrp_ variable identifies the hypergroup number for analysis variables.
- The _Source_ and _Target_ variables have the values of the vertices that each edge connects.
- The _Sindex_ and _Tindex_ variables identify the vertex index (0, 1, 2, and so on) for the _Source_ and _Target_ variables.
- The _SindexH_ and _TindexH_ variables identify the vertex index (0, 1, 2, and so on) for the _Source_ and _Target_ variables, within each hypergroup subgraph.
- The _XCoordS_ and _YCoordS_variables identify the coordinates of the source vertex.
- The _XCoordT_ and _YCoordT_ variables identify the coordinates of the target vertex.
- The _SColor_ and _TColor_ variables identify the index to associate with the source and target vertices.

The following display shows the output for the sample data set.

***Figure 4.5*** *Sample _TEMPEDGES_ Table*

| Selected Records from Table _T_BDDBBAC6_10DE6460 | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| _HypGrp_ | _Source_ | _Target_ | _Sindex_ | _Tindex_ | _SindexH_ | _TindexH_ | _SourceX_ | _SourceY_ | _TargetX_ | _TargetY_ | _SColor_ | _TColor_ | Model |
| 0 | Tina | AutoEmporium | 7 | 0 | 4 | 0 | 22.964985888 | 72.419689977 | 44.551263029 | 43.477576325 | 0 | 0 | Focus |
| 0 | Tony | AutoEmporium | 9 | 0 | 3 | 0 | 72.608918148 | 67.309276841 | 44.551263029 | 43.477576325 | 0 | 0 | Sonic |
| 0 | Tom | Prestige | 8 | 6 | 1 | 2 | 77.035014112 | 27.580310023 | 60.703527865 | 59.372245511 | 0 | 0 | TT |
| 0 | Tom | AutoEmporium | 8 | 0 | 1 | 0 | 77.035014112 | 27.580310023 | 44.551263029 | 43.477576325 | 0 | 0 | ATS |
| 1 | Blake | AutoMall | 4 | 1 | 4 | 0 | 49.284334819 | 80.403378428 | 50.390746059 | 44.640587013 | 0 | 0 | Accord |
| 1 | Bob | AutoMall | 5 | 1 | 3 | 0 | 83.229266616 | 58.819111574 | 50.390746059 | 44.640587013 | 0 | 0 | RAV4 |
| 1 | Bart | AutoMall | 2 | 1 | 2 | 0 | 16.770733384 | 56.903975263 | 50.390746059 | 44.640587013 | 0 | 0 | Civic |
| 1 | Beth | AutoMall | 3 | 1 | 1 | 0 | 75.929412381 | 19.596621572 | 50.390746059 | 44.640587013 | 0 | 0 | Accord |

### Additional Tables and Columns in Result Tables

The previous sections describe the temporary table names and columns that are produced by default with the HYPERGROUP statement. The statement offers many options that can add additional columns and additional tables. For example, the preceding sections include columns that are related to assigning color values to the vertices. If you specify the NOCOLOR option, then those columns are not included. The same is true for the NOCOORD option. Specifying NOCOORD eliminates the columns that are related to coordinates and improve processing times because calculating coordinates is a computationally intensive task.

If you specify the CENTRALITY option, then the result tables are affected. For more information, see "Centrality Measures" on page 177.

### *ODS Table Names*

The HYPERGROUP statement generates the following ODS tables.

| ODS Table Name | Description | Option |
|---|---|---|
| HypGrpTables | Temporary hypergroup table names | Default |
| Namespace | Hypergroup namespaces for a table | When analysis variables are specified with the namespace syntax. |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## KDE Statement

The KDE statement calculates kernel-density estimates of the distribution of one or more numeric variables from an in-memory table. You can choose between normal, tricube, and quadratic kernel functions. The default is to use a normal kernel function. The number of points returned are determined by the center region of a multi-threaded, inverse finite Fourier transform.

### Syntax

**KDE** *variable-list* <*/ options*>;

### Required Argument

**variable-list**
specifies a one or more numeric variables.

### KDE Statement Options

**BANDWIDTH=*b***
specifies the standardized bandwidth of the kernel function. The default bandwidths are optimal values that minimize the asymptotic mean integrated squared errors of the kernel function. The actual bandwidth for the kernel estimator is a multiple of the standardized bandwidth, the inter-quartile range of the data, and $n^{-1/5}$. Larger values for bandwidth result in smoother density estimates. However, specifying a bandwidth that is too large can result in density estimates that omit important aspects of the distribution at finer granularity.

**KERNEL= NORMAL | TRICUBE | QUADRATIC**
specifies the kernel function.

**Default** NORMAL

**MAX=*number***
specifies the largest value to consider in the density calculation. If a value is not specified, then the largest value in the data range is used, subject to the WHERE clause.

**Alias** UPPER=

**MIN=*number***
specifies the smallest value to consider in the density calculation. If a value is not specified, the smallest value in the data range is used, subject to the WHERE clause.

**Alias** LOWER=

**MULTIPLER=*number***
specifies a scaling factor for the calculated density.

**Default** 1

**NPOINTS=*n***
specifies the number of points from which to calculate the center region of the inverse finite Fourier transform. The value of the NPOINTS= option is adjusted to the largest integer of power of 2 that is equal to or smaller than *n*. For example, specifying NPOINTS=40 is adjusted to 32. The number of density points returned depends on the distribution of the data.

**Default** 512

**Range** 16 to 512

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SCALE= PERCENT | COUNT | PROPORTION**
specifies the units in which the density is calculated.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### *ODS Table Names*

The KDE statement generates the following ODS table for each analysis variable.

| ODS Table Name | Description | Option |
|---|---|---|
| KDE | Kernel density estimation results | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# LOGISTIC Statement

The LOGISTIC statement can model binary data with logit, probit, log-log, and complementary log-log link functions. It can also model binomial data with the same set of link functions.

## Syntax

**LOGISTIC** *dependent-variable* <(*class-variables*)> = *model-effects* </ *options*>
**LOGISTIC** *event-variable* / *trial-variable* <(*class-variables*)> = *model-effects* </ *options*>

### *Required Arguments*

***dependent-variable***
specifies the variable to model. This variable is also referred to as the response variable.

**TIP** The LOGISTIC statement produces a response profile table that shows the ordered values of the dependent variable. By default, the smaller ordered value is the event that is modeled. For example, if the dependent variable has values 0 and 1, the statement models the probability that the dependent variable takes on the value 0. You can change the ordering of the dependent variable values with the DESCENDING option.

***event-variable***
specifies the name of the variable that indicates the count of positive responses.

***model-effects***
specifies a list of variables to use for modeling the dependent variable.

***trial-variable***
specifies the name of the variable that indicates the total number of trials.

## Optional Argument

***class-variables***
specifies a list of variables to use as classification variables. The variables in this list take the place of the CLASS statement in traditional SAS procedures.

## LOGISTIC Statement Options

**ALLIDVARS**
requests that all variables in the input table are treated as ID variables when a scoring table is produced. In other words, if this option is specified, all variables from the input table, including computed columns, are transferred to the scoring table. This option has no effect unless you specify the SCORE option.

**ALPHA=*number***
specifies a number between 0 and 1 from which to determine the confidence level for approximate confidence intervals of the parameter estimates. The default is α = 0.05, which leads to 100 x (1- α)% = 95% confidence limits for the parameter estimates.

**Default** 0.05

**CI**
specifies to add confidence intervals to the table of parameter estimates. The confidence level is 100*(1-α)% where α is determined by the ALPHA= option. The default value is α = 0.05. This value is equivalent to a 95% confidence limit.

**Default** 0.05

**CLASSFORMATS=("*format-name1*" <, "*format-name2*" …> )**
specifies the formats for the classification variables in the model. If you do not specify the CLASSFORMATS= option, the default format is applied for the classification variable. That default format was determined when the table was originally loaded into the server. In the following example, the CLASSFORMAT= values apply to variables x1 and x2.

**Alias** CLASSFMT=

**Example**
```
logistic y (x1 x2) = x3-x7 / classformats=("YN.", "F8.");
```

**CODE <(*code-generation-options*)>**
requests that the server produce SAS scoring code based on the actions that it performed during the analysis. The server generates DATA step code. By default, the code is replayed as an ODS table by the procedure as part of the output of the statement. More frequently, you might want to write the scoring code to an external file by specifying options.

The scoring code computes the predicted value of the response variable on the data scale (the inverse link scale) and prefixes the name with "P_". For example, if the response variable is **Y**, the generated code stores the predicted value as **P_Y**. The name of the variable is truncated to fit within the SAS name length requirements.

**COMMENT**
specifies to add comments to the code in addition to the header block. The header block is added by default.

**FILENAME='*path*'**
specifies the name of the external file to which the scoring code is written. This suboption applies only to the scoring code itself. If you request that the server generate IMSTAT programming statements with the IMSTAT suboption, then these statements are saved as an ODS table.

**Alias** FILE=

**FORMATWIDTH=*k***
specifies the width to use in formatting derived numbers such as parameter estimates in the scoring code. The server applies the BEST format, and the default format for code generation is BEST20.

**Alias** FMTW=

**Range** 4 to 32

**IMSTAT**
specifies to generate IMSTAT programming statements that reproduce the analysis in addition to the scoring code. For example, this option is helpful when you perform variable selection and you want to capture the modeling code that reflects only the selected variables.

**IMSTATONLY**
specifies to generate the IMSTAT programming statements only. No scoring code is produced.

**LABELID=*id***
specifies a group identifier for group processing. The identifier is an integer and is used to create array names and statement labels in the generated code.

**LINESIZE=*n***
specifies the line size for the generated code.

**Alias** LS=

**Default** 72

**Range** 64 to 256

**NOTRIM**
requests that the comparison of the formatted values for class variables and group-by variables is based on the full format width with padding. By default, the leading and trailing blanks are removed from the formatted values.

**REPLACE**
specifies to overwrite the external file with the new contents if the file already exists. This option has no effect unless you specify the FILENAME= option.

**DESCENDING**
specifies to model the largest ordered value for the dependent variable instead of the smallest. This option is useful for modeling responses with the value of 1 instead of modeling for value 0.

**Alias** DESC

**EXCLUDE=(*list-of-ODS-tables*)**
specifies the result tables that you want to exclude from being generated on the server and from being sent to the SAS session. The GLM statement can generate the following tables:

| Table Name | Table Alias | Description | Condition |
|---|---|---|---|
| ModelInfo | | Information about the model—constant across groups or partitions. | This table is shown by default. |
| ClassLevels | Class | Information about the classification variables, such as the number of levels and their values. | This table is shown when classification variables are present in the model. |
| ConvStatus | Convergence | Convergence status of optimization | This table is shown by default. |
| Dimensions | Dim | Model dimensions | This table is shown by default. |
| FitStatistics | Fit | Fit statistics customary for regression models | This table is shown when it is requested with the SELECT= option. |
| GlobalTest | Global | Test of the hypothesis that the model fits as well as a null model without explanatory variables | This table is shown my default. |
| IterHistory | IterHist | Iteration history | This table is shown when the ITDETAILS option is used or when the table is requested with the SELECT= option. |
| ParmEstimates | ParameterEstimates<br>Pest | The solutions for the linear model coefficients | This table is shown when there are no classification variables in the model. |
| ResponseProfile | Resp | Information about the values of the binary response variable such as the level order and frequency | This table is shown when modeling binary data. (When the events/trials syntax is not used.) |

| Table Name | Table Alias | Description | Condition |
|---|---|---|---|
| Tests3 | | Type III tests of model effects | This table is shown when the effects contain classification variables and the NOSTDERR option is not specified. |

Whether a table is shown by default or not, you can request any table with the SELECT= option in the LOGISTIC statement. The Condition column in the table identifies when a table is produced by default. For example, if the response variable is binary, the server generates the ResponseProfile table.

**FORMATS=("*format-specification*" <,…> )**
specifies the formats for the GROUPBY variables. If you do not specify the FORMATS= option, or if you omit the entry for a GROUPBY variable, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
   statement / groupby=(a b) formats=("8.3", "$10");
quit;
```

**FCONV=*r***
specifies a relative function convergence criterion. For all techniques except NMSIMP, termination requires a small relative change of the function value in successive iterations. Suppose that Ψ is the $p \times 1$ vector of parameter estimates in the optimization, and the objective function at the *k*th iteration is denoted as $f(\Psi)^k$. Then, the FCONV criterion is met if

$$\frac{\left(f\left(\Psi^{(k)}\right) - f\left(\Psi^{(k-1)}\right)\right)}{f\left(\Psi^{(k-1)}\right)} \leq r$$

**Default** $r=10^{-FDIGITS}$ where FDIGITS is $-\log_{10}(e)$ and e is the machine precision.

**FREQ=*variable-name***
specifies the numeric variable that provides frequencies for the analysis. For example, if the FREQ= variable has the value 5, then it implies that the record represents five such observations with identical values for the modeling variables. If you specify a FREQ= variable, then only the observations with a value that is not missing and greater than zero for the variable are used in the analysis.

**GCONV=*r***
specifies a relative gradient convergence criterion. For all optimization techniques except CONGRA and NMSIMP, termination requires that the normalized predicted function reduction is small. The default value is $r$ = 1e-8. Suppose that Ψ is the $p \times 1$ vector of parameter estimates in the optimization with *i*th element $\Psi_i$. The objective function, its $p \times 1$ gradient vector, and its $p \times p$ Hessian matrix are denoted, $f(\Psi)$, $g(\Psi)$, and $H(\Psi)$, respectively. Then, if superscripts denote the iteration count, the normalized predicted function reduction at iteration $k$ is

$$\frac{g\left(\Psi^{(k)}\right)' H\left(\Psi^{(k)}\right)^{-1} g\left(\Psi^{(k)}\right)}{f\left(\Psi^{(k)}\right)}$$

The GCONV convergence criterion is assumed to be met if that value is less than or equal to *r*.

Note that it is possible that the relative gradient reduction is small, even if one or more gradients is still substantial in absolute value. If this situation occurs, you can disable the GCONV criterion by setting *r*=0. If the optimization would have stopped early due to meeting the GCONV criterion, the iterative process usually takes one more step until the gradients are small in absolute value.

**GROUPBY=(*variable-list*)**
specifies the names of the Group-by variables in the order of the grouping hierarchy. If no variable names are specified, the model is fit across the entire table—possibly subject to a WHERE clause.

If you work on a partitioned table, you can also use the PARTITION option to fit the model for a specific partition or separately for all partitions. Operations on partitions are much more efficient than a group-by analysis.

Because fitting logistic models requires an iterative method, the Group-by analysis for these models is a data-parallel technique where the model in each group is fit separately, assigning different rows of the group to different threads.

**GROUPFILTER=(*filter-options*)**
specifies a section of the group-by hierarchy to be included in the computation. With this option, you can request that the server performs the analysis for only a subset of all possible groupings. The subset is determined by applying the group filter to a temporary table that you generate with the GROUPBY statement.

You can specify the following suboptions in the GROUPFILTER option:

**DESCENDING**
specifies the top section or the bottom section of the groupings to be collected. If the DESCENDING option is specified, the top LIMIT=*n* (where $n > 0$) groupings are collected. Otherwise, the bottom LIMIT=*n* groupings are collected.

**Alias** DESC

**LIMIT=*n***
specifies the maximum number of distinct groupings to be collected, where integer $n >= 0$. If *n* is zero, then all distinct groupings (up to $2^{31}-1$) that satisfy the boundary constraints, such as LOWERSCORE=*f*, are collected.

**CAUTION** **High Cardinality Data Sets** Setting *n* to zero with high-cardinality data sets can significantly delay the response of the server.

**SCOREGT=*f***
specifies the exclusive lower bound for the numeric scores of the distinct groupings to collect.

**Alias** SGT=

**SCORELT=*f***
specifies the exclusive upper bound for the numeric scores of the distinct groupings to collect.

**Alias** SLT=

**VALUEGT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive lower bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VGT=

**VALUELT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive upper bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VLT=

**TABLE=*table-with-groupby-results***
specifies the in-memory table from which to load the group-by hierarchy. If the TABLE= option is not specified, then all other GROUPFILTER= options are ignored.

The following program request all the groupings of State, City, and then Trade_In_Model in the Cars_Program_All table. The groupings are ordered by the maximum value of New_Vehicle_Msrp for each grouping:

```
proc imstat;
    table example.cars_program_all;
    groupby state city trade_in_model / temptable
                weight=new_vehicle_msrp
                agg   =(max)
                order =weight;
run;
```

The TEMPTABLE option in the GROUPBY statement directs the server to save all the groupings in a temporary in-memory table. The following DISTINCT statement requests the count of the distinct unformatted values of Sales_Type for each of the selected groupings of State, City, and Trade_In_Model.

```
    table example.cars_program_all;
    distinct sales_type / groupfilter=(
                table  =mylasr.&_TEMPLAST_
                scoregt=40000
                valuelt=("FL","Ft Myers","")
                limit  =20
                descending);
run;
```

This example considers only groupings that have maximum values of the New_Vehicle_Msrp above 40,000 and with formatted values that are less than State="FL" and City="Ft Myers." The empty quotation marks result in no restriction on Trade_In_Model values. These groupings are ordered according to the maximum values of New_Vehicle_Msrp. Because of the DESCENDING option, this example collects the 20 top groupings within the specified group-by range for the DISTINCT analysis.

**Interaction** If you specify the GROUPFILTER= option, then the GROUPBY= and FORMATS= options have no effect.

**IDVARS=(*variable-list*)**
**IDVARS=*variable-name***
specifies the variables from the active table to transfer to the temporary table that is created by scoring the input table. This option has no effect unless the SCORE option is also specified. (See the SCORE option for details about which variables are

added to the temporary table by default.) The IDVARS= option should be used to transfer additional columns from the input table to the scoring table.

**Alias** ID=

**Tip** Instead of this option, you can specify the ALLIDVARS option to transfer all variables from the input table to the scoring table.

**ITDETAILS**
requests to add details about the iterative model fitting process (an iteration history) to the ODS output tables.

**Alias** ITDETAIL

**KEYORDER**
requests that the results for a partitioned analysis are displayed in the order of the partition keys. If this option is not specified, then results are displayed by using the partitions on the first worker node followed by the partitions on the second node, and so on. Without this option, the results are likely to have random ordering of the partitions. The KEYORDER option makes result collection less efficient but produces a natural, predictable order.

**LINK=*function***
specifies the link function to use for the model fitting process. See the following list for the available functions:

- LOGIT
- PROBIT
- LOGLOG
- CLOGLOG

**Default** LOGIT

**MAXFUNC=*n***
specifies the maximum number *n* of function calls in the iterative model fitting process. The default value depends on the optimization technique as follows:

| Optimization Technique | Default Number of Function Calls |
|---|---|
| TRUREG, NRRIDG, and NEWRAP | 125 |
| QUANEW and DBLDOG | 500 |
| CONGRA | 1000 |
| NMSIMP | 3000 |

**Alias** MAXFU=

**MAXITER=*i***
specifies the maximum number of iterations in the iterative model fitting process. The default value depends on the optimization technique as follows:

| Optimization Technique | Default Number of Iterations |
|---|---|
| TRUREG, NRRIDG, and NEWRAP | 50 |
| QUANEW and DBLDOG | 200 |
| CONGRA | 400 |
| NMSIMP | 1000 |

**Alias** MAXIT=

**MAXTESTLEV=*n***
specifies the maximum number of levels in an effect for which the server generates Type III tests. The idea behind the MAXTESTLEV= option is that testing effects for significance that have a large number of levels is typically not meaningful. The effects tend to be highly significant anyway, but determining the exact significance level is computationally intensive. The default value is 300 and implies that no test statistics are produced for any effect that has more than 300 levels.

**Default** 300

**NAME=*SAS-name***
specifies the name to use for identifying the model in the server output and in the temporary table of results generated by the TEMPTABLE option. SAS name rules apply. For example, the following statements add the 'Model' entry to the ModelInformation table.

```
proc imstat;
   table hps.neuralgia;
   logistic pain = treatment sex duration / name = LogisModel
run;
```

| Model Information | |
|---|---|
| Model | LogisModel |
| Data Source | HPS.NEURALGIA |
| Response Variable | Pain |
| Distribution | Binary |
| Link Function | Logit |

**NOCLPRINT <=*n*>**
specifies the number of levels for each classification variables to show in the Class Level Information ODS table. If you do not specify the NOCLPRINT option, all unique values are shown in the order of the class variable levelization. If you specify NOCLPRINT=*n*, then the values are shown for those classification variables that have less than *n* levels only. The value for *n* must be at least 1.

If you specify the NOCLPRINT option without specifying a value for *n*, then $n = 0$ is assumed. This enables you to get a listing of the classification variables in the model.

This might be useful if you did not identify classification variables explicitly—without listing their (possibly many) levels.

For example, the following Class Level Information table is displayed with NOCLPRINT=4. Because the number of levels for variable Smoking_Status exceeds 4, the values are not displayed.

| Class Level Information for Table HPS.HEART | | |
|---|---|---|
| **Class** | **Levels** | **Values** |
| **Weight_Status** | 3 | Normal Overweight Underweight |
| **Smoking_Status** | 5 | not printed |

**NOINT**
suppresses the inclusion of an intercept in the model. By default, all models contain an intercept term.

**NOPREPARSE**
prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NOSTDERR**
prevents the computation of the covariance matrix and the standard errors of the parameter estimates. When you specify this option, the Type III tests for the model effects are also not available.

**Alias** NOSTD

**OFFSET=*variable-name***
specifies the offset variable for the analysis. An offset variable can be thought of as a regressor variable whose regression coefficient is known to be 1. Offsets are used to shift the linear predictors by a certain amount. For example, an offset can be used to accommodate constants in the underlying model. For example, a model for the probability of being seropositive is as follows:

$$\pi = 1 - \exp(-\beta X)$$

After applying the log function, the model on the linear scale is as follows:

$$\log(-\log(1-\pi)) = \log(\beta) + \log(X)$$

You can model this relationship with a complementary log-log link (LINK=CLOGLOG), and the offset variable log(X). The term log(β) is then estimated by the intercept of the model.

**PARTITION <=*partition-key*>**

When you specify this option and the table is partitioned, the results are calculated separately for each value of the partition key. In other words, the partition variables function as automatic GROUPBY variables. This mode of executing calculations by partition is more efficient than using the GROUPBY= option. With a partitioned table, the server takes advantage of knowing that observations for a partition cannot be located on more than one worker node.

If you do not specify a *partition-key*, the analysis is performed for all partitions. If you do specify a *partition-key*, the analysis is carried out for the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid partition key values for a table.

You can specify a *partition-key* in two ways. You can supply a single quoted string that is passed to the server, or you can specify the elements of a composite key separated by commas. For example, if you partition a table by variables GENDER and AGE, with formats $1 and BEST12, respectively, then the composite partition key has a length of 13. You can specify the partition for the 11-year-old females as follows:

```
statement / partition="F          11"; /* passed directly to the server */
statement / partition="F","11";        /* composed by the procedure */
```

If you choose the second format, the procedure composes a key based on formatting information from the server.

**Alias** PART=

**ROLEVAR=*variable-name***

specifies a variable in the in-memory table that defines whether an observation belongs to the training set, the validation set, or is to be excluded from the analysis. The role variable can have a numeric or character type, and it can be a temporary computed variable.

If the role variable data type is numeric, the values of *variable-name* are interpreted as follows:

- value = 1: this observation is in the training set
- value = 2: this observation is in the validation set
- any other value: this observation is to be excluded from the analysis

If the role variable data type is character, the values of *variable-name* are interpreted as follows:

- If the first non-blank character is 't' or 'T', then the observation is in the training set.
- If the first non-blank character is 'v' or 'V', then he observation is in the validation set.
- Any other value for the first non-blank character, including an all blank entry, leads to the exclusion of the observation from the analysis.

**Alias** ROLE=

**Interactions** You can divide the data at random into training and validation sets by providing the VALIDATE= and SEED= options.

If you specify both the ROLEVAR= option and the VALIDATE= options, then the ROLEVAR= setting supersedes the VALIDATE= option.

**SCORE <(*score-statistic1score-statistic2…*)>**
requests that the active table be scored after the model is fit and the results be stored in a temporary table. The server automatically adds all model variables to the temporary table with the score results. These results include the response variable, the class variables, all explanatory variables from which effects are formed, and the WEIGHT=, and FREQ= variables.

In addition, if the active table is partitioned or ordered, the partition variables and order-by variables are transferred from the input table to the temporary table. The temporary table is partitioned and ordered in the same way as the active table.

If the analysis uses the GROUPBY= option, the variables in the group-by list are also transferred to the scoring table. If you want to transfer additional variables, you can specify them with the IDVARS= option.

If you do not specify the list of score statistics, default statistics are computed. These statistics are identified with Yes in the Default column in the table below. You can request that the following statistics be computed for each observation:

| Keyword and Aliases | Column Name | Description | Default |
|---|---|---|---|
| PRED, PREDICTED, LINP | _PRED_ | Predicted linear predictor value | Yes |
| RESID, RESIDUAL, R | _RESID_ | Raw residual (on a linear scale) | Yes |
| LEVERAGE, H | _LEVERAGE_ | Measure of how extreme an observation is in the regressor space | Yes |
| ILINK, MEAN, PROB | _ILINK_ | Inversely linked linear predictor, the predicted mean of the response | Yes |
| PEARSON, RESCHI | _PEARSON_ | Pearson residual, also known as the Chi-square residual | Yes |
| DEVRESID, RESDEV | _DEVRESID_ | Deviance residual | Yes |
| LIKEDIST, LD, RESLIKE | _LIKEDIST_ | Likelihood displacement | Yes |

| Keyword and Aliases | Column Name | Description | Default |
|---|---|---|---|
| STDRES, STDRESCHI | _STDRESCHI_ | Standardized Pearson Chi-square residual | Yes |
| STDP | _STDP_ | Standard error of the mean predicted value | No |
| LCLM, LOWERMEAN | _LCLM_ | Lower confidence limit for the mean of the predicted value | No |
| UCLM, UPPERMEAN | _UCLM_ | Upper confidence limit for the mean of the predicted value | No |
| LCL, LOWERPRED | _LCL_ | Lower confidence limit for the predicted value | No |
| UCL, UPPERPRED | _UCL_ | Upper confidence limit for the predicted value | No |
| DIFDEV | _DIFDEV_ | Change in the deviance due to the deletion of the observation | No |
| DIFCHISQ | _DIFCHISQ_ | Change in the Pearson statistic due to deletion of the observation | No |

The calculation of these statistics agrees with the LOGISTIC procedure. You can consult the documentation for the procedure in the *SAS/STAT User's Guide* for details about the calculations. The output statistics of the LOGISTIC procedure referred to as PREDICTED, LOWER, and UPPER are equivalent to the _ILINK_, _LCLM_, and _UCLM_ statistics.

If you specify SCORE(_ALL_), then the server calculates and adds all the possible output statistics to the temporary table. The confidence levels for the LCLM, LCL, UCLM, and UCL confidence bounds are determined from the significance level specified in the ALPHA= option.

The interpretation of the LCL/UCL and LCLM/UCLM bounds is slightly different in the LOGISTIC statement as compared to the GLM statement. In a GLM model, the distinction between LCLM and LCL bounds is that the former apply to predictions of the mean (expected value) of an observation, whereas the latter apply to prediction of a new observation that has not been used in modeling the data. In the LOGISTIC statement all confidence bounds are bounds for predicting an expected value. The difference between LCLM/UCLM and LCL/UCL bounds here relates to whether the

bound applies to the mean scale (the data scale), or the scale of the linear predictor. The LCLM and UCLM bounds apply to the mean (=data) scale—these are confidence bounds for the predicted probability. The LCL/UCL bounds apply to the linear scale—these are confidence bounds for the linear predictor η.

**SELECT=(*list-of-ODS-tables*)**
specifies the list of ODS tables that you want to display for the analysis. The specified list replaces the default tables that are generated by the server and displayed. See the EXCLUDE= option for the list of default tables and the table names that you can display.

**SHOWSELECTED**
requests that the server perform variable selection for the model. A backward selection method is used, where the significance level for an effect to remain in the model is determined by the SLSTAY= option. This option performs variable selection like the VARSEL option, but in contrast to the latter option, it displays output only for the selected effects.

**Alias** SHOWSEL

**SLSTAY=*α***
specifies the significance level used in determining whether effects should stay in the model during variable selection.

**Default** 0.1

**Range** 0 to 1

**TECHNIQUE=**
specifies the optimization technique.

Valid values are as follows:

| | |
|---|---|
| CONGRA (CG) | performs a conjugate-gradient optimization. |
| DBLDOG (DD) | performs a version of the double-dogleg optimization. |
| DUQUANEW (DQN) | performs a (dual) quasi-Newton optimization. |
| NMSIMP (NS) | performs a Nelder-Mead simplex optimization. |
| NONE | specifies not to perform any optimization. This value can be used to perform a grid search without optimization. |
| NEWRAP (NRA) | performs a (modified) Newton-Raphson optimization that combines a line-search algorithm with ridging. |
| NRRIDG (NRR) | performs a (modified) Newton-Raphson optimization with ridging. |
| QUANEW (QN) | performs a quasi-Newton optimization. If you specify this technique, but specify bounds for any parameter, the server automatically performs DUQUANEW. |
| TRUREG (TR) | performs a trust-region optimization. |

The factors that go into choosing a particular optimization technique for a particular problem are complex. Trial and error can be involved. For many optimization problems, computing the gradient takes more computer time than computing the function value. Computing the Hessian sometimes takes much more computer time and memory than computing the gradient, especially when there are many

parameters. Unfortunately, first-order optimization techniques that do not use some type of Hessian or Hessian approximation usually require more iterations than second-order techniques that use a Hessian matrix. As a result, the total run time of first-order techniques can be longer. Techniques that do not use the Hessian also tend to be less reliable. For example, they can terminate more easily at stationary points than at global optima.

The TRUREG, NEWRAP, and NRRIDG algorithms are second-order algorithms.

The server computes first and second derivatives of the objective function with respect to the parameters in analytic form wherever possible. Finite-difference approximations for derivatives are used only when the derivatives of functions are not known. In most cases, finite-difference approximations are not necessary.

For more information about the algorithms, see *SAS/STAT User's Guide*.

**Alias** TECH=

**Default** NRRIDG

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**Interaction** For information about the interaction between the TEMPTABLE, CODE, and SCORE options, see "Temporary Tables, Generated Code, and Scoring" on page 253.

**VALIDATE=*f***
specifies the proportion *f* in the validation data set.

**Alias** VALPROP=

**Range** 0 to 1

**Interaction** If you specify both the ROLEVAR= option and the VALIDATE= option, then the ROLEVAR= setting supersedes the VALIDATE= option.

**VARSELECTION**
specifies that the server perform variable selection for the model. A backward selection method is used, where the significance level for an effect to remain in the model is determined by the SLSTAY= option. In contrast to the SHOWSEL option, all effects are reported in the IMSTAT output.

**Alias** VARSEL

**WEIGHT=*variable-name***
specifies the numeric variable to use as a weighing variable in solving the linear model.

## Details

### *ODS Table Names*

The ODS tables that can be generated with the GLM statement are described in the "EXCLUDE=(*list-of-ODS-tables*)" on page 188.

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# MDSUMMARY Statement

The MDSUMMARY statement calculates a multi-dimensional summary for numeric variables.

## Syntax

**MDSUMMARY** *variable-list* </ <*set-specification,…*> *options*>;

### *Optional Arguments*

***variable-list***
specifies one or more numeric variables. If you do not specify this option, then all numeric variables in the table are used.

**set-specification, …**
specifies the three elements for generating a set. Separate each set-specification with a comma.

**GROUPBY=*variable-name***
**GROUPBY=(*variable-list*)**
specifies the list of GROUP BY variables for this set-specification. A GROUPBY= specification is required.

**FORMATS=("*format-specification*",…)**
specifies the formats for the GROUPBY= variables. If you do not specify the FORMAT= option, the default format is applied for that variable. Enclose each format specification in quotation marks and separate each format specification with a comma.

You can omit the assignment of a format for a GROUPBY= variable by entering an empty string. For example, **`FORMATS=("$10.", "", "BEST4.")`** specifies to format the first variable, with **`$10`** and the third variable with **`BEST4.`**. The default format is applied to the second variable. The FORMATS= element of the set-specification is optional.

**FILTER="*expression*"**
specifies an optional WHERE clause for this set-specification. The filter is applied separately for each set and possibly in addition to an overall WHERE clause.

### *MDSUMMARY Statement Options*

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**GROUPBYLIMIT=*n***
specifies the maximum number of levels in a GROUPBY set. When the software determines that there are at least *n* levels in the GROUPBY set, it abandons the action, returns a message, and does not produce a result set. You can specify the GROUPBYLIMIT= option if you want to avoid creating excessively large result sets in GROUPBY operations.

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**NOPREPARSE**
prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**RAWORDER**
specifies that the ordering of the GROUPBY variables is based on the raw values of the variables instead of the formatted values.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports

the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the &_TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

## Details

### *ODS Table Names*

The MDSUMMARY statement generates the following ODS table for each set specification.

| ODS Table Name | Description | Option |
|---|---|---|
| Summary | Descriptive statistics | Default |
| TempTable | Information about a temporary table | TEMPTABLE |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## NEURAL Statement

The NEURAL statement trains feed-forward artificial neural networks (ANN) and can use the trained networks to score data sets. When you do not specify a target variable, the server encodes the input nodes for the purpose of dimensionality reduction.

**Examples:** "Example 15: Training and Validating a Neural Network" on page 280
"Example 16: Predicting Email Spam and Assessing the Model" on page 284

### Syntax

**NEURAL** <*target-variable*> <*/ options*>;

#### *Optional Argument*

***target-variable***
specifies the variable to model. If you do not specify a variable, then the server encodes the input nodes for the purpose of variable reduction.

#### *NEURAL Statement Options*

**ACTIVATION=(*activation-function-for-a-hidden-layer …*)**
specifies the activation function for the neurons on each hidden layer. The following functions are available:

IDENTITY specifies the identify function. Values range between -∞ and ∞. For an input value of $t$, the function returns the same value, $t$.

LOGISTIC specifies the logistic function. Values range between zero and one. For an input value of $t$, the function returns $\frac{1}{1+e^{-t}}$.

EXP specifies the exponential function. Values range between zero and ∞. For an input value of $t$, the function returns $e^t$.

SIN specifies the sine function. Values range between zero and 1, inclusively. For an input value of $t$, the function returns $\sin(t)$.

TANH specifies the hyperbolic tangent function. Values range between –1 and 1. For an input value of $t$, the function returns $\tanh(t) = 1 - \frac{2}{1+e^{2t}}$.

**Aliases** ACT=
ACTIVATE=

**Default** TANH

**ARCHITECTURE= GLIM | MLP | DIRECT**
specifies the network architecture to be trained. The GLIM architecture (Generalized Linear Model) specifies a two-layer perceptron (one is the input layer and the other is the output layer) without hidden layers or units. The MLP architecture specifies a

multilayer perceptron with one or more hidden layers. The DIRECT architecture is an extension of MLP with direct connections between the input layer and the output layer.

**Alias** ARCH=

**Default** GLIM.

**ASSESS**
specifies that predicted probabilities are added to the scored data for all the levels of the nominal target variable when scoring is performed with the TEMPTABLE and LASRANN= options. It adds variables _NN_Level_ and _NN_P_ to the results. You can use these predicted probabilities in an ASSESS statement.

**Interaction** You must specify the TEMPTABLE and LASRANN= options along with this option.

**BIAS=*r***
specifies a fixed bias value for all the hidden and output neurons. In this case, the bias parameters are fixed and are not optimized.

**CODE <(*code-generation-options*)>**
requests that the server produce SAS scoring code based on the actions that it performed during the analysis. The server generates DATA step code. By default, the code is replayed as an ODS table by the procedure as part of the output of the statement. Frequently, you might want to write the scoring code to an external file by specifying options.

**COMMENT**
specifies to add comments to the code in addition to the header block. The header block is added by default.

**FILENAME='*path*'**
specifies the name of the external file to which the scoring code is written. This suboption applies only to the scoring code itself.

**Alias** FILE=

**Interaction** If you do not specify this option, then the scoring code is displayed in an ODS result table.

**FORMATWIDTH=*k***
specifies the width to use in formatting derived numbers such as parameter estimates in the scoring code. The server applies the BEST format, and the default format for code generation is BEST20.

**Alias** FMTW=

**Range** 12 to 32

**LABELID=*id***
specifies a group identifier for group processing. The identifier is an integer and is used to create array names and statement labels in the generated code.

**LINESIZE=*n***
specifies the line size for the generated code.

**Alias** LS=

**Default** 72

**Range** 64 to 256

**NOTRIM**
specifies to format the variables using the full format width with padding. By default, leading and trailing blanks are removed from the formatted values.

**REPLACE**
specifies to overwrite the external file if a file with the specified name already exists. The option has no effect unless you specify the FILENAME= option.

**COMBINATION=(*combination-function-for-a-hidden-layer …*)**
specifies the combination function for the neurons on each hidden layer. The available functions are described in the following list. When the COMBINATION= option is not specified, the hidden units use the LINEAR function.

ADD adds all the incoming values without using any weights or biases. The function is defined as $\sum_i x_i$.

LINEAR uses a linear combination of the incoming values and weights. The function is defined as $\text{bias}_j + \sum_i w_{\text{ij}} x_i$

RADIAL uses a radial basis function with equal heights and unequal widths for all units in the layer. The function is defined as $\text{bias}_j^2 \cdot \left(w_{\text{ij}} - x_i\right)^2$

**DELTA=*r***
specifies the annealing parameter when performing a simulated annealing (SA) global optimization. Without the DELTA= option, the step size option (STEP=) and the temperature option (T=) are used to perform a Monte Carlo (MC) global optimization. With the addition of the DELTA= option, the optimization becomes simulated annealing where the temperature is scaled by DELTA × T at every MC step.

**DETAILS**
specifies to display the convergence status and iteration results for training the network.

**Alias** DETAIL

**ERROR=*error-function***
specifies the error function to train the network. For interval variables, the default error function is NORMAL. For nominal variables, the default error function is ENTROPY. The available functions are as follows:

GAMMA uses the gamma distribution. The values of the target variable must be greater than zero.

ENTROPY uses the cross or relative entropy for independent targets. The values of the target variable must be between zero and one.

NORMAL uses the normal distribution. The target variable can have any value.

POISSON uses the Poisson distribution. The values of the target variable must be greater than or equal to zero.

**FCONV=*r***
specifies a relative function convergence criterion.

**Default** $1e^{-5}$

**FORMATS=("*format-specification*", …)**
specifies the formats for the input variables. If you do not specify the FORMATS= option, the default format is applied for that variable. Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
    neural x / input=(a b) formats=("8.3", "$10");
quit;
```

**GCONV=*r***
specifies a relative gradient convergence criterion.

**Default** $1e^{-5}$

**Interaction** When TECH=NSIMP, there is no default value.

**HIDDEN=(*positive-integer* …)**
specifies the number of hidden neurons for each hidden layer in the feed-forward artificial neural network model. For example, **`HIDDEN=(5 3)`** specifies two hidden layers. The first hidden layer has five hidden neurons and the second has three hidden neurons.

**Alias** HIDDENS=

**Interaction** When the HIDDEN= option is specified, the default architecture is MLP.

**IMPUTE**
specifies to impute the output values with available target values when data is being scored. In this case, you are interested only in predicting the missing values of the target variable. This option can be used only when scoring is performed with the TEMPTABLE and LASRANN= options.

**INPUT=*variable-name***
**INPUT=(*variable-name1 <variable-name2*, …> )**
specifies the input variables for the network. You can add the target variable to the input list if you want to assign a format to the target variable by using the FORMATS= option. The number of input neurons on the input layer is determined by the number of input variables. The number of input variables equals the total number of levels from the nominal variables plus the number of interval variables.

**LAMBDA=*λ***
specifies the weight decay number. The value must be zero or greater.

**Default** 0

**LASRANN=*table-name***
specifies the table that contains the weights from a previously trained model. When the RESUME option is used with the LASRANN= option, training for that model resumes using the previously obtained weights as the new starting weights. Otherwise, the weights are used to score the active table.

**Alias** ANNLASR=

**LINESEARCH=*i***
specifies the line-search method for the CONGRA and QUANEW optimization techniques.

**Default** 2

**Range** An integer between 1 and 8.

**LISTNODE= ALL | INPUT | OUTPUT | HIDDEN**
specifies the nodes to be included in the temporary table that is generated when you score a table with the NEURAL statement. When encoding of the input nodes is requested, the default is HIDDEN. This option is particularly useful when encoding is applied to reduce the dimension of the input nodes. By reusing the node output values, machine learning algorithms in the NEURAL, CLUSTER, DECISIONTREE, and RANDOMWOODS statements can use the newly encoded vectors as input.

ALL specifies to include all the nodes in the temporary table.

HIDDEN specifies to include the hidden nodes only.

INPUT specifies to include the input nodes only.

OUTPUT specifies to include the output nodes only.

**LOWER=*r***
specifies a lower bound for the network weights.

**Default** –10.0

**MAXFUNC=*n***
specifies the maximum number *n* of function calls in the iterative model fitting process. The default value depends on the optimization technique as follows:

| Optimization Technique | Default Number of Function Calls |
|---|---|
| TRUREG, NRRIDG, and NEWRAP | 125 |
| QUANEW and DBLDOG | 500 |
| CONGRA | 1000 |
| NMSIMP | 3000 |
| LBFGS | Number of iterations × 10 |

**Alias** MAXFU=

**MAXITER=*i***
specifies the maximum number of iterations in the model fitting process. The default value is 10 for all techniques.

**MISSING= MIN | MEAN | MAX**
specifies how to impute missing values for the input or target variables. When the MISSING= option is not specified, the observations with missing values are ignored. The MIN request imputes missing values for each variable with the minimum value. Similarly, the MAX request imputes the maximum value, and the MEAN request

imputes the mean value. For nominal variables, a new missing category is created for the missing values.

**MULTINET=*i***
specifies the number of networks to select out of the number of tries specified in the NUMTRIES= option. The networks with the smallest errors are selected as the set of best networks. When data is scored, the most frequent predicted values among the selected networks are used to make the final predictions. This option is required when performing Monte Carlo or simulated annealing optimizations. Those optimizations also use the DELTA=, STEP=, and T= options.

**NOPREPARSE**
prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NOBIAS**
specifies no bias parameters for the hidden and output units.

**NOMINAL=*variable-name***
**NOMINAL=(*variable-list*)**
specifies the numeric variables to use as nominal variables. For the nominal input variables, neurons are created for every level. The values are coded as 0 or 1 indicator variables. Character variables are nominal and do not need to be included in the list.

**NUMTRIES=*i***
specifies the number of optimizations to perform with different weight initializations when training networks. The network with the smallest error is selected as the best network. This option is required when performing Monte Carlo or simulated annealing global optimizations that also use the DELTA=, STEP=, and T= options.

**Default** 1

**RESUME**
specifies to resume a training optimization and use the weights that were obtained from previous training. The initial weights for resuming the optimization are read from a temporary table with the LASRANN= option. The specified framework of model options must be the same as the previous framework.

**Interaction** The RESUME option cannot be used with the MULTINET= option.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within

the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SCOREDATA=*table-name***
specifies the in-memory table that contains the scoring data. The table must exist in-memory on the server. The NEURAL statement in the IMSTAT procedure does not transfer a local data set to the server.

**SEED=*number***
specifies the random number seed to use for generating random numbers. The random numbers are used to initialize the network weights.

**STD= MIDRANGE | NONE | STD**
specifies the standardization to use on the input interval variables.

| | |
|---|---|
| MIDRANGE | Variables are scaled to a midrange of 0 and a half-range of 1. |
| NONE | Variables are not altered. |
| STD | Variables are scaled to a mean of 0 and a standard deviation of 1. |

**Default** STD

**STEP=*r***
specifies a step size for perturbations on the network weights when performing Monte Carlo or simulated annealing global optimizations.

**Default** 0.01

**T=*r***
specifies the artificial temperature parameter when performing Monte Carlo or simulated annealing global optimizations.

**Default** 1000

**TARGETACT=*activation-function-for-target-variable***
specifies the activation function for the neurons on the output layer. The available functions are IDENTITY, LOGISTIC, EXP, SIN, TANH, and SOFTMAX. The definitions of these functions are described in the "ACTIVATION=(*activation-function-for-a-hidden-layer* …)" on page 203 option. The SOFTMAX function is unique to this option and is described as follows:

SOFTMAX performs the multiple logistic function. Values range between zero and one. For an input value of t, the function returns

$$\frac{e^t}{\sum_j \text{exponentials}}$$

When the TARGETACT= option is not specified, the SOFTMAX function is used for nominal variables, and the IDENTITY function is used for interval variables. When the target variable is not provided for the purpose of encoding the input nodes, the SOFTMAX function is used.

**Alias** TARACT=

**TARGETCOMB=*combination-function-for-target-variable***
specifies the combination function for the neurons on the target output nodes. The available functions are ADD, LINEAR, and RADIAL. The definitions of these functions are described in the "COMBINATION=(*combination-function-for-a-hidden-layer* …)" on page 205 option.

**Alias** TARCOMB=

**Default** LINEAR

**TECHNIQUE=*value***
specifies the optimization technique for the iterative model-fitting process. The valid values are as follows:

| | |
|---|---|
| CONGRA (CG) | performs a conjugate-gradient optimization. |
| DBLDOG (DD) | performs a version of the double-dogleg optimization. |
| LBFGS | performs a limited-memory Broyden–Fletcher–Goldfarb–Shanno optimization. |
| NMSIMP (NS) | performs a Nelder-Mead simplex optimization. |
| QUANEW (QN) | performs a quasi-Newton optimization. |
| TRUREG (TR) | performs a trust-region optimization. |

The factors that go into choosing a particular optimization technique for a particular problem are complex. Trial and error can be involved.

**Alias** TECH=

**Default** LBFGS

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the &_TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**TIMEOUT=*seconds***
specifies the maximum number of seconds that the server should run the statement. If the time-out is reached, the server terminates the request and generates an error and error message. By default, there is no time-out.

**UPPER=*r***
specifies an upper bound for the network weights.

**Default** 10.0

**VARS=*variable-name***
**VARS=(*variable-name1* <, *variable-name2*, …> )**
specifies the names of the variables to transfer from the active table to a temporary table that contains the scoring results. This option is ignored unless you score an in-memory table and the TEMPTABLE option is specified. The observations with these variables are copied to the generated temporary table.

**Alias** IDVARS=

**WEIGHT=*variable-name***
specifies a variable to weight the prediction errors (the difference between the output of the network value and the target value specified in the input data set) for each observation during training.

## Details

### *ODS Table Names*

The NEURAL statement generates the following ODS tables.

| ODS Table Name | Description | Option |
|---|---|---|
| ANNWeightInfo | Parameter estimates | Default, not shown when TEMPTABLE is specified. |
| CodeGen | Generated DATA step code for scoring | CODE |
| ConvergenceStatus | Convergence status of optimization | DETAILS |
| ModelInfo | Model information | Default |
| OptIterHistory | Iteration history | DETAILS |
| ScoreInfo | Score information | LASRANN= |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# OPTIMIZE Statement

The OPTIMIZE statement performs a non-linear optimization of an objective function that is defined through a SAS program. The expression defined in the SAS program and its analytic first and second derivatives are compiled into executable code. The code is then executed in multiple threads against the data in an in-memory table. Like all other IMSTAT statements, the calculations are performed by the server. You can choose from several first-order and second-order optimization algorithms.

## Syntax

**OPTIMIZE** <*options*>;

### OPTIMIZE Statement Options

**ALPHA=*number***
specifies a number between 0 and 1 from which to determine the confidence level for approximate confidence intervals of the parameter estimates. The default is α = 0.05, which leads to 100 x (1- α)% = 95% confidence limits for the parameter estimates.

**Default** 0.05

**BOUNDS=(*boundary-specification* <, *boundary-specification*,…> )**
specifies boundary values for the parameters. A boundary-specification is specified in the following form:

*parameter-name operator value*

**parameter-name**
specifies the parameter

**operator**
is one of **>=**, **GE**, **<=**, **LE**, **>**, **GT**, **<**, **LT**, **=**, **EQ**.

**value**
specifies the boundary value

**Alias** BOUND=

**Example** BOUNDS=(s2 > 0, beta2 >= 0.2)

**CODE=*file-reference***
specifies a file reference to the SAS program that defines the objective function. The program must make an assignment to the reserved symbol **_OBJFNC_**. The server then minimizes the negative of that function (or maximize the function). In other words, you should specify **_OBJFNC_** to be the function that you want to maximize across the in-memory table. The actual optimization is carried out as a minimization problem.

**Alias** PGM=

**DEFSTART=*value***
specifies the default starting value for parameters whose starting value has not been specified. The default value, 1, might not work well depending on the optimization.

**Alias** DEFVAL=

**Default** 1

**DUD**
specifies that you do not want to use analytic derivatives in the optimization. The option name is an acronym for "do not use derivatives." Instead, the server calculates gradient vectors and Hessian matrices from finite difference approximations. Generally, you should not rely on derivatives calculated from finite differences if analytic derivatives are available. However, this option is useful in situations where the objective function is not calculated independently for each row of data. If derivatives of the objective function depend on lagged values, which are themselves functions of the parameters, then finite difference derivatives are called for.

**Alias** NODERIVATIVES

**FCONV=*r***

specifies a relative function convergence criterion. For all techniques except NMSIMP, termination requires a small relative change of the function value in successive iterations. Suppose that Ψ is the $p \times 1$ vector of parameter estimates in the optimization, and the objective function at the *k*th iteration is denoted as $f(\Psi)^k$. Then, the FCONV criterion is met if

$$\frac{\left(f\left(\Psi^{(k)}\right) - f\left(\Psi^{(k-1)}\right)\right)}{f\left(\Psi^{(k-1)}\right)} \leq r$$

**Default** $r=10^{-FDIGITS}$ where FDIGITS is $-\log_{10}(e)$ and e is the machine precision.

**GCONV=*r***

specifies a relative gradient convergence criterion. For all optimization techniques except CONGRA and NMSIMP, termination requires that the normalized predicted function reduction is small. The default value is $r$ = 1e-8. Suppose that Ψ is the $p \times 1$ vector of parameter estimates in the optimization with *i*th element $\Psi_i$. The objective function, its $p \times 1$ gradient vector, and its $p \times p$ Hessian matrix are denoted, $f(\Psi)$, $g(\Psi)$, and $H(\Psi)$, respectively. Then, if superscripts denote the iteration count, the normalized predicted function reduction at iteration $k$ is

$$\frac{g\left(\Psi^{(k)}\right)' H\left(\Psi^{(k)}\right)^{-1} g\left(\Psi^{(k)}\right)}{f\left(\Psi^{(k)}\right)}$$

The GCONV convergence criterion is assumed to be met if that value is less than or equal to $r$.

Note that it is possible that the relative gradient reduction is small, even if one or more gradients is still substantial in absolute value. If this situation occurs, you can disable the GCONV criterion by setting $r$=0. If the optimization would have stopped early due to meeting the GCONV criterion, the iterative process usually takes one more step until the gradients are small in absolute value.

**ITDETAIL**

requests that the server produce an iteration history table for the optimization. This table displays the objective function, its absolute change, and the largest absolute gradient across the iterations.

**MAXFUNC=*n***

specifies the maximum number *n* of function calls in the iterative model fitting process. The default value depends on the optimization technique as follows:

| Optimization Technique | Default Number of Function Calls |
|---|---|
| TRUREG, NRRIDG, and NEWRAP | 125 |
| QUANEW and DBLDOG | 500 |
| CONGRA | 1000 |
| NMSIMP | 3000 |

**Alias** MAXFU=

**MAXITER=*i***
specifies the maximum number of iterations in the iterative model fitting process. The default value depends on the optimization technique as follows:

| Optimization Technique | Default Number of Iterations |
|---|---|
| TRUREG, NRRIDG, and NEWRAP | 50 |
| QUANEW and DBLDOG | 200 |
| CONGRA | 400 |
| NMSIMP | 1000 |

**Alias** MAXIT=

**MAXTIME=*t***
specifies an upper limit of *t* seconds of CPU time for the optimization process. The default value is the largest floating-point double representation value for the hardware used by the SAS LASR Analytic Server. Note that the time specified by the MAXTIME= option is checked only once at the end of each iteration. The time is measured on the root node for the server. Therefore, the actual running time can be longer than the value specified by the MAXTIME= option.

**MINITER=*i***
specifies the minimum number of iterations.

**Alias** MINIT=

**Default** 0

**NBEST=*k***
requests that only the *k* best points in the starting value grid are reproduced in the "Starting Values" table. By default, the objective function is initially evaluated at all points in the starting value grid and the "Starting Values" table contains one row for each point on the grid. If you specify the NBEST= option, then only the *k* points with the smallest objective function value are shown.

**Alias** BEST=

**NOEMPTY**
requests that result sets for optimizations without usable data are not generated.

**NOPREPARSE**
specifies to prevent pre-parsing and pre-generating the program code that is referenced in the CODE= option. If you know the code is correct, you can specify this option to save resources. The code is always parsed by the server, but you might get more detailed error messages when the procedure parses the code rather than the server. The server assumes that the code is correct. If the code fails to compile, the server indicates that it could not parse the code, but not where the error occurred.

**Alias** NOPREP

**NOSTDERR**

specifies to prevent calculating standard errors of the parameter estimates. The calculation of standard errors requires the derivation of the Hessian or cross-product Jacobian. If you do not want standard errors, *p*-values, or confidence intervals for the parameter estimates, then specifying this option saves computing resources.

**Alias** NOSTD

**PARAMETERS=(*parameter-specification* <, *parameter-specification…*> )**

specifies the parameters in the optimization and the starting values. You do not have to specify parameters and you do not have to specify starting values. If you omit the starting values, the default starting value is assigned. This default value is 1.0 and can be modified with the DEFSTART= option.

If you do not specify the parameter names, the server assumes that all symbols in your SAS program are parameters if they do not match column names in the in-memory table or are not special or temporary symbols. This might not be what you want and you should examine the "Starting Values" and "Parameter Estimates" table in that case to make sure that the server designated the appropriate quantities as parameters in the optimization.

In the first example that follows, Intercept is assigned a starting value of 6. The remaining parameters start at 0 because the DEFSTART= option is 0.

In the second example that follows, the server evaluates the objective function initially for the Cartesian product set of all the parameter vectors. The server evaluates $1 \times 3 \times 2 \times 1 = 6$ parameter vectors. The optimization then starts from the vector associated with the best objective function value.

**Alias** PARMS=

**Examples** DEFSTART=0; PARMS=(Intercept = 6, a_0, b_0, c_0, x_1, x_2, x_3);

PARMS=(beta1 = -3.22, beta2 = 0.5 0.47 0.6, beta3 = -2.45 -2.0, s2 = 0.5);

**RESTRICT=(*one-restriction* <, *one-restriction*> )**

specifies linear equality and inequality constraints for the optimization. A single restriction takes on the general form

```
coefficient parameter ... coefficient parameter operator value
```

Inequality restrictions are expressed as constraints greater than (>) or greater than or equal (>=) than the right hand side value.

The first example that follows shows the restriction $\beta_1 - 2\,\beta_2 > 3$.

The second example that follows shows how to use more than one restriction. Restrictions are separated by commas and the second example requests that the estimates for parameters dose1 and dose2 are the same, as well as the estimates for logd1 and logd2.

**Examples** RESTRICT=(1 beta1 -2 beta2 > 3)

RESTRICT=(1 dose1 -1 dose2 = 0, 1 logd1 -1 logd2 = 0)

**SAVE=*table-name***

saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within

the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**

requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**TECHNIQUE=**

specifies the optimization technique.

Valid values are as follows:

| | |
|---|---|
| CONGRA (CG) | performs a conjugate-gradient optimization. |
| DBLDOG (DD) | performs a version of the double-dogleg optimization. |
| DUQUANEW (DQN) | performs a (dual) quasi-Newton optimization. |
| NMSIMP (NS) | performs a Nelder-Mead simplex optimization. |
| NONE | specifies not to perform any optimization. This value can be used to perform a grid search without optimization. |
| NEWRAP (NRA) | performs a (modified) Newton-Raphson optimization that combines a line-search algorithm with ridging. |
| NRRIDG (NRR) | performs a (modified) Newton-Raphson optimization with ridging. |
| QUANEW (QN) | performs a quasi-Newton optimization. |
| TRUREG (TR) | performs a trust-region optimization. |

The factors that go into choosing a particular optimization technique for a particular problem are complex. Trial and error can be involved. For many optimization problems, computing the gradient takes more computer time than computing the function value. Computing the Hessian sometimes takes much more computer time and memory than computing the gradient, especially when there are many parameters. Unfortunately, first-order optimization techniques that do not use some type of Hessian or Hessian approximation usually require more iterations than second-order techniques that use a Hessian matrix. As a result, the total run time of first-order techniques can be longer. Techniques that do not use the Hessian also tend to be less reliable. For example, they can terminate more easily at stationary points than at global optima.

The TRUREG, NEWRAP, and NRRIDG algorithms are second-order algorithms.

The server computes first and second derivatives of the objective function with respect to the parameters in analytic form wherever possible. Finite-difference approximations for derivatives are used only when the derivatives of functions are not known. In most cases, finite-difference approximations are not necessary.

For more information about the algorithms, see *SAS/STAT User's Guide*.

**Alias** TECH=

**Default** DUQUANEW

## Details

### *ODS Table Names*

The OPTIMIZE statement generates the following ODS tables.

| ODS Table Name | Description | Option |
|---|---|---|
| OptParameters | Starting values for optimization | Default |
| OptIterHistory | Iteration history | ITDETAILS |
| ConvergenceStatus | Convergence status | Default |
| OptFitStatistics | Fit statistics | Default |
| OptParameterEstimates | Parameter estimates for optimization | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# PERCENTILE Statement

The PERCENTILE statement computes percentiles for one or more numeric variables.

**Examples:** "Example 1: Calculating Percentiles and Quartiles" on page 255
"Example 8: Storing Temporary Variables" on page 372

## Syntax

**PERCENTILE** <*variable-list*> <*/ options*>;

### *Optional Argument*

***variable-list***
specifies one or more numeric variables. If you do not specify this option, then all numeric variables in the table are used.

### PERCENTILE Statement Options

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**EPSILON=*number***
specifies the tolerance used for determining the convergence of the iterative algorithm for the percentile calculation.

**Alias** EPS=

**Default** 1e-5

**FORMATS=("*format-specification*",…)**
specifies the formats for the GROUPBY= variables. If you do not specify the FORMATS= option, or if you omit the entry for a GROUPBY variable, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
   percentile x / groupby=(a b) formats=("8.3", "$10");
quit;
```

**GROUPBY=(*variable-list*)**
specifies a list of variable names, or a single variable name, to use as GROUPBY variables in the order of the grouping hierarchy. If you do not specify any GROUPBY variable names, then the calculation is performed across the entire table —possibly subject to a WHERE clause.

**GROUPBYLIMIT=*n***
specifies the maximum number of levels in a GROUPBY set. When the software determines that there are at least *n* levels in the GROUPBY set, it abandons the action, returns a message, and does not produce a result set. You can specify the GROUPBYLIMIT= option if you want to avoid creating excessively large result sets in GROUPBY operations.

**MAXITER=*i***
specifies the maximum number of iterations for the algorithm. The percentile algorithm is iterative. You can limit the number of iterations with the MAXITER= option. You can also control the computational demand with the EPSILON= option. That option affects the tolerance criterion by which the convergence of the iterative algorithm is judged. Whether the percentile calculation has converged is displayed separately for each of the calculated percentiles.

**Alias** ITER=

**Default** 10

**MERGEBINS=*b***
specifies the number of bins to create when a numeric GROUPBY variable exceeds the MERGELIMIT=*n* specification. If you specify a MERGELIMIT, but do not specify a value for the MERGEBINS= option, the server automatically calculates the number of bins.

**MERGELIMIT=*n***
specifies that when the number of unique values in a numeric GROUPBY variable exceeds *n*, the variable is automatically binned and the GROUPBY structure is determined based on the binned values of the variable, rather than the unique formatted values.

For example, if you specify MERGELIMIT=500, any numeric GROUPBY variable with more than 500 unique formatted values is binned. Instead of returning results for more than 500 groups, the results are returned for the bins. You can specify the number of bins with the MERGEBINS= option.

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**NOPREPARSE**
prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NOTEMPPART**
specifies that the temporary table generated by the TEMPTABLE option is not partitioned by the GROUPBY= variables. When you request a temporary table with the PERCENTILE statement, by default, the server partitions the table and the size of a partition is equal to the number of analysis variables in the *variable-list* of the PERCENTILE statement. When the number of groups is large, this can result in many small partitions, and requires extra memory resources to store the partition information for the temporary table. By specifying this option, the temporary table is organized similarly to the default table, but is not a partitioned table.

**Alias** NOTP

**PARTITION <=*partition-key*>**
When you specify this option and the table is partitioned, the results are calculated separately for each value of the partition key. In other words, the partition variables function as automatic GROUPBY variables. This mode of executing calculations by partition is more efficient than using the GROUPBY= option. With a partitioned table, the server takes advantage of knowing that observations for a partition cannot be located on more than one worker node.

If you do not specify a *partition-key*, the analysis is performed for all partitions. If you do specify a *partition-key*, the analysis is carried out for the specified key value

only. You can use the PARTITIONINFO statement to retrieve the valid partition key values for a table.

You can specify a *partition-key* in two ways. You can supply a single quoted string that is passed to the server, or you can specify the elements of a composite key separated by commas. For example, if you partition a table by variables GENDER and AGE, with formats $1 and BEST12, respectively, then the composite partition key has a length of 13. You can specify the partition for the 11-year-old females as follows:

```
statement / partition="F          11"; /* passed directly to the server */
statement / partition="F","11";        /* composed by the procedure */
```

If you choose the second format, the procedure composes a key based on formatting information from the server.

**Alias** PART=

**RAWORDER**
specifies that the ordering of the GROUPBY variables is based on the raw values of the variables instead of the formatted values.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the &_TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**VALUES=(*percentiles*)**
specifies the values for which to calculate the percentiles. The default is to calculate the 25th, 50th, and 75th percentile. These are also known as the first, second, and third quartile. The second quartile is the median.

**Alias** VALS=

**Range** 0 to 100

**Example** The following statement requests the 10th, 20th, …., 90th percentile:
```
percentile invoice / values=(10 20 30 40 50 60 70 80 90);
```

## Details

### *ODS Table Names*

The PERCENTILE statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| Percentiles | Quantiles and percentiles | Default |
| TempTable | Information about a temporary table | TEMPTABLE |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# RANDOMWOODS Statement

The RANDOMWOODS statement builds a random forest of decision trees. Each tree is constructed from a bootstrap sample of the data, drawn with replacement, and is constructed from only a subset of the variables specified in the INPUT= option.

## Syntax

**RANDOMWOODS** *target-variable* </ *options*>;

### Required Argument

***target-variable***
specifies a single column in the in-memory table as the target variable. The variable can be a temporary calculated column.

### RANDOMWOODS Statement Options

**ADDTREES**
requests that the temporary table that is generated by scoring a random forest is enhanced with information about the votes of the individual trees. The process of scoring a random forest means that each tree votes on the predicted value and the predicted value for the forest is obtained by majority vote. This option adds the votes for each tree in the forest. By default, only the overall vote is reported in the temporary table.

**ASSESS**
specifies that predicted probabilities are added to the temporary result table for the event levels. You can use these predicted probabilities in an ASSESS statement.

**BOOTSTRAP=*f***
specifies the fraction of the data in the bootstrap sample.

**Default** $f = 1 - \exp(-1)$

**Range** 0 to 1

**CODE <(*code-generation-options*)>**
requests that the server produce SAS scoring code based on the actions that it performed during the analysis. The server generates DATA step code. By default, the code is replayed as an ODS table by the procedure as part of the output of the statement. More frequently, you might want to write the scoring code to an external file by specifying options.

The scoring code computes the predicted value of the response variable on the data scale (the inverse link scale) and prefixes the name with "RF_". For example, if the response variable is `Y`, the generated code stores the predicted value as `RF_Y`. The name of the variable is truncated to fit within the SAS name length requirements.

**COMMENT**
specifies to add comments to the code in addition to the header block. The header block is added by default.

**FILENAME='*path*'**
specifies the name of the external file to which the scoring code is written. This suboption applies only to the scoring code itself.

**Alias** FILE=

**FORMATWIDTH=*k***
specifies the width to use in formatting derived numbers such as parameter estimates in the scoring code. The server applies the BEST format, and the default format for code generation is BEST20.

**Alias** FMTW=

**Range** 4 to 32

**LABELID=*id***
specifies a group identifier for group processing. The identifier is an integer and is used to create array names and statement labels in the generated code.

**LINESIZE=*n***
specifies the line size for the generated code.

| | |
|---|---|
| **Alias** | LS= |
| **Default** | 72 |
| **Range** | 64 to 256 |

**NOTRIM**
specifies to format the variables using the full format width with padding. By default, leading and trailing blanks are removed from the formatted values.

**REPLACE**
specifies to overwrite the external file if a file with the specified name already exists. The option has no effect unless you specify the FILENAME= option.

**EVENT=("*event1*" <, "*event2*"> …)**
specifies the event names of the target variable. This option is combined with the WEIGHT= option to specify the weight for each specific event. Observations with the specified event are reweighted with the value from the WEIGHT= option. This option is useful for rare-event sampling.

**FORMATS=("*format-specification*",…)**
specifies the formats for the input variables. If you do not specify the FORMATS= option, the default format is applied for that variable. Enclose each format specification in quotation marks and separate each format specification with a comma.

**GAIN**
specifies that the splitting criterion is changed to information gain. Typically, this criterion intends to generate trees with more nodes than information gain ratio.

**GREEDY**
specifies how to perform splitting under specific circumstances.

Assuming that one variable has $q$ levels, when binary splitting is performed and $q$ is less than 15, or option MAXBRANCH > 2 and $q < 12$, all possible binary splits are enumerated and the split with the largest gain or gain ratio is chosen for the variable.

When $q$ is less than 1024 and splitting is not just binary, local greedy searches are applied to determine the optimum local split. Specifically, when the variable is numeric, $q$ levels (similar to $q$ bins) are sorted by value.

When the variable is nominal, the $q$ levels are ordered by random weights. The best binary splitting is applied until the desired number of branches is reached. Only a local optimum can be found with this technique.

For values of $q \geq 1024$, the default $k$-means clustering algorithm is applied to determine the splits.

**IMPUTE**
specifies how to treat observations with nonmissing values for the target variable during scoring. When this option is specified, the observed values are used as the predicted values. That is, the observed value is assumed to be known without error. Only the observations with missing values for the target variable are then scored against the random forest, based on their values for the input variables.

This option is useful if you want to replace missing values of a target variable with classified values that are based on the random forest.

**INPUT=*variable-name***
**INPUT=(*variable-list*)**
specifies the variables to use for building the tree. You can add the target variable to the input list if you want to assign a format to the target variable by using the FORMATS= option. Any numeric variable that is not specified in the NOMINAL= option is binned according to the NBINS= specification.

In random forest implementations, all of the input variables do not participate in the construction of the trees. Each tree is built from a subset of the input variables. You can use the M= option to affect the selection of these input variables.

**LEAFSIZE=*m***
specifies the minimal number of observations on each node. When the number of observations on a tree node falls short of the specified leaf size *m*, the node is changed into a leaf during the building of the tree.

**Interaction** Specifying the LEAFSIZE option affects the pruning of the tree.

**M=*k***
specifies the number of input variables used to build a tree. The *k* variables are selected at random from the list of input variables for each tree. If not specified, then *k* defaults to the square root of the number of input variables, rounded up to the nearest integer.

**MAXBRANCH=*n***
specifies the maximum number of children (branches) allowed for each level of the tree.

**Default** 2

**MAXLEVEL=*n***
specifies the maximum number of tree levels.

**Default** 6

**NBINS=*k***
specifies the number of bins used in the calculation of the tree. The number of bins affects the accuracy of the tree and increases with *k* at the expense of computing time and memory consumption.

**Default** 2

**NBINSTARGET=*k***
specifies the number of bins to use for a numeric target variable. The number of bins affects the accuracy of the tree. The accuracy increases as values of *k* increase. However, computing time and memory consumption also increase as values of *k* increase. When *k* is greater than zero, the numeric target variable is binned into equally sized bins first and then the bins are used to perform the classification.

**Default** 0

**NOERROR**
specifies that the out-of-bag error is not computed when building a random decision forest. This option is useful to speed up the building process.

**NOMINAL=*variable-name***
**NOMINAL=(*variable-list*)**
specifies the numeric variables to use as nominal variables. Binning is not applied to the specified variables. The target variable is always treated as a nominal variable and does not need to be listed.

**NOMISSOBS**
specifies to ignore observations that have missing values in the analysis variables when building a decision tree. When scoring a data set, any observations with missing values in the analysis variables for the decision tree are ignored when this option is specified.

When this option is not specified, the RANDOMWOODS statement builds a tree by applying the following policy for missing values:

- For an interval variable, the smallest machine value is assigned.
- For a nominal variable, missing values are represented by a separate level.

**NOPREPARSE**
prevents pre-parsing and pre-generating the program code that is referenced in the CODE= option. If you know the code is correct, you can specify this option to save resources. The code is always parsed by the server, but you might get more detailed error messages when the procedure parses the code rather than the server. The server assumes that the code is correct. If the code fails to compile, the server indicates that it could not parse the code, but not where the error occurred.

**Alias** NOPREP

**NTREE=*n***
specifies the number of trees to build for the random forest.

**Default** 1

**REG**
specifies to build the random decision forest using regression trees. Minimal cost-complexity pruning is applied to prune the trees.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SCOREDATA=*table-name***
specifies the in-memory table that contains the scoring data. The table must exist in-memory on the server. The RANDOMWOODS statement in the IMSTAT procedure does not transfer a local data set to the server.

If you do not specify a table name for this option, the active table is used as the scoring input.

**SEED=*s***
specifies the random number seed for the random number generator in the server. The default value, zero, implies that the random number stream is based on the computer clock. Negative seed values also lead to random number streams that are based on the computer clock. If you want a reproducible random number sequence between runs, specify a value that is greater than zero.

**Default** 0

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
specifies to save the results of the RANDOMWOODS statement in a temporary table.

If you build a random forest, the temporary table contains information about the forest.

If you score a random forest, the temporary table contains the predicted values for the observations in the input table that you scored. The temporary table also contains the variables that you specified to transfer from the input table and other statistics. This option is required when you perform scoring so that you can access the predictions for each observation. The IMSTAT procedure then displays the name of the table and stores it in the _TEMPSCORE_ macro variable, provided that the scoring action was successful. Observations from the table that you scored can be transferred to the temporary table using the VARS= option.

**TIMEOUT=*s***
specifies the maximum number of seconds that the statement should run in the server. If the computation does not complete before the time-out is reached, execution stops and the server generates an error message. There is no default time-out.

**TREEINFO**
specifies to display information about individual tress, when you build a tree. For example, the table that is shown can display which variables are used in each tree. The option has no effect if you store the tree in a temporary table.

**TREELASR=*table-name***
specifies the in-memory table that contains the information for the random forest if you want to score a table against the random forest.

The data set whose observations are to be scored is specified in the SCOREDATA= option. If you do not specify the SCOREDATA= option, the active table is used as scoring input.

**Alias** LASRTREE=

**VARS=*variable-name***
**VARS=(*variable-name1* <, *variable-name2*, …> )**
specifies the variables to transfer from the input table to the temporary table in the server that contains the results of scoring a decision tree. This option has no effect unless you specify the TEMPTABLE option and you score a decision tree.

**WEIGHT=**
specifies the weight for each corresponding event in the EVENT= option.

## Details

### ODS Table Names

The RANDOMWOODS statement generates the following ODS tables.

| ODS Table Name | Description | Option |
|---|---|---|
| ForestInfo | Basic information about a random forest | Default |
| GeneratedCode | Generated SAS code from modeling task | CODE= |
| ForestVarImpInfo | Variable importance in a random forest | Default |
| ForestScoreInfo | Basic information about scoring in a random forest | TREELASR= |
| TreeInfo | Basic information about trees in a random forest | TREEINFO |
| TempTable | Information about a temporary table | TEMPTABLE |

The information in the ForestVarImpInfo table is similar to the DTreeVarImpInfo table that is produced with the STAT option to the DESCISIONTREE statement. The difference is that the RANDOMWOODS statement uses multiple trees and the server computes the reduction from all trees.

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# REGCORR Statement

The REGCORR statement calculates and returns the results for linear, quadratic, or cubic polynomial regression models.

**Example:** "Example 11: Fitting a Regression Model" on page 271

## Syntax

**REGCORR** <*variable-list*> <*/ options*>;

### Optional Argument

***variable-list***

specifies one or more numeric variables. If you do not specify this option, then all numeric variables in the table are used.

### REGCORR Statement Options

**NBEST=*n***

specifies that results are returned only for the *n* regression with the highest R-square value (the highest coefficient of determination) . If *n* is smaller than the number of regressions computed by the statement, then the actual number of computed regression is returned.

**ORDER=1 | 2 | 3**
**ORDER=-1 | –2 | –3**

specifies the highest polynomial degree in the regression model. By default, ORDER=1, and the model is a simple linear regression. Specify ORDER=2 for a quadratic model and ORDER=3 for a cubic model.

If you specify a negative value for the ORDER= option, the server evaluates the best model for each variable combination based on statistical principles. For example, if you specify ORDER=–2, the server returns results for a linear regression provided that the removal of the quadratic term does not result in a poorer model—as judged statistically. Similarly, with ORDER=–3, you might get results for a cubic, quadratic, or a linear regression. The results depend on which model is deemed to fit best. The evaluation of the models is done by the same rules that apply for the backward selection method in the REG procedure—that is, coefficients that are not significant at the 0.1 significance level are removed. Furthermore, if a higher-order term remains in the model, the lower-order polynomials are not being evaluated (for example, if the quadratic term is needed, the server does not try to remove the linear term).

**Default** 1

**SAVE=*table-name***

saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***

specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**

specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**VARWITH**

if this option is specified, the *variable-list* of size *k* is interpreted to consist of one response variable and *k*–1 regressors. Otherwise, the *variable-list* is used to compute all pairs of regressions where the response variable cycles through the left-hand side of the list. For example, if *variable-list* is **`x1`**, **`x2`**, **`x3`**, and **`x4`**, then the REGCORR statement computes the following regression models:

| Response Variable | Regressor Variable |
|---|---|
| x1 | x2 |
| x1 | x3 |
| x1 | x4 |
| x2 | x3 |
| x2 | x4 |
| x3 | x4 |

If the VARWITH option is specified, the list of regressions models changes as follows:

| Response Variable | Regressor Variable |
|---|---|
| x1 | x2 |
| x1 | x3 |
| x1 | x4 |

## Details

### *ODS Table Names*

The REGCORR statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| Regression | Linear Regression | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# SUMMARY Statement

The SUMMARY statement is used to calculate descriptive statistics such as the sample mean, sample variance, number of observations, sum of squares, and so on. If you specify one or more variables in the GROUPBY= option, the results are produced separately for each combination of the GROUPBY variables.

**Examples:** "Example 1: Partitioning a Table into a Temporary Table" on page 362
"Example 4: Deleting Rows and Saving a Table to HDFS" on page 368

## Syntax

**SUMMARY** <*variable-list*> </ *options*>;

### Optional Argument

***variable-list***
specifies one or more numeric variables. If you do not specify this option, then all numeric variables in the table are used.

### SUMMARY Statement Options

**AGGREGATE=(*aggregation-methods*)**
lists the aggregator on which the ordering of the result set is based.

The available aggregation methods are as follows:

| | |
|---|---|
| CSS | corrected sum of squares |
| CV | coefficient of variation |
| MAX | maximum value |
| MEAN | arithmetic mean |
| MIN | minimum value |
| N | number of observations |
| NMISS | number of missing observations |
| PROBT | *p*-value for the t-statistic |
| STD | standard deviation |
| STDERR | standard error |
| SUM | sum of the nonmissing values |
| TSTAT | t-statistic for the null hypothesis that the mean equals zero |
| USS | uncorrected sum of squares |
| VAR | sample variance |

**Alias** AGG=

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**FORMATS=("*format-specification*",…)**
specifies the formats for the GROUPBY= variables. If you do not specify the FORMATS= option, or if you omit the entry for a GROUPBY variable, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
   summary x*y / groupby=(a b) formats=("8.3", "$10");
quit;
```

**GROUPBY=(*variable-list*)**
specifies a list of variable names, or a single variable name, to use as GROUPBY variables in the order of the grouping hierarchy. If you do not specify any GROUPBY variable names, then the calculation is performed across the entire table —possibly subject to a WHERE clause.

**GROUPBYLIMIT=*n***
specifies the maximum number of levels in a GROUPBY set. When the software determines that there are at least *n* levels in the GROUPBY set, it abandons the action, returns a message, and does not produce a result set. You can specify the GROUPBYLIMIT= option if you want to avoid creating excessively large result sets in GROUPBY operations.

**GROUPFILTER=(*filter-options*)**
specifies a section of the group-by hierarchy to be included in the computation. With this option, you can request that the server performs the analysis for only a subset of all possible groupings. The subset is determined by applying the group filter to a temporary table that you generate with the GROUPBY statement.

You can specify the following suboptions in the GROUPFILTER option:

**DESCENDING**
specifies the top section or the bottom section of the groupings to be collected. If the DESCENDING option is specified, the top LIMIT=*n* (where $n > 0$) groupings are collected. Otherwise, the bottom LIMIT=*n* groupings are collected.

**Alias** DESC

**LIMIT=*n***
specifies the maximum number of distinct groupings to be collected, where integer $n >= 0$. If *n* is zero, then all distinct groupings (up to $2^{31}-1$) that satisfy the boundary constraints, such as LOWERSCORE=*f*, are collected.

**CAUTION** **High Cardinality Data Sets** Setting *n* to zero with high-cardinality data sets can significantly delay the response of the server.

**SCOREGT=*f***
specifies the exclusive lower bound for the numeric scores of the distinct groupings to collect.

**Alias** SGT=

**SCORELT=*f***
specifies the exclusive upper bound for the numeric scores of the distinct groupings to collect.

**Alias** SLT=

**VALUEGT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive lower bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VGT=

**VALUELT=("*format-name1*" <, "*format-name2*" …> )**
specifies the exclusive upper bound of the group-by variable's formatted values for the distinct groupings to collect.

**Alias** VLT=

**TABLE=*table-with-groupby-results***
specifies the in-memory table from which to load the group-by hierarchy. If the TABLE= option is not specified, then all other GROUPFILTER= options are ignored.

The following program request all the groupings of State, City, and then Trade_In_Model in the Cars_Program_All table. The groupings are ordered by the maximum value of New_Vehicle_Msrp for each grouping:

```
proc imstat;
    table example.cars_program_all;
    groupby state city trade_in_model / temptable
                weight=new_vehicle_msrp
                agg   =(max)
                order =weight;
run;
```

The TEMPTABLE option in the GROUPBY statement directs the server to save all the groupings in a temporary in-memory table. The following DISTINCT statement requests the count of the distinct unformatted values of Sales_Type for each of the selected groupings of State, City, and Trade_In_Model.

```
    table example.cars_program_all;
    distinct sales_type / groupfilter=(
                table  =mylasr.&_TEMPLAST_
                scoregt=40000
                valuelt=("FL","Ft Myers","")
                limit  =20
                descending);
run;
```

This example considers only groupings that have maximum values of the New_Vehicle_Msrp above 40,000 and with formatted values that are less than State="FL" and City="Ft Myers." The empty quotation marks result in no restriction on Trade_In_Model values. These groupings are ordered according to the maximum values of New_Vehicle_Msrp. Because of the DESCENDING option, this example collects the 20 top groupings within the specified group-by range for the DISTINCT analysis.

**Interaction** If you specify the GROUPFILTER= option, then the GROUPBY= and FORMATS= options have no effect.

**LIMIT=*n***
limits the size of the result set returned to the SAS client. For example, the following SUMMARY statement returns the size (in number of records) of the largest partition for Table1.

Because of the PARTITION option, the rows are processed by partition and the summary request returns one row per partition. The ORDERBY= specification requests that the results are ordered by the variable Amount, and that the result is sorted in descending order with respect to the number of observations (AGGREGATE=N). Once the results are arranged this way, the first observation in the set is returned (LIMIT=1).

**Example**

```
proc imstat data=mylasr.Table1;
    summary Amount / partition orderby=(Amount) desc
                     aggregate=(N) limit=1;
run;
```

**MERGEBINS=*b***
specifies the number of bins to create when a numeric GROUPBY variable exceeds the MERGELIMIT=*n* specification. If you specify a MERGELIMIT, but do not specify a value for the MERGEBINS= option, the server automatically calculates the number of bins.

**MERGELIMIT=*n***
specifies that when the number of unique values in a numeric GROUPBY variable exceeds *n*, the variable is automatically binned and the GROUPBY structure is determined based on the binned values of the variable, rather than the unique formatted values.

For example, if you specify MERGELIMIT=500, any numeric GROUPBY variable with more than 500 unique formatted values is binned. Instead of returning results for more than 500 groups, the results are returned for the bins. You can specify the number of bins with the MERGEBINS= option.

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**NOPREPARSE**
prevents the procedure from preparsing and pregenerating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to preparse and pregenerate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from preparsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NOTEMPPART**
specifies that the temporary table generated by the TEMPTABLE option is not partitioned by the GROUPBY= variables. When you request a temporary table with the SUMMARY statement, by default, the server partitions the table and the size of a partition is equal to the number of analysis variables in the *variable-list* of the SUMMARY statement. When the number of groups is large, this can result in many small partitions, and requires extra memory resources to store the partition information for the temporary table. By specifying this option, the temporary table is organized similarly to the default table, but is not a partitioned table.

**Alias** NOTP

**ORDERBY=(*variable-list*)**
specifies the variables to use for ordering the result set. If a variable is not one of the numeric variables in the *variable-list* for the SUMMARY statement, it is assumed to be one of the GROUPBY variables.

**ORDERBYDESC**

specifies the sort order for the result set. The default is ascending order. Specify the ORDERBYDESC option to sort in descending order. Note that this option is different from setting the DESCENDING option. The DESCENDING option affects the order of the values for the GROUPBY variables.

**PARTITION <=*partition-key*>**

When you specify this option and the table is partitioned, the results are calculated separately for each value of the partition key. In other words, the partition variables function as automatic GROUPBY variables. This mode of executing calculations by partition is more efficient than using the GROUPBY= option. With a partitioned table, the server takes advantage of knowing that observations for a partition cannot be located on more than one worker node.

If you do not specify a *partition-key*, the analysis is performed for all partitions. If you do specify a *partition-key*, the analysis is carried out for the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid partition key values for a table.

You can specify a *partition-key* in two ways. You can supply a single quoted string that is passed to the server, or you can specify the elements of a composite key separated by commas. For example, if you partition a table by variables GENDER and AGE, with formats $1 and BEST12, respectively, then the composite partition key has a length of 13. You can specify the partition for the 11-year-old females as follows:

```
statement / partition="F          11"; /* passed directly to the server */
statement / partition="F","11";        /* composed by the procedure */
```

If you choose the second format, the procedure composes a key based on formatting information from the server.

**Alias** PART=

**RAWORDER**

specifies that the ordering of the GROUPBY variables is based on the raw values of the variables instead of the formatted values.

**SAVE=*table-name***

saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**

requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated

number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the &_TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

**Interaction** The TEMPTABLE option requires a group-by analysis or a partitioned analysis with this statement.

**TWELVEBIN**
specifies to augment the summary results with a 12-bin histogram. This option has no effect when the summaries are computed in GROUPBY or partitioned mode.

## Details

### *ODS Table Names*

The SUMMARY statement generates the following ODS tables.

| ODS Table Name | Description | Option |
|---|---|---|
| Summary | Descriptive Statistics | Default, when TEMPTABLE is not specified |
| TempTable | Information about a temporary table | TEMPTABLE |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## TEXTPARSE Statement

The TEXTPARSE statement performs text analytics on the active in-memory table. You can separate the documents in the table into terms, derive topics based on weighted term frequencies, and project the active table onto the latent space defined by the topic-discovered topics.

**See:** For backround information, see Chapter 16, “Text Analytics in SAS LASR Analytic Server,” on page 465.

### Syntax

**TEXTPARSE** TXT=*text-variable* ID=*document-ID* <*options*>;

#### *Required Arguments*

**TXT=*text-variable***
specifies the name of the variable that contains the text to analyze.

**Alias** VARIABLE=

**ID=*document-ID***
specifies the name of the variable that identifies the documents in the table uniquely. The values are typically a row number or other value that identifies the rows. The document ID is important to perform joins of the result tables.

**Alias** DOCID=

#### *TEXTPARSE Statement Options*

**CELLWGT= NONE | LOG**
specifies how elements in the term × document matrix are weighted. Elements in the matrix are assigned weight $w_i * g(f_{ij})$, where $w_i$ is the term weight for the *i*th term, $f_{ij}$ is the frequency of appearance of this term in document *j*.

If CELLWGT=LOG, then $g(f_{ij}) = \log_2(f_{ij}+1)$. The logarithmic function tempers the influence of very frequent terms.

**Default** LOG

**ENTITIES= NONE | STD**
determines whether the entity extractor should use the standard list of entities. When ENTITIES=STD, entity extraction is enabled and standard entities are used. Terms such as "George W. Bush" are then recognized as an entity and given the corresponding entity role and attribute. For this example, the entity role is PERSON and the attribute is Entity. Although the entity is treated as the single term, "george w bush," the individual tokens "george," "w," and "bush" are also included.

**Default** NONE

**EXACTWEIGHT**
specifies not to round the weights that are aggregated during topic derivation. By default, the calculated weights are rounded to the nearest .001.

**Alias** NOWTRND

**KEEP=(*variable-list*)**
**KEEP=*variable-name***
specifies one or more variables to transfer from the input data to the temporary table with the document projection. You can use _ALL_ for all variables, _NUMERIC_ for all numeric variables, and other valid variable list names. By default, only the document ID (ID=) is transferred to the projected document table so that it can be used to join with the active table.

**LANGUAGE="*name*"**
specify a single language to use for the input data. Values are as follows:

- "chinese" (This value applies to simplified and traditional Chinese.)
- "dutch"
- "english"
- "finnish"
- "french"
- "german"
- "italian"
- "japanese"
- "korean"
- "portuguese"
- "russian"
- "spanish"
- "turkish"

**Default** "english"

**NONOUNGROUPS**
specifies not to use the noun group extractor. By default, the server extracts noun groups and returns maximal groups and subgroups (which do not include groups that contain determiners or prepositions). If stemming is turned on, then noun group elements are also stemmed.

**Alias** NONG

**NOSTEMMING**
specifies not to stem words. By default, words are stemmed and terms such as "advises" and "advising" are mapped to the parent term "advise."

**Alias** NOSTEM

**NOTAGGING**
specifies not to tag terms. By default, terms are tagged and the server identifies a term's part of speech based on context clues. The identified part of speech is provided in the Role variable of the TERMS table.

**NUMLABELS=*n***
specifies the number of terms to use in labeling a topic. By default, the n = 5 terms with the largest weight are used in constructing a label for the topic.

**Alias** NLABELS=

**Default** 5

**REDUCEF=*n***
specifies the minimum document frequency of terms. By default, n = 4 and implies that a term is not kept for analysis unless it occurs in at least four documents.

**Default** 4

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SELECT <=> (*list-of-temporary-tables*)**
specifies the results the server should store as temporary tables. By default, the server generates the Terms table, which contains terms, their parent-child relationships, and weights. If you specify the NUMTOPICS= option, the server also generates the Topics table. You can specify **`SELECT=(_ALL_)`** to generate all of the tables.

The possible values for the list specification are shown in the following table:

| Table Name | Table Alias | Description |
|---|---|---|
| TERMS | TERM | Contains summary information about the terms in the document collection. |
| TERMDOC | BAGOFWORDS<br>BOW<br>PARENT<br>PARENTS | Contains a compressed representation of the sparse term-by-document frequency matrix in transactional style. The matrix is represented as a set of (row, column, value) triples. |
| V | SVDV | Contains the V matrix of the singular-value decomposition. |
| U | SUDV | Contains the rotated U matrix of the singular-value decomposition. |

| Table Name | Table Alias | Description |
|---|---|---|
| PROJECTION | DOCPRO<br>PROJ | Contains the projections of the columns of the term-by-document frequency matrix onto the columns of U. Because each column of the term-by-document frequency matrix corresponds to a document, the output forms a new representation of the input documents in a space with much lower dimensionality. |
| TOPICS | | Contains the topics and a label constructed from the most highly weighted terms. This is typically a small table, as the number of topics is limited by *k*, the value of the singular-value decomposition or by the value specified in the NUMTOPICS= option. |
| TERMTOPICS | TERMBYTOPICS | A sparse representation of the terms by topic using the term ID and topic ID. This table might be useful in joins involving terms or topics. |

For information about the tables, see "Output Tables for the TEXTPARSE Statement" on page 468.

**START=*table-name***
specifies the name of the in-memory table that contains the terms that are to be kept for the analysis. These terms are displayed in the Terms result table with a keep status of "Y." The START= table must have variable that is named Term and can also have a variable that is named Role.

**Alias** STARTLIST=

**Interaction** If you specify both the START= option and the STOP= option, the STOP= specification takes precedence.

**STOP=*table-name***
specifies the name of the in-memory table that contains the terms to exclude from the analysis. The STOP= table must contain a variable that is named Term and can also have a variable that is named Role.

**Alias** STOPLIST=

**Interaction** If you specify both the START= option and the STOP= option, the STOP= specification takes precedence.

**SVD(*singular-value-decomposition-options*)**
specifies how to perform the singular-value decomposition (SVD). The server carries out this decomposition whenever you request a result table that depends on topics, or if you request to save the V or U matrix of the decomposition. You can specify the following SVD options inside the parentheses:

**K=*k***
specifies the number of dimensions to be extracted by SVD. This number is equal to the number of topics for topic generation. If you specify the TOPICS= (NUMTOPICS=) option, then the value of *k* is automatically set to match the value given in the TOPICS= option.

If the value of *k* is too large, then the server might process for an unnecessarily long time.

**Default** If you request topic generation and do not specify the K= or MAXK= option, then *k* = 10.

**Interaction** If you specify both the K= and MAXK= options, the K= option takes precedence.

**MAXK=*m***
specifies the maximum value that the server should return as the recommended value of *m*. If the RESOLUTION= option is specified to recommend the value of *k*, then this option limits that value to at most *m*. The HPTMINE procedure attempts to calculate (as opposed to recommends) *k* dimensions when it performs the singular-value decomposition.

**Interaction** If you specify both the K= and MAXK= options, the K= option takes precedence.

**RESOLUTION=LOW | MED | HIGH**
specifies the recommended number of dimensions (resolution) for the singular value decomposition. If you specify this option, you must also specify the MAXK= option. A low-resolution singular value decomposition returns fewer dimensions than a high-resolution singular value decomposition. This option recommends the value of *k* (the number of topics) heuristically based on the value specified in the MAXK= option.

Assume that the MAXK=*n* option and the singular value decomposition with *n* dimensions accounts for *t*% of the total variance. If you specify RES=HIGH, the server always recommends the maximum number of dimensions. That is, *k*=*n*. If you specify RES=MED, the server recommends a value for *k* that explains (5/6) × *t*% of the total variance. If you specify RES=LOW, the server recommends a value for *k* that explains (2/3) × *t*% of the total variance.

**TOL=$\varepsilon$**
specifies the maximum allowable tolerance for the singular value.

**Default** The value of epsilon on the machine where the server is running.

**SYNONYMS=*table-name***
specifies the name of an in-memory table that contains user-defined synonyms to use in the analysis. The table specifies parent-child relationships that enable you to map child terms to a representative parent. The synonym relationship is indicated in the Terms result table and is also reflected in the term-by-document result table known as the Termdoc or Parent table.

The specified table must have either the two variables Term and Parent, or the four variables Term, Parent, Termrole, and Parentrole. When stemming is enabled (the default), the relationships provided by the SYNONYMS= table take precedence over relationships that are identified through term stemming.

**Alias** SYN=

**TERMWGT=ENTROPY | MI | NONE**
specifies how terms are weighted. TERMWGT=ENTROPY specifies that terms are weighted using the entropy formulation. If you specify TERMWGT=MI, then terms are weighted using the mutual information formulation. Specifying TERMWGT=NONE suppresses term weighting. See the documentation for the HPTMINE procedure for the details about computing term weights.

If you specify TERMWGT=MI, then you must specify a target variables with the TARGET= option.

**Default** ENTROPY

**TOPICS=*n***
specifies the number of topics to generate. When you specify *n*, the server automatically produces a table of topics with up to *n* entries. You can also request the Topics table with the SELECT= option. Specifying TOPICS=*n* is equivalent to requesting topics based on a singular-value decomposition with *n*=*k* factors.

**Alias** NUMTOPICS=

**Interaction** You can use the NUMLABELS= option to control the number of terms to use in labeling the topic.

**TARGET=*target-variable***
specifies a variable that contains information about the category that a document belongs to. If specified, the target variable is used in computing term weights. For example, it is used with TERMWGHT=MI.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### *ODS Table Names*

The TEXTPARSE statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| TextParseSummary | Summary Information from parsing documents | Default |

The ODS table includes the temporary tables names for the tables that are requested in the "SELECT = (list-of-temporary-tables)".

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## TOPK Statement

The TOPK statement calculates and selects the top-k and bottom-k distinct values of a variable based on a user-specified ranking order. The distinct values can be reported as raw or formatted values. The ranking can be based on the raw value, the formatted value, the frequency count, or based on a calculated score derived from the values of a weight variable. You can also specify aggregate functions to roll up multiple weight values into a single score for a distinct value.

### Syntax

**TOPK** <*variable-list*> <*/ options*>;

#### *Optional Argument*

***variable-list***
specifies one or more numeric variables. If you do not specify this option, then all numeric variables in the table are used.

#### *Topk Statement Options*

**AGGREGATE=(*aggregation-methods*)**
specifies the aggregation methods for which WEIGHT= variable values are rolled up into rank order score for distinct values. If no WEIGHT= variable is specified, then this option is ignored.

The available aggregation methods are as follows:

MAX specifies to use the maximum value of the weight values

MEAN specifies to use the arithmetic mean of the weight values

MIN specifies to use the minimum value of the weight values

SUM specifies to use the sum of the weight values

**Alias** AGG=

**Default** SUM

**FORMATS=("*format-specification*",…)**
specifies the formats for the variables. If you do not specify the FORMATS= option, or if you omit the entry for a variable, the default format is applied for that variable.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**Example**
```
proc imstat data=lasr1.table1;
   topk x1 x2 / formats=("10.2", 10.2");
quit;
```

**FREQ=*variable-name***
specifies the numeric frequency variable to use for calculating the rank order score for distinct values. This option is valid when ORDER=FREQ or when AGGREGATE= is N, SUM, or MEAN only.

**K1=*n***
specifies the maximum number of distinct values to include in the top-k list.

**Alias** TOPK=

**Default** 1

**Range** 1 to 1000

**K2=*n***
specifies the maximum number of distinct values to include in the bottom-k list.

**Alias** BOTTOMK=

**Default** 1

**Range** 1 to 1000

**DESCENDING**
specifies that the levels of the GROUPBY variables are to be arranged in descending order.

**Alias** DESC

**ORDER= FREQ | VALUE | WEIGHT**
specifies the rank ordering to apply to the distinct values when no WEIGHT= variable is specified. The following rank orders are valid in the TOPK request.

The available ordering methods are as follows:

| | |
|---|---|
| FREQ | specifies to order by frequency count |
| VALUE | specifies to order by raw or formatted values of the variable |
| WEIGHT | specifies to order by the aggregate values of the WEIGHT= variable |

**Default** FREQ

**WEIGHT=*variable-name***
specifies the numeric weight variable to use for calculating the rank order score. If you specify ORDER= and WEIGHT=, then the WEIGHT= variable takes priority over ORDER.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### *ODS Table Names*

The TOPK statement generates the following ODS tables for each analysis variable.

| ODS Table Name | Description | Option |
|---|---|---|
| TOPK | Top/Bottom K Distinct Values | Default |
| BTMK | Top/Bottom K Distinct Values | Default |
| TOPKMISC | Misc. Info for Top/Bottom K Distinct Values | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# TRANSFORM Statement

The TRANSFORM statement can perform multiple transformations on a set of input variables. Each transformation can perform imputation, outlier detection and treatment, functional transformation, binning, and output.

**Example:** "Example 17: Transforming Variables with Imputation and Binning" on page 291

## Syntax

**TRANSFORM** (*request1*) <(*request2*) …> </ *options*>;

### *Request Options*

***request***
specifies the transform phases to perform on the input variables. If a request is not specified properly, it is ignored but does not stop other requests from processing. If

you do not specify a request at all, the TRANSFORM statement bins the numeric variables from the active table with the BUCKET(5) method.

A request can specify up to five phases. The five phases are as follows:

- Imputation phase

  performs a simple imputation using the mean, median, or a specified value

- Outlier phase

  detects outliers and can treat them by trimming or Winsorization

- Functional phase

  computes various functions for rescaling or standardization

- Binning phase

  performs discretization with either supervised or unsupervised methods

- Output phase

  scores the input variables according to the specified requests, generates summarization tables, and can score code

For more information, see "Details" on page 251.

**NAME="*request-name*"**
specifies the transformation request name. The names should be unique among the requests. If you do not specify unique names or do not provide names, then unique names are generated. Specifying a name can be useful because the name is displayed in the output and is used as a prefix to names of the transformed variables.

**INPUT=*variable-name***
**INPUT=(*variable-name1 variable-name2 …*)**
specifies the input variables to transform. If you do not specify input variables, then all numeric variables in the table are transformed.

**Alias** TRANSVAR=

### *Imputation Phase Options*

**IMPUTE= NONE | MEAN | MEDIAN | VALUE(*number*)**
specifies how to impute missing values for the input variables. The imputed values can be the mean, median, or a user-specified value.

**Default** NONE

### *Outlier Phase Options*

**OUTLIER <=(*outlier-boundaries <outlier-treatment>* )>**
specifies how to define outlier boundaries and how to treat outliers. The options for defining the *outlier-boundaries* are either IQR or PERC. The options for defining *outlier-treatment* are TRIM or WINSOR.

**IQR <(*m*)>**
specifies outliers based on a multiple of the interquartile range. Values outside the range [Q1 – *m**IQR, Q3 + *m**IQR] are considered outliers.

**Default** 1.5

**PERC <(*p*)>**
**PERC <(*lower upper*)>**
specifies outliers based on percentiles. If you specify a value for *p*, then it specifies the percentage of the distribution to be defined as outliers. Specifically, it defines the *p*/2 and (100-*p*/2) percentiles to use as the lower and upper bounds. Alternatively, you can specify the percentiles directly with the *lower* and *upper* options. This option also enables you to define nonsymmetric bounds. The default percentiles for the lower and upper bounds are 5 and 95 (or p=10).

**TRIM**
specifies to remove outliers by assigning them to missing values.

**WINSOR**
specifies to replace outliers with the lower and upper bounds determined by the outlier definition.

**Default** The default outlier boundaries are IQR(1.5). The default outlier treatment is TRIM.

### Functional Phase Options

**FUNC=*function-name* <(*arguments*)>**
specifies a function for rescaling or standardization to use during the functional transformation phase. The available functions are as follows:

**ABS**
returns the absolute value of each of the variables specified in the INPUT= option.

**ARCSIN**
returns the angular transformation for proportional data for each of the variables specified in the INPUT= option. Mathematically, the function is computed as arcsine(sqrt(*x*)).

**BOXCOX <(λ)>**
returns the Box-Cox transformation for each of the variables specified in the INPUT= option.

**Default** 2

**CENTER <(LOC=*location*)>**
returns the distance to the mean or median value for each of the variables specified in the INPUT= option. Values for *location* are MEAN or MEDIAN.

**Default** LOC=MEAN

**COSH**
returns the hyperbolic cosine for each of the variables specified in the INPUT= option.

**EXP**
returns the exponential value for each of the variables specified in the INPUT= option.

**INVERSE**
returns the value of 1/*variable* for each of the variables specified in the INPUT= option.

**LOG <(*base*)>**
returns the logarithm for each of the variables specified in the INPUT= option. If you do not specify a value for *base*, then the natural logarithm is used.

**POWER <(*power*)>**
returns the power transform for each of the variables specified in the INPUT= option.

**Default** 2

**RANGE <(MIN=*lower* MAX=*upper*)>**
returns the value of each variable specified in the INPUT= option, subject to the upper bound and lower bound of the range.

**Default** MIN=0 and MAX=1

**SQRT**
returns the square root for each of the variables specified in the INPUT= option.

**STANDARDIZE <(LOC=*location* SCALE=*scale-specification*)>**
returns the standardized values for each variable in the INPUT= option. The value of *location* can be MEDIAN or MEAN.

**SCALE=IQR <(*m*)>**
**SCALE=STD <(*m*)>**
specifies the scale as a multiple of the interquartile range or a multiple of the standard deviation as follows.

**Defaults** LOC=MEAN and SCALE=STD(1)

For SCALE=IQR, the default value is 1.5. For SCALE=STD, the default value is 1.

**TANH**
returns the hyperbolic tangent for each of the variables specified in the INPUT= option.

### *Binning Phase Options*

**BIN=*method* <(*options*)>**
specifies a supervised or unsupervised discretization method. The unsupervised methods determine cutpoints without a target variable.

The unsupervised methods are as follows:

**BUCKET <(*n*)>**
specifies to create equal width bins according to the minimum value, maximum value, and the number of bins requested in *n*.

**Default** 5

**QUANTILE <(*n*)>**
specifies to create equal frequency bins.

**Default** 5

The following supervised methods require that you specify the TARGET= option. With the supervised methods, the server combines empty bins with their left neighbors. As a result, the actual number of bins produced might be less than the specified MIN= value. To enforce the MIN= constraint, permit empty bins by

specifying the EMPTYBINS option in the TRANSFORM statement. The supervised methods are as follows:

**CHIMERGE <( <MIN=*n*> <MAX=*n*> )>**
specifies to use the chi-merge algorithm for binning.

**DTREE <(*options*)>**
specifies to construct a one-level decision tree and discretize the variables by one-to-one mapping of the leaves of the decision tree to bins. You can use the following options to control the discretization.

**GAIN**
specifies to use the information gain as criteria, instead of information gain ratio.

**MAX=*n***
specifies the maximum number of bins.

**Alias** MAXBRANCH=

**Default** The default value for MAX= option is the number of levels (classes) of the TARGET= variable.

**MDLP <( <MIN=*n*> <MAX=*n*> )>**
specifies to use minimum description length principle for binning.

**Default** BIN=BUCKET(5)

**EVENT=("*event-category1*" <"*event-category2*" …> )**
specifies the event category for binary target variables specified in the TARGET= option. Event categories are needed for the WOE, IV, or GINI index binning evaluation statistics. Multiple event categories are matched in pairs with multiple target variables based on the order they are listed. If more events are listed than target variables, the remaining events are ignored. If more target variables are listed than events, the last event value is used for the remaining target variables.

**Example**
```
target=(saleFlag over18Flag) event=('1' '1')
```

**TARGET=*variable-name***
**TARGET=(*variable-name1 variable-name2* …)**
specifies target variables for performing supervised binning or for computing evaluation statistics with the EVALSTATS= option. Multiple target variables are matched in pairs with multiple input variables based on the order the variables are listed. If more target variables are listed than input variables, the remaining target variables are ignored. If more input variables are listed than target variables, the last target variable is used for the remaining input variables.

**Alias** EVALVAR=

### *Output Phase Options*

**CODE <(*code-generation-options*)>**
requests that the server produce SAS scoring code based on the actions that it performed during the analysis. The server generates DATA step code. By default, the code is replayed as an ODS table by the procedure as part of the output of the statement. More frequently, you might want to write the scoring code to an external file by specifying options.

For each transformation request, the scoring code applies all the phases to the variables specified in the INPUT= option. The scoring code then creates result variables that are computed from the transform. The result variables are named uniquely by prefixing the NAME= option value of the request to the names of the input variables. If you do not specify the NAME= option, then the server uses _TRAN*n*_ as a prefix.

**COMMENT**
specifies to add comments to the code in addition to the header block. The header block is added by default.

**FILENAME='*path*'**
specifies the name of the external file to which the scoring code is written. This suboption applies only to the scoring code itself.

**Alias** FILE=

**FORMATWIDTH=*k***
specifies the width to use in formatting derived numbers such as parameter estimates in the scoring code. The server applies the BEST format, and the default format for code generation is BEST20.

**Alias** FMTW=

**Range** 12 to 32

**LINESIZE=*n***
specifies the line size for the generated code.

**Alias** LS=

**Default** 100

**Range** 100 to 256

**DETAILS**
specifies to display the bin details that result from a binning transformation phase.

**SCORE**
specifies to score the input variables according to the transformation requests. The scored results and the variables specified in the INPUT=, TARGET=, and FREQ= options are stored in a temporary table. If you want to transfer additional variables, you can specify them with the IDVARS= option.

**Alias** TEMPTABLE

## *TRANSFORM Statement Options*

**ALLIDVARS**
requests that all variables in the input table are treated as ID variables when a scoring table is produced. In other words, if this option is specified, all variables from the input table, including computed columns, are transferred to the scoring table. This option has no effect unless you specify the SCORE option.

**ALPHA=*number***
specifies a number between 0 and 1 from which to determine the confidence level when the BIN= option uses the CHIMERGE= method. The default is $\alpha = 0.05$, which leads to $100 \times (1- \alpha)\% = 95\%$ confidence limits for the parameter estimates.

**Default** 0.05

**BININIT=(*binning-initialization*)**
specifies how to initialize the bins for supervised binning methods (MDLP, CHIMERGE, or DTREE). The *n* value for the BUCKET and QUANTILE options is the starting number of bins. The default value is 100.

The three options are as follows:

**BUCKET <(*n*)>**
specifies to use equal width binning.

**EXACT**
**DISTINCT**
specifies to use the distinct levels of the input variables. This binning initialization method is not compatible with outlier and functional phases. If you specify this initialization method, then transformation requests that contain an outlier phase or functional phase are ignored.

**QUANTILE <(*n*)>**
specifies to use equal frequency binning.

**BINMISSING**
specifies to place missing values in a separate bin during the binning phase.

**EMPTYBINS**
avoids the default merging of bins with no observations (empty bins) with their left neighbors. This option is applicable to the MDLP binning method. Empty bins can occur from initializing the number of bins with the BININIT= option set to BUCKET or QUANTILE.

**EVALSTATS=(*list-of-binning-evaluation-statistics*)**
specifies to compute binning evaluation statistics. These statistics are computed from two-way contingency tables between the scored variables and a target variable. The available statistics are as follows:

| | |
|---|---|
| CHISQ | chi-square statistic |
| FTEST | F-test statistic |
| G2 | $G^2$ log-likelihood-ratio statistic |
| GINI | Gini index statistic |
| IV | information value statistic |
| WOE | weight of evidence statistic |

The CHISQ and G2 statistics are valid for all target variables. The WOE, IV, and GINI statistics are valid for binary target variables, The FTEST statistic is valid for continuous target variables only. If statistics are not specified, the default statistics are CHISQ and G2. Transformation requests that are incompatible with the specified evaluation statistics are ignored.

**FREQ=*variable-name***
specifies a numeric variable with a value that represents the frequency of the observation. For example, if the FREQ= variable has the value 5 for a given observation, then that observation represents five observations. The FREQ= option is not available for transform requests that require percentiles in any of the transformation phase. Consequently, if the FREQ= option is specified and percentiles are required, then all such transformation requests are ignored.

**IDVARS=(*variable-list*)**
**IDVARS=*variable-name***
specifies the variables from the active table to transfer to the temporary table that is created by scoring the input table. This option has no effect unless the SCORE option is also specified. (See the SCORE option for details about which variables are added to the temporary table by default.) The IDVARS= option should be used to transfer additional columns from the input table to the scoring table.

**Alias** ID=

**Tip** Instead of this option, you can specify the ALLIDVARS option to transfer all variables from the input table to the scoring table.

**PERCEPSILON=*number***
specifies the convergence tolerance for the iterative algorithm that is used to compute percentiles. Percentiles are calculated in the outlier transformation phase and can also be calculated in the binning transformation phase if quantiles are requested.

**Default** $1e^{-5}$

**PERCMAXITER=*i***
specifies the maximum number of iterations in the percentile algorithm. The percentile algorithm is iterative and avoids the cost of copying and sorting the data with multiple passes through the data. You can limit the number of iterations with the PERCMAXITER= option. You can also control the computational demand with the PERCEPSILON= option, which affects the tolerance criterion by which the convergence of the iterative algorithm is judged. If the percentile computations for a particular variable do not converge, the transformation that depends on those percentiles is not performed.

**Default** 10

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

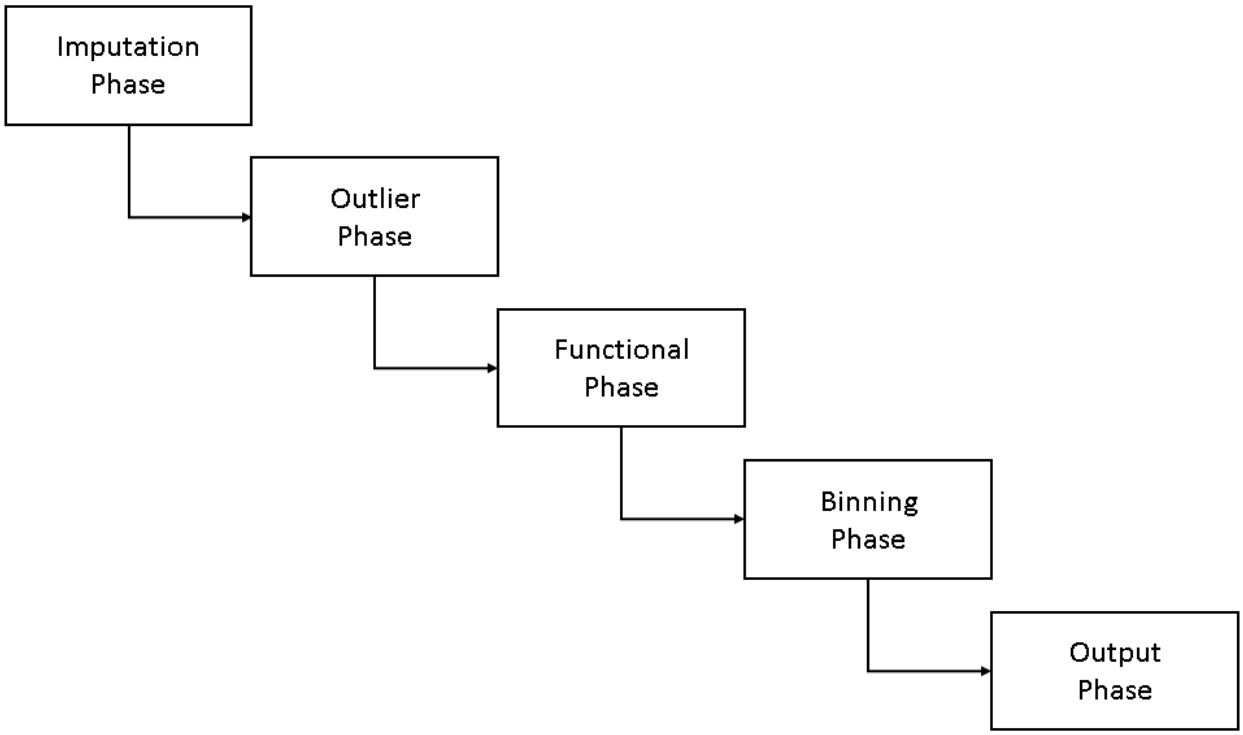
**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### *Understanding Transformation Requests*

A transformation request can perform up to five types of transformation phases. The phase names and the sequence of execution are shown in the following figure:




The following list summarizes the details:

- You can specify the phases in any order in a transform request. The server executes them in the sequence that is shown in the figure.
- You can specify any subset of the phases that you want to use or all five phases. For example, in one TRANSFORM statement you can use one request to impute values and bin them. In another request, you can determine outliers, apply a functional transformation, and then bin the variables.
- You can specify a phase more than once to modify a set of variables in a different way. For example, the following code fragment imputes missing values of the Oxygen variable by replacing with the mean value. For the RunPulse variable, missing values are replaced with the value 155.

```
transform (impute=mean input=(oxygen) name="impute")
          (impute=value(155) input=(runpulse) name="imputepulse155");
```

### *ODS Table Names*

The TRANSFORM statement generates the following ODS tables.

| ODS Table Name | Description | Option |
|---|---|---|
| TransInfo | Phase-wise information about the transformations | Default |
| NumVarSummary | Summary statistics for numeric variables | Default |
| NomVarSummary | Summary statistics for nominal variables | TARGET= or BININIT=EXACT |

| ODS Table Name | Description | Option |
|---|---|---|
| VarTransInfo | Variable transformation information | Default |
| BinDetails | Detailed binning information for numeric variables | BIN= and DETAILS |
| TempTable | Information about a temporary table | SCORE or TEMPTABLE |

## QUIT Statement

The QUIT statement is used to end the procedure execution. When the procedure reaches the QUIT statement, all resources allocated by the procedure are released. You can no longer execute procedure statements without invoking the procedure again. However, the connection to the server is not lost, because that connection was made through the SAS LASR Analytic Server engine. As a result, any subsequent invocation of the procedure that uses the same libref executes almost instantaneously because the engine is already connected to the server.

**Interaction:** Using a DATA step or another procedure step is equivalent to issuing a QUIT statement. If there is an error during the procedure execution, it is also equivalent to issuing a QUIT statement.

### Syntax

**QUIT**;

# Temporary Tables, Generated Code, and Scoring

Several of the analytic statements for the IMSTAT procedure offer the TEMPTABLE, CODE, and SCORE options. The following table summarizes the interactions between those options and the procedure output.

| Statement Options | | | Output Result | | |
|---|---|---|---|---|---|
| **TEMPTABLE** | **CODE** | **SCORE** | **ModelInfo** | **Scoring Code** | **Scored Data Set** |
| | | | ODS | Not created | Not created |
| | Y | | ODS | ODS (or the FILE= option) | Not created |
| | | Y | ODS | Not created | _TEMPLAST_ and _TEMPSCORE_ |
| Y | | | _TEMPLAST_ | Not created | Not created |
| Y | Y | | _TEMPLAST_ | Not created | Not created |

| Statement Options | | | Output Result | | |
|---|---|---|---|---|---|
| TEMPTABLE | CODE | SCORE | ModelInfo | Scoring Code | Scored Data Set |
| Y | Y | Y | _TEMPLAST_ | Not created | _TEMPSCORE_ |

## Generated Code and Locale Sensitivity

Several of the analytic statements offer a CODE option that can be used to produce and store code for analytic models. The code can be used for scoring data sets with the models.

Because formatting numeric variables with thousands separators and decimal points is locale-sensitive, in rare circumstances, the scoring code can be locale-sensitive too. This can occur when a numeric variable that takes on non-integral values is used as a classification or in a GROUPBY= option. The following code demonstrates how locale-specific code can be generated:

```
options locale=fr_FR;

data example.cars;
  set sashelp.cars;
run;

proc imstat data=example.cars;
  glm mpg_city = weight / groupby=(enginesize)
    code(file="/tmp/glmcars.sas" replace linesize=256);
quit;
```

For the SAS LASR Analytic Server 2.7 release, the locale is specified in the file header. For locales that are sensitive to the formatting of numeric variables, the NLBEST format is used. The following example is for fr_FR:

```
   /*-----------------------------------------
   SAS Code Generated by LASR Analytic Server
     Date              : 02 juin 2015 12 h 54
     Locale            : fr_FR
     Model Type        : Regression
     Group-By variable : EngineSize
     Response variable : MPG_City
     ----------------------------------------*/

...
   length _EngineSize_ $12; drop _EngineSize_;
   _EngineSize_ = left(trim(put(EngineSize,NLBEST12.)));

...
     select (_EngineSize_);
      when ('1,3') do;
...
```

If the locale in which you intend to use the code does not match the locale specified in the file header, then you need to take some action. You can run the SAS program that

produced the scoring code again in the preferred locale or you can modify the scoring code.

# Examples: IMSTAT Procedure (Analytics)

## Example 1: Calculating Percentiles and Quartiles

### Details

If you specify the PERCENTILE statement without variables or options, you obtain results for the 25th, 50th, and 75th percentile. These are also known as the first quartile, the median, and the third quartile. This is done for all numeric non-CLASS variables in the table.

This PROC IMSTAT demonstrates the default behavior for calculating percentiles for a single variable and then demonstrates using GROUPBY= variables and generating results for nonstandard percentiles.

### Program

```
libname example sasiola host="grid001.unx.sas.com" port=10010 tag='hps';

data example.prdsale; set sashelp.prdsale; run;

proc imstat data=example.prdsale;
    percentile actual; 1
run;
    percentile actual / groupby=(region division); 2
run;
    percentile actual / values=(3 5 10 90 95 97); 3
quit;
```

### Program Description

1. This PERCENTILE statement generates the default output for the Actual variable.
2. The quartiles for the Actual variable are calculated for the groups of Region and Division.
3. The VALUES= option is used to specify the percentiles to calculate.

### Output

The following results include the column that is named Converged. This column indicates whether the iterative percentile algorithm converged for the variable and percentile. It is possible that some percentiles can fail to converge while others do

converge. The converged percentiles match those computed with the UNIVARIATE or MEANS procedures using their default definition of a quantile.

***Output 4.1*** *Default Output for a Single Variable*

| Percentiles and Quantiles for Table WORK.PRDSALE | | | |
|---|---|---|---|
| Column | Percentile | Value | Converged |
| ACTUAL | 25 | 261.0000000 | Yes |
| ACTUAL | 50 | 503.0000000 | Yes |
| ACTUAL | 75 | 756.5000000 | Yes |

***Output 4.2*** *Results for Actual Grouped by Region and Division*

| Percentiles and Quantiles for Table WORK.PRDSALE | | | | | |
|---|---|---|---|---|---|
| REGION | DIVISION | Column | Percentile | Value | Converged |
| EAST | CONSUMER | ACTUAL | 25 | 260.5000000 | Yes |
| EAST | CONSUMER | ACTUAL | 50 | 494.5000000 | Yes |
| EAST | CONSUMER | ACTUAL | 75 | 745.5000000 | Yes |
| EAST | EDUCATION | ACTUAL | 25 | 277.5000000 | Yes |
| EAST | EDUCATION | ACTUAL | 50 | 548.5000000 | Yes |
| EAST | EDUCATION | ACTUAL | 75 | 759.5000000 | Yes |
| WEST | CONSUMER | ACTUAL | 25 | 243.5000000 | Yes |
| WEST | CONSUMER | ACTUAL | 50 | 508.5000000 | Yes |
| WEST | CONSUMER | ACTUAL | 75 | 750.0000000 | Yes |
| WEST | EDUCATION | ACTUAL | 25 | 225.0000000 | Yes |
| WEST | EDUCATION | ACTUAL | 50 | 465.5000000 | Yes |
| WEST | EDUCATION | ACTUAL | 75 | 774.0000000 | Yes |

***Output 4.3*** *Results for a Single Variable Using the VALUES= Option*

| Percentiles and Quantiles for Table WORK.PRDSALE | | | |
|---|---|---|---|
| Column | Percentile | Value | Converged |
| ACTUAL | 3 | 38.0000000 | Yes |
| ACTUAL | 5 | 59.0000000 | Yes |
| ACTUAL | 10 | 111.0000000 | Yes |
| ACTUAL | 90 | 900.0000000 | Yes |
| ACTUAL | 95 | 950.0000000 | Yes |
| ACTUAL | 97 | 968.0000000 | Yes |

## Example 2: Retrieving Box Values

### Details

This PROC IMSTAT example demonstrates retrieving the statistics for a box plot. The BOXPLOT statement does not generate a plot, it generates values that can be used to create a plot.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.cars; 1
    set sashelp.cars;
run;

proc imstat data=example.cars; 2
    boxplot;
quit;
```

### Program Description

1. The sashelp.cars data set is loaded to memory on the SAS LASR Analytic Server.
2. The in-memory table is referenced with the DATA= option and then the BOXPLOT statement is used.

### Output

The arithmetic mean is reported because the center line of the box plot is sometimes drawn at the mean and not the median. In some displays the median is shown as a line and the box is augmented with a graphic symbol at the position of the mean. The low and high whiskers are values of actual observations in the data set. These values might be the minimum or maximum values in the data set if the value for that observation equals the value nearest 1.5 times the inter-quartile range from the edge of the box. The box in box plot is drawn from the first quartile to the third quartile.

***Output 4.4*** *BOXPLOT Statement Results*

| Box Plot Elements for Table WORK.CARS | | | | | | |
|---|---|---|---|---|---|---|
| Column | First Quartile | Median | Third Quartile | Mean | Low Whisker | Hi Whisker |
| MSRP | 20330 | 27635 | 39215 | 32775 | 10280 | 65000 |
| Invoice | 18851 | 25295 | 35733 | 30015 | 9875.00 | 59912 |
| Engine Size | 2.3500 | 3.0000 | 3.9000 | 3.1967 | 1.3000 | 6.0000 |
| Cylinders | 4.0000 | 6.0000 | 6.0000 | 5.8075 | 3.0000 | 8.0000 |
| Horsepower | 165.00 | 210.00 | 255.00 | 215.89 | 73.0000 | 390.00 |
| MPG_City | 17.0000 | 19.0000 | 21.5000 | 20.0607 | 12.0000 | 28.0000 |
| MPG_Highway | 24.0000 | 26.0000 | 29.0000 | 26.8435 | 17.0000 | 36.0000 |
| Weight | 3103.00 | 3474.50 | 3978.50 | 3577.95 | 1850.00 | 5287.00 |
| Wheelbase | 103.00 | 107.00 | 112.00 | 108.15 | 93.0000 | 124.00 |
| Length | 178.00 | 187.00 | 194.00 | 186.36 | 154.00 | 218.00 |

## Example 3: Retrieving Box Plot Values with the NOUTLIERLIMIT= Option

### Details

When you specify the NOUTLIERLIMIT= option, the IMSTAT procedure requests outlier information for the variables. When outliers are reported for a variable, pay attention to the last two columns of the display (columns Lo Bin and Hi Bin). These two columns let you know whether the values displayed in the outlier columns are actual data values, or counts in bins. For more information, see the information in the Output section.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.cars;
    set sashelp.cars;
run;

proc imstat data=example.cars;
    boxplot / noutlierlimit=7; 1

    ods output boxplot=outliers; 2
quit;

proc print data=outliers noobs;
  var column outlo1-outlo5 outhi1-outhi10 binlo binhi;
run;
```

### Program Description

1. The program example requests that the raw values for up to seven high outliers and up to seven low outliers are retrieved. If there are more than seven outliers, the procedure returns the binned values for the outlying values.

2. The ODS statement and the PRINT procedure that follows are display purposes only.

### Output

In example that follows, using the CARS data set, several variables exhibit outliers on the low end. For example, there are two outlying values for the MPG_City variable. Since the Lo Bin column of the result table is set to **No** for this variable, the values, 10 for OiutLo1 and 10 for OutLo2, are actual values in the data. Note that these values are smaller than the lower whisker value of 12. (See the previous example.) Similarly, the Horsepower variable shows several outliers on the high end of the distribution and the Hi Bin column is set to **No**. This lets you know that the values 493, 450, 500, and so on, represent actual values in the CARS table.

On the other hand, the Hi Bin column for the MSRP variable is set to **Yes**. This lets you know that more outliers were found than the specified NOUTLIERLIMT= limit of 7. The outliers are then placed in bins and the binned counts are reported. For example, there are 14 values in the first bin of MSRP outliers, 8 values in the second bin, 1 value in the fourth bin, and no value in the fifth bin, and so on.

***Output 4.5*** *BOXPLOT Statement Results with the NOUTLIERS= Option*

| Column | OutLo1 | OutLo2 | OutLo3 | OutLo4 | OutLo5 | OutHi1 | OutHi2 | OutHi3 | OutHi4 | OutHi5 | OutHi6 | OutHi7 | OutHi8 | OutHi9 | OutHi10 | BinLo | BinHi |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MSRP | _ | _ | _ | _ | _ | 14 | 8 | 1 | 0 | 3 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| Invoice | _ | _ | _ | _ | _ | 15 | 7 | 1 | 0 | 2 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| EngineSize | _ | _ | _ | _ | _ | 6.8 | 8.3 | _ | _ | _ | _ | _ | _ | _ | _ | 0 | 0 |
| Cylinders | _ | _ | _ | _ | _ | 10 | 12 | 10 | 12 | 12 | _ | _ | _ | _ | _ | 0 | 0 |
| Horsepower | _ | _ | _ | _ | _ | 477 | 493 | 493 | 500 | 450 | 493 | 420 | _ | _ | _ | 0 | 0 |
| MPG_City | 10 | 10 | _ | _ | _ | 15 | 3 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 2 | 0 | 1 |
| MPG_Highway | 16 | 16 | 13 | 12 | 14 | 9 | 3 | 3 | 1 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| Weight | _ | _ | _ | _ | _ | 7 | 2 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| Wheelbase | 89 | 89 | _ | _ | _ | 2 | 4 | 5 | 2 | 0 | 0 | 1 | 1 | 0 | 2 | 0 | 1 |
| Length | 153 | 144 | 143 | 150 | 153 | 3 | 4 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |

## Example 4: Retrieving Distinct Value Counts and Grouping

### Details

The DISTINCT statement calculates the count of unique raw values of variables. The following example shows how the DISTINCT statement is used with the GROUPBY= option to count the unique values within groups. The results are stored in a temporary table and then the FETCH statement is used to order the results and view them.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.cars;
    set sashelp.cars;
run;

proc imstat;
    table example.cars;
    distinct / groupby=(origin type) temptable; 1
run;

    table example.&_templast_;
```

```
    /* columninfo; */  2
    fetch origin type _Column_ _N_ _NMiss_ /
          orderby=(origin type _N_) desc=(_N_) format to=20;  3
quit;
```

### Program Description

1. The DISTINCT statement calculates the unique values of all variables in the table (numeric and character) and then groups them by the unique combinations of the Origin and Type columns. The results are stored in a temporary table.
2. The results of the COLUMNINFO statement are not shown, but in practice, listing the columns in the temporary table is helpful to understand the column names that are created.
3. The FETCH statement retrieves the first 20 rows from the temporary table. The results are sorted by ascending values of Origin and Type, and descending values of the distinct count, _N_.

### Output

In the following output, the rows are sorted lexically on values of Origin and Type, and then by the column with the greatest number of distinct values.

***Output 4.6*** *DISTINCT Statement Results After Grouping and Sorting*

| Selected Records from Table _T_FD641A91_138BAE60 | | | | |
|---|---|---|---|---|
| Origin | Type | _Column_ | _N_ | _NMiss_ |
| Asia | Hybrid | MSRP | 3 | 0 |
| Asia | Hybrid | Horsepower | 3 | 0 |
| Asia | Hybrid | Weight | 3 | 0 |
| Asia | Hybrid | EngineSize | 3 | 0 |
| Asia | Hybrid | Wheelbase | 3 | 0 |
| Asia | Hybrid | MPG_City | 3 | 0 |
| Asia | Hybrid | Model | 3 | 0 |
| Asia | Hybrid | Invoice | 3 | 0 |
| Asia | Hybrid | Length | 2 | 0 |
| Asia | Hybrid | MPG_Highway | 2 | 0 |
| Asia | Hybrid | Cylinders | 2 | 0 |
| Asia | Hybrid | Make | 2 | 0 |
| Asia | Hybrid | Type | 1 | 0 |
| Asia | Hybrid | DriveTrain | 1 | 0 |
| Asia | Hybrid | Origin | 1 | 0 |
| Asia | SUV | Invoice | 25 | 0 |
| Asia | SUV | MSRP | 25 | 0 |
| Asia | SUV | Weight | 25 | 0 |
| Asia | SUV | Model | 25 | 0 |
| Asia | SUV | Horsepower | 18 | 0 |

## Example 5: Performing a Cluster Analysis

### Details

You can perform a clustering analysis for all variables in an in-memory table by simply issuing a CLUSTER statement. However, specifying the variables to analyze and options can be specified to provide more a meaningful analysis.

The following SAS statements load the famous Iris flower data of R.A. Fisher to memory, and then perform *k*-means clustering on four of the variables.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';
```

```
data example.iris;
    set sashelp.iris;
run;

proc imstat data=example.iris;
   cluster SepalLength SepalWidth PetalLength PetalWidth / 1
       maxiter=50
       numclus=3
       nsamp  =2
       conv   =1.e-06
       init   =rand
       freq   =Species; 2
quit;
```

### Program Description

1. The four variables to analyze are specified in the CLUSTER statement.
2. Species is specified as the frequency variable and is used to cluster the four variables.

### Output

***Output 4.7*** *CLUSTER Statement Results for the Iris Data Set*

K-Means Clustering Information for 4 variables in Table WORK.IRIS

| Cluster ID | Number of Obs | Root Mean Square of STD | Maximum Distance from Seed to Obs. | Minimum Distance from Seed to Obs. | Within Cluster Sum of Square | Nearest Cluster | Distance between Cluster Centroids |
|---|---|---|---|---|---|---|---|
| 0 | 61 | 3.9943 | 16.4680 | 2.3571 | 3829.08 | 1 | 17.8842 |
| 1 | 39 | 4.0890 | 15.5156 | 2.3945 | 2541.38 | 0 | 17.8842 |
| 2 | 50 | 2.7803 | 12.4803 | 0.6618 | 1515.10 | 0 | 33.4949 |

Frequency Clustering Information for variable Species in Table WORK.IRIS

| Cluster ID | Level | Formatted | Frequency |
|---|---|---|---|
| 0 | 0 | Setosa | 0 |
| 0 | 1 | Versicolor | 47 |
| 0 | 2 | Virginica | 14 |
| 1 | 0 | Setosa | 0 |
| 1 | 1 | Versicolor | 3 |
| 1 | 2 | Virginica | 36 |
| 2 | 0 | Setosa | 50 |
| 2 | 1 | Versicolor | 0 |
| 2 | 2 | Virginica | 0 |

## Example 6: Performing a Pairwise Correlation

### Details

This PROC IMSTAT example demonstrates how to perform a pairwise correlation for all the numeric variables in the Iris data set.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.iris;
    set sashelp.iris;
run;

proc imstat data=example.iris;
   corr;
quit;
```

### Output

The output does not display a correlation matrix in the statistical sense. It is a collection of pairwise correlations.

| Pairwise Correlations for Table WORK.IRIS | | | | | |
|---|---|---|---|---|---|
| Column | Row | SepalLength | SepalWidth | PetalLength | PetalWidth |
| SepalLength | 1 | 1.0000 | -0.1176 | 0.8718 | 0.8179 |
| SepalWidth | 2 | -0.1176 | 1.0000 | -0.4284 | -0.3661 |
| PetalLength | 3 | 0.8718 | -0.4284 | 1.0000 | 0.9629 |
| PetalWidth | 4 | 0.8179 | -0.3661 | 0.9629 | 1.0000 |

## Example 7: Crosstabulation with Measures of Association and Chi-Square Tests

### Details

To compute measures of association of the row and column variable, you can add the MEASURE option to the CROSSTAB statement. ASSOCIATION is an alias for the MEASURE option. You can also request Chi-Square statistics for the test of independence between row and column variable with the CHISQ option.

The following statements request a crosstabulation of the Cylinders and Origin variables for the CARS data set. The statements also request measures of association and Chi-Square statistics. The NOMISS option is used to exclude levels of the variables that correspond to missing values.

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.cars; set sashelp.cars; run;

proc imstat data=example.cars;
    crosstab cylinders * origin / 1
        measures
        chisq
        nomiss; 2
quit;
```

### Program Description

1. The variables to use for the columns and rows are specified in the CROSSTAB statement.
2. The NOMISS option excludes levels that have missing values. Note that by default the FREQ procedure excludes levels with missing values, whereas the IMSTAT procedure includes those as valid levels.

### Output

***Output 4.8*** *Crosstabulation Results*

| Cross-tabulation of Cylinders by Origin for Table WORK.CARS with Aggregator N | | | |
|---|---|---|---|
| Cylinders | Asia | Europe | USA |
| 3 | 1 | 0 | 0 |
| 4 | 74 | 25 | 37 |
| 5 | 0 | 7 | 0 |
| 6 | 69 | 54 | 67 |
| 8 | 12 | 34 | 41 |
| 10 | 0 | 0 | 2 |
| 12 | 0 | 3 | 0 |

***Output 4.9*** *Measures of Association Results*

| Measures of Association of Cylinders by Origin for Table WORK.CARS | | | | |
|---|---|---|---|---|
| Statistic | Value | Asymptotic Standard Error | Lower 95% Confidence Limit | Upper 95% Confidence Limit |
| Gamma | 0.3433 | 0.0580 | 0.2295 | 0.4570 |
| Kendall's Tau-B | 0.2316 | 0.0404 | 0.1525 | 0.3108 |
| Stuart's Tau-c | 0.2294 | 0.0402 | 0.1506 | 0.3082 |
| | | | | |
| Somers' D C\|R | 0.2327 | 0.0403 | 0.1538 | 0.3117 |
| Somers' D R\|C | 0.2305 | 0.0405 | 0.1511 | 0.3100 |
| | | | | |
| Lambda Asymmetric C\|R | 0.1519 | 0.0275 | 0.0980 | 0.2058 |
| Lambda Asymmetric R\|C | 0.0212 | 0.0501 | 0.0000 | 0.1194 |
| Lambda Symmetric | 0.0909 | 0.0268 | 0.0384 | 0.1435 |
| | | | | |
| Uncertainty Coefficient C\|R | 0.0776 | 0.0156 | 0.0469 | 0.1082 |
| Uncertainty Coefficient R\|C | 0.0713 | 0.0134 | 0.0451 | 0.0974 |
| Uncertainty Coefficient Symmetric | 0.0743 | 0.0144 | 0.0461 | 0.1025 |

***Output 4.10*** *Chi-Square Statistics Results*

| Chi-Square Statistics of Cylinders by Origin for Table WORK.CARS | | | |
|---|---|---|---|
| Statistic | DF | Value | |
| Chi-Square | 12 | 68.7854 | <.0001 |
| Likelihood Ratio Chi-Square | 12 | 72.3057 | <.0001 |

## Example 8: Training and Validating a Decision Tree

### Details

This PROC IMSTAT example demonstrates how to use the DECISIONTREE statement to generate a decision tree and then use a validation data set for scoring against the tree.

The data for this example is available from the Machine Learning Repository of the University of California at Irvine.

Frank, A. and A. Asuncion. 2010. UCI Machine Learning Repository. Irvine, CA: University of California, School of Information and Computer Science. Available at http://archive.ics.uci.edu/ml. Accessed on December 4, 2012.

### Program

```
libname mylib 'path-to-datasets';
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.bank_train_1; set mylib.bank_train_1;
data example.bank_valid_1; set mylib.bank_valid_1;run;

proc imstat data=example.bank_train_1;
    decisiontree subscribe_term_deposit /
        nbins      =10
        maxlevel   =7
        maxbranches=4
        input      =(age job marital_status education
                     default balance housing loan contact
                     day month duration campaign previous
                     poutcome)
        nominal    =(contact default education housing
                     job loan marital_status month
                     poutcome)
        multvar
        prune
        leafsize   =5
        save       =DTreeTab; 1

/*   ods output dtree=example.banktree_train_1; 2 */
run;

   decisiontree subscribe_term_deposit /
        treetab   =DTreeTab
        scoredata =example.bank_valid_1
        detail
        save      =DTreeScoreTab;
run;
/*
    table example.bank_valid_1;run; 3
    decisiontree subscribe_term_deposit /
        treedata=example.banktree_train_1; 4
*/

    free DTreeTab DTreeScoreTab;
quit;
```

### Program Description

1. The SAVE= option stores the result table so that it can be used in subsequent statements. It is named DTreeTab.
2. As an alternative to the SAVE= option, the ODS OUTPUT statement can also be used to save the result table.
3. To use the table that was stored with the ODS OUTPUT statement, the TABLE statement switches the active table to bank_valid_1.
4. The TREEDATA= option specifies the decision tree that was saved with the ODS OUTPUT statement.

### Output

***Output 4.11*** *Partial Results for the Decision Tree Created from the Training Data*

| Tree Node Information for 29 nodes in Table WORK.BANK_TRAIN_1 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Node Number | Tree Level | Parent Node | Parent | Node Type | Node Name | Gain Ratio | Number of Observations | Target Value |
| 0 | 0 | -1 | | CLASS | poutcome | 0.1952278335 | 40689 | no |
| 1 | 1 | 0 | poutcome | NUM | duration | 0.1301732794 | 39321 | no |
| 2 | 1 | 0 | poutcome | NUM | balance | 0.0643359505 | 1368 | ye |
| 3 | 2 | 1 | duration | CLASS | month | 0.1154551349 | 34569 | no |
| 4 | 2 | 1 | duration | NUM | age | 0.0636420716 | 4752 | no |
| 5 | 2 | 2 | balance | CLASS | contact | 0.052032778 | 1363 | ye |
| 6 | 2 | 2 | balance | LEAF | subscribe_term_deposit | 0 | 5 | ye |
| 7 | 3 | 3 | month | NUM | age | 0.0695583366 | 33292 | no |
| 8 | 3 | 3 | month | LEAF | subscribe_term_deposit | 0.0273350231 | 1277 | no |
| 9 | 3 | 4 | age | LEAF | subscribe_term_deposit | 0.2804464392 | 19 | ye |
| 10 | 3 | 4 | age | NUM | duration | 0.0362803449 | 4732 | no |

***Output 4.12*** *Partial Results for Classification Information Generated with the DETAIL Option*

| Tree Scoring Results for 4521 records in Table WORK.BANK_VALID_1 (Mis-Classification Rate=0.10418) | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Target Value | Target Level | Mis-Classification | Number of Nodes | NodeList1 | NodeList2 | NodeList3 | NodeList4 | NodeList5 | NodeList6 | NodeList7 |
| no | 0 | Yes | 5 | 0 | 1 | 3 | 7 | 14 | | |
| no | 0 | No | 5 | 0 | 1 | 3 | 7 | 14 | | |
| no | 0 | No | 5 | 0 | 1 | 4 | 10 | 15 | | |
| no | 0 | No | 5 | 0 | 1 | 3 | 7 | 14 | | |
| no | 0 | No | 5 | 0 | 1 | 3 | 7 | 14 | | |
| no | 0 | No | 5 | 0 | 1 | 3 | 7 | 14 | | |
| no | 0 | No | 5 | 0 | 1 | 3 | 7 | 14 | | |
| no | 0 | No | 5 | 0 | 1 | 3 | 7 | 14 | | |
| no | 0 | No | 5 | 0 | 1 | 4 | 10 | 15 | | |
| ye | 1 | No | 6 | 0 | 2 | 5 | 11 | 17 | 23 | |

## Example 9: Storing and Scoring a Decision Tree

### Details

You can store the representation of a decision tree in an in-memory table on the server and at the same time score the input table. This process generates two temporary tables: the temporary table with the tree representation and the temporary table with the scoring results.

This enables you to compute decision trees for high-cardinality problems. The results from tree building and tree scoring are available to you without transferring large amounts of data between the SAS client and the server. You can query the tree (for a drill-down, for example) and query the scoring information efficiently, using the storing and querying features of the server. Also, by storing them as temporary tables, you can process them with other IMSTAT procedure statements.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';
```

```
data example.train example.validate;
   set sashelp.heart;
   flag=ranuni(12345);
   if flag <= .7 then output example.train;
   else output example.validate;
run;

proc imstat;
  table example.train;
  decisiontree Weight / input=(Sex DeathCause
                               Chol_Status
                               BP_Status Weight
                               Smoking_Status)
                        nbinstarget=5
                        temptable   1
                        vars=(Sex DeathCause
                              Chol_Status
                              BP_Status Weight
                              Smoking_Status)
                        nomissobs;
run;

  table example.&_temptree_;  2
  /* tableinfo; */
  /* columninfo; */
  fetch _CI0_ _CI1_ _Val0_ _Val1_ _Parent_ -- _TargetUpperbd_
        / from=1 to=10 format;
run;

  table example.&_tempscore_;   3
  /* tableinfo; */
  /* columninfo; */
  where _NodeList2_=6;
  fetch / from=1 to=5 format;
run;

  where;  /* clear the WHERE clause */

  decisiontree Weight / treelasr=example.&_temptree_  4
                        scoredata=example.validate
                        temptable
                        vars=(Sex DeathCause
                              Chol_Status
                              BP_Status Weight
                              Smoking_Status);
run;

  table example.&_tempscore_;
  fetch / from=1 to=5 format;
run;
```

### Program Description

1. The TEMPTABLE option specifies to save the decision tree and the scoring results in in-memory tables on the server.

2. The &_TEMPTREE_ macro variable is used to access the representation of the decision tree and the following FETCH statement prints a subset of the variables from the first ten rows.
3. The &_TEMPSCORE_ macro variable is used to access the scoring table. The following FETCH statement prints the first five rows.
4. The TREELASR= option and SCOREDATA= option demonstrate how to score an input table explicitly that is already in memory.

## Output

***Output 4.13*** *Partial Results for the Decision Tree Created from the Heart Data Set*

| Selected Records from Table _T_BC324DFF_7FE9FAB66BE8 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| _CI0_ | _CI1_ | _Val0_ | _Val1_ | _Parent_ | _ParentName_ | _NodeType_ | _NodeName_ | _Gain_ | _NumObs_ | _TargetLowerbd_ | _TargetUpperbd_ |
| 0.5062565172 | 0.3879040667 | | | -1 | | CLASS | Sex | 0.1013667052 | 1918 | 113.6 | 160.2 |
| 0.3529411765 | 0.3529411765 | Desirable | | 15 | Chol_Status | LEAF | Weight | 0 | 17 | 113.6 | 160.2 |
| 0.6 | 0.4 | High | Desirable | 30 | Chol_Status | LEAF | Weight | 0 | 20 | 67 | 113.6 |
| 0.5160377358 | 0.4028301887 | Male | | 0 | Sex | CLASS | BP_Status | 0.0358150502 | 1060 | 160.2 | 206.8 |
| 0.5510204082 | 0.2244897959 | High | Borderline | 15 | Chol_Status | LEAF | Weight | 0 | 49 | 160.2 | 206.8 |
| 0.634032634 | 0.2296037296 | Female | | 0 | Sex | CLASS | BP_Status | 0.055274468 | 858 | 113.6 | 160.2 |
| 0.4285714286 | 0.3571428571 | Desirable | | 16 | Chol_Status | LEAF | Weight | 0 | 14 | 113.6 | 160.2 |
| 0.5716666667 | 0.3216666667 | High | | 1 | BP_Status | CLASS | DeathCause | 0.0138436095 | 600 | 160.2 | 206.8 |
| 0.5384615385 | 0.4230769231 | High | Borderline | 16 | Chol_Status | LEAF | Weight | 0 | 52 | 160.2 | 206.8 |
| 0.5086956522 | 0.4434782609 | Normal | Optimal | 1 | BP_Status | CLASS | Smoking_Status | 0.0309618307 | 460 | 113.6 | 160.2 |

***Output 4.14*** *Partial Results for the Scoring Table*

| Selected Records from Table _T_BC324E1E_7FE9FAB66B58 | | | | | | | |
|---|---|---|---|---|---|---|---|
| Sex | DeathCause | Chol_Status | BP_Status | Weight | Smoking_Status | _TargetLowerbd_ | _TargetUpperbd_ |
| Female | Cancer | Desirable | Optimal | 120 | Heavy (16-25) | 113.6 | 160.2 |
| Female | Coronary Heart Disease | High | Normal | 195 | Non-smoker | 113.6 | 160.2 |
| Female | Cancer | High | Normal | 117 | Non-smoker | 113.6 | 160.2 |
| Female | Coronary Heart Disease | High | Normal | 221 | Light (1-5) | 113.6 | 160.2 |
| Female | Cerebral Vascular Disease | Desirable | Optimal | 152 | Light (1-5) | 113.6 | 160.2 |

When the tree is scored with the DECISIONTREE statement (the last statement in the example), the misclassification rate information is printed to the SAS log.

***Output 4.15*** *Misclassification Rate Information*

```
NOTE: The misclassification rate for scoring the decision tree is 0.368401
      using table EXAMPLE.HEART with 5209 records out of 5209.
```

## Example 10: Performing a Multi-Dimensional Summary

### Details

This PROC IMSTAT example demonstrates creating multi-dimensional summaries of the Prdsale data set. Three set specifications are shown in the example. There is no limit to the number of set specifications that you can specify.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.prdsale; set sashelp.prdsale; run;

proc imstat data=example.prdsale;
    mdsummary actual / 1
/* 1 */  groupby=(region prodtype)
         formats=("$", "$")
         filter="(NOT (REGION='WEST'))",

/* 2 */  groupby=(region prodtype division)
         formats=("$", "$", "$")
         filter="(NOT (REGION='WEST')) AND (NOT (PRODUCT='SOFA'))"

/* 3 */  groupby=(region prodtype division year)
         formats=("$", "$", "$", "f4.") 2
         filter="(NOT (REGION='WEST'))";
run;
```

### Program Description

1. This MDSUMMARY statement uses the variable that is named actual. You can analyze more than one variable by adding the variable names before the slash.
2. Formats are enclosed in quotation marks and separated by commas. Numeric formats are also enclosed in quotation marks.

### Output

***Output 4.16*** *Results for Summarization of Actual by Region and Prodtype*

| Summary Statistics for Table WORK.PRDSALE | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| REGION | PRODTYPE | Column | Min | Max | N | Sum | Mean | Std Dev. | Coefficient of Variation | Number Missing |
| EAST | FURNITURE | ACTUAL | 4.0000 | 999.00 | 288 | 146471 | 508.58 | 283.91 | 55.8241 | 0 |
| EAST | OFFICE | ACTUAL | 13.0000 | 1000.00 | 432 | 223790 | 518.03 | 284.22 | 54.8653 | 0 |

***Output 4.17*** *Results for Summarization of Actual by Region, Prodtype, and Division*

| Summary Statistics for Table WORK.PRDSALE | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| REGION | PRODTYPE | DIVISION | Column | Min | Max | N | Sum | Mean | Std Dev. | Coefficient of Variation | Number Missing |
| EAST | FURNITURE | CONSUMER | ACTUAL | 15.0000 | 983.00 | 72 | 37608 | 522.33 | 266.56 | 51.0334 | 0 |
| EAST | FURNITURE | EDUCATION | ACTUAL | 9.0000 | 993.00 | 72 | 36262 | 503.64 | 290.27 | 57.6353 | 0 |
| EAST | OFFICE | CONSUMER | ACTUAL | 13.0000 | 1000.00 | 216 | 108686 | 503.18 | 286.72 | 56.9812 | 0 |
| EAST | OFFICE | EDUCATION | ACTUAL | 14.0000 | 996.00 | 216 | 115104 | 532.89 | 281.58 | 52.8406 | 0 |

***Output 4.18*** *Results for Summarization of Actual by Region, Prodtype, Division, and Year*

| Summary Statistics for Table WORK.PRDSALE | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| REGION | PRODTYPE | DIVISION | YEAR | Column | Min | Max | N | Sum | Mean | Std Dev. | Coefficient of Variation | Number Missing |
| EAST | FURNITURE | CONSUMER | 1993 | ACTUAL | 5.0000 | 983.00 | 72 | 37652 | 522.94 | 290.67 | 55.5827 | 0 |
| EAST | FURNITURE | CONSUMER | 1994 | ACTUAL | 63.0000 | 991.00 | 72 | 34918 | 484.97 | 272.17 | 56.1210 | 0 |
| EAST | FURNITURE | EDUCATION | 1993 | ACTUAL | 4.0000 | 999.00 | 72 | 37929 | 526.79 | 291.69 | 55.3715 | 0 |
| EAST | FURNITURE | EDUCATION | 1994 | ACTUAL | 32.0000 | 991.00 | 72 | 35972 | 499.61 | 284.58 | 56.9600 | 0 |
| EAST | OFFICE | CONSUMER | 1993 | ACTUAL | 30.0000 | 995.00 | 108 | 55612 | 514.93 | 284.78 | 55.3060 | 0 |
| EAST | OFFICE | CONSUMER | 1994 | ACTUAL | 13.0000 | 1000.00 | 108 | 53074 | 491.43 | 289.48 | 58.9063 | 0 |
| EAST | OFFICE | EDUCATION | 1993 | ACTUAL | 14.0000 | 996.00 | 108 | 56336 | 521.63 | 280.21 | 53.7179 | 0 |
| EAST | OFFICE | EDUCATION | 1994 | ACTUAL | 20.0000 | 987.00 | 108 | 58768 | 544.15 | 283.80 | 52.1558 | 0 |

## Example 11: Fitting a Regression Model

### Details

This IMSTAT procedure example demonstrates using higher-order polynomial models and also model selection.

Both REGCORR statements in the example request fitting a regression model of the response variable, Sales, and the regressor variable, Inventory. If no variables are specified, then the procedure fits a model for each pair of numeric non-CLASS variables in the table.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.shoes; set sashelp.shoes; run;

proc imstat data=example.shoes;
    regcorr sales inventory / order=2; 1
run;

    regcorr sales inventory / order=-3; 2
run;
quit;
```

### Program Description

1. This REGCORR statement uses the ORDER=2 option to specify a quadratic model. ORDER=1 requests a linear model and ORDER=3 requests a cubic model.

2. Specifying -3 for the ORDER= option indicates that the procedure perform model selection from linear, quadratic, and cubic models. The procedure finds the best-fitting polynomial that is most appropriate, according to statistical principles.

### Output

The following display shows the results for the two REGCORR statements. The first display shows the results for the quadratic model that was requested with ORDER=2. The second display shows the results for letting the procedure determine the best fit of the model. In this case, because the Quadr column has a nonzero value, that indicates the procedure determined that the quadratic model fits the data best. If the Cubic column been nonzero, that would indicate that the procedure selected the cubic model as the most appropriate.

| Linear Regression for Table WORK.SHOES | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Y | X | Intercept | Slope | Quadr | SS(Error) | SS(Total) | R-Square | Correlation Coefficient | Number of Obs | Mean(X) | Mean(Y) | Std(X) | Std(Y) |
| Sales | Inventory | 1259.40 | 0.3257 | 1.46E-8 | 5.755E11 | 6.567E12 | 0.9124 | . | 395 | 250899 | 85700 | 351515 | 129107 |

| Linear Regression for Table WORK.SHOES | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Y | X | Intercept | Slope | Quadr | Cubic | SS (Error) | SS (Total) | R-Square | Correlation Coefficient | Number of Obs | Mean (X) | Mean (Y) | Std(X) | Std(Y) |
| Sales | Inventory | 1259.40 | 0.3257 | 1.46E-8 | 0 | 5.755E11 | 6.567E12 | 0.9124 | . | 395 | 250899 | 85700 | 351515 | 129107 |

## Example 12: Forecasting and Automatic Modeling

### Details

This IMSTAT procedure example demonstrates using the FORECAST statement in its simplest use and when used with independent variables.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.pricedata;
    set sashelp.pricedata;
    where region=1 and product=1 and line=1;
run;

proc imstat data=example.pricedata;
    forecast date / vars =sale  1
                    lead =4
                    info;
run;

    forecast date / vars =sale
                    lead =4
                    info
                    indep=(price discount);  2

    ods output forecast out=work.forecast2;  3
quit;

proc sgplot data=work.forecast2;
```

```
    format date monyy7.; /* monyy5.; */
    band x=date lower=lower upper=upper /
       legendlabel="95% CLI" name="band";

    series x=date y=predict / lineattrs=GraphPrediction name="predict";
    series x=date y=actual / name="actual";

    keylegend "actual" "predict" "band" / location=inside
        position=bottomright;
run;
```

### Program Description

1. The first FORECAST statement shows the simplest usage. The Sale variable is forecasted and the Date variable is used as the time stamp for identifying the time series. The LEAD=4 option specifies to forecast four intervals into the future.
2. The second FORECAST statement is similar to the first, but specifies independent variables in the data. In this case, the server performs time series model building and variable selection. Variables Price and Discount are candidates for the independent variables.
3. The ODS statement is used to save the results of the second forecast in a temporary SAS data set that is named Forecast2. You can use the data set with the SGPLOT procedure or other graphics procedures for plotting the forecast. For information about plots, see "SGPLOT Procedure" in *SAS ODS Graphics: Procedures Guide*.

### Output

The following display shows the results for the two FORECAST statements. The first display shows the results for the forecast information and then the forecasted time series of the Sale variable in the Pricedata data set.

The Date column contains the value of the time stamp. Observed values of the time series are identified by a nonmissing value for the variable named Actual variable. For example, the mean value of Sale at Date=13880 is 355.00. The Predict column contains the predicted value under the chosen model and the Residual column is the difference between the observed value in the Actual column and the predicted value.

The StdErrPred column contains the standard error of the predicted value. This is a measure of the precision of predicting the value of Sale for the particular time stamp under the model used. The Lower and Upper columns are the confidence limits for the prediction.

The observations with missing values for column Actual at the end of the table contain the forecasted value in column Predict. Notice how the value of the prediction standard error grows quickly as the forecast extends beyond the observed time stamps. The width of the confidence interval grows accordingly. The further that you predict into the future, the less precise the prediction is. The result table contains several columns not shown in

the following display. These columns identify the table, the analysis variable, and the aggregator. You can materialize those columns by writing the table to a SAS data set.

| Time Series Forecast Information for Table HPS.PRICEDATA | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Time Stamp | Interval | Units | Seasonality | NStamps | NMissT | NPeriods | NFit | NFor | Run Status | Model Used |
| date | MONTH | d | 12 | 60 | 0 | 60 | 60 | 64 | OK | Simple Exponential Smoothing |

| Time Series Forecast for Table HPS.PRICEDATA | | | | | | |
|---|---|---|---|---|---|---|
| date | Actual | Predict | Residual | StdErr Pred | Lower | Upper |
| 13880 | 355.00 | 388.81 | -33.8069 | 56.1985 | 278.66 | 498.95 |
| 13911 | 398.00 | 388.13 | 9.8730 | 56.1985 | 277.98 | 498.27 |
| 13939 | 387.00 | 388.33 | -1.3256 | 56.1985 | 278.18 | 498.47 |
| 13970 | 380.00 | 388.30 | -8.2989 | 56.1985 | 278.15 | 498.45 |
| 14000 | 555.00 | [illegible] | [illegible] | 56.1985 | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | 46.6890 | [illegible] | [illegible] | 507.04 |
| 15645 | 344.00 | 395.91 | -51.9058 | 56.1985 | 285.76 | 506.05 |
| 15675 | 373.00 | 394.86 | -21.8619 | 56.1985 | 284.71 | 505.01 |
| 15706 | . | 394.42 | . | 56.1985 | 284.28 | 504.57 |
| 15737 | . | 394.42 | . | 56.2098 | 284.25 | 504.59 |
| 15765 | . | 394.42 | . | 56.2212 | 284.23 | 504.61 |
| 15796 | . | 394.42 | . | 56.2325 | 284.21 | 504.64 |

The second FORECAST statement specifies independent variables in the data. In this case, the server performs time series model building and variable selection and then returns the best-fitting time series model and values for the selected independent variables.

The forecast information table indicates that an ARIMA model with variable Price as the independent variable was chosen as the best-fitting model. Note that in automatic modeling mode it is possible that none of the independent variables specified in the INDEP= option are used in the final model. The model then falls back to an exponential smoothing model as in previous FORECAST statement.

In addition, when one or more independent variables are selected for the model, the output includes a table with the values for the independent variables. Notice that the independent variables are also forecast into the lead horizon. The last time stamp in the

input data set for the dependent and independent variables is Date=15675 with Price having an observed value of 52.3.

| Time Series Forecast Information for Table HPS.PRICEDATA | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Time Stamp | Interval | Units | Seasonality | NStamps | NMissT | NPeriods | NFit | NFor | NIndep | Run Status | Model Used |
| date | MONTH | d | 12 | 60 | 0 | 60 | 60 | 64 | 2 | OK | ARIMA: sale ~ P = 1 + INPUT: price |

| Time Series Forecast for Table HPS.PRICEDATA | | | | | | |
|---|---|---|---|---|---|---|
| date | Actual | Predict | Residual | StdErr Pred | Lower | Upper |
| 13880 | 355.00 | 379.29 | -24.2926 | 19.7512 | 340.58 | 418.00 |
| 13911 | 398.00 | 366.28 | 31.7237 | 19.7512 | 327.56 | 404.99 |
| 13939 | 387.00 | 389.32 | -2.3163 | 19.7512 | 350.60 | 428.03 |
| 13970 | 380.00 | 383.42 | -3.4223 | 19.7512 | 344.71 | 422.13 |
| 14000 | 555.00 | 581.90 | -26.8965 | 19.7512 | 543.18 | 620.61 |
| 14031 | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | 373.66 | -25.6572 | [illegible] | 334.95 | 412.37 |
| 15645 | 344.00 | 362.53 | -18.5256 | 19.7512 | 323.81 | 401.24 |
| 15675 | 373.00 | 360.38 | 12.6177 | 19.7512 | 321.67 | 399.09 |
| 15706 | . | 389.51 | . | 19.7512 | 350.80 | 428.23 |
| 15737 | . | 391.13 | . | 22.4078 | 347.21 | 435.05 |
| 15765 | . | 392.02 | . | 23.1142 | 346.72 | 437.32 |
| 15796 | . | 392.52 | . | 23.3130 | 346.83 | 438.21 |

| Time Series Independent Variables for Table HPS.PRICEDATA | |
|---|---|
| price | date |
| 52.300000 | 13880 |
| 52.300000 | 13911 |
| 52.300000 | 13939 |
| 52.300000 | 13970 |
| 44.455000 | 14000 |
| [illegible] | [illegible] |
| [illegible] | 15614 |
| 52.300000 | 15645 |
| 52.300000 | 15675 |
| 51.772694 | 15706 |
| 51.770643 | 15737 |
| 51.768655 | 15765 |
| 51.766731 | 15796 |

## Example 13: Forecasting with Goal Seeking

### Details

This IMSTAT procedure example demonstrates using FORECAST statement with a goal-seeking analysis.

Goal seeking is based on numerical optimization of control variables in order to produce a desired forecast. You can think of it as an inverse prediction method. Normal prediction techniques produce a predicted value when given a series of inputs. An inverse prediction method specifies the desired predicted value and then asks to find the inputs that generate it. For time series forecasting, the inverse prediction method is called goal seeking. Instead of producing a forecast for values of independent variables that you provide, you provide the target forecast (the goal). Then, the numerical optimization attempts to find the values of the independent variables that generate the goal values for the chosen model.

The independent variables in the time series model are divided into two categories. Those whose values can be modified during goal seeking are called controllable variables. The values of other independent variables are immutable during goal seeking. You specify the variables that can be modified during goal seeking in the CONTROL= option. You specify the variable that cannot be modified (the target) with the GOAL= option.

### Program

```
data work.pricedata;   1
    set sashelp.pricedata;
    where region=1 and product=1 and line=1;
run;

data work.goalsale;
    retain slast 0;
    keep region product line;
    keep date gsale;
    set pricedata end=last;
    if sale ne . then slast=sale;
    if last then do;
        gsale=slast;
        do i=1 to 4;  2
            gsale=1.05 * gsale;
            date=intnx("month", date, 1);
            output;
        end;
        stop;
    end;
run;

data work.merged;
    merge pricedata goalsale;
    by date;
run;

/*
proc print data=merged;  3
    var date sale price discount gsale;
    where date > '01jul2002'd;
run;
*/

proc imstat;
     forecast data=merged date / dep    =sale  4
                                 control=(price discount)
                                 goal   =gsale
                                 info
                                 lead   =4
                                 host   ="grid001.example.com"  5
                                 port   =10010;
quit;
```

### Program Description

1. The DATA step places a subset of the Sashelp.Pricedata data set into the temporary Work library.
2. The purpose of this DATA step is to generate four additional observations for the variable Gsale. The values for this variable represent the sales goal to attain.
3. The PRINT procedure can be used to view the last few observations from the original data set with the observed values for Sale and the target values for variable Gsale.
4. The FORECAST statement requests a goal seeking analysis for time stamp Date, with dependent variable Sale, control variables Price and Discount, and the goal variable Gsale.
5. This example also demonstrates how the DATA= option can be used with the HOST= and PORT= option to analyze a data set that is not in memory. This feature is unique to this statement. In this example, the Merged data set is transferred from the temporary Work library to the server and then analyzed.

### Output

The forecast information table shows that the automatic modeling step determined that the best-fitting model was an ARIMA model with independent variable Price.

Based on this model, the missing values for the control variables are replaced with forecasted values and the result is passed to the goal-seeking analysis. It produces the second and third tables.

The first sixty observations in the forecast table are the forecasted values from the automatic modeling step. The four observations at the end of the table are the result of goal seeking. Notice that the values in the Predict column for these four observations match the values for the Gsale column in the Merged data set. (The data set is not shown here, but is available if you remove the comments from the PROC PRINT statement.) The numerical optimization converged and the goal was met.

The third table, the independent variables table, shows the value for the control variable (Price). The first sixty values for Price match the values in the input table. The last four

values for the Price variable are the values for the control variable that produces the goal forecast.

| Time Series Forecast Information for Table | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Time Stamp | Interval | Units | Seasonality | NStamps | NMissT | NPeriods | NFit | NFor | Run Status | Model Used |
| date | MONTH | d | 12 | 64 | 0 | 60 | 60 | 64 | OK | ARIMA: sale ~ P = 1 + INPUT: price |

| Time Series Forecast for Table | | | | | | |
|---|---|---|---|---|---|---|
| date | Actual | Predict | Residual | StdErr Pred | Lower | Upper |
| 13880 | 355.00 | 379.29 | -24.2926 | 19.7512 | 340.58 | 418.00 |
| 13911 | 398.00 | 366.28 | 31.7237 | 19.7512 | 327.56 | 404.99 |
| 13939 | 387.00 | 389.32 | -2.3163 | 19.7512 | 350.60 | 428.03 |
| 13970 | 380.00 | 383.42 | -3.4223 | 19.7512 | 344.71 | 422.13 |
| 14000 | 555.00 | 581.90 | -26.8965 | 19.7512 | 543.18 | [illegible] |
| [illegible] | [illegible] | [illegible] | -18.5256 | [illegible] | 323.81 | 401.24 |
| 15675 | 373.00 | 360.38 | 12.6177 | 19.7512 | 321.67 | 399.09 |
| 15706 | . | 391.65 | . | 19.7512 | 352.94 | 430.36 |
| 15737 | . | 411.23 | . | 22.4078 | 367.31 | 455.15 |
| 15765 | . | 431.79 | . | 23.1142 | 386.49 | 477.10 |
| 15796 | . | 453.38 | . | 23.3130 | 407.69 | 499.08 |

| Time Series Independent Variables for Table | |
|---|---|
| price | date |
| 52.300000 | 13880 |
| 52.300000 | 13911 |
| 52.300000 | 13939 |
| 52.300000 | 13970 |
| 44.455000 | [illegible] |
| [illegible] | [illegible] |
| 52.300000 | 15675 |
| 51.689815 | 15706 |
| 50.990857 | 15737 |
| 50.225732 | 15765 |
| 49.405624 | 15796 |

## Example 14: Aggregating Time Series Data

### Details

This IMSTAT procedure example demonstrates using AGGREGATE statement with time series data.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.stocks;
    set sashelp.stocks;
run;

proc imstat data=example.stocks;
```

```
    where stock="IBM";
    aggregate close (agg=min) close (agg=max) close (agg=mean)  1
              / id=date idstart='01jan1998'd idend='31dec1998'd   2
                interval="quarter";  3
run;

    where same and date >= '01jan2003'd;  4

    aggregate close (agg=n) close (agg=mean) close (agg=stddev)
              / id=date idfmt="yyq6." interval="quarter" temptable;  5

    table example.&_templast_;
    fetch / format;  6

quit;
```

### Program Description

1. The three aggregate expressions calculate the minimum, maximum, and mean values of the Close variable.
2. The Date variable is used to identify the time associated with each observation. The IDSTART= and IDEND= options limit the series to a single year.
3. The INTERVAL= option specifies to aggregate the observations by quarter.
4. The WHERE SAME clause is used to add a clause to the existing WHERE statement. In this case, it is used to subset the time series.
5. The second AGGREGATE statement calculates the count, mean, and standard deviation of the Close variable. The Date variable is formatted as YYQ6. and the observations are aggregated by quarter. The TEMPTABLE option saves the results of the aggregation to a temporary in-memory table.
6. The TABLE statement sets the temporary table as the active table. The FETCH statement prints the formatted values of the table.

### Output

The first table shows the results of the first AGGREGATE statement. The second table shows the name of the temporary in-memory table that has the results of the second AGGREGATE statement. The third table shows the contents of the temporary table. The Date_f column shows the formatted values of the Date column that were applied with

the IDFMT= option. The other column headings include the name of the analysis variable, Close, and the aggregate method.

| Aggregation Statistics for HPS.STOCKS | | | | |
|---|---|---|---|---|
| Date | Column | Min | Max | Mean |
| 01JAN98 | Close | 98.75 | 104.44 | 102.353333 |
| 01APR98 | Close | 114.81 | 117.5 | 116.06 |
| 01JUL98 | Close | 112.62 | 132.5 | 124.54 |
| 01OCT98 | Close | 148.5 | 184.38 | 166.003333 |

| Temporary Table Information for Table HPS.STOCKS | |
|---|---|
| Statement | AGGREGATE |
| Temporary Table | _T_4B111C05_7F588B3EED68 |
| Table Type | AGGREGATE |

| Selected Records from Table _T_4B111C05_7F588B3EED68 | | | | |
|---|---|---|---|---|
| Date | Date_f | _Close_N_ | _Close_Mean_ | _Close_Std_ |
| 2003Q1 | 2003Q1 | 3 | 78.193333333 | 0.2400694344 |
| 2005Q2 | 2005Q2 | 3 | 75.376666667 | 1.1002878411 |
| 2003Q4 | 2003Q4 | 3 | 90.9 | 1.6300920219 |
| 2005Q1 | 2005Q1 | 3 | 92.46 | 1.0252804494 |
| 2004Q1 | 2004Q1 | 3 | 95.856666667 | 3.7367677655 |
| 2004Q2 | 2004Q2 | 3 | 88.303333333 | 0.2484619354 |
| 2004Q3 | 2004Q3 | 3 | 85.833333333 | 1.1927419391 |
| 2004Q4 | 2004Q4 | 3 | 94.19 | 4.4152123392 |
| 2003Q2 | 2003Q2 | 3 | 85.146666667 | 2.7782248529 |
| 2003Q3 | 2003Q3 | 3 | 83.863333333 | 3.8868667759 |
| 2005Q3 | 2005Q3 | 3 | 81.433333333 | 1.7665031371 |
| 2005Q4 | 2005Q4 | 3 | 84.326666667 | 3.9638533441 |

## Example 15: Training and Validating a Neural Network

### Details

This IMSTAT procedure example demonstrates using the NEURAL statement to train and validate a neural network.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';
```

```
data example.iris;
    set sashelp.iris;
    part=ranuni(12345);
run;


proc imstat data=example.iris;
   where part <= .75;
   neural species / seed=12345
       input=(sepallength sepalwidth petallength petalwidth) 1
       nominal=(species)
       hidden=(2)
       maxiter=1000 numtries=10
       lower=-20 upper=20
       code=(file="/data/iris_score.sas" replace)  2
       /* details */  3
       temptable;  4
   run;

   where part > .75;
   neural species /
       lasrann=example.&_TEMPLAST_  5
       idvars=species  6
       temptable;
   run;

   table example.&_TEMPLAST_;  7
   fetch / to=5 format;
   run;
quit;
```

### Program Description

1. The NEURAL statement models values of the species variable based on the input variables.
2. The CODE= option is used to save the scoring code to a file.
3. The DETAILS option is omitted from the results but can be useful during model development because it provides details about the iterations.
4. The TEMPTABLE option is used to save the weights from the training exercise to a temporary table.
5. The LASRANN= option is used to reference the temporary table that has the weights from the training exercise.
6. Specifying IDVARS= adds the target variable to the scored data set. This makes it easy to see both the actual value and the predicted value.
7. The TABLE statement is used to access the temporary table that includes the scoring results.

To score other data with the scoring program in the **`/data/iris_score.sas`** file, you can use the SCORE statement with the CODE= option.

### Output

The first display shows the results of the first NEURAL statement that is used to train the network.

**The IMSTAT Procedure**

| Model Information | |
|---|---|
| Model | Artificial Neural Network |
| Number of Observations Used | 114 |
| Number of Observations Read | 150 |
| Number of nodes | 9 |
| Number of nodes (input) | 4 |
| Number of nodes (output) | 3 |
| Number of nodes (hidden) | 2 |
| Number of hidden layers | 1 |
| Number of weight parameters | 12 |
| Number of bias parameters | 5 |
| Architecture | MLP |
| Number of neural nets | 1 |
| Object Value | 19.3622531964 |

| Temporary Table Information for Table WORK.IRIS | |
|---|---|
| Statement | NEURAL |
| Temporary Table | _T_E1EB126F_FE58350 |
| Table Type | NEURAL |

The second display shows the results of the second NEURAL statement that is used to validate the model. The Selected Records table shows the first five records of the scoring results table that were read with the FETCH statement.

| Score Information by Model | |
|---|---|
| **Model** | Artificial Neural Network |
| **Number of nodes** | 9 |
| **Number of nodes (input)** | 4 |
| **Number of nodes (output)** | 3 |
| **Number of nodes (hidden)** | 2 |
| **Number of hidden layers** | 1 |
| **Number of weight parameters** | 12 |
| **Number of Observations Used** | 36 |
| **Number of Observations Read** | 150 |
| **Misclassification Error (%)** | 0 |

| Temporary Table Information for Table WORK.IRIS | |
|---|---|
| **Statement** | NEURAL |
| **Temporary Table** | _T_E21CE769_10F787A0 |
| **Table Type** | Scoring results |

| Selected Records from Table _T_E21CE769_10F787A0 | | |
|---|---|---|
| **Species** | **NN_Species** | **NN_P_Species** |
| Setosa | Setosa | 0.9977987582 |
| Setosa | Setosa | 0.9977945245 |
| Setosa | Setosa | 0.9977945497 |
| Setosa | Setosa | 0.9977985977 |
| Setosa | Setosa | 0.9977987616 |

## Example 16: Predicting Email Spam and Assessing the Model

### Details

This IMSTAT procedure example demonstrates using the NEURAL statement to train a neural network. The ASSESS option is used to generate predicted probabilities and include them in the scored data. The ASSESS statement is then used to generate lift and receiver operating characteristic (ROC) information.

The data set is from a study on classifying whether an email is junk email (coded as 1) or not (coded as 0). The data were collected in Hewlett-Packard labs and donated by George Forman. The data set contains 4,601 observations with 58 variables. The response variable is a binary indicator of whether an email is considered spam or not. The 57 variables are continuous variables that record frequencies of some common words and characters in emails and lengths of uninterrupted sequences of capital letters. The data set is publicly available at the UCI Machine Learning repository (Asuncion and Newman, 2007).

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';


%let base = http://archive.ics.uci.edu/ml/machine-learning-databases;
data spambase;
   infile "&base/spambase/spambase.data" device=url dsd dlm=',';
   input Make Address All _3d Our Over Remove Internet Order Mail Receive
       Will People Report Addresses Free Business Email You Credit Your Font
       _000 Money Hp Hpl George _650 Lab Labs Telnet _857 Data _415 _85
       Technology _1999 Parts Pm Direct Cs Meeting Original Project Re Edu
       Table Conference Semicol Paren Bracket Bang Dollar Pound Cap_Avg
       Cap_Long Cap_Total Class;
run;

data example.spambase;
    set spambase;
    part = ranuni(12345);
run;

proc imstat data=example.spambase;   1
    where part <= .75;
    neural class / seed=12345
      input=(make--cap_total)
      nominal=(class)
      hidden=(10) act=(logistic)
      numtries=5 maxiter=50 tech=congra
      maxfunc=2147483647 fconv=1e-4
      lower=-20 upper=20
      temptable /* details */;  2
run;

    neural class / seed=12345  3
      resume lasrann=example.&_templast_
```

```
      input=(make--cap_total)
      nominal=class
      hidden=(10) act=(logistic)
      tech=congra maxiter=50
      maxfunc=2147483647
      fconv=1e-4 lower=-20 upper=20
      temptable /* details */;
run;

    where part > .75;
    neural class / lasrann=example.&_templast_
      input=(make--cap_total)
      nominal=class
      temptable assess  4
      vars=(class);
run;

    table example.&_templast_;
    where strip(_NN_Level_) eq '1';
    assess _NN_P_ / y=class event='1'  5
       nbins=20 step=0.05;
    ods output liftinfo=work.liftdata;
    ods output rocinfo=work.rocdata;
quit;

proc sgplot data=work.liftdata;  6
    title 'Lift Chart';
    series x=depth y=Cumlift /
       markers markerattrs=(symbol=circlefilled);
    series x=depth y=CumliftBest;
    yaxis label=' ' grid;
run;

data work.endpoint;  7
    sensitivity=0;
    specificity=1;
run;

data work.rocdata1;
    set work.rocdata work.endpoint;
    one_minus_specificity=1-specificity;
run;

proc sort data=work.rocdata1;
    by one_minus_specificity;
run;

/* Plot ROC curve */
ods graphics on / width=480px height=480px;
proc sgplot data=work.rocdata1;  8
    title 'ROC Curve';
    series x=one_minus_specificity y=sensitivity /
      lineattrs=(color=blue);
    series x=one_minus_specificity y=one_minus_specificity /
      lineattrs=(color=black);
    yaxis grid;
```

```
quit;
```

### Program Description

1. The first NEURAL statement is used to pretrain several shallow neural networks, starting from different points to avoid creating a neural network that is ineffective due to poor initial values.
2. The TEMPTABLE option is used to store the parameter estimates from the training in an in-memory table.
3. The second NEURAL statement selects the best neural network from the pretrained neural networks and resumes the analysis to train a much deeper neural network as the final model.
4. The ASSESS option specifies to add predicted probabilities to the scored data for all the levels of the nominal target variable. In this example, two levels are created because the variable named class has two values, 0 or 1. The scored data are stored in a temporary table.
5. The ASSESS statement uses the scoring result to perform model assessment. The probabilities of all levels are output, but we need the probabilities of the event level only. The WHERE clause is used to select the rows with event level only. The strip function is applied to remove the blanks in the character variable _NN_Level_.
6. The SGPLOT procedure is used to plot a lift chart.
7. The first DATA step adds the (0, 1) end point to the data set. The results of the ASSESS statement do not always include the end point. The second DATA set adds the end point to the ROC data set and also calculates a new variable.
8. The SGPLOT procedure is used to plot the ROC curve. The ODS WIDTH= and HEIGHT= options are used to ensure that the plot is square.

### Output

The first display shows the results of the first NEURAL statement that is used to train the network.

**The IMSTAT Procedure**

| Model Information | |
|---|---|
| Model | Artificial Neural Network |
| Number of Observations Used | 3429 |
| Number of Observations Read | 4601 |
| Number of nodes | 69 |
| Number of nodes (input) | 57 |
| Number of nodes (output) | 2 |
| Number of nodes (hidden) | 10 |
| Number of hidden layers | 1 |
| Number of weight parameters | 580 |
| Number of bias parameters | 12 |
| Architecture | MLP |
| Number of neural nets | 1 |
| Object Value | 978.174707132 |

| Temporary Table Information for Table HPS.SPAMBASE | |
|---|---|
| Statement | NEURAL |
| Temporary Table | _T_FBB6501A_7F8566869C90 |
| Table Type | NEURAL |

The second display shows the results of the second NEURAL statement that is used to resume training and develop a deeper model.

| Model Information | |
|---|---|
| Model | Artificial Neural Network |
| Number of Observations Used | 3429 |
| Number of Observations Read | 4601 |
| Number of nodes | 69 |
| Number of nodes (input) | 57 |
| Number of nodes (output) | 2 |
| Number of nodes (hidden) | 10 |
| Number of hidden layers | 1 |
| Number of weight parameters | 580 |
| Number of bias parameters | 12 |
| Architecture | MLP |
| Number of neural nets | 1 |
| Object Value | 611.926107029 |

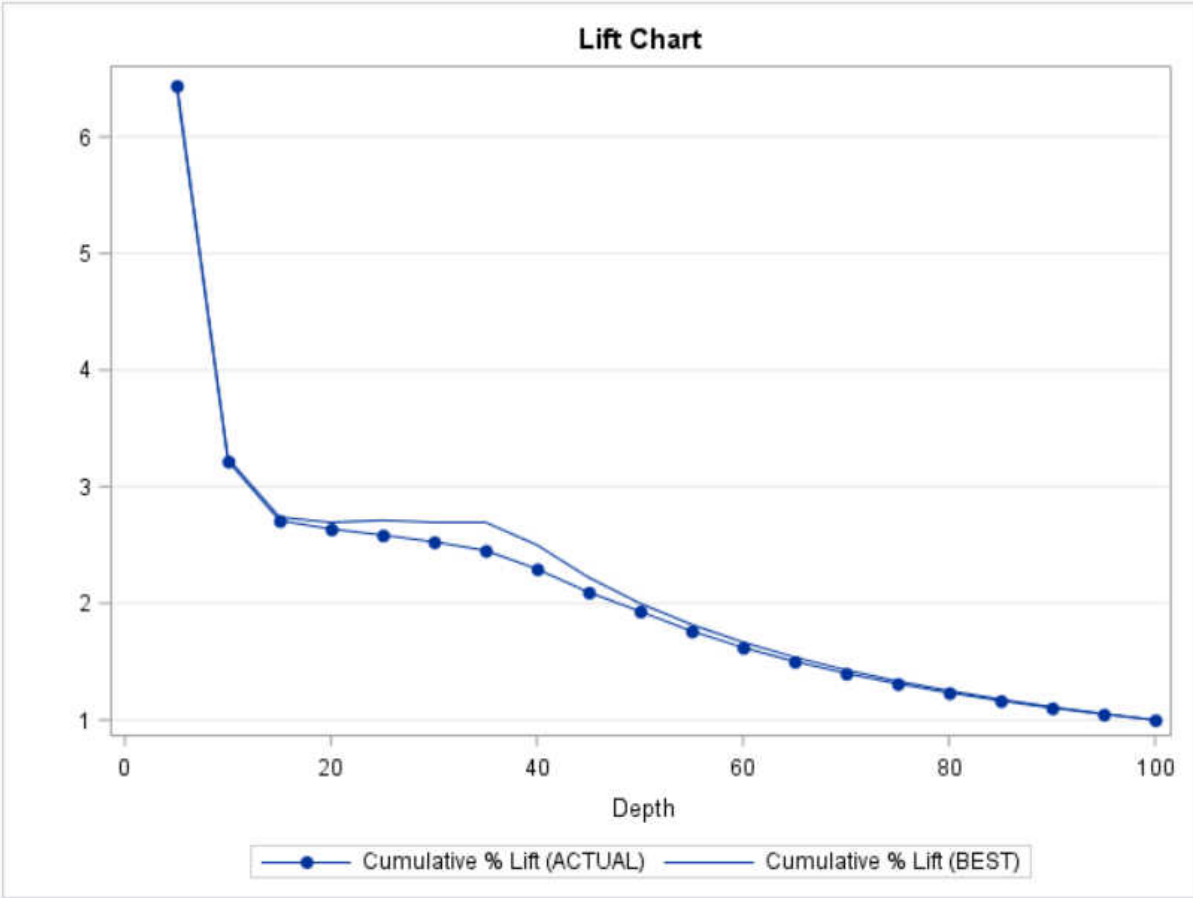
| Temporary Table Information for Table HPS.SPAMBASE | |
|---|---|
| Statement | NEURAL |
| Temporary Table | _T_FBB710F3_7F857815EC90 |
| Table Type | NEURAL |

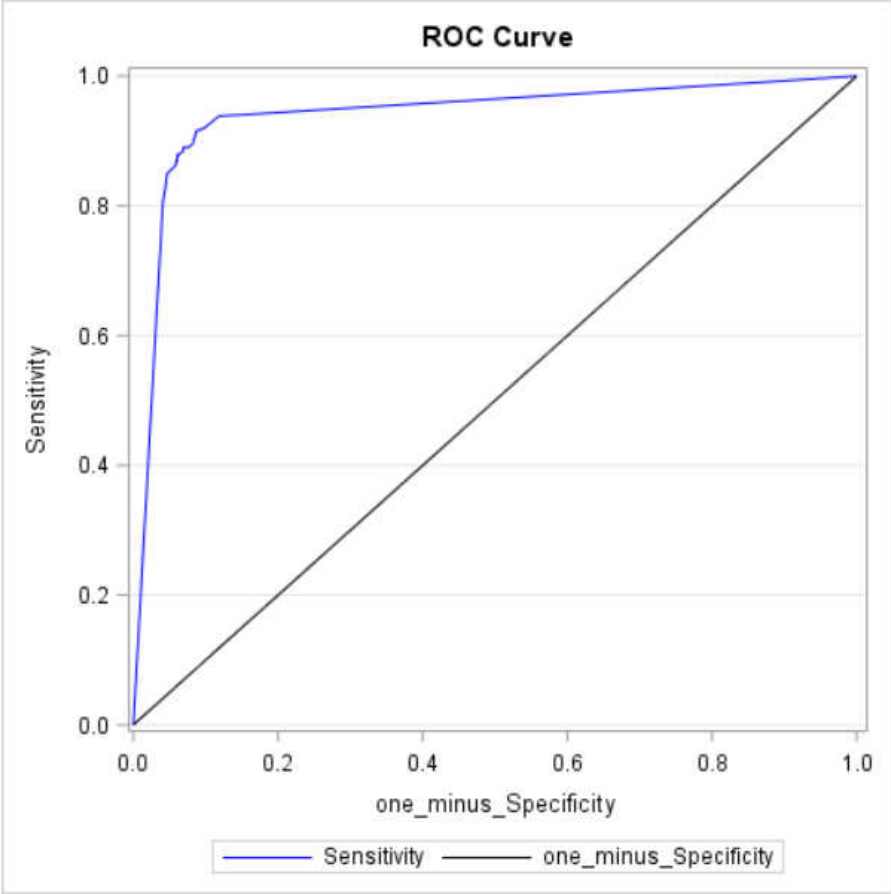
The third display shows the results of the ASSESS statement that provides the scoring results. Two additional ODS tables are created, a lift information table and a ROC information table. These are not shown here because the tables are wide.

| Score Information by Model | |
|---|---|
| **Model** | Artificial Neural Network |
| **Number of nodes** | 69 |
| **Number of nodes (input)** | 57 |
| **Number of nodes (output)** | 2 |
| **Number of nodes (hidden)** | 10 |
| **Number of hidden layers** | 1 |
| **Number of weight parameters** | 580 |
| **Number of Observations Used** | 1172 |
| **Number of Observations Read** | 4601 |
| **Misclassification Error (%)** | 8.61774744027 |

| Temporary Table Information for Table HPS.SPAMBASE | |
|---|---|
| **Statement** | NEURAL |
| **Temporary Table** | _T_FBB77C03_7F8564989DA0 |
| **Table Type** | Scoring results |

The following two displays show the lift chart and the ROC curve for the scoring results.






## Example 17: Transforming Variables with Imputation and Binning

### Details

This IMSTAT procedure example demonstrates using the TRANSFORM statement to impute values for missing variables, bin the variables into equal weight bins, and then use the FREQUENCY statement to see the distribution of values in the bins.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.fitness1;  1
   input Oxygen RunTime RunPulse @@;
   datalines;
44.609  11.37  178     45.313  10.07  185
54.297   8.65  156     59.571    .      .
49.874   9.22    .     44.811  11.63  176
  .     11.95  176          .  10.85    .
39.442  13.08  174     60.055   8.63  170
50.541    .      .     37.388  14.03  186
44.754  11.12  176     47.273    .      .
51.855  10.33  166     49.156   8.95  180
40.836  10.95  168     46.672  10.00    .
```

```
46.774  10.25    .     50.388  10.08  168
39.407  12.63  174     46.080  11.17  156
45.441   9.63  164       .      8.92    .
45.118  11.08    .     39.203  12.88  168
45.790  10.47  186     50.545   9.93  148
48.673   9.40  186     47.920  11.50  170
47.467  10.50  170
;
run;


proc imstat;
    table ml.fitness1;  2
    transform
      (impute=mean input=(oxygen runtime) name="meanBinned"   bin=bucket(10))  3
      (impute=median input=(runpulse)     name="medianBinned" bin=bucket(10))
       / details score code(file="/data/scorefitness1.sas" comment replace);  4
run;

  table example._&TEMPLAST_;
  fetch / format to=5;   5
run;

  frequency me:;  6
quit;
```

### Program Description

1. The data are loaded serially into the server with a DATA step.
2. The TABLE statement references the in-memory table that was created with the DATA step.
3. The first transform request is to impute values of the Oxygen and RunTime variables using the mean values for each variable. The imputed values and nonmissing original values are binned into 10 equal width bins. The second transform request imputes values of the RunPulse variable using the median value. Values for that variable are also binned into 10 equal width bins.
4. The DETAILS option specifies to display information about the bins. The SCORE option specifies to create an in-memory temporary table that has results of imputing and binning the variables. The CODE option, with the FILE= suboption, writes DATA step scoring code to a file on the SAS client.
5. The TABLE statement references the temporary in-memory table that has the scored results. The FETCH statement displays the first five rows from the table.
6. The FREQUENCY statement displays the distribution of the three binned variables. Because the NAME= options for both transformation requests begin with "me," meanBinned and medianBinned, the names of the scored variables can be referenced with the colon wildcard operator.

### Output

The first display shows the results of the TRANSFORM statement.

**The IMSTAT Procedure**

| Transformation Description | |
|---|---|
| Number of Transformation Variables | Binning Method |
| 2 | Bucket |
| 1 | Bucket |

| Summary Statistics for the Numeric Variables | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Variable Name | Number Transformations | Number Obs | Number Missing | Min | Max | Sum | Mean | Std Dev. |
| Oxygen | 1 | 28 | 3 | 37.3880 | 60.0550 | 1319.25 | 47.1162 | 5.4130 |
| RunPulse | 1 | 22 | 9 | 148.00 | 186.00 | 3781.00 | 171.86 | 10.1432 |
| RunTime | 1 | 28 | 3 | 8.6300 | 14.0300 | 299.27 | 10.6882 | 1.3799 |

| Variable Transformation Results | | | |
|---|---|---|---|
| Variable Name | Transformation Name | Result Variable | Number of Bins |
| Oxygen | meanBinned | meanBinned_Oxygen | 10 |
| RunPulse | medianBinned | medianBinned_RunPulse | 10 |
| RunTime | meanBinned | meanBinned_RunTime | 10 |

The second display shows the binning information that was requested with the DETAILS option.

| Continuous Variables Binning Information | | | | | | | |
|---|---|---|---|---|---|---|---|
| Transformation Name | Variable Name | Bin Id | Range | Number of Obs in Bin | Min Value in Bin | Max Value in Bin | Sum of Values in Bin |
| meanBinned | Oxygen | 1 | 37.388 - 39.6547 | 4 | 37.388 | 39.442 | 155.44 |
| meanBinned | Oxygen | 2 | 39.6547 - 41.9214 | 1 | 40.836 | 40.836 | 40.836 |
| meanBinned | Oxygen | 3 | 41.9214 - 44.1881 | 0 | . | . | . |
| meanBinned | Oxygen | 4 | 44.1881 - 46.4548 | 8 | 44.609 | 46.08 | 361.916 |
| meanBinned | Oxygen | 5 | 46.4548 - 48.7215 | 6 | 46.672 | 48.673 | 284.779 |
| meanBinned | Oxygen | 6 | 48.7215 - 50.9882 | 5 | 49.156 | 50.545 | 250.504 |
| meanBinned | Oxygen | 7 | 50.9882 - 53.2549 | 1 | 51.855 | 51.855 | 51.855 |
| meanBinned | Oxygen | 8 | 53.2549 - 55.5216 | 1 | 54.297 | 54.297 | 54.297 |
| meanBinned | Oxygen | 9 | 55.5216 - 57.7883 | 0 | . | . | |
| meanBinned | Oxygen | [illegible] | [illegible] | 2 | [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | 41.55 |
| meanBinned | RunTime | 5 | 10.79 - 11.33 | 5 | 10.85 | 11.17 | 55.17 |
| meanBinned | RunTime | 6 | 11.33 - 11.87 | 3 | 11.37 | 11.63 | 34.5 |
| meanBinned | RunTime | 7 | 11.87 - 12.41 | 1 | 11.95 | 11.95 | 11.95 |
| meanBinned | RunTime | 8 | 12.41 - 12.95 | 2 | 12.63 | 12.88 | 25.51 |
| meanBinned | RunTime | 9 | 12.95 - 13.49 | 1 | 13.08 | 13.08 | 13.08 |
| meanBinned | RunTime | 10 | 13.49 - 14.03 | 1 | 14.03 | 14.03 | 14.03 |
| medianBinned | RunPulse | 1 | 148 - 151.8 | 1 | 148 | 148 | 148 |
| medianBinned | RunPulse | 2 | 151.8 - 155.6 | 0 | . | . | . |
| medianBinned | RunPulse | 3 | 155.6 - 159.4 | 2 | 156 | 156 | 312 |
| medianBinned | RunPulse | 4 | 159.4 - 163.2 | 0 | . | . | . |
| medianBinned | RunPulse | 5 | 163.2 - 167 | 2 | 164 | 166 | 330 |
| medianBinned | RunPulse | 6 | 167 - 170.8 | 6 | 168 | 170 | 1014 |
| medianBinned | RunPulse | 7 | 170.8 - 174.6 | 2 | 174 | 174 | 348 |
| medianBinned | RunPulse | 8 | 174.6 - 178.4 | 4 | 176 | 178 | 706 |
| medianBinned | RunPulse | 9 | 178.4 - 182.2 | 1 | 180 | 180 | 180 |
| medianBinned | RunPulse | 10 | 182.2 - 186 | 4 | 185 | 186 | 743 |

The third display shows the first five records from the temporary in-memory table that has the scored values. In this case, the original variables, Oxygen, RunPulse, and

RunTime, are paired with the imputed and binned values. The NAME= option from each transform request is used as a prefix to the new variables.

| Temporary Table Information for Table HPS.FITNESS1 | |
|---|---|
| **Statement** | TRANSFORM |
| **Temporary Table** | _T_4FEF21E_139C6F10 |
| **Table Type** | Scoring results |

| Selected Records from Table _T_4FEF21E_139C6F10 | | | | | |
|---|---|---|---|---|---|
| **Oxygen** | **RunPulse** | **RunTime** | **meanBinned_Oxygen** | **meanBinned_RunTime** | **medianBinned_RunPulse** |
| 44.609 | 178 | 11.37 | 4 | 6 | 8 |
| 45.313 | 185 | 10.07 | 4 | 3 | 10 |
| 54.297 | 156 | 8.65 | 8 | 1 | 3 |
| 59.571 | . | . | 10 | 4 | 7 |
| 49.874 | . | 9.22 | 6 | 2 | 7 |

The following display shows the results of the FREQUENCY statement for the meanBinned_Oxygen variable. The results for the meanBinned_RunTime and medianBinned_RunPulse variables is similar.

| Frequencies for Column meanBinned_Oxygen in Table _T_4FEF21E_139C6F10 | | | |
|---|---|---|---|
| **Level** | **Formatted** | **Value** | **Frequency** |
| 1 | 1 | 1 | 4 |
| 2 | 2 | 2 | 1 |
| 3 | 4 | 4 | 8 |
| 4 | 5 | 5 | 9 |
| 5 | 6 | 6 | 5 |
| 6 | 7 | 7 | 1 |
| 7 | 8 | 8 | 1 |
| 8 | 10 | 10 | 2 |

*Chapter 5*

# IMSTAT Procedure (Data and Server Management)

# Overview: IMSTAT Procedure (Data and Server Management)

### *What Do the Data and Server Management Statements for the IMSTAT Procedure Do?*

This portion of the IMSTAT procedure is used to manage in-memory tables and SAS LASR Analytic Server instances.

# Concepts: IMSTAT Procedure

### *Partitioned Tables*

A SAS LASR Analytic Server table can be partitioned, and it can also be ordered within each partition. A partition is a collection of observations that share the same key value and are located on the same worker node. The key value can be constructed from one or more variables. The partitioning keys are created according to the formatted values of the specified variables. Partitioning of data is an important tool in managing distributed data sources. The process of partitioning can consume computing and network resources as well as create greater imbalance in the data compared to a round-robin distribution. However, partitioned data can be accessed more quickly and subsequent procedure statements can execute more quickly.

You can achieve a partitioned table as follows:

- load a table with the SAS LASR Analytic Server engine using the PARTITION= data set option
- load a SASHDAT file that has been previously partitioned into HDFS by using the PARTITION= data set option of the SASHDAT engine
- re-partition data from one table into a temporary table (you can then make it a regular table with the PROMOTE statement)

- create a temporary table with a GROUPBY= option. The temporary table is partitioned by the formatted values of the GROUPBY= variables.

For more information, see "Data Partitioning and Ordering" on page 25.

### RUN-Group Processing

The IMSTAT procedure supports RUN-group processing. RUN-group processing enables you to submit RUN groups without ending the procedure. This feature is particularly useful for running SAS interactively. You can start the procedure with the PROC IMSTAT statement and then execute statements like SUMMARY and FREQUENCY. Each statement runs when it reaches a RUN statement.

To use RUN-group processing, you start the procedure and then submit multiple RUN-groups. A RUN-group is a group of statements that contains at least one action statement and ends with a RUN statement. As long as you do not terminate the procedure, it remains active and you do not need to resubmit the PROC statement.

To end RUN-group processing, submit a RUN CANCEL statement. Statements that have not been submitted are terminated. To stop the procedure, submit a QUIT statement. Statements that have not been submitted are terminated as well.

### WHERE Clause Processing

There are two important features for WHERE clause processing that are related to the IMSTAT procedure. The first is that the WHERE clause is applied to the data by the server. When you use a WHERE clause to subset data, the subsetting is performed by the server. Only the rows that meet the WHERE clause criteria are affected by subsequent operations.

The second important feature for WHERE clause processing is related to the RUN-group processing that the IMSTAT procedure supports. You can modify the WHERE clause between statements. Unless a WHERE clause is specified in a RUN block, no subsetting of rows occurs. In the following code example, the SUMMARY statement in the first RUN-group is not subject to a WHERE clause. The FREQUENCY statement in the second RUN-group applies only to observations for which Division='EDUCATION.'

```
proc imstat data=example.prdsale(tag=sashelp);
    summary actual predict / groupby=(region);
run;
    where division='EDUCATION';
    frequency prodtype;
run;
```

If you specify WHERE clauses in different RUN blocks, the clauses replace each other. A note is written to the SAS log to indicate the change. For example, the SUMMARY statement in the following code example applies to observations for which the Division='CONSUMER.' The FREQUENCY statement applies to observations for which Region='EAST.'

```
proc imstat data=example.prdsale(tag=sashelp);
    where division='CONSUMER';
    summary actual predict / groupby=(region);
run;
    where region='EAST';
    frequency prodtype;
run;
```

When the FREQUENCY statement runs, the following line is added to the SAS log.

```
NOTE: WHERE clause has been replaced.
```

WHERE clauses can remain active across RUN statements. The following example is the same as the previous example, except that the second WHERE clause is not submitted.

```
proc imstat data=example.prdsale(tag=sashelp);
    where division='CONSUMER';
    summary actual predict / groupby=(region);
run;
    /* where region='EAST'; */
    frequency prodtype;
run;
```

In this case, the SAS log includes the following note.

```
NOTE: A WHERE clause remains active from a previous RUN
      block: '(division='CONSUMER')'.
```

You can clear a WHERE clause by submitting **WHERE;**.

Each time you access a different table with the TABLE statement, the WHERE clause is cleared. In following example, the second FREQUENCY statement is not restricted to observations for which Region='EAST' because the TABLE statement that accesses Prdsal2 clears the WHERE clause.

```
proc imstat;
    table example.prdsale(tag=sashelp);
    where region='EAST';
    frequency prodtype;
run;

    table example.prdsal2(tag=sashelp);
    frequency prodtype;
run;
```

The SAS log indicates that the WHERE clause is no longer applied.

```
NOTE: The WHERE statement is cleared when you open a LASR Analytic Server
      table with the TABLE statement.
```

## *Temporary Tables*

A temporary table is an in-memory table that contains the result set of a procedure statement. Instead of transferring the results to the client SAS session, the results remain in the server and only the name of the temporary table is transferred to the client. You can then use other procedure statements with the temporary table.

Temporary tables can be partitioned and the SUMMARY, CROSSTAB, DISTINCT, and PERCENTILE statements perform this action. For non-partitioned data, you can also generate temporary tables with the SUMMARY and CROSSTAB statements, provided that you request a GROUPBY analysis.

The following DATA step shows how to create a partitioned table on the variables Country and Region.

```
data lasr.prdsale(partition=(country region));
    set sashelp.prdsale;
run;
```

The following statements generate a summary analysis for variables Actual and Predict in each of the partitions.

```
proc imstat;
   table lasr.prdsale;
   summary actual predict / partition;
run;
```

The output for the previous statements is as follows:

**The SAS System**

**The IMSTAT Procedure**

| Summary Statistics for Table WORK.PRDSALE | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COUNTRY | REGION | Column | Min | Max | N | Sum | Mean | Std Dev. | Std Error | Coefficient of Variation | Number Missing |
| CANADA | EAST | ACTUAL | 5.0000 | 999.00 | 240 | 127485 | 531.19 | 281.61 | 18.1781 | 53.0160 | 0 |
| CANADA | EAST | PREDICT | 0 | 986.00 | 240 | 120646 | 502.69 | 284.29 | 18.3510 | 56.5540 | 0 |
| CANADA | WEST | ACTUAL | 3.0000 | 1000.00 | 240 | 119505 | 497.94 | 296.53 | 19.1406 | 59.5507 | 0 |
| CANADA | WEST | PREDICT | 6.0000 | 1000.00 | 240 | 112373 | 468.22 | 275.99 | 17.8150 | 58.9443 | 0 |
| GERMANY | EAST | ACTUAL | 13.0000 | 1000.00 | 240 | 124547 | 518.95 | 287.71 | 18.5714 | 55.4406 | 0 |
| GERMANY | EAST | PREDICT | 4.0000 | 993.00 | 240 | 117579 | 489.91 | 292.43 | 18.8762 | 59.6901 | 0 |
| GERMANY | WEST | ACTUAL | 3.0000 | 996.00 | 240 | 121451 | 506.05 | 289.17 | 18.6658 | 57.1429 | 0 |
| GERMANY | WEST | PREDICT | 0 | 981.00 | 240 | 113975 | 474.90 | 280.49 | 18.1056 | 59.0637 | 0 |
| U.S.A. | EAST | ACTUAL | 4.0000 | 984.00 | 240 | 118229 | 492.62 | 282.26 | 18.2200 | 57.2983 | 0 |
| U.S.A. | EAST | PREDICT | 1.0000 | 1000.00 | 240 | 120587 | 502.45 | 301.35 | 19.4518 | 59.9757 | 0 |
| U.S.A. | WEST | ACTUAL | 6.0000 | 984.00 | 240 | 119120 | 496.33 | 285.67 | 18.4401 | 57.5567 | 0 |
| U.S.A. | WEST | PREDICT | 22.0000 | 999.00 | 240 | 121135 | 504.73 | 280.10 | 18.0801 | 55.4944 | 0 |

As an alternative, you can leave the result set in an in-memory table by adding the TEMPTABLE option to the SUMMARY statement:

```
   summary actual predict / partition temptable;
run;
```

The previous SAS statements generate the following output in the SAS session.

**The SAS System**

**The IMSTAT Procedure**

| Temporary Table Information for Table WORK.PRDSALE | |
|---|---|
| Statement | SUMMARY |
| Temporary Table | _T_9B073B87_10F2B178 |

The temporary table is assigned a name by the server. When the IMSTAT procedure ends, any temporary tables created during the procedure run are removed from the server. Since the generated name is not predictable, the procedure assigns the name of the most recently generated temporary table to the _TEMPLAST_ macro variable.

You can use the TABLE statement to switch the active table to the temporary table and perform analyses. Make sure that the statement that generated the temporary table is

separated from the next statement with a RUN statement. Otherwise, you receive an error that the table specified in the TABLE statement does not exist. The temporary table does not exist at parse time, it is created at run time when the statement is executed.

The following statements retrieve information about the temporary table, the formatted values for (up to) the first twenty rows, and perform a summarization:

```
    table lasr.&_templast_
      tableinfo;
      columninfo;
      fetch / from=1 to=20 format;
      summary;
  quit;
```

The output for the TABLEINFO, COLUMNINFO, and FETCH statements is not shown. The results for the SUMMARY statement are as follows:

Summary Statistics for Table _T_9B077204_10F2B178

| Column | Min | Max | N | Sum | Mean | Std Dev. | Std Error | Coefficient of Variation | Number Missing |
|---|---|---|---|---|---|---|---|---|---|
| _Min_ | 0 | 22.0000 | 12 | 67 | 5.5833 | 6.2298 | 1.7984 | 111.58 | 0 |
| _Max_ | 981.00 | 1000.00 | 12 | 11922 | 993.50 | 7.5619 | 2.1829 | 0.7611 | 0 |
| _N_ | 240.00 | 240.00 | 12 | 2880 | 240.00 | 0 | 0 | 0 | 0 |
| _NMiss_ | 0 | 0 | 12 | 0 | 0 | 0 | 0 | . | 0 |
| _Mean_ | 468.22 | 531.19 | 12 | 5985.96667 | 498.83 | 17.0597 | 4.9247 | 3.4199 | 0 |
| _Sum_ | 112373 | 127485 | 12 | 1436632 | 119719 | 4094.34 | 1181.93 | 3.4199 | 0 |
| _Std_ | 275.99 | 301.35 | 12 | 3437.59539 | 286.47 | 7.3829 | 2.1313 | 2.5772 | 0 |
| _StdErr_ | 17.8150 | 19.4518 | 12 | 221.895828 | 18.4913 | 0.4766 | 0.1376 | 2.5772 | 0 |
| _Var_ | 76170 | 90809 | 12 | 985354.753 | 82113 | 4263.82 | 1230.86 | 5.1926 | 0 |
| _USS_ | 70820047 | 86672749 | 12 | 952904056 | 79408671 | 4419132 | 1275694 | 5.5651 | 0 |
| _CSS_ | 18204667 | 21703393 | 12 | 235499786 | 19624982 | 1019053 | 294175 | 5.1926 | 0 |
| _CV_ | 53.0160 | 59.9757 | 12 | 689.727293 | 57.4773 | 2.1126 | 0.6098 | 3.6755 | 0 |
| _T_ | 25.8303 | 29.2212 | 12 | 323.848629 | 26.9874 | 1.0171 | 0.2936 | 3.7687 | 0 |
| _PRT_ | 7.49E-81 | 4.06E-71 | 12 | 7.4196E-71 | 6.18E-72 | 1.22E-71 | 3.52E-72 | . | 0 |

## Creating Temporary Tables with GROUPBY Variables

Temporary tables play an important role in partitioning in the server. Temporary tables that are created by the DISTINCT, SUMMARY, CROSSTAB, or PARTITION statements are partitioned.

If the input table is not partitioned, you can still use temporary tables with the SUMMARY and CROSSTAB statements, provided that you perform a group-by analysis. The temporary table that is created by the server is automatically partitioned by the group-by variables. This potentially involves a redistribution of the groups in the result set to transfer all the result records for a particular group to the same worker node.

## Creating Temporary Variables

You can use temporary variables in a table. For example, if a table has variables that are named x1, x2, and x3, you can calculate the summary statistics for variable d1 = x1 + x2 / 3*x3. One way is to declare d1 as a temporary variable of the table (with data set options for the input table). You can use the temporary variables (temporary for the duration of each statement) with DATA step expression scripts.

```
proc imstat data=lasrlib.table1(array=(d,1));
```

```
    summary d1 x1 -- x3 / tempnames=d1 tempexpress="d1 = x1 + x2 / 3*x3;";
run;
    summary d1 / tempnames=d1 tempexpress="d1 = mean(x1, x2);";
quit;
```

Because the temporary variable exists for the duration of the statement, its name can be reused in subsequent statements. The second SUMMARY statement uses the same name, d1, for the temporary variable but it has a different value.

You can also create temporary character variables. The following example creates a program that concatenates the first character of the Type variable and the first character of the Origin variable.

```
libname lasrlib sasiola host="hostname.example.com" port=10010 tag=hps;

data lasrlib.cars; set sashelp.cars; run;

data _null_;
    file 'concat.sas';
    put "c1 = substr(type,1,1) || substr(origin,1,1);";
run;

filename fref 'concat.sas';

proc imstat;
    table lasrlib.cars(tempnames=(c1 $));  1
    summary horsepower / groupby=c1 tn=(c1) te=fref;  2
run;
    crosstab c1*type / tn=(c1) te=fref;
run;
```

1 The TEMPNAMES= data set option for the SAS LASR Analytic Server engine reserves the variable name, c1, for the temporary variable that does not exist. The $ indicates that it is a character variable.

2 The temporary variable name can be used in a variety of expressions. The TEMPNAMES= and TEMPEXPRESS= options must be specified in every statement that uses the temporary variable.

## Input Tables and Data Set Options

The IMSTAT procedure works with in-memory tables only, and the SAS LASR Analytic Server engine is used to identify the tables to use.

The following data set options are not supported when used with the IMSTAT procedure to specify an input table:

- OBS=
- FIRSTOBS=
- DROP=
- KEEP=
- RENAME=

## Syntax: IMSTAT Procedure (Data and Server Management)

**PROC IMSTAT** <*options*>;
- **BALANCE** <*/options*>;
- **COLUMNINFO** <*/ options*>;
- **COMPRESS** <*/ options*>;
- **COMPUTE** *column-name* "*SAS-statements*" <*/ option*>;
- **CREATETABLE** *table-name column-specification1* <*column-specification2…*> <*/ options*>;
- **DELETEROWS** <*/ options*>;
- **DISTRIBUTIONINFO** <*/ option*>;
- **DROPCOLUMN** *column-name*;
- **DROPTABLE** <*libref.member-name*>;
- **FETCH** <*variable-list*> <*/ options*>;
- **FREE** *resource-specification*;
- **LIFETIME** *time-specification* <MODE= ABSOLUTE | LASTUSE >;
- **NUMROWS**;
- **PARTITION** *variable-list* <*/ options*>;
- **PARTITIONINFO** <*/ options*>;
- **PROMOTE** *member-name* <*/ options*>;
- **PURGETEMPTABLES** <*options*>;
- **REPLAY** *table-list*;
- **SAVE** <*options*>;
- **SCHEMA** *dim-specification1* <*dim-specification2 …*> <*/ options*>;
- **SCORE** CODE=*file-reference* <*options*>;
- **SERVERINFO** <*option*>;
- **SERVERPARM** <*options*>;
- **SERVERTERM** <*options*>;
- **SERVERWAIT** <*options*>;
- **SET** *set-specification1* <*set-specification2 …*> <*/ options*>;
- **STORE** <*table-name* <[*table-number*]>>
  - (*row-number* | _ALL_ | _LAST_ | *row-list* | WHERE=(*where-clause*) <,>
  - *column-number* | _ALL_ | COLS=*column-list*) = *macro-variable-name* <*/ options*>;
- **TABLE** <*libref.member-name*>;
- **TABLEINFO** <*/ options*>;
- **UNCOMPRESS** <*/ options*>;
- **UPDATE** *variable1=value1* <*variable2=value2 …*> <*/options*>;

**QUIT**;

| Statement | Task | Example |
|---|---|---|
| BALANCE | Rebalancing a table | Ex. 3 |
| DELETEROWS | Deleting rows and saving a table to HDFS | Ex. 4 |
| PARTITION | Partitioning a table into a temporary table | Ex. 1 |
| PROMOTE | Promoting temporary tables to regular tables | Ex. 2 |
| SCHEMA | Creating a star schema | Ex. 5 |
| SET | Appending tables | Ex. 6 |

# PROC IMSTAT (Data and Server Management) Statement

Manages in-memory tables in a SAS LASR Analytic Server instance.

## Syntax

**PROC IMSTAT** <*options*>;

### *Summary of Optional Arguments*

BATCHMODE
DATA=*libref.member-name*
FMTLIBXML=*file-reference*
IMMEDIATE
NODATE
NOPREPARSE
NOPRINT
NOTIMINGMSG
PGMMSG
SIGNER="*authorization-web-service-uri*"
TEMPTABLEINFO
TEMPTABLESQUEEZE
UCA

### *Optional Arguments*

**BATCHMODE**

By default, the IMSTAT procedure operates in interactive mode. If your program contains errors that prevent SAS from parsing or executing statements, the errors are reported in the SAS log, but they do not stop the procedure. If the errors are fatal errors such as running out of memory on the SAS client, the procedure stops.

In contrast, when the BATCHMODE option is specified in the PROC IMSTAT statement, the procedure behaves with respect to error handling as if it were not an

interactive procedure. Whenever an error occurs, the procedure terminates and sets the SYSERR macro variable.

**Alias** BATCH

**DATA=*libref.member-name***
specifies the table to access from memory. The libref must be assigned from a SAS LASR Analytic Server engine LIBNAME statement.

**FMTLIBXML=*file-reference***
specifies the file reference for a format stream. For more information, see "FMTLIBXML" in the LASR procedure.

**IMMEDIATE**
specifies that the procedure executes one statement at a time rather than accumulating statements in RUN blocks.

**Alias** SINGLESTEP

**NODATE**
specifies to suppress the display of time and date-dependent information in results from the TABLEINFO statement.

**NOPREPARSE**
prevents the procedure from pre-parsing and pre-generating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as-is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to pre-parse and pre-generate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from pre-parsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**NOPRINT**
This option suppresses the generation of ODS tables and other printed output in the IMSTAT procedure. You can use this option to suppress printed output during execution of the actions, and then use the REPLAY statement to print the tables at a later point in the procedure execution.

**NOTIMINGMSG**
When an action completes successfully, the IMSTAT procedure generates a SAS log message that contains the execution time of the request. Specify this option to suppress the message.

**Alias** NOTIME

**PGMMSG**
specifies to capture messages associated with user-provided SAS statements on the server in a temporary table. Messages are produced when parsing errors occur, when code generation fails, or by PUT statements in a SAS program.

You can use this option as a debugging feature for SAS code that you submit through temporary column expressions. The macro variable _PGMMSG_ is used in the IMSTAT procedure to capture the name of the table. The _TEMPLAST_ macro variable is also updated in case this temporary table is the most recently created temporary table.

**Alias** PROGMST

**SIGNER="*authorization-web-service-uri*"**
specifies the URI for the SAS LASR Authorization web service. For more information, see *SAS Visual Analytics: Administration Guide*.

**Example** SIGNER="https://server.example.com/SASLASRAuthorization"

**TEMPTABLEINFO**
specifies to add additional information for temporary tables to the ODS table that is created on the SAS client. The information includes the time at which the temporary table was created in the server, the number of rows, and the number of columns.

**Alias** TEMPINFO

**TEMPTABLESQUEEZE**
requests that the temporary tables generated in the PROC IMSTAT session are automatically squeezed (compressed). You can use the INFO option in the COMPRESS statement to determine the compression ratio that was applied to the table.

**Alias** TEMPSQUEEZE

**UCA**
specifies that you want to use Unicode Collation Algorithms (UCA) to determine the ordering of character variables in the GROUPBY= operations and other operations that depend on the order of formatted values.

**Alias** UCACOLLATION

## BALANCE Statement

The BALANCE statement creates a temporary table from the active table and re-balances it so that the number of rows on the worker nodes are balanced as evenly as possible. The rows are balanced within ± 1 row of each other.

**Example:** "Example 3: Rebalancing a Table" on page 365

### Syntax

**BALANCE** </ *options*>;

#### *Without Arguments*

The re-balancing removes any observations marked as deleted or marked for purging in the active table. A WHERE clause is observed when the data are rebalanced.

One case for re-balancing is if the data distribution for a table has become uneven due to block movement within the Hadoop Distributed File System. This can occur when nodes

fail in Hadoop or Hadoop processes have exited on some nodes. Another situation where re-balancing is useful is when a partitioned table has uneven distribution across the worker nodes due to uneven sizes of the partition. This can affect the performance of all actions running in the LASR Analytic Server since typically the nodes with the most records determine the overall performance.

Rebalancing of a table removes partition and ordering information from the table.

The BALANCE statement can be used with non-distributed servers as well. However, it is less important because all records of a table reside on the same machine. It might be useful, however, to derive from a partitioned table a new table that is subject to a WHERE clause, has deleted records removed, and is not partitioned.

### *Optional Arguments*

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### *ODS Table Names*

The BALANCE statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| TempTable | Information about a temporary table | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## COLUMNINFO Statement

The COLUMNINFO statement is used to return information for all the columns in an in-memory table.

## Syntax

**COLUMNINFO** </ *options*>;

### COLUMNINFO Statement Options

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

## Details

### ODS Table Names

The COLUMNINFO statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| ColumnInfo | Information about columns in a SAS LASR Analytic Server table | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# COMPRESS Statement

The COMPRESS statement is used to create a compressed in-memory table from another in-memory table. The compressed table is stored as a temporary table.

## Syntax

**COMPRESS** </ *options*>;

### *COMPRESS Statement Options*

**INFO**
requests the server to report information about the compression state of a table, but does not perform any compression. On a compressed table, the report includes information about the compressed size and compression ratio. On an uncompressed table, the results include the uncompressed size only. The option is also useful to find out how much memory a table consumes.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

## Details

### *Interactions*

You can apply compression to regular in-memory tables, to partitioned tables, and to partitioned and ordered tables. You can apply compression with an active WHERE clause and with tables that contain rows that are marked for deletion (purging). Rows that do not pass the WHERE clause or rows that are marked for deletion are not transferred to the temporary table. You cannot compress a view.

If a table is already compressed, the statement produces information about the compressed and uncompressed size of the table. No further compression is performed.

### *ODS Table Names*

The COMPRESS statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| Compress | Information from compressing or uncompressing tables | Default |
| TempTable | Information about a temporary table | When INFO is not specified and COMPRESS generates a compressed table. |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## Example

```
data lasr.prdsale;
    set sashelp.prdsale;
run;

proc imstat data=lasr.prdsale;
```

```
    compress / info;  1
run;
    compress;
    table lasr.&_templast_;
    promote prdsalecomp;  2
run;
    table lasr.prdsalecomp;
    compress / info;  3
quit;
```

1 The INFO option does not attempt to compress the table. The compression information is shown in Figure 5.1 on page 311.

2 The PROMOTE statement is used to make the compressed table into a permanent table and give it a name.

3 The INFO option shows the compression information for the compressed table. See Figure 5.2 on page 311.

***Figure 5.1*** *Compression Information for an Uncompressed Table*

The IMSTAT Procedure

| Compression Information | |
|---|---|
| Data Source | HPS.PRDSALE |
| Table | HPS.PRDSALE |
| Size | 1.7e+02KB |
| Compressed Size | . |
| Compression Ratio | . |

***Figure 5.2*** *Compression Information for a Compressed Table*

| Compression Information | |
|---|---|
| Data Source | HPS.PRDSALECOMP |
| Table | HPS.PRDSALECOMP |
| Size | 1.7e+02KB |
| Compressed Size | 15KB |
| Compression Ratio | 12 |

## COMPUTE Statement

The COMPUTE statement adds a permanent computed column to an in-memory table.

## Syntax

**COMPUTE** *column-name file-reference* <*/ option*>;

**COMPUTE** *column-name* "*SAS-statements*" <*/ option*>;

### Required Argument

***column-name***

specifies the name to use for the computed column. The name cannot already be in use in the table.

### COMPUTE Statement Options

**NOPREPARSE**

prevents the procedure from pre-parsing and pre-generating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as-is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to pre-parse and pre-generate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from pre-parsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

## CREATETABLE Statement

The CREATETABLE statement is used to create an empty in-memory table by specifying column definitions. This is useful when you want to append tables, rows, or stream data into and empty table.

## Syntax

**CREATETABLE** *table-name column-specification1* <*column-specification2…*> <*/ options*>;

### Required Arguments

***table-name***

specifies the name of the table to create.

***column-specification***

specifies the name of a column. Numeric columns are created with the name only. Character columns are created by specifying the name and then the column length, enclosing the length in parenthesis.

### *CREATETABLE Statement Options*

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option when you do not have an active table.

**LABEL="*label*"**
assigns the specified label to the table.

**NOPROXY**
specifies that the table creation does not go through the step where the requesting process writes the table signature files. If this option is specified, then the user ID that started the server owns the table signature files.

**ORDERBY=(*variable-list*)**
specifies the variable or variables to use for ordering the observations within a partition. This option is ignored unless you also specify the PARTITION= option.

**PARTITION=(*variable-list*)**
specifies the variable or variables to use for partitioning the table. Separate variable names with a space.

**PERM=*mode***
specifies the permission setting for accessing a table. The mode value is specified as an integer value such as 755. The mode corresponds to the mode values that are used for UNIX file access permissions.

For Windows servers, the UNIX mode setting is not applicable. Access is controlled according to permissions that you set manually on the signature files directory.

| | |
|---|---|
| **Alias** | PERMISSION= |
| **Range** | 600 to 777 |

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server. If you do not specify a PORT= value, then behavior of the CREATETABLE statement depends on whether an in-memory table is active. If there is no active table, then the procedure attempts to connect to the server using the LASRPORT macro variable. If a table is active, the information is gathered for the server that is implied by the libref of the active table.

**SQUEEZE**
specifies to create the in-memory table in compressed form. Rows that are added to the table are compressed by the server as they are added.

| | |
|---|---|
| **Alias** | COMPRESS |
| **Interaction** | This option cannot be used with both the PARTITION= and ORDERBY= options. |

**TAG='*server-tag*'**
specifies the tag of the new table in the server. If you do not specify this option, then the default tag, `WORK` is used. The tag and the value for *table-name* are used to create the name of the new table in the server.

## Details

### *Creating Tables and User-Defined Formats*

If you want to associate the new table with user-defined formats, you can provide the XML for the formats in the FMTLIBXML= option in the PROC IMSTAT statement. The following code provides a simple example.

```
libname myfmts "/path/to/formats";
proc format library=myfmts;
  value region 00 -< 20 = "Northeast"
               20 -< 45 = "Midwest"
               45 -< 60 = "Southeast"
               60 -< 90 = "West";

  value $storesz 'S' = "Small"
                 'M' = "Medium"
                 'L' = "Large";
run;

options fmtsearch=(myfmts);

filename myfmtxml 'myfmt.xml';
libname  myfmtxml XML92 xmltype=sasfmt tagset=tagsets.XMLsuv;

proc format library=myfmts cntlout=myfmtxml.allfmts; 1
run;

libname example sasiola host="grid001.example.com" port=10010 tag='hps';

proc imstat fmtlibxml=myfmtxml; 2
  createtable sales storeid region size(1) predicted actual /
     label="Sales Table" tag="hps" host="grid001.example.com" port=10010;
quit;

/* stream or append data into the table */
data example.sales(append=yes);
  input storeid region size $1. predicted actual;
datalines;
233 10 L 10040 12343
459 33 L 10070 11567
327 45 M 5700  5478
546 76 M 5400  5790
;;;;
run;

proc imstat data=example.sales;
  fetch / format=("", "region.", "$storesz6.", "comma9.", "comma9."); 3
quit;
```

1 The steps up to this point are common for working with formats such as XML. For a similar example, see "Example 9: Working with User-Defined Formats and the FMTLIBXML= Option" on page 61.

2 The FMTLIBMXL= option associates the user-defined formats with the table that is created with the CREATETABLE statement. The Region and Storesz formats are not

assigned to any variables, but can be applied to variables with any statement that supports the FORMAT= option.

3 The Region and Size variables are formatted with the user-defined formats. The $storesz6. format specifies a length of 6 because that is the longest string in the format definition. If you do not specify the length, then the length of the variable is used and is equivalent to specifying $storesz1. as the format.

The following figure shows the results of the FETCH statement with the user-defined formats applied to the Region and Size variables.

| Selected Records from Table HPS.SALES | | | | |
|---|---|---|---|---|
| storeid | region | size | predicted | actual |
| 233 | Northeast | Large | 10,040 | 12,343 |
| 459 | Midwest | Large | 10,070 | 11,567 |
| 327 | Southeast | Medium | 5,700 | 5,478 |
| 546 | West | Medium | 5,400 | 5,790 |

## DELETEROWS Statement

The DELETEROWS statement is used to mark rows as deleted, to undelete rows, and to purge rows from an in-memory table. Rows that are marked for deletion or purging are not included in the calculations performed by the server.

**Interaction:** If a WHERE clause is active, the rows that meet the criteria are marked for deletion or purging. When no WHERE clause is active, a delete request marks all rows for deletion (they can be undeleted), but the PURGE option removes the rows that are already marked for deletion rather than removing all rows.

**Example:** "Example 4: Deleting Rows and Saving a Table to HDFS" on page 368

### Syntax

**DELETEROWS** </ *options*>;

#### *DELETEROWS Statement Options*

**PURGE**

specifies to remove from memory the rows that are marked for deletion. The memory that was used by the rows is freed. The purged rows cannot be undeleted. One use case for purging rows is to remove older records from an in-memory table after new records were appended to the table.

If a WHERE clause is active, the rows that meet the criteria are purged. If no WHERE clause is active, then the PURGE request removes the rows that are already marked for deletion. It does not remove all the rows in the table. This was implemented to prevent accidentally removing all rows from a table.

**SAVE=*table-name***

saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within

the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**UNDELETE**
specifies to clear the deletion mark from rows.

If a WHERE clause is active, only the rows that meet the criteria have the deletion mark cleared. A row that has been marked for purging from the table cannot be undeleted.

## Details

### *ODS Table Names*

The DELETEROWS statement generates an ODS table that is named **`DeleteRows`**. For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# DISTRIBUTIONINFO Statement

The DISTRIBUTIONINFO statement returns the number of partitions and the number of records on each worker node in the SAS LASR Analytic Server instance. This information provides the approximate distribution characteristics of the data across the worker nodes. If you want more details about the data distribution, then use the PARTITIONINFO statement.

## Syntax

**DISTRIBUTIONINFO** </ *option*>;

### *DISTRIBUTIONINFO Statement Options*

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

## Details

### ODS Table Names

The DISTRIBUTIONINFO statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| DistributionInfo | Distribution information for a LASR Analytic Server table | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# DROPCOLUMN Statement

The DROPCOLUMN statement removes a column that was created with the COMPUTE statement from an in-memory table.

## Syntax

**DROPCOLUMN** *column-name*;

### Required Argument

***column-name***
specifies the name of the computed column to remove. The column must have been created with the COMPUTE statement.

## Example

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.cars;
   set sashelp.cars;
run;

proc imstat nopreparse;
   table example.cars;
   compute ratio "ratio = mpg_city / mpg_highway";  1
   /* columninfo; */
run;

   dropcolumn ratio;  2
   /* columninfo; */
quit;
```

1 The COMPUTE statement adds the permanent computed column Ratio to the active table.

2 The DROPCOLUMN statement removes the permanent computed column that is named Ratio from the table.

## DROPTABLE Statement

The DROPTABLE statement is used to remove an in-memory table from a server. You must specify a table as an active table with the DATA= procedure option or in a TABLE statement before you can use the DROPTABLE statement. Once a table is active, you can specify that table, another table, or a temporary table.

### Syntax

**DROPTABLE** <*libref.member-name*>;

#### *Optional Argument*

**libref.member-name**
specifies the name of the in-memory table to remove from the server. If you do not specify the table, then the active table is dropped.

## FETCH Statement

The FETCH statement is used to retrieve rows from an in-memory table. You can use the FETCH statement to retrieve calculated columns that are calculated according to a script as part of the request. The columns that are calculated this way do not persist beyond the time it takes to execute in the server.

**Tip:** By default, the FETCH statement retrieves 20 rows. You can specify the FROM= and TO= options to change the behavior.

### Syntax

**FETCH** <*variable-list*> </ *options*>;

#### *Optional Argument*

***variable-list***
specifies the numeric and character variables to retrieve.

#### *FETCH Statement Options*

**ARRAYSTART=*n***
specifies the starting element of an array when the record of an in-memory table represents a variable array. This is the case, for example, when a pricing cube from SAS High-Performance Risk is loaded into a server. There might then be 10,000 columns for a variable. Specifying the ARRAYSTART= and ARRAYLENGTH= options enables you to page through the data more conveniently.

**ARRAYLENGTH=*k***
specifies the length of the array to fetched when the record of an in-memory table represents a variable array. Use this option with the ARRAYSTART= option.

**DESCENDING=*variable-name***
**DESCENDING=(*variable-list*)**
specifies which variables of the ORDERBY= list are used with descending sort order. Specifying the DESCENDING= option by itself has no effect. The option is specified in addition to the ORDERBY= option. The following example specifies to fetch data on columns A and B of the active table ordered by ascending unformatted values of B and descending unformatted values of C.

**Alias** DESC=

**Example**
```
fetch a b / orderby=(b c) descending=c;
```

**FORMAT <=("*format-specification*", …)>**
specifies the formats to use for the variables. By default, the FETCH statement retrieves the unformatted values. If you specify the FORMAT option without a list of format names, then the server applies the default format that is associated with each variable.

Be aware that when you retrieve unformatted values and you create an output data set with the OUT= option, the variable information such as formats and labels are transferred to the output data set.

If you want to use the default format for a variable, specify an empty string. The following example uses the default format for column A and the $10 format for column B.

**Alias** FORMATS=

**Example**
```
fetch a b / formats=("", "$10");
```

**FROM=*first-index***
specifies the index of the first row to retrieve (inclusive). The value for *first-index* is 1-based.

**Default** FROM=1

**Interaction** The value for FROM= is applied after the evaluation of a WHERE clause.

**NOPREPARSE**
prevents the procedure from pre-parsing and pre-generating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as-is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to pre-parse and pre-generate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from pre-parsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**ORDERBY=*variable-name***
**ORDERBY=(*variable-list*)**
specifies one or more variables by which to order the results. The default sort order of the ORDERBY= variables is ascending in unformatted values and follows location and collation rules. If you want to arrange some ORDERBY= variables in descending sort order, then list the variable names in the DESCENDING= option (in addition to listing them in the ORDERBY= option).

If you want to assign a format to ORDERBY= variables, you can use the ORDERBYFORMAT= option. That option can also be used to specify which variables are sorted by formatted values and which variables are sorted by unformatted values.

**ORDERBYFORMAT=("*format-specification*", …)**
specifies the formats to use for sorting of the ORDERBY= variables. By default, if you specify an ORDERBY= variable or variable list, the server sorts by the ascending unformatted values. If you want to apply unformatted value ordering for some ORDERBY= variables, and formatted value ordering for other ORDERBY= variables, you can specify the keyword **_RAW_** for the variables to sort by unformatted value.

The following example specifies to retrieve unformatted values of columns Make, Model, Type, Invoice, and Mpg_City. The rows are retrieved in the order of ascending formatted value of Type, using the $ format, and descending unformatted values of Invoice.

**Example**
```
fetch make model type invoice mpg_city /
    orderby=(type invoice)
    desc   =invoice
    orderbyformat=("$", "_RAW_");
```

**OUT=*libref.member-name***
specifies the name of the data set to store the result set of the FETCH statement.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since

many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**TO=*last-index***
specifies the index of the last row to retrieve (inclusive). The value for *last-index* is 1-based.

**Default** The default value is FROM=*first-index* + 19.

**Interaction** The value for TO= is applied after the evaluation of a WHERE clause.

## Details

### *ODS Table Names*

The FETCH statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| Fetch | Fetched rows from a LASR Analytic Server table | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# FREE Statement

The FREE statement is used to release resources from STORE statements or SAVE= options. If you save a result table (one table or sets of tables) with the SAVE= option in any of the analytic statements of the IMSTAT procedure, then you can release the resources with the FREE statement. Once a table has been freed, you can reuse the table name. While a saved table exists, you cannot create a table by the same name.

## Syntax

**FREE** *table-list*;

**FREE** _ALL_;

**FREE** TABLE=*one-table*;

**FREE** MACRO=*macro-variable-name*;

### FREE Statement Options

***table-list***
specifies a list of tables to release.

**_ALL_**
specifies to release the resources for all the tables that were saved throughout the procedure execution.

**TABLE=*one-table***
specifies the name of the table to release.

**MACRO=*macro-variable-name***
specifies the name of a macro variable to release.

## LIFETIME Statement

The LIFETIME statement enables you to associate an expiration date with an in-memory table. You must have sufficient authorization, which is equivalent to the authorization to drop the table. The server checks periodically if any tables have expired and drops the expired ones. This frees all resources associated with those tables.

### Syntax

**LIFETIME** *time-specification* <MODE= ABSOLUTE | LASTUSE >;

### Required Argument

***time-specification***
specifies the duration (in seconds) that the table is to remain in memory. The minimum value is 1 second.

### LIFETIME Statement Options

**MODE=ABSOLUTE | LASTUSE**
specifies how to use the *time-specification*. If MODE=ABSOLUTE is specified, then the server drops the table after the specified number of seconds. If MODE=LASTUSE is specified, then the server drops the table the specified number of seconds after the last successful access to the table.

**Default** ABSOLUTE

## NUMROWS Statement

The NUMROWS statement identifies how many rows satisfy a selection criterion. The selection observes the WHERE clause and records marked for deletion or purging.

## Syntax

**NUMROWS**;

## Details

### ODS Table Names

The NUMROWS statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| NumRows | Number of rows | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# PARTITION Statement

The PARTITION statement is used to generate a temporary table from the active table and partition it according to the statement options. This re-partitioning of tables is supported for distributed and non-distributed SAS LASR Analytic Server.

**Examples:** "Example 1: Partitioning a Table into a Temporary Table" on page 362
"Example 7: Appending a Non-Partitioned Table to a Partitioned Table" on page 371

## Syntax

**PARTITION** *variable-list* <*/ options*>;

### PARTITION Statement Options

**DESCENDING=(*variable-list*)**
**DESCENDING=*variable-name***

specifies the variables in the ORDERBY= list to use with a descending sort order. Specifying the DESCENDING= option by itself has no effect on ordering within a partition. The option is specified in addition to the ORDERBY= option.

**Alias** DESC=

**Example** The following statement requests partitioning of the active table by variables A and B, and ordering within the partition by ascending value of column C and descending value of column D:

```
partition a b / orderby=(c d) descending=d;
```

**FORMATS=("*format-specification*" <,…> )**

specifies the formats for the PARTITIONBY= variables. If you do not specify the FORMATS= option, the default format is applied for that variable. The format of a partitioning variable is important because the equality of the partition keys is determined from the formatted values.

Enclose each format specification in quotation marks and separate each format specification with a comma.

**NOMISSING**
specifies that missing values are excluded in the determination of partition keys. By default, observations with missing values are included.

**Alias** NOMISS

**ORDERBY=(*variable-list*)**
**ORDERBY=*variable-name***
specifies the variable or variables to use for ordering the observations within a partition. By default, the sort order for the ORDERBY= variables is ascending in raw values and follows location and collation rules. If you want to order some ORDERBY= variables in descending sort order, then specify the variable names in the DESCENDING= option (in addition to listing them in the ORDERBY= option).

The ORDERBY= variables are transferred automatically to the partitioned temporary table, whether you list them in the VARS= option or not.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

**VARIABLES=(*variable-list*)**
**VARIABLES=*variable-name***
specifies the variable or variables to include in the temporary table in addition to the partitioning variables. If you do not specify the VARS= option, then all the variables are transferred from the active table. Temporary calculated columns are also transferred to the temporary table.

**Alias** VARS=

## Details

### *ODS Table Names*

The PARTITION statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| TempTable | Information about a temporary table | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement. However, the information that is available in the ODS table, the temporary table name, is automatically placed in the &_TEMPLAST_ macro variable.

# PARTITIONINFO Statement

The PARTITIONINFO statement produces very detailed information about the partitions of the data for each node of the server. Be aware that depending on the number of partitions, the result table can be very large. If you use the statement on non-partitioned data, it reports the number of bytes and records for each node. These results are similar to the result set you get from a DISTRIBUTIONINFO statement.

## Syntax

**PARTITIONINFO** <*/ options*>;

### *PARTITIONINFO Statement Options*

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

## Details

### *ODS Table Names*

The PARTITIONINFO statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| PartitionInfo | Partition Information for a LASR table | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## PROMOTE Statement

The PROMOTE statement is used to change a temporary table to a regular in-memory table. The currently active table must be a temporary table and the table identified with the *member-name* parameter must not already exist in the server. Promoting a temporary table requires authorization to add a table to the server. You can also specify a tag for the in-memory table with the TAG= option.

**Example:** "Example 5: Creating a Star Schema" on page 369

### Syntax

**PROMOTE** *member-name* <*/ options*>;

#### *Required Argument*

***member-name***
specifies the name to use for the table that is promoted.

#### *PROMOTE Statement Options*

**PERM=*mode***
specifies the access permission for the newly created table as an integer. The mode value is specified as an integer value such as 755. The mode corresponds to the mode values that are used for UNIX file access permissions. You must specify a value that preserves Read and Write permission for your user ID.

For Windows servers, the UNIX mode setting is not applicable. Access is controlled according to permissions that you set manually on the signature files directory.

| | |
|---|---|
| **Alias** | PERMISSION= |
| **Range** | 600 to 777 |

**TAG='*server-tag*'**
specifies the tag to use for naming the table. If no TAG= option is specified, then the TAG= option from the LIBNAME statement is used. If the LIBNAME statement does not specify the TAG= option, then the name of the libref is used as the server tag.

## PURGETEMPTABLES Statement

The PURGETEMPTABLES removes all temporary tables from a server. The action requires server-level authorization because it removes temporary tables created by all users. To execute this command successfully, you must have the same authorization that is required to terminate the server.

## Syntax

**PURGETEMPTABLES** <*options*>;

### *Without Arguments*

The server to use is identified from the active table. If you do not have an active table, then you can connect to a specific server with the HOST= and PORT= options.

### *PURGETEMPTABLES Statement Options*

**HOST="*host-name*"**
specifies the host name for the server. Use this option with the PORT= option.

**PORT=*number***
specifies the port number for the server.

# REPLAY Statement

The REPLAY statement enables you to display a previously saved result table or set of tables. The REPLAY statement displays the saved result tables regardless of the NOPRINT option. This enables you to suppress output generation with the NOPRINT option and to then display the tables that you want in a different order.

## Syntax

**REPLAY** *table-list*;

### *Optional Argument*

***table-list***
specifies the saved result tables to display.

## Example: Display Result Tables in a Different Order

The following SAS statements suppress the display of output with the NOPRINT option. Then, the tables are displayed in the reverse order.

```
proc imstat data=sales.prdsale noprint;
    fetch country region actual / save=salestab from=1 to=5;
    fetch predict actual / save=predicttab from=1 to=10;
    replay predicttab salestab;
quit;
```

# SAVE Statement

The SAVE statement enables you to save an in-memory table as a SASHDAT table or a CSV file. The table must be the active table. You specify the active table with the DATA= option in the IMSTAT procedure or with the TABLE statement.

**Example:** "Example 4: Deleting Rows and Saving a Table to HDFS" on page 368

## Syntax

**SAVE** <*options*>;

### *Optional Arguments*

**BLOCKSIZE**
specifies the block size to use for distributing the data set. Suffix values are B (bytes), K (kilobytes), M (megabytes), and G (gigabytes).

**Interaction** If the in-memory table is partitioned, the BLOCKSIZE= specification is ignored. The server determines the block size based on the size of the partitions.

**COPIES=*n***
specifies the number of replications to make for the data set (beyond the original blocks). The default value is 1. You can specify COPIES=0 if you do not need replications for redundancy.

**CSV**
specifies to save the table as a comma-separated value file. The first line of the file includes the variable names.

**Interaction** The SQUEEZE option is ignored if you specify CSV.

**Examples**

```
  table hps.stocks;
  save path="/hps/stocks2.csv" fullpath replace csv;
run;
```

To load a CSV file to memory, you can use the SASHDAT engine and PROC LASR as follows:

```
libname hpscsv sashdat host="grid001.example.com"
    install="/opt/TKGrid" path="/hps";

proc lasr add data=hpscsv.stocks2(filetype=csv getnames=yes)
     port=10010;
  performance host="grid001.example.com";
run;
```

**ENCRYPTKEY=*key-value***
specifies a key value for AES (Advanced Encryption Standard) encryption.

**See** For more information, see "ENCRYPT= Data Set Option" on page 447.

**FULLPATH**
specifies that the value for the PATH= option specifies the full path for the file, including the filename (without the SASHDAT extension). The filename portion of the quoted string is expected to be in lowercase characters.

**PATH='*HDFS-path*'**
specifies the directory in which to store the table as a SASHDAT file. The value is case sensitive. The filename for the SASHDAT file that is stored in the path is always lowercase.

*Note:* If the PATH= option is not specified, the server attempts to save the table in the /user/*userid* directory. The *userid* is the user ID that started the server instance.

**REPLACE**
specifies that the SASHDAT file should be overwritten if it already exists.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SIGNERFILEPOLICY**
specifies to apply the encryption policies stored in metadata to the file.

**See** For more information, see "SIGNERFILEPOLICY Data Set Option" on page 454.

**SQUEEZE**
specifies to save the table in the SASHDAT file in compressed form.

*Note:* You must specify this option even if the in-memory table is compressed. By default, tables are saved in uncompressed form, regardless of whether the in-memory table is compressed or not.

## SCHEMA Statement

The SCHEMA statement is used to define a simple star schema in the server from a single fact table and one or more dimension tables. The result of the SCHEMA statement is a temporary table that you can use as it is, or with the PROMOTE statement.

**Example:** "Example 5: Creating a Star Schema" on page 369

### Syntax

**SCHEMA** *dim-specification1 <dim-specification2 …> </ options>*;

#### *Required Argument*

***dim-specification***
specifies how to use the dimension table with the fact table. You must specify the variables to use as keys for the fact table (*fact-key*) and the dimension table (*dim-key*). The variables do not need to have the same name, but they do need to have the same type.

*dim-table-name* (*fact-key* = *dim-key* </ <PREFIX = *dim-prefix*> <TAG='*server-tag*'> <, *variable-list*>>)

***dim-table-name***
specifies the name of the dimension table.

***fact-key***
specifies the variable name in the fact table to use.

***dim-key***
specifies the variable name from the dimension table to use.

**PREFIX=*dim-prefix***
specifies a prefix to use for naming variables in the schema. If you do not specify PREFIX=, then up to the first sixteen characters of the *dim-table-name* are used as the dimension prefix for naming the variables in the schema.

**Alias** NAME=

**TAG='*server-tag*'**
specifies the server tag to use for identifying the dimension table.

***variable-list***
specifies the variables from the dimension table to join with the fact table. By default, all variables except the dimension key are transferred from the dimension table. The dimension key is never transferred because a corresponding value is available through the *fact-key*.

### SCHEMA Statement Options

**LABELPREFIX**
specifies that the dimension prefix (or the dimension name, when PREFIX= is not specified), is used to modify existing column labels from the dimension tables. This enables you to identify the original table for a column by the label.

**MODE=VIEW | TABLE**
specifies whether the rows of the schema are materialized when the statement executes in the server. The default is MODE=VIEW and implies that the server resolves the relations in the tables but defers the resolution (formation) of the rows until the view is accessed. If you specify MODE=TABLE, then the table is resolved (flattened) when the statement executes. A view consumes much fewer resources (almost none), but data access is slower compared to a flattened table.

**Default** VIEW

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### Assumptions

The server makes the following assumptions with regard to fact and dimension tables:

- Dimension tables do not have repeat values for the dimension keys. If a key value appears multiple times, the first record that matches the key is used.

- The relation between the fact table and a dimension table is expressed by one pair of keys. That is, one variable in the fact table defines the relation to one variable in the dimension table.
- The variable names of keys in the fact table and dimension tables do not have to be the same.
- The look ups for the keys are performed based on raw values with fuzzing for doubles.
- If the fact table or a dimension table contains pre-levelized CLASS variables, the class-variable signature is removed when the schema is created.
- Partitioning and order-by information is preserved when the schema is created. However, only partitioning of the fact table is taken into consideration and the resulting table or view is partitioned by the same variables as the fact table.
- The relations are resolved when the schema is created. This strategy makes passes through the data more efficient.

### *About Views and Tables*

When the SCHEMA statement executes, a temporary table is generated and the name of that temporary table is returned to the client as the result set. You use the &_TEMPLAST_ macro variable to refer to the star schema.

By default, the server creates a view from the schema definition. The temporary table then has columns for all variables in the schema. The relations have been resolved, but the rows of the view have not been formed.

You can request that row resolution takes place when the temporary table is formed. The result is a flattened temporary table where the rows of the schema are materialized.

There are advantages and disadvantages to using views and flattened tables. The primary consideration is whether there is enough memory for the data volume. The following list identifies some of the considerations:

- A flattened table consumes memory when the statement executes. If the memory requirement of the fully joined schema exceeds the capacity of the machine, the statement fails. For example, if you intend to work with relations that expand to seven terabytes of memory, then you cannot flatten the table unless you have that much memory on your system.
- If a flattened table can be held in memory, data access is faster because it is a regular in-memory table.
- A view does not consume memory until it is accessed. At that time, the table is never materialized fully in memory. The joined rows are formed only when a buffer is needed. This enables you to work with views that exceed the memory capacity of the system.
- The performance difference between resolving a view at run time and accessing a flattened table is difficult to quantify. It depends, for example, on the number of columns to resolve and the data access pattern. A request that passes through the data multiple times suffers a greater performance hit (compared to a flat table) than a single-pass request.

Some operations are not supported with views (but are supported with materialized schemas):

- You cannot append tables or rows to a view.
- You cannot perform row updates of the view.

- You cannot re-partition a view.
- You cannot use a view in another schema.

If a view is based on a partitioned fact table and you want to change the partition key, then re-partition the fact table and re-create the view with another SCHEMA statement.

A view is static. For example, if you append rows to the fact table, the Append operation succeeds and every new access to the fact table can use the appended rows. However, the view is not affected by the addition of rows to the fact table. The view resolves to the state of the fact table when the view was formed.

If you want a schema to change with appends and updates, then you can materialize it and then append or update the flattened table. Likewise, you can append or update the fact table and dimension tables, drop the view, and re-create it.

Using a view as the fact table or as a dimension table in a SCHEMA statement is not supported.

### *ODS Table Names*

The SCHEMA statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| TempTable | Information about a temporary table | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement. However, the information that is available in the ODS table, the temporary table name, is automatically placed in the &_TEMPLAST_ macro variable.

## SCORE Statement

The SCORE statement applies scoring rules to an in-memory table. The results can take different forms. They can be displayed as tables in the SAS session, output data sets on the client, or temporary tables in the server. The most common use of the SCORE statement is to execute DATA step code on some or all rows of an in-memory table and to produce corresponding output records.

**Requirement:** The program that you use with the SCORE statement must use DATA step syntax. DS2 syntax is not supported.

**Note:** There are some restrictions on the DATA step statements that you can use with the SCORE statement. See "Restrictions in DATA Step Processing" on page 460 for information.

**Examples:** "Example 1: Joining Data during Scoring with a DATA Step Hash Object" on page 337

"Example 2: Using the RETAIN Statement" on page 339

### Syntax

**SCORE** CODE=*file-reference* <*options*>;

### Required Argument

**CODE=*file-reference***
specifies a file reference to the SAS program that performs the scoring.

For information about the code that you can use, see "Restrictions in DATA Step Processing" on page 460.

**Alias** PGM=

### SCORE Statement Options

**DROP=(*variable-list*)**
**DROP=*variable-name***
specifies one or more variables that you want to drop from the input data when transferring results to the scoring results. By default, the SCORE statement does not transfer any variables from the input table to the scoring results. If you specify the KEEP= option, only the specified variables are moved to the result. If you specify the DROP= option, all variables except the specified variables are moved to the result set.

**Interaction** Do not use the DROP= and KEEP= options together. The KEEP= option takes precedence.

**DSRETAIN**
requests that the scoring code implements DATA step retention behavior for output symbols. By default, the server automatically retains output variables, whereas the DATA step sets variables to missing unless they are retained explicitly with a RETAIN statement. Specifying this option means that the automatic retention behavior is disabled in the server and you must specify RETAIN statements in your DATA step program to retain values.

**Alias** NOAUTORETAIN

**HASHDATA(*table-name1* <, *table-name2*…> )**
specifies the names of one or more in-memory tables in server that are used as input tables for DATA step hash objects. The names are the actual table names in the server, not names where tags are masked by the libref. The names must match the names that you use in the DECLARE HASH statements in the scoring program.

The server checks your authorization to access the secondary tables. The scoring action fails with a permission error if you are not authorized to use any one of the tables. The scoring action also fails if the names that you specify in the HASHDATA option do not match the names that are used in the scoring program.

**Alias** HASH

**Example** "Example 1: Joining Data during Scoring with a DATA Step Hash Object" on page 337

**KEEP=(*variable-list*)**
**KEEP=*variable-name***
specifies one or more variables that you want to transfer from the input data to the scoring results. You can use _ALL_ for all variables, _NUMERIC_ for all numeric variables, and other valid variable list names. If this option is not specified, then no variables are transferred from the input data (the table that is being scored), unless they are assigned values in the scoring code.

**Alias** TABLEVARS=

**NOPREPARSE**
specifies to prevent pre-parsing and pre-generating the program code that is referenced in the CODE= option. If you know the code is correct, you can specify this option to save resources. The code is always parsed by the server, but you might get more detailed error messages when the procedure parses the code rather than the server. The server assumes that the code is correct. If the code fails to compile, the server indicates that it could not parse the code, but not where the error occurred.

**Alias** NOPREP

**OUT=*libref.member-name***
specifies the name of an output data set in which to store the scoring results. If the result set contains variables that match those in the input data set, then format information is transferred to the output data set. The OUT= option and the TEMPTABLE option are mutually exclusive. If you specify the OUT= option, a temporary table is not created in the server.

**Alias** DATA=

**PARTITION <=*partition-key*>**
specifies to take advantage of partitioning for tables that are partitioned. When this option is specified, the scoring code is executed in the order of the partitions. If the data are also ordered within the partition, the observations are processed in that order. If the scoring code uses the reserved symbols **`__first_in_partition`** or **`__last_in_partition`**, then the data are also processed in partitioned order. Although the observations are processed in a specific order, the execution occurs in concurrent threads (in parallel). Different threads are assigned to work on different partitions.

If you do not specify the optional *partition-key*, then the analysis is performed for all partitions. If you do specify a *partition-key*, then the analysis is performed for the partitions that match the specified key value only. You can use the PARTITIONINFO statement to retrieve the valid *partition-key* values for a table.

You can specify a *partition-key* in two ways. You can supply a single quoted string that is passed to the server, or you can specify the elements of a composite key separated by commas. For example, if you partition a table by variables GENDER and AGE, with formats $1 and BEST12, respectively, then the composite partition key has a length of 13. You can specify the partition for the 11 year-old females as follows:

```
score / partition="F          11"; /* passed directly to the server */
score / partition="F","11";        /* composed by the procedure */
```

If you choose the second format, the procedure composes a key based on formatting information from the server.

**Alias** PART=

**Interaction** This option is effective when used with partitioned in-memory tables only.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within

the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**SETSIZE**
requests that the server estimate the size of the result set. The procedure does not create a result table if the SETSIZE option is specified. Instead, the procedure reports the number of rows that are returned by the request and the expected memory consumption for the result set (in KB). If you specify the SETSIZE option, the SAS log includes the number of observations and the estimated result set size. See the following log sample:

```
NOTE: The LASR Analytic Server action request for the STATEMENT
      statement would return 17 rows and approximately
      3.641 kBytes of data.
```

The typical use of the SETSIZE option is to get an estimate of the size of the result set in situations where you are unsure whether the SAS session can handle a large result set. Be aware that in order to determine the size of the result set, the server has to perform the work as if you were receiving the actual result set. Requesting the estimated size of the result set does consume resources on the server. The estimated number of KB is very close to the actual memory consumption of the result set. It might not be immediately obvious how this size relates to the displayed table, since many tables contain hidden columns. In addition, some elements of the result set might not be converted to tabular output by the procedure.

**SYMBOLS=(*symbol-list*)**
specifies one or more symbols that are calculated in the scoring code that you want to transfer as columns to the scoring results. If the SYMBOLS= option is not specified, then all symbols that are assigned values in the program—and that are not just placeholders for intermediate calculations—are transferred to the results. If you use a large program with many assignments, you might want to use the SYMBOLS= option to limit the columns in the results.

**Alias** SYM=

**TEMPTABLE**
generates an in-memory temporary table from the result set. The IMSTAT procedure displays the name of the table and stores it in the _TEMPLAST_ macro variable, provided that the statement executed successfully.

When the IMSTAT procedure exits, all temporary tables created during the IMSTAT session are removed. Temporary tables are not displayed on a TABLEINFO request, unless the temporary table is the active table for the request.

## Details

### *Using Partitioning and Scoring*

To help manage how output is generated, options in the SCORE statement can be brought to bear together with special syntax elements in the scoring code. For example, the PARTITION<=> option can be used to specify that the scoring code is executed separately for each partition or for a specific partition of the data only. If you want to control precisely which observations are used to generate output records, you can use the **`__lasr_output`** symbol in your SAS program. When this symbol is set to 1, the row is output. You can also use the **`__first_in_partition`** and **`__last_in_partition`** variables to programmatically determine the first and last observation in a partition.

The following SAS code is an example:

```
__lasr_output = 0;
if __first_in_partition then do; 1
   totalmsrp = msrp;
   minmsrp   = msrp;
   numCars   = 1;
end; else do;
   totalmsrp + msrp;
   numCars   + 1;
   if (msrp < minmsrp) then minmsrp = msrp;
end;
orgdrive = Origin || drivetrain; 2
mpgdiff = mpg_highway - mpg_city;
if __last_in_partition then 3
   __lasr_output = 1;
```

1 For the first observation within a partition, three variables are initialized. The minimum MSRP, the total MSRP, and the number of records in the partition are then computed.

2 The variable ORGDRIVE is obtained by concatenating the strings of the ORIGIN and DRIVETRAIN variables

3 When the last record within the partition is reached, the **__lasr_output** automatic variable is set to 1, this is used to add of the current record to the result set.

The execution of the SCORE code observes the active WHERE clause in the IMSTAT run block—in other words, the scoring code is executed only for those observations that meet the WHERE condition if a WHERE clause is active.

The following example loads the SASHELP.CARS data set partitioned by the TYPE variable, and executes the previous code sample.

```
data lasrlib.cars(partition=(type));
    set sashelp.cars;
run;

filename fref '/path/to/scorepgm.sas';

proc imstat;
   table lasrlib.cars;
   score pgm=fref / partition;
run;
```

The PARTITION option in the SCORE statement requests the server to execute the code separately for each partition of the data. Because the code outputs one observation per partition and there are six unique values of the TYPE variable in the SASHELP.CARS data set, the scoring results show six rows:

Scoring Results for Table WORK.CARS

| totalmsrp | minmsrp | numCars | orgdrive | mpgdiff | Type |
|---|---|---|---|---|---|
| 59760 | 19110 | 3.000000 | Asia Front | -8.000000 | Hybrid |
| 2087415 | 17163 | 60.000000 | EuropeAll | 5.000000 | SUV |
| 7800688 | 10280 | 262.000000 | EuropeFront | 7.000000 | Sedan |
| 2615966 | 18345 | 49.000000 | Asia Rear | 6.000000 | Sports |
| 598593 | 12800 | 24.000000 | Asia All | 3.000000 | Truck |
| 865216 | 11905 | 30.000000 | EuropeAll | 7.000000 | Wagon |

The SCORE statement does not support the temporary expressions that are available in other IMSTAT statements. This is because you can compute all necessary temporary variables in the scoring code.

### ODS Table Names

The SCORE statement generates the following ODS tables.

| ODS Table Name | Description | Option |
|---|---|---|
| TempTable | Information about a temporary table | TEMPTABLE |
| Fetch | Fetching rows from the table of a LASR Analytic Server | Default, when TEMPTABLE is not specified |

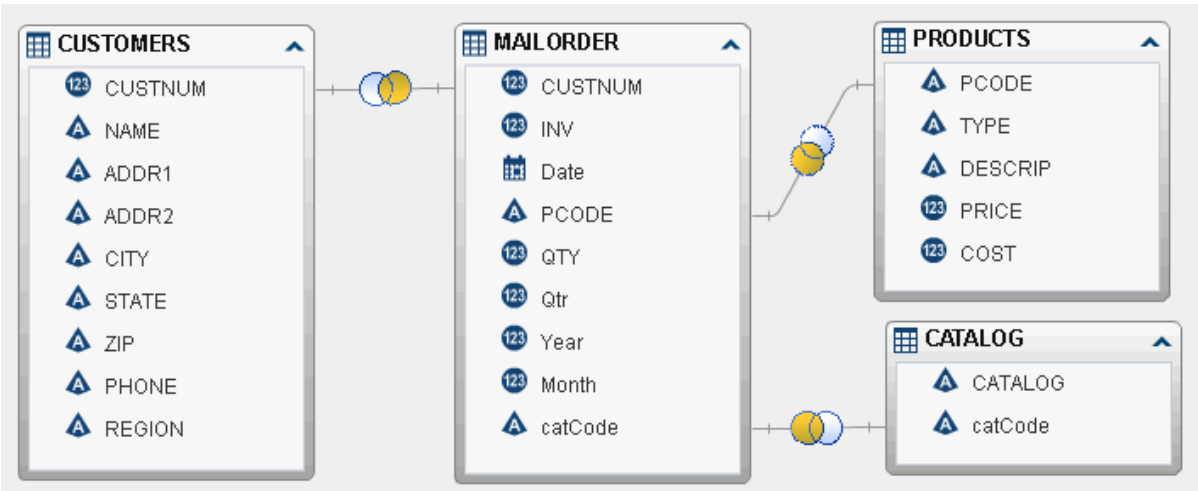
For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## Examples

### Example 1: Joining Data during Scoring with a DATA Step Hash Object

You can use many SAS functions in the scoring code that is processed by the server, including some functionality from the DATA step hash object. You can specify an in-memory table in the DECLARE statement to populate a hash object from a table.

This example shows how to use a fact table and dimension tables to instantiate one hash object for each dimension table. This effectively produces a join of the fact and dimension tables that is resolved at run time of the scoring program. The following display shows the relations:




The details of the relationships are as follows:

- MailOrder is the fact table.
- Customers joins to MailOrder on the CustNum column.
- Products joins to MailOrder on the PCode column.

- Catalog joins to MailOrder on the CatCode column.

The following code shows how to create a table with scoring code that combines the columns Date and Year from the fact table with the columns Catalog, Cost, Price, Type, Name, City, and State from the dimension tables. The code limits the results to customers from the state of New Jersey.

***File 5.1*** *Scoring Program*

```
length catcode $6;  1
length pcode   $6;
length catalog $20;
length type    $15;
length name    $32;
length city    $20;
length state   $3 ;
length cost    8  ;
length price   8  ;

declare hash cst(dataset:"star.customers");  2
rc = cst.defineKey ('custnum');  3
rc = cst.defineData('name'   );  4
rc = cst.defineData('city'   );
rc = cst.defineData('state'  );
rc = cst.defineDone();

rc_cst = cst.find();
if ((rc_cst=0) and (state = 'NJ ')) then do;  5

   declare hash cat(dataset:"star.catalog");
   rc = cat.defineKey ('catcode');
   rc = cat.defineData('catalog');
   rc = cat.defineDone();

   declare hash prd(dataset:"star.products");
   rc = prd.defineKey ('pcode');
   rc = prd.defineData('type' );
   rc = prd.defineData('price');
   rc = prd.defineData('cost' );
   rc = prd.defineDone();

   rc_cat = cat.find();
   rc_prd = prd.find();
   if (rc_cat=0 and rc_prd=0) then output;  6
end;
```

1 The LENGTH statements define the variables that are accessed from the dimension tables through the hash objects. The specified types and lengths must match the data types and lengths in the dimension tables.

2 Three hash objects are declared in the program, one for each dimension table. The DECLARE HASH statement defines and names the object. The DATASET option specifies the in-memory table to use for populating the hash object. The names specified in the program code are the actual table names in the server, and not the names where a libref masks the tag.

3 The DEFINEKEY function defines the key for the hash object. The keys in the example are the same that would be used in a star schema to join the dimension tables to the fact table.

4 The DEFINEDATA function lists the columns from the dimension tables that you want to make available to the scoring step through the hash object.

5 Because this example uses the STATE variable as a filter, the program determines whether a record matches New Jersey first. Only then does the program look for matching records in the other hash objects. The FIND function is used to locate a matching record for the particular hash object. The code compares the State variable against the literal 'NJ ' after the CST.FIND() call. Otherwise, the State variable would not have been updated with the value that corresponds to the current record in the fact table.

6 A record that passes the filter and can be successfully looked up in the hash objects is output to the scoring result set.

For information about the syntax, see "Using DATA Step Component Objects" in the *SAS Language Reference: Concepts* and “Dictionary of Hash and Hash Iterator Object Language Elements” in *SAS Component Objects: Reference*.

The following example shows how the scoring program is used with the HASHDATA option:

```
filename fref './hash_schema.txt';  1

proc imstat;
   table lasr.mailorder;
   score pgm=fref hashdata(star.catalog,  2
                           star.products,
                           star.customers)
                  symbols(catalog cost price type name city state)  3
                  keep   (date year)
                  temptable;  4
run;
```

1 The scoring program is saved in a file that is named **`hash_schema.txt`**. The file is referenced with a FILENAME statement.

2 The HASHDATA option specifies the names of the dimension tables. In this example, the table names do not match the libref that is used for accessing the fact table. Specify the tables in the HASHDATA option as they are shown in the results of the TABLEINFO / HOST="*hostname.example.com*" PORT=*number* statement. The names in the DATASET options in the DECLARE HASH statements of the scoring program must match.

3 The SYMBOLS option specifies the columns to transfer from the dimension tables to the result set. Although these are columns in an in-memory table, they are not columns in the active table, and therefore must be transferred explicitly to the result set. The KEEP option specifies the columns from the active table, MailOrder, to transfer to the result set.

4 The TEMPTABLE option specifies to store the result set as an in-memory table. Otherwise, the result set is transferred to your SAS session.

### *Example 2: Using the RETAIN Statement*

You can use the RETAIN statement in your program, but be aware of the following requirements for predictable results:

- The active table must be partitioned by one or more variables.

- The active table must be ordered by one or more variables.
- You must specify the DSRETAIN option and the PARTITION option.

If the data in the active table are not partitioned and ordered, then the results are unpredictable and are unlikely to be useful.

```
libname example host="grid001.example.com" port=10010 tag="hps";

data example.prdsale(partition=(COUNTRY REGION DIVISION PRODTYPE PRODUCT)
   orderby=(year month));
   set sashelp.prdsale;
run;

filename ret "/data/retain.pgm";
data _null_;
file "/data/retain.pgm";
put "retain prev;";  1
put "if __first_in_partition then do;";
put "  prev = predict;";
put "end;";
put "diff = predict - prev;";
put "prev = predict;";
put "put 'n = ' _n_;"; 2
run;

proc imstat pgmmsg;  3
  table example.prdsale;
  score code=ret dsretain temptable keep=(actual predict quarter month)
    partition="CANADA", "EAST", "CONSUMER", "FURNITURE", "SOFA"; 4
run;
  table example.&_TEMPLAST_;
  fetch / format to=5;
run;
  table example.&_PGMMSG_;
  fetch / format to=5;
run;
```

1 The RETAIN statement is used in the DATA step program to submit to the server.

2 The PUT statement is not necessary for the analysis. It is included in the example to demonstrate how a PUT statement adds records to the &_PGMMSG_ temporary table instead of the SAS log.

3 The PGMMSG option is required for the server to create the &_PGMMSG_ table.

4 The SCORE statement includes the DSRETAIN option so that only the variables specified in a RETAIN statement are retained. The TEMPTABLE option is used to generate a temporary table for the scored results. The PARTITION option is required in order to use the **__first_in_partition** variable. In this example, it also specifies the partition to score. This can be used to avoid computing values for partitions that you are not interested in.

The following display shows the results. The FETCH statement used the TO= option to limit the results to the first five rows.

| Temporary Table Information for Table HPS.PRDSALE | |
|---|---|
| Statement | SCORE |
| Temporary Table | _T_C6ACA482_7F80C80FB120 |
| Table Type | Program messages |

| Temporary Table Information for Table HPS.PRDSALE | |
|---|---|
| Statement | SCORE |
| Temporary Table | _T_1919DDB5_7F80C80FB5E8 |
| Table Type | SCORE |

Selected Records from Table _T_1919DDB5_7F80C80FB5E8

| ACTUAL | PREDICT | QUARTER | MONTH | prev | diff | COUNTRY | REGION | DIVISION | PRODTYPE | PRODUCT |
|---|---|---|---|---|---|---|---|---|---|---|
| $5.00 | $425.00 | 1 | Jan | 425 | 0 | CANADA | EAST | CONSUMER | FURNITURE | SOFA |
| $164.00 | $215.00 | 1 | Feb | 215 | -210 | CANADA | EAST | CONSUMER | FURNITURE | SOFA |
| $422.00 | $948.00 | 1 | Mar | 948 | 733 | CANADA | EAST | CONSUMER | FURNITURE | SOFA |
| $424.00 | $544.00 | 2 | Apr | 544 | -404 | CANADA | EAST | CONSUMER | FURNITURE | SOFA |
| $854.00 | $764.00 | 2 | May | 764 | 220 | CANADA | EAST | CONSUMER | FURNITURE | SOFA |

Selected Records from Table _T_C6ACA482_7F80C80FB120

| Rank | Thread | Text |
|---|---|---|
| 5 | 0 | n = 1 |
| 5 | 0 | n = 2 |
| 5 | 0 | n = 3 |
| 5 | 0 | n = 4 |
| 5 | 0 | n = 5 |

As expected, the Diff variable shows the difference between the value for Predict and the retained value of Predict that is stored in the Prev variable. The second table shows the first five records from the &_PGMMSG_ table. Be aware that attempting to use _N_ in a program with a distributed server is more complicated than in a DATA step in a Base SAS session.

## SERVERINFO Statement

The SERVERINFO statement returns information about the SAS LASR Analytic Server.

### Syntax

**SERVERINFO** </ *option*>;

#### *SERVERINFO Statement Options*

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option.

**NORANKS**
specifies to omit the list of host names for the worker nodes. This option reduces the output of the SERVERINFO option considerably for large environments.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server. If you do not specify a PORT= value, then behavior of the SERVERINFO statement depends on whether an in-memory table is active. If there is no active table, then the procedure attempts to connect to the server using the LASRPORT macro variable. If a table is active, the information is gathered for the server that is implied by the libref of the active table.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

## Details

### *ODS Table Names*

The SERVERINFO statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| ServerInfo | Information about a LASR Analytic Server | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# SERVERPARM Statement

The SERVERPARM statement enables you to change some global settings for the server if you have sufficient authorization. The user account that starts the server has privileges to modify server parameters.

## Syntax

**SERVERPARM** <*options*>;

### *SERVERPARM Statement Options*

**CONCURRENT=*number***
specifies the number of concurrent requests that can execute in the server. Once the threshold is met, the requests are queued and then executed as the currently running requests complete.

**Alias** NACTIONS=

**Default** 20

**EXTERNALMEM=*pct***
specifies the percentage of memory that can be allocated before the server stops transferring data to external processes such as external actions and the SAS High-

Performance Analytics procedures. If the percentage is exceeded, the server stops transferring data.

**Default** 75

**Tip** If the external processes are running on a separate cluster, you can increase this value to 100%.

**HADOOPHOME="*path*"**
specifies the path for the HADOOP_HOME environment variable. Changing this variable is useful for migrating SASHDAT files from one Hadoop installation to another.

Setting the HADOOP_HOME environment variable is a server-wide change. All requests, by all users, for reading files from HDFS and saving files, use the specified HADOOP_HOME. This can cause unexpected results if users are not aware of the change.

*Note:* If you are using this option to migrate SASHDAT files, then consider starting a server for that exclusive purpose.

**Alias** HADOOP=

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server. If you do not specify a PORT= value, then behavior of the SERVERPARM statement depends on whether an in-memory table is active. If there is no active table, then the procedure attempts to connect to the server using the LASRPORT macro variable. If a table is active, the information is gathered for the server that is implied by the libref of the active table.

**TABLECEILING=*n* M | G**
specifies a process virtual memory limit (in megabytes or gigabytes) for the server. After the limit is met, adding tables and appending rows to tables is rejected by the server. This option provides a soft limit that enables the server to continue to function in a restricted way. Memory use might increase above the setting if the server requires memory to perform an analysis. However, new data, including temporary tables, cannot be added to the server. The limit applies to all virtual memory used by the server, not just the virtual memory that is used by tables. Setting the value to zero removes the limit.

This option has no effect for non-distributed servers. For non-distributed servers, you can specify a virtual memory size limit with the MEMSIZE system option.

**Default** Unlimited

**Applies to** Distributed SAS LASR Analytic Server

**TABLEMEM=*pct***
specifies the percentage of memory that can be allocated before the server rejects requests to add tables or append data. If the percentage is exceeded, adding a table or appending rows to tables fails. These operations continue to fail until the percentage is reset or the memory usage on the server drops below the threshold.

This option has no effect for non-distributed servers. For non-distributed servers, the memory limits can be controlled with the MEMSIZE system option.

*Note:* The specified *pct* value does not specify the percentage of memory allocated to in-memory tables. It is the percentage of all memory used by the entire machine that—if exceeded—prevents further addition of data to the server. The memory used is not measured at the process or user level, it is computed for the entire machine. In other words, if operating system processes allocate a lot of memory, then loading tables into the server might fail. The threshold is not affected by memory that is associated with SASHDAT tables that are loaded from HDFS.

**Alias** MEMLOAD=

**Default** 75

**TEMPNAMES=YES | NO**
specifies whether the server writes the full name of temporary tables to the server log file (TEMPNAMES=YES) or whether the names are masked (TEMPNAMES=NO). Because the name of the temporary table provides access to the table, the server does not display the full names of temporary tables, by default.

This option can be useful if you need to debug a series of requests that use temporary tables as input. By changing the handling of temporary names with TEMPNAMES=YES, you can see the full temporary table names in the log file.

## SERVERTERM Statement

The SERVERTERM statement sends a termination request to the server that is identified through the statement options. You must have sufficient authorization for this request to succeed.

### Syntax

**SERVERTERM** <*options*>;

#### *SERVERTERM Statement Options*

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server.

## SERVERWAIT Statement

The SERVERWAIT statement suspends execution of the IMSTAT procedure until the server that it uses receives a termination request.

### Syntax

**SERVERWAIT** <*options*>;

### SERVERWAIT Statement Options

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server.

## SET Statement

The SET statement is used to append in-memory tables to each other. The result of the operation is not a temporary table, but the appending of rows from the secondary tables to the active table.

**Examples:** "Example 6: Appending Tables" on page 370
"Example 7: Appending a Non-Partitioned Table to a Partitioned Table" on page 371

### Syntax

**SET** *set-specification1 <set-specification2 …> </ options>*;

### Required Argument

***set-specification***
specifies the table to append to the active table and options. You can list multiple set-specifications. A table can be used in more than one set-specification, and you can specify the active table in a set-specification.

*table-name* <(TAG='*server-tag*')>)

***table-name***
specifies the table to append to the active table.

**TAG='*server-tag*'**
specifies the server tag to use for identifying the table to append.

### SET Statement Options

**DROP**
specifies that the secondary tables (the tables specified in the set-specifications) are dropped from the server after the statement executes successfully. If the active table is listed in a set-specification, it is not dropped.

**NOPARTITION**
specifies to append the secondary tables and undo the partitioning of the active table. If the active table is partitioned, and you append partitioned tables to it, then the server rejects the request unless all the tables have the same partitioning variables, in the same order, and have the same key length. When this option is specified, the active table is no longer partitioned if the SET statement succeeds.

**Alias** NOPART

**WHEREWITHALL**
specifies to apply the WHERE clause to the active table, in addition to the secondary tables to append. By default, the rows of the secondary tables that are appended to the active table are filtered according to the WHERE clause. Rows marked for

deletion or purging are not appended to the main table. By default, the WHERE clause does not filter rows of the active table. If you want the WHERE clause to apply to the active table, specify this option.

**Alias** ALLWHERE

## STORE Statement

The STORE statement enables you to assign the contents of previously saved tables to macro variables. You can reuse the results from one statement as input for subsequent statements in the same IMSTAT procedure.

### Syntax

**STORE** *table-name* <[*table-number*]>
(*row-number* | _ALL_ | _LAST_ | *row-list* | WHERE="*where-clause*" <,>
*column-number* | _ALL_ | COLS=*column-list*) = *macro-variable-name* </ *options*>;

#### *Required Arguments*

***column-number***
specifies the column number to access as it appears in the default output or with the REPLAY statement. Be aware that hidden columns that might appear in an output data set when an ODS table is converted to a SAS data set are not counted. The first column is numbered 1. You can specify _ALL_ as an alternative, or specify the column names in the COLS= option.

***macro-variable-name***
specifies the name of a macro variable to use for storing the value.

***table-name***
specifies the saved result table to use.

***row-number***
specifies the row number to access as it appears in the default output or with the REPLAY statement. The first row is numbered 1. You can specify _ALL_, _LAST_, a numeric list of rows (row-list), or a WHERE clause as alternatives. When you specify a row-list, any row number less than 1, greater than the number of rows, and duplicate row numbers, are ignored.

If you specify a WHERE= clause, you can use Boolean expressions like NOT, BETWEEN, CONTAINS, LIKE, and mathematical expressions. The following operators are available:

- prefix +, prefix -
- <>, ><, ** (max, min, and power)
- *, /
- infix +, infix -
- || (concatenation)
- =, ^=, <, <=, >, >=, IN(*set*), IS NULL, IS MISSING
- AND, &

- OR, |

**Examples** The following row-list accesses rows 0, 1, and 2 from the results.

```
(0, 1, 2)
```

The following row-list accesses rows 0, 1, 2, 4, 8, 12, 16, and 20.

```
(0 to 2 by 1, 4, 8 to 20 by 4)
```

## Optional Arguments

**[*table-number*]**
specifies the table to use for accessing multi-part tables. In the following example, the HISTOGRAM statement generates one histogram table for the Cylinders variable and a second for the EngineSize variable. The two histogram tables are stored in the temporary HistTab table. In order to access the second histogram table, the `[2]` is used. If you do not specify a table-number, the first table is used.

**Example**

```
proc imstat data=mylasr.cars(tag=sashelp);;
    histogram Cylinders EngineSize / save=HistTab;
    store HistTab[2](2,6) = Engsz_Pct;
quit;
%put &Engsz_Pct;
```

**CONTROL="*control-string*"**
specifies the string that used to format cell values before they are stored in the macro variable. The control string must include the same number of placeholders as the number of columns. The default placeholder is `%`. There are two more placeholders besides `%`, `#` and `^`. If the *i*-th placeholder in CONTROL= is:

- `#`, then it is a placeholder for the value of the *i*-th relevant column in the first relevant row
- `^`, then it is a placeholder for the value of the *i*-th relevant column in the last relevant row

In the preceding statements, first means the relevant row with the smallest row number and last means the relevant row that with the largest row number. See Example 4 on page 353.

If you read character values that you want to use in a WHERE clause, you might need to enclose the placeholders in quotation marks.

**LEFT="*left-side-string*"**
specifies the string to assign as a prefix to the macro variable.

**NODUPS**
specifies to ignore duplicate formatted cell values.

**Restriction** This option applies to numeric values only and when only one column is accessed from the results.

**RIGHT="*right-side-string*"**
specifies the string to assign as a suffix to macro variable.

**SEPARATOR="*separator-string*"**
specifies the string to use for separating the formatted cell values.

**Default** " " (the space character)

## Details

### *Using the STORE Statement*

The simplest use of the STORE statement is to read a value from a cell in the results, assign it to a macro variable, and use it in subsequent statements. The following statement is copied from part of Example 3 and demonstrates reading the value of a single cell into a macro variable.

```
store mpgtab (_last_, cols=Mean) = avgmpgcity;
```

More sophisticated uses of the STORE statement are possible. Example 2 on page 350 shows how variable names are read from a results table and used in a subsequent programming statement.

Perhaps the most sophisticated use of the STORE statement is to construct a string that can be used in a WHERE clause. Such a use typically requires use of the LEFT=, CONTROL=, SEPARATOR=, and RIGHT= options. The following steps describe the concept and flow:

1. Specify the cells from the results to use by specifying the rows and columns.
2. Use options to control how to construct the macro value, made up of cell values and the following:
   a. A string to prefix to the left side of the macro variable. Typically, "(" is common.
   b. A control string to use for formatting cell values for each row.
   c. A string to use for separating the formatted control strings. The control string and the separator are used.
   d. A string to use as a suffix for the right side of the macro variable. Typically, ")" is common.

For examples that demonstrate using these options, see Example 3 and Example 4.

### *Listing Column Names from ODS Tables*

In order to use the STORE statement, you need to save the ODS table output with a statement that supports the SAVE= option, such as **`summary mpg_city / save=mpgtab;`**. The ODS output is shown in Example 3 on page 352.

You can reference the mean value with a statement like **`store mpgtab(1,6) = meanMpgCity;`** because the **`Mean`** column is the sixth column. However, it is more robust to reference columns by name, such as **`store mpgtab(where="Column eq 'MPG_City'", cols=Mean) = meanMpgCity;`**.

To determine the column names, like **`Column`** and **`Mean`**, you need to know the ODS table name for the statement in the IMSTAT procedure and to use the TEMPLATE procedure. The following example shows how to identify the column names that are available in a saved table from the SUMMARY statement.

```
proc template;
    source LASR.IMSTAT.Summary;  1
run;
```

1 The LASR.IMSTAT portion of the source statement is common to all statements in the IMSTAT procedure. The name of the table, such as Summary, is provided in the documentation for each statement.

```
define table Lasr.Imstat.Summary;
   notes "Descriptive Statistics";
   dynamic TableName;
   column Table GroupBy Column Min Max N Sum Mean Std StdErr CV CSS
          USS TValue ProbT NMiss Bin;
   header h1;

   define h1;
      text "Summary Statistics for Table " TableName;
      space = 1;
      spill_margin;
…
```

The SAS log shows the column names. **`Table`** is not used unless the ODS output is written to a SAS data set. The **`GroupBy`** column is not used because the GROUPBY= option was not used in the SUMMARY statement. The remaining columns are available and you must specify the column name as it is shown. The names are case-sensitive.

## Examples

### *Example 1: Accessing Multi-Part Table Output and Storing a Single Value*

The following STORE statement reads the value from the second row of the Percent column into the Bin2Pct macro variable.

```
proc imstat data=example.cars(tag=sashelp);
    histogram Cylinders EngineSize / save=HistTab;
    store HistTab[2](2,cols= percent) = Bin2Pct;
quit;
%put &Bin2Pct;
```

The following display shows the HISTOGRAM output for the EngineSize variable.

| Histogram for Column EngineSize in Table SASHELP.CARS | | | | | |
|---|---|---|---|---|---|
| Bin Number | Min | Mid | Max | Frequency | Percent |
| 1 | 1.2 | 1.5 | 1.8 | 23 | 5.3738 |
| 2 | 1.8 | 2.1 | 2.4 | 84 | 19.6262 |
| 3 | 2.4 | 2.7 | 3 | 71 | 16.5888 |
| 4 | 3 | 3.3 | 3.6 | 117 | 27.3364 |
| 5 | 3.6 | 3.9 | 4.2 | 37 | 8.6449 |
| 6 | 4.2 | 4.5 | 4.8 | 63 | 14.7196 |
| 7 | 4.8 | 5.1 | 5.4 | 15 | 3.5047 |
| 8 | 5.4 | 5.7 | 6 | 10 | 2.3364 |
| 9 | 6 | 6.3 | 6.6 | 6 | 1.4019 |
| 10 | 6.6 | 6.9 | 7.2 | 1 | 0.2336 |
| 11 | 7.2 | 7.5 | 7.8 | 0 | 0 |
| 12 | 7.8 | 8.1 | 8.4 | 1 | 0.2336 |

The PUT statement shows the value from the Bin2Pct macro variable in the SAS log. The macro variable can show a different number of significant digits than the table output.

```
NOTE: The string 19.626168224 related to table histtab has been stored in the
         macro variable bin2pct.

%put &bin2pct;
19.626168224
```

### *Example 2: Storing Variable Names*

The following DISTINCT statement calculates the number of unique values for the numeric variables and stores the results in DistinctTab.

```
proc imstat data=example.cars(tag=sashelp);
    distinct _numeric_ / save=distincttab;
run;
```

The following display shows the results.

| Number of Distinct Values in Table SASHELP.CARS | | |
|---|---|---|
| Column | Number of Distinct Values | Number of Missing Values |
| MSRP | 410 | 0 |
| Invoice | 425 | 0 |
| Engine Size | 43 | 0 |
| Cylinders | 8 | 2 |
| Horsepower | 110 | 0 |
| MPG_City | 28 | 0 |
| MPG_Highway | 33 | 0 |
| Weight | 348 | 0 |
| Wheelbase | 40 | 0 |
| Length | 67 | 0 |

The following STORE statement reads the results and filters for rows with a number of distinct values that is greater than 100. The values from the first column (the variable names) are stored in the macro variable VarList1.

```
    store distincttab (where="NDistinct > 100", 1) = varlist1;
run;
```

The SAS log shows the constructed string:

```
NOTE: The string MSRP Invoice Horsepower Weight related to table distincttab
        has been stored in the macro variable varlist1.
```

The following CORR statement uses the variable names that were stored in the VarList1 macro variable.

```
    corr &varlist1;
run;
```

The following display shows the results of the CORR statement.

| Pairwise Correlations for Table SASHELP.CARS | | | | | |
|---|---|---|---|---|---|
| Column | Row | MSRP | Invoice | Horsepower | Weight |
| MSRP | 1 | 1.0000 | 0.9991 | 0.8269 | 0.4484 |
| Invoice | 2 | 0.9991 | 1.0000 | 0.8237 | 0.4423 |
| Horsepower | 3 | 0.8269 | 0.8237 | 1.0000 | 0.6308 |
| Weight | 4 | 0.4484 | 0.4423 | 0.6308 | 1.0000 |

### *Example 3: Storing Values from Multiple Cells*

```
proc imstat data=example.cars(tag=sashelp);
    summary mpg_city / save=mpgtab;
run;
```

The following display shows the results. The overall average Mpg_City for all makes and models is highlighted.

| Summary Statistics for Table SASHELP.CARS | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Column | Min | Max | N | Sum | Mean | Std Dev. | Std Error | Coefficient of Variation | Number Missing |
| MPG_City | 10.0000 | 60.0000 | 428 | 8586 | 20.0607 | 5.2382 | 0.2532 | 26.1118 | 0 |

The following STORE statement stores the overall average Mpg_City value in the AvgMpgCity macro variable. Then, a summary of the same Mpg_City variable is requested, but grouped by Make. The results are saved in SummaryTab.

```
 store mpgtab (_last_, cols=Mean) = avgmpgcity;
run;
    summary mpg_city / groupby=make save=summarytab;
run;
```

The following display shows the part of the results.

| Summary Statistics for Table SASHELP.CARS | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Make | Column | Min | Max | N | Sum | Mean | Std Dev. | Std Error | Coefficient of Variation | Number Missing |
| Acura | MPG_City | 17.0000 | 24.0000 | 7 | 136 | 19.4286 | 2.6992 | 1.0202 | 13.8930 | 0 |
| Audi | MPG_City | 14.0000 | 23.0000 | 19 | 351 | 18.4737 | 2.3891 | 0.5481 | 12.9322 | 0 |
| BMW | MPG_City | 16.0000 | 21.0000 | 20 | 374 | 18.7000 | 1.5927 | 0.3561 | 8.5174 | 0 |
| Buick | MPG_City | 15.0000 | 20.0000 | 9 | 170 | 18.8889 | 1.6915 | 0.5638 | 8.9549 | 0 |
| Cadillac | MPG_City | 13.0000 | 18.0000 | 8 | 132 | 16.5000 | 2.0000 | 0.7071 | 12.1212 | 0 |

The following STORE statement accesses the results (from SummaryTab) and filters for rows with an above average Mpg_City value. The LEFT=, CONTROL=, SEPARATOR=, and RIGHT= options are used to build a string, substituting each value of Make for % in the CONTROL= option.

```
store summarytab (where="Mean > &avgmpgcity",cols=Make) = highmpgmakes /
      left="Make in (" control="'%'" separator=", " right=")"; run;
```

The SAS log shows the constructed string. Notice how the LEFT= value is used to begin the string, and the CONTROL= and SEPARATOR= values are used for each row in the results. The RIGHT= value is used to end the string.

```
NOTE: The string Make in ('Honda', 'Hyundai', 'Kia', 'MINI', 'Mazda',
        'Mitsubishi', 'Oldsmobile', 'Pontiac', 'Saab', 'Saturn', 'Scion',
        'Subaru', 'Suzuki', 'Toyota', 'Volkswagen') related to table
        summarytab has been stored in the macro variable highmpgmakes.
```

Finally, the constructed string is used in a WHERE clause. The CROSSTAB statement shows the frequency of each MPG_City value for each Make that is specified in the WHERE clause.

```
where &highmpgmakes;
    crosstab mpg_city*make;
run;
```

The following display shows part of the crosstabulation results.

| Cross-tabulation of MPG_City by Make for Table SASHELP.CARS with Aggregator N | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MPG_City | Honda | Hyundai | Kia | MINI | Mazda | Mitsubishi | Oldsmobile | Pontiac | Saab | Saturn | Scion | Subaru | Suzuki | Toyota | Volkswagen |
| 12 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 |
| 15 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 16 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 17 | 1 | 2 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 18 | 2 | 0 | 0 | 0 | 3 | 3 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 2 | 2 |
| 19 | 0 | 3 | 0 | 0 | 0 | 0 | 1 | 2 | 1 | 0 | 0 | 2 | 1 | 2 | 1 |

### *Example 4: Storing Values from Vertically Stacked Cells*

The following PRECENTILE statement calculates the quantiles for the Mpg_City variable and saves the results in MpgTab.

```
proc imstat data=example.cars(tag=sashelp);
    percentile mpg_city / save=mpgtab;
run;
```

The following display shows the results.

| Percentiles and Quantiles for Table SASHELP.CARS | | | |
|---|---|---|---|
| Column | Percentile | Value | Converged |
| MPG_City | 25 | 17.0000000 | Yes |
| MPG_City | 50 | 19.0000000 | Yes |
| MPG_City | 75 | 21.5000000 | Yes |

To create a WHERE clause that uses the values for the third quartile of the Mpg_City variable (between 19 and 21.5), the following STORE statement is used.

```
    store mpgtab (where="Pctl >= 50", cols=Value Value) = q3 /
          control="(mpg_city between # and ^)";
run;
```

The STORE statement performs the following:

- Filters the results to the rows where Percentile is greater than or equal to 50 (two rows).
- Reads the Value column (twice).
- Substitutes the first value in the first row for the # placeholder. Substitutes the second value from last row in the ^ placeholder, because it is the second placeholder.
- Stores the resulting string (mpg_city between 19 and 21.5) in the Q3 macro variable.

The SAS log shows the constructed string:

```
NOTE: The string (mpg_city between 19 and 21.5) related to table mpgtab has been
         stored in the macro variable q3.
```

The following statements then use a constructed string in a WHERE clause and request a summary of the Mpg_City variable that is grouped by Make.

```
where &q3.;
    summary mpg_city / groupby=make;
run;
```

The following display shows part of the results.

| Summary Statistics for Table SASHELP.CARS | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Make | Column | Min | Max | N | Sum | Mean | Std Dev. | Std Error | Coefficient of Variation | Number Missing |
| Acura | MPG_City | 20.0000 | 20.0000 | 1 | 20 | 20.0000 | 0 | 0 | 0 | 0 |
| Audi | MPG_City | 20.0000 | 21.0000 | 6 | 121 | 20.1667 | 0.4082 | 0.1667 | 2.0244 | 0 |
| BMW | MPG_City | 19.0000 | 21.0000 | 13 | 256 | 19.6923 | 0.6304 | 0.1748 | 3.2014 | 0 |
| Buick | MPG_City | 19.0000 | 20.0000 | 6 | 119 | 19.8333 | 0.4082 | 0.1667 | 2.0584 | 0 |
| Chevrolet | MPG_City | 19.0000 | 21.0000 | 5 | 100 | 20.0000 | 1.0000 | 0.4472 | 5.0000 | 0 |

## TABLE Statement

The TABLE statement is used to specify the in-memory table to use for subsequent IMSTAT procedure statements. You can use this statement to switch between different in-memory tables in a single run of the IMSTAT procedure.

**Example:** "Example 5: Creating a Star Schema" on page 369

### Syntax

**TABLE** <*libref.member-name*>;

#### *Optional Argument*

***libref.member-name***

specifies the libref for the SAS LASR Analytic Server and the table name.

If you do not specify the libref and member-name, the procedure closes the table that is currently open.

### Example

A common use for the TABLE statement is to reference a temporary table with the *libref*.**&_TEMPLAST_** macro variable. Temporary tables are in-memory tables that are created with the results of a statement that supports the TEMPTABLE option.

**TIP** Make sure that the statement that generated the temporary table is separated from the next statement with a RUN statement. The temporary table does not exist at parse time, it is created at run time when the statement is executed.

```
proc imstat data=lasrlib.sales2012;
    partition customerid;
run;

    table lasrlib.&_templast_;
run;

    /*
     *  More statements for the partitioned table.
     *  The PROMOTE statement can be used to convert the
     *  temporary table to a regular table.
     */
quit;
```

## TABLEINFO Statement

The TABLEINFO statement is used to return information about an in-memory table. This information includes the table name, label, number of rows and column, owner, encoding, and the time of table creation. If no table is in use, then information is returned for the in-memory tables for the server specified in the HOST= and PORT= options.

### Syntax

**TABLEINFO** </ *options*>;

#### *TABLEINFO Statement Options*

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option.

**PARTVARS**
specifies to include information about partition and orderby variables in the output of the TABLEINFO statement. This enables you to retrieve the names of those variables. If a table is not partitioned or ordered, "N/A" is displayed.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server. If you do not specify a PORT= value, then behavior of the TABLEINFO statement depends on whether an in-memory table is active. If there is no active table, then the procedure attempts to connect to the server using the LASRPORT macro variable. If a table is active, the information is gathered for the server that is implied by the libref of the active table.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

## Details

### ODS Table Names

The TABLEINFO statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| TableInfo | Information about tables on a LASR Analytic Server | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

# UNCOMPRESS Statement

The UNCOMPRESS statement is used to create a regular in-memory table from a compressed in-memory table. The result is stored as a temporary table.

## Syntax

**UNCOMPRESS** </ *options*>;

### UNCOMPRESS Statement Options

**INFO**
requests the server to report information about the compression state of a table, but does not perform uncompression. On a compressed table, the report includes information about the compressed size and compression ratio. On an uncompressed table, the results include the uncompressed size only. The option is also useful to find out how much memory a table consumes.

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

## Details

### ODS Table Names

The UNCOMPRESS statement generates the following ODS tables.

| ODS Table Name | Description | Option |
|---|---|---|
| Compress | Information from compressing or uncompressing tables | Default |
| TempTable | Information about a temporary table | When the INFO option is not used, a temporary table is generated. |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## Example

```
data lasr.prdsalecomp(squeeze=yes);  1
    set sashelp.prdsale;
run;

proc imstat data=lasr.prdsalecomp;
    uncompress / info;  2
run;
    uncompress;
    table lasr.&_templast_;
    promote prdsalenocomp;  3
run;
    table lasr.prdsalenocomp;
    uncompress / info;  4
quit;
```

1 The SQUEEZE=YES data set option is used so that the server compresses the table as it is loaded to memory.

2 The INFO option is used to report the compression information for the table.

3 The PROMOTE statement is used to make the uncompressed table into a permanent table and give it a name.

4 The INFO option is used to report the compression information for the table.

# UPDATE Statement

The UPDATE statement performs rowwise updates of the data in an in-memory table.

## Syntax

UPDATE *variable1=value1 <variable2=value2 …> </ options>*;

UPDATE DATA=*libref.member-name </ options>*;

### Required Arguments

***variable***

specifies the name of the variable to update.

*Note:* Because DATA= is a keyword, you cannot specify it as a variable name. For an example of how to modify a variable named Data, see "Example 2: Update with a Data Set".

***value***

specifies the value to assign to the variable.

**DATA=*libref.member-name***

specifies the libref and table name of a SAS data set to use for updating the in-memory table. The data set must contain the variables and values that you want to update. You can specify a **_WHERE_** variable in the data set to apply as a filter to the

particular set of update values. This clause in the data set augments the overall WHERE clause, if one is specified.

### UPDATE Statement Options

**CODE=*file-reference***
specifies a file reference to a SAS program to use for the row update (an update script). You can combine the specification of a SAS program through the CODE= option with the name-value pair specification or the DATA= specification for bulk updates. The updates that are specified in the name-value pair and DATE= specifications are performed first and then the update script executes on the modified row to produce the update.

**Alias** PGM=

**NOPREPARSE**
prevents the procedure from pre-parsing and pre-generating code for temporary expressions, scoring programs, and other user-written SAS statements.

When this option is specified, the user-written statements are sent to the server "as-is" and then the server attempts to generate code from it. If the server detects problems with the code, the error messages might not to be as detailed as the messages that are generated by SAS client. If you are debugging your user-written program, then you might want to pre-parse and pre-generate code in the procedure. However, if your SAS statements compile and run as you want them to, then you can specify this option to avoid the work of parsing and generating code on the SAS client.

When you specify this option in the PROC IMSTAT statement, the option applies to all statements that can generate code. You can also exclude specific statements from pre-parsing by using the NOPREPARSE option in statements that allow temporary columns or the SCORE statement.

**Alias** NOPREP

**SAVE=*table-name***
saves the result table so that you can use it in other IMSTAT procedure statements like STORE, REPLAY, and FREE. The value for *table-name* must be unique within the scope of the procedure execution. The name of a table that has been freed with the FREE statement can be used again in subsequent SAVE= options.

**TEMPEXPRESS="*SAS-expressions*"**
**TEMPEXPRESS=*file-reference***
specifies either a quoted string that contains the SAS expression that defines the temporary variables or a file reference to an external file with the SAS statements.

**Alias** TE=

**TEMPNAMES=*variable-name***
**TEMPNAMES=(*variable-list*)**
specifies the list of temporary variables for the request. Each temporary variable must be defined through SAS statements that you supply with the TEMPEXPRESS= option.

**Alias** TN=

## Details

### Usage Notes

It is common to use the UPDATE statement with a WHERE clause. The clause filters the rows to which the updates are applied. If you are unsure about the number of rows that can be updated, use the NUMROWS statement to determine how many rows would be affected by the rowwise update.

*Note:* The ODS output of the UPDATE statement indicates the number of rows that were updated. This value actually indicates the number of rows that met the condition in the WHERE clause, not the number of rows that were modified.

You can update the values of ORDERBY variables, but you cannot update the value of variables that are used for constructing partition keys.

You cannot update the values of permanent computed variables. Their values are determined by the SAS program that originally defined them.

### ODS Table Names

The UPDATE statement generates the following ODS table.

| ODS Table Name | Description | Option |
|---|---|---|
| RowUpdate | Number of Rows Considered for an Update | Default |

For information about using the ODS table with SAVE= option, see the Details on page 348 section of the STORE statement.

## Examples

### Example 1: Update with a SAS Program

```
libname example sasiola host="grid001.example.com" port=10010 tag=hps;

data example.prdsale;
    set sashelp.prdsale;
run;

filename pgm "some-path/update.pgm";  1
data _null_;
    file pgm;
    put "if quarter=3 and product eq 'SOFA' then do;";
    put "  predict = 1.05 * predict;";
    put "end;";
    put "if quarter=3 and product ne 'SOFA' then do;";
    put "  predict = 1.03 * predict;";
    put "end;";
run;

proc imstat data=example.prdsale;  2
    /* columninfo; */
    /* fetch / format; */
```

```
    update / code=pgm;
run;

    where quarter=3;
    fetch / format;
quit;
```

1 The FILENAME statement and the DATA statement are used to write a simple SAS program to a file. The file reference, Pgm, is used later in the UPDATE statement.

2 In the PROC IMSTAT example, the COLUMNINFO and FETCH statements are enclosed in comments, but they are helpful for helping you understand the data. The UPDATE statement uses the CODE= option to specify the file reference. The WHERE statement and FETCH statements are used to display the first 20 rows that were updated.

### *Example 2: Update with a Data Set*

```
libname example sasiola host="grid001.example.com" port=10010 tag=hps;

data example.tableA;  1
  do x=11 to 20;
    data = x;
    output;
  end;
run;

data pgm;  2
  length _where_ $64;
  _where_ = "x between 5 and 13";
  data = 10;
output;
  _where_ = "x between 14 and 30";
  data = 15;
output;
run;

proc imstat data=example.tableA;
  fetch / format; run;
quit;

proc imstat data=lasr.tableA;
  update data=pgm;  3
run;
  fetch / format orderby=(x);
quit;
```

1 The SAS LASR Analytic Server engine and a DATA step are used to create a simple in-memory table with columns named x and Data.

2 A data set, named Pgm, is created in the Work library. The data set describes two updates to make an in-memory table. The _WHERE_ column is used to specify the rows to modify.

3 The DATA= option is used to specify the data set with the update instructions.

The following display shows the original in-memory table before it is modified by the UPDATE statement.

**The IMSTAT Procedure**

| Selected Records from Table HPS.TABLEA | |
|---|---|
| x | data |
| 11 | 11 |
| 12 | 12 |
| 13 | 13 |
| 14 | 14 |
| 15 | 15 |
| 16 | 16 |
| 17 | 17 |
| 18 | 18 |
| 19 | 19 |
| 20 | 20 |

The following display shows the number of rows that were considered for an update (the number of rows that met the _WHERE_ criteria) and the result of the UPDATE statement.

| Number of Rows Updated in Table HPS.TABLEA | | |
|---|---|---|
| Update Data Set | Observation | Number of Rows |
| WORK.PGM | 1 | 3 |
| WORK.PGM | 2 | 7 |

| Selected Records from Table HPS.TABLEA | |
|---|---|
| x | data |
| 11 | 10 |
| 12 | 10 |
| 13 | 10 |
| 14 | 15 |
| 15 | 15 |
| 16 | 15 |
| 17 | 15 |
| 18 | 15 |
| 19 | 15 |
| 20 | 15 |

## QUIT Statement

The QUIT statement is used to end the procedure execution. When the procedure reaches the QUIT statement, all resources allocated by the procedure are released. You can no longer execute procedure statements without invoking the procedure again. However, the connection to the server is not lost, because that connection was made through the SAS LASR Analytic Server engine. As a result, any subsequent invocation of the procedure that uses the same libref executes almost instantaneously because the engine is already connected to the server.

**Interaction:** Using a DATA step or another procedure step is equivalent to issuing a QUIT statement. If there is an error during the procedure execution, it is also equivalent to issuing a QUIT statement.

### Syntax

**QUIT**;

# Examples: IMSTAT Procedure (Data and Server Management)

## Example 1: Partitioning a Table into a Temporary Table

### Details

This PROC IMSTAT example demonstrates partitioning a table as it is loaded to memory and then saving it to a temporary table with different partitioning variables.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.prdsale(partition=(country region)); 1
    set sashelp.prdsale;
run;

proc imstat data=example.prdsale;
    partitioninfo;
    summary actual predict / partition; 2
run;

    /* partition the active table, example.prdsale, by region and prodtype */
    partition region prodtype; 3
run;
    table example.&_templast_; 4
run;
    partitioninfo;
    summary actual predict / partition; 5
```

```
quit;
```

## Program Description

1. The Prdsale data set is loaded into memory and partitioned by the unique combinations of the formatted values for the Country and Region variables.
2. The procedure examines the partitioning of the table and requests a summarization of the Actual and Predict variables by the partition values (unique combinations of Country and Region).
3. In order to accommodate a different data access pattern, the table is partitioned by unique combinations of the Region and Prodtype variables. The table is stored in a temporary table and the name is assigned to the _TEMPLAST_ macro variable.
4. The TABLE statement references the _TEMPLAST_ macro variable and sets the temporary table as the active table. All statements that follow use the temporary table.
5. As with the previous SUMMARY statement, the partitioning is examined and the summary is requested for the Actual and Predict variables by the unique combinations of the Region and Prodtype variables.

## Output

***Output 5.1*** *Partitions for Prdsale When Partitioned by Country and Region*

| Partitions and their Properties by Node in Table WORK.PRDSALE | | | | |
|---|---|---|---|---|
| LASR Node | Partition Number | Partition Key | Size in kBytes | Number of Records |
| [illegible] | 1 | CANADA EAST | 28.125 | 240 |
| [illegible] | 1 | CANADA WEST | 28.125 | 240 |
| [illegible] | 1 | GERMANY EAST | 28.125 | 240 |
| [illegible] | 1 | GERMANY WEST | 28.125 | 240 |
| [illegible] | 1 | U.S.A. EAST | 28.125 | 240 |
| [illegible] | 1 | U.S.A. WEST | 28.125 | 240 |

***Output 5.2*** *Summary Statistics for Prdsale When Partitioned by Country and Region*

| Summary Statistics for Table WORK.PRDSALE | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COUNTRY | REGION | Column | Min | Max | N | Sum | Mean | Std Dev. | Std Error | Coefficient of Variation | Number Missing |
| CANADA | EAST | ACTUAL | 5.0000 | 999.00 | 240 | 127485 | 531.19 | 281.61 | 18.1781 | 53.0160 | 0 |
| CANADA | EAST | PREDICT | 0 | 986.00 | 240 | 120646 | 502.69 | 284.29 | 18.3510 | 56.5540 | 0 |
| CANADA | WEST | ACTUAL | 3.0000 | 1000.00 | 240 | 119505 | 497.94 | 296.53 | 19.1406 | 59.5507 | 0 |
| CANADA | WEST | PREDICT | 6.0000 | 1000.00 | 240 | 112373 | 468.22 | 275.99 | 17.8150 | 58.9443 | 0 |
| GERMANY | EAST | ACTUAL | 13.0000 | 1000.00 | 240 | 124547 | 518.95 | 287.71 | 18.5714 | 55.4406 | 0 |
| GERMANY | EAST | PREDICT | 4.0000 | 993.00 | 240 | 117579 | 489.91 | 292.43 | 18.8762 | 59.6901 | 0 |
| GERMANY | WEST | ACTUAL | 3.0000 | 996.00 | 240 | 121451 | 506.05 | 289.17 | 18.6658 | 57.1429 | 0 |
| GERMANY | WEST | PREDICT | 0 | 981.00 | 240 | 113975 | 474.90 | 280.49 | 18.1056 | 59.0637 | 0 |
| U.S.A. | EAST | ACTUAL | 4.0000 | 984.00 | 240 | 118229 | 492.62 | 282.26 | 18.2200 | 57.2983 | 0 |
| U.S.A. | EAST | PREDICT | 1.0000 | 1000.00 | 240 | 120587 | 502.45 | 301.35 | 19.4518 | 59.9757 | 0 |
| U.S.A. | WEST | ACTUAL | 6.0000 | 984.00 | 240 | 119120 | 496.33 | 285.67 | 18.4401 | 57.5567 | 0 |
| U.S.A. | WEST | PREDICT | 22.0000 | 999.00 | 240 | 121135 | 504.73 | 280.10 | 18.0801 | 55.4944 | 0 |

***Output 5.3*** *Partitions for Prdsale When Partitioned by Region and Prodtype*

| Partitions and their Properties by Node in Table _T_BA5820CB_446FB438 | | | | |
|---|---|---|---|---|
| LASR Node | Partition Number | Partition Key | Size in kBytes | Number of Records |
| [illegible] | 1 | WEST OFFICE | 50.625 | 432 |
| [illegible] | 1 | EAST FURNITURE | 33.75 | 288 |
| [illegible] | 1 | EAST OFFICE | 50.625 | 432 |
| [illegible] | 1 | WEST FURNITURE | 33.75 | 288 |

***Output 5.4*** *Summary Statistics for Prdsale When Partitioned by Region and Prodtype*

| Summary Statistics for Table _T_BA5820CB_446FB438 | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| REGION | PRODTYPE | Column | Min | Max | N | Sum | Mean | Std Dev. | Std Error | Coefficient of Variation | Number Missing |
| EAST | FURNITURE | ACTUAL | 4.0000 | 999.00 | 288 | 146471 | 508.58 | 283.91 | 16.7296 | 55.8241 | 0 |
| EAST | FURNITURE | PREDICT | 0 | 1000.00 | 288 | 141116 | 489.99 | 299.16 | 17.6280 | 61.0542 | 0 |
| EAST | OFFICE | ACTUAL | 13.0000 | 1000.00 | 432 | 223790 | 518.03 | 284.22 | 13.6745 | 54.8653 | 0 |
| EAST | OFFICE | PREDICT | 12.0000 | 994.00 | 432 | 217696 | 503.93 | 288.06 | 13.8595 | 57.1640 | 0 |
| WEST | FURNITURE | ACTUAL | 3.0000 | 996.00 | 288 | 144154 | 500.53 | 297.06 | 17.5046 | 59.3491 | 0 |
| WEST | FURNITURE | PREDICT | 0 | 1000.00 | 288 | 137202 | 476.40 | 285.65 | 16.8320 | 59.9604 | 0 |
| WEST | OFFICE | ACTUAL | 6.0000 | 1000.00 | 432 | 215922 | 499.82 | 285.74 | 13.7477 | 57.1689 | 0 |
| WEST | OFFICE | PREDICT | 6.0000 | 999.00 | 432 | 210281 | 486.76 | 274.62 | 13.2125 | 56.4170 | 0 |

## Example 2: Promoting Temporary Tables to Regular Tables

### Details

The SUMMARY, SCORE, CROSSTAB, PERCENTILE, and DISTINCT statements offer a TEMPTABLE option. When you specify the TEMPTABLE option, the results of the statement are written to a temporary table. The PARTITION statement also results in a temporary table. If you want to keep the table, you can use the PROMOTE statement to convert the table from being a temporary table to a regular table. Once you do this, other users can access the data.

### Program

```
libname example sasiola host="grid001.unx.sas.com" port=10010 tag='hps';

data example.prdsale; set sashelp.prdsale; run;

proc imstat data=example.prdsale;
    summary / groupby=(country) temptable; 1
run;
    table example.&_templast_; 2
run;
    promote sum_by_country; 3
run;
    table example.sum_by_country; 4
run;
    fetch / format to=10; 5
quit;
```

### Program Description

1. The TEMPTABLE option stores the results of the SUMMARY statement to a temporary table. If this table is not promoted before the QUIT statement, it is removed from memory.
2. The TABLE statement references the _TEMPLAST_ macro variable and sets the temporary table as the active table. All statements that follow use the temporary table.
3. The PROMOTE statement converts the temporary table to a regular table with the name Sum_By_Country. The table is associated with the current library through the libref, Example. The SAS log also includes a note that indicates how to specify the libref and table name.
4. The TABLE statement makes the table the active table explicitly by specifying the libref and table name. The Sum_By_Country table is not removed from memory when the IMSTAT procedure terminates.
5. All the subsequent statements that follow the TABLE statement use the newly promoted table.

The example does not show the use of SAS LASR Analytic Server engine server tags. You can use server tags with the PROMOTE statement as show in the following code sample.

```
proc imstat data=example.prdsale;
    summary / groupby=(region) temptable;
run;

    table example.&_templast_;
run;
    promote sum_by_region / tag="sales";
run;
    table example.sum_by_country (tag="sales"); 4
run;
quit;
```

As shown in the previous example, the TAG= option is used in the PROMOTE statement. To access the table, the TABLE statement uses the TAG= data set option.

As shown in the following sample, the SAS log indicates the libref, table name, and server tag to use for accessing the table.

***Log 5.1*** *SAS Log for the PROMOTE Statement with the TAG= Option*

```
NOTE: The temporary table _T_BE5C2602_45A0DCB8 was successfully promoted to the
      LASR Analytic Server table WORK.SUM_BY_COUNTRY. You can access this table
with
      the TABLE statement as table EXAMPLE.sum_by_country(tag='sales').
```

## Example 3: Rebalancing a Table

### Details

It might be beneficial to rebalance the rows of a table if the data access patterns do not take advantage of partitioning or if the HDFS block distribution becomes uneven.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

proc imstat immediate;
  table example.table1;
  distributioninfo; 1
  balance;
  droptable; 2
  table example.&_templast_; 3
  promote table1; 4
  table example.table1;
  distributioninfo; 5
  /* save path="/hps" replace; */ 6
quit;
```

### Program Description

1. The DISTRIBUTIONINFO statement displays the number of rows from Table1 on each machine in the cluster.
2. The DROPTABLE statement is used to drop the active table, Table1.
3. The BALANCE statement rebalanced Table1 into a temporary table. The TABLE statement is used with the &_TEMPLAST_ macro variable to access the temporary table.
4. The PROMOTE statement changes the temporary table into a regular in-memory table with the original table name, Table1.
5. After setting the Table1 as the active table with the TABLE statement, the DISTRIBUTIONINFO statement displays the nearly homogenous distribution of rows.
6. The SAVE statement can be used to save the table back to HDFS with the homogeneous block distribution.

### Output

The following output shows the partial display for the first DISTRIBUTIONINFO statement. One machine has zero rows and another machine has approximately twice the number of rows.

***Output 5.5*** *Uneven Row Distribution*

| Number of Partitions and Records by Node in Table [illegible] | | |
|---|---|---|
| LASR Node | Number of Partitions | Number of Records |
| [illegible] | 0 | 165971 |
| [illegible] | 0 | 165984 |
| [illegible] | 0 | 0 |
| [illegible] | 0 | 165984 |
| [illegible] | 0 | 165984 |
| [illegible] | 0 | 165984 |
| [illegible] | 0 | 165984 |
| [illegible] | 0 | 331955 |
| [illegible] | 0 | 165984 |

The following output shows the homogenous distribution of rows after the BALANCE statement is used.

***Output 5.6*** *Homogenous Row Distribution*

| Number of Partitions and Records by Node in Table [illegible] | | |
|---|---|---|
| LASR Node | Number of Partitions | Number of Records |
| [illegible] | 0 | 165977 |
| [illegible] | 0 | 165977 |
| [illegible] | 0 | 165977 |
| [illegible] | 0 | 165977 |
| [illegible] | 0 | 165977 |
| [illegible] | 0 | 165977 |
| [illegible] | 0 | 165977 |
| [illegible] | 0 | 165977 |
| [illegible] | 0 | 165977 |

## Example 4: Deleting Rows and Saving a Table to HDFS

### Details

The server can delete rows from in-memory tables and also save tables to HDFS. The following example demonstrates using WHERE clause processing across RUN-group boundaries to copy a subset of an in-memory table to HDFS and then delete the subset from memory.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

data example.prdsale; set sashelp.prdsale; run;

proc imstat data=example.prdsale;
    where year=1994 and quarter=1; 1

    save path="/dept/sales/y1994q1" copies=1 fullpath; 2
run;
    deleterows / purge; 3
run;

    where; 4
    summary actual;
run;
```

### Program Description

1. Once the WHERE clause is specified, it applies to the statements that follow it. It also crosses RUN boundaries.
2. The SAVE statement is subject to the WHERE clause. As a result, the records from the Prdsale table that meet the WHERE clause are saved to /dept/sales/y1994q1.sashdat. The FULLPATH option is used to specify the table name instead of using the name of the active table. This is particularly useful when saving temporary tables.
3. The DELETEROWS statement is also subject to the WHERE clause. The records that were just saved to HDFS are now deleted and purged from memory. (The DELETEROWS statement without the PURGE option would mark the records for deletion and exclude them from being used in calculations, but it does not free the memory resources.)
4. The WHERE clause is cleared and the SUMMARY statement that follows is performed against all the remaining records in the Prdsale table.

This pattern of using a WHERE clause to subset an in-memory table, save the records to HDFS, and then delete them can be combined with the APPEND data set option of the SAS LASR Analytic Server engine. You can create a sliding window for keeping months or years of data in memory for analysis, yet keeping it up-to-date by appending the most recent records.

### Output

| Information from Saving Table SALES.PRDSALE | |
|---|---|
| Path | [illegible] |
| Number of Records | 180 |
| Block Size (kBytes) | 2.5 |

| Status of Records in Table SALES.PRDSALE | | | | |
|---|---|---|---|---|
| Total Number of Records | Records Marked Usable | Records Marked Delete | Records Marked Purge | Purged (This Action) |
| 1440 | 1260 | 0 | 180 | 180 |

## Example 5: Creating a Star Schema

### Details

The following example demonstrates using the SCHEMA statement to join dimension tables with a fact table.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';

proc imstat;
   table example.mailorder; 1
   schema catalog  (catCode=CatCode)
          products (pcode  =pcode  )
          customers(custnum=custnum); 2
run;

   table example.&_templast_ 3
run;
   columninfo;
quit;
```

### Program Description

1. Table Example.MailOrder is set as the active table. This table is the fact table for the star schema.
2. The SCHEMA statement joins the tables Catalog, Products, and Customers to the active table, MailOrder. The columns to use as keys for joining each table are enclosed in parenthesis.
3. The result of the SCHEMA statement is a temporary table or view. Use the &_TEMPLAST_ macro variable to refer to the star schema. If you want to persist the star schema, use the PROMOTE statement.

### Output

The following output shows the temporary table name and how the dimension table names are used as prefixes for the column names.

| Temporary Table Information for Table HPS.MAILORDER | |
|---|---|
| Statement | SCHEMA |
| Temporary Table | _T_BAD4C8F8_7F10C99A4F48 |
| Table Type | SCHEMA |

| Column Information for Table _T_BAD4C8F8_7F10C99A4F48 | | | | | |
|---|---|---|---|---|---|
| Id | Column | Type | Length | Format | Label |
| 1 | CUSTNUM | Num | 8 | BEST12. | |
| 2 | INV | Num | 8 | BEST12. | |
| 3 | Date | Num | 8 | DATE9. | |
| 4 | PCODE | Char | 6 | $CHAR6. | |
| 5 | QTY | Num | 8 | BEST12. | |
| 6 | Qtr | Num | 8 | BEST12. | |
| 7 | Year | Num | 8 | BEST12. | |
| 8 | Month | Num | 8 | BEST12. | |
| 9 | catCode | Char | 6 | $6. | |
| 10 | catalog_CATALOG | Char | 20 | $20. | Catalog |
| 11 | products_TYPE | Char | 15 | $F15. | TYPE |
| 12 | products_DESCRIP | Char | 30 | $F30. | DESCRIP |
| 13 | products_PRICE | Num | 8 | DOLLAR8.2 | RCOST |
| 14 | products_COST | Num | 8 | DOLLAR8.2 | WCOST |
| 15 | customers_NAME | Char | 32 | $32. | NAME |
| 16 | customers_ADDR1 | Char | 32 | $32. | |
| 17 | customers_ADDR2 | Char | 32 | $32. | |
| 18 | customers_CITY | Char | 20 | $20. | CITY |
| 19 | customers_STATE | Char | 3 | $3. | STATE |
| 20 | customers_ZIP | Char | 10 | $10. | |
| 21 | customers_PHONE | Char | 18 | $18. | |
| 22 | customers_REGION | Char | 10 | $10. | |

## Example 6: Appending Tables

### Details

The following example demonstrates using the SET statement to append tables with the active table.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';
libname hdfs sashdat host="grid001.example.com" install="/opt/TKGrid" path="/hps"; 1
```

```
proc lasr add data=hdfs.january port=10010; 2
    performance host="grid001.example.com" nodes=all;
run;

proc lasr add data=hdfs.february port=10010;  3
    performance host="grid001.example.com" nodes=all;
run;

data example.march;  4
    set otherlib.march;
run;

proc imstat;
    table example.january; 5
    set february / drop;  6
    set march;  7
run;

   save path="/hps/qtr1" copies=1 replace fullpath;  8
quit;
```

### Program Description

1. The value for the TAG= option in the SAS LASR Analytic Server LIBNAME statement matches the PATH= value for the SASHDAT engine LIBNAME statement.
2. The first table, January, is loaded to memory from HDFS.
3. The second table, February, is loaded to memory from HDFS. The tables are still independent in-memory tables.
4. The third table, March, is loaded from another library into the server with the SAS LASR Analytic Server engine.
5. The first table, January, is set as the active table.
6. The second table, February, is appended to the active table. The DROP option specifies to remove the February table from memory as soon as the SET statement completes.
7. The third table, March, is appended to the active table. This table remains in memory.
8. The February and March tables are now appended to the active table, January. The SAVE statement saves the table to HDFS with the name Qtr1.

## Example 7: Appending a Non-Partitioned Table to a Partitioned Table

### Details

The following example demonstrates how to append a table that is not partitioned to an in-memory table that is partitioned. The SET statement is used to append the table.

*Note:* As an alternative, if the table to append is not already in memory, you can append the rows to the partitioned in-memory table with the SAS LASR Analytic Server engine. For more information, see "APPEND= Data Set Option" on page 429.

### Program

```
libname example sasiola host="grid001.example.com" port=10010 tag='hps';
libname hdfs sashdat host="grid001.example.com" install="/opt/TKGrid" path="/hps"; 1

proc lasr add data=hdfs.transactions(partition=(customerid)) port=10010; 2
    performance host="grid001.example.com" nodes=all;
run;

proc lasr add data=hdfs.recenttrans(partition=(dateid)) port=10010;  3
    performance host="grid001.example.com" nodes=all;
run;

proc imstat;
    table example.recenttrans; 4
    partition customerid;
run;
    table example.transactions;  5
    set &_templast_ / drop;  6
quit;
```

### Program Description

1. The value for the TAG= option in the SAS LASR Analytic Server LIBNAME statement matches the PATH= value for the SASHDAT engine LIBNAME statement.
2. The first table, Transactions, is loaded to memory from HDFS. The table is partitioned by values of the CustomerId variable.
3. The second table, RecentTrans, is loaded to memory from HDFS. The table is partitioned by values of the DateId variable.
4. The second table, RecentTrans, is set as the active table and then partitioned into a temporary table with the PARTITION statement. The temporary table is partitioned by values of the CustomerId variable.
5. The first table, Transactions, is set as the active table.
6. The temporary table is appended to the active table. he DROP option specifies to remove the temporary table from memory as soon as the SET statement completes.

## Example 8: Storing Temporary Variables

### Details

Many statements offer a SAVE= option that is used to save the result table of the statement for use in other IMSTAT procedure statements. You can use the STORE statement to assign a value from the saved result table to a macro variable.

### Program

```
libname example sasiola host="grid001.unx.sas.com" port=10010 tag='hps';

data example.prdsale(partition=(country region)); set sashelp.prdsale; run;

proc imstat data=example.prdsale immediate noprint; 1
```

```
    percentile actual / partition="U.S.A.", "EAST" save=tab1; 2
run;
    percentile actual / partition="CANADA", "EAST" save=tab2;
run;

    store tab1(3,5) = us_SeventyFivePct; 3
run;
    store tab2(3,5) = ca_SeventyFivePct;
run;
%put %sysevalf(&us_SeventyFivePct - &ca_SeventyFivePct);

    replay tab2; 4
run;

    free tab1 tab2; 5
    free macro=us_SeventyFivePct;
    free macro=ca_SeventyFivePct;
quit;
```

### Program Description

1. The NOPRINT option suppresses displaying the results tables.
2. The results tables for the PERCENTILE statements are saved to temporary tables.
3. The STORE statements access the results tables and store the value from the fifth column in the third row (the 75th percentile) to macro variables.
4. The REPLAY statement displays the results table for the second PERCENTILE statement.
5. The FREE statements releases the memory used by results tables and the macro variables.

### Log Output

The SAS log describes how the values are stored to the macro variables.

```
     store tab1(3,5) = us_SeventyFivePct;
NOTE: The numeric value 746.5 from row 3, column 5 of table tab1 has been stored
      in the macro variable us_SeventyFivePct.
   run;

     store tab2(3,5) = ca_SeventyFivePct;
NOTE: The numeric value 759.5 from row 3, column 5 of table tab2 has been stored
      in the macro variable ca_SeventyFivePct.
```

*Chapter 6*

# IMXFER Procedure



## Overview: IMXFER Procedure

### *What Does the IMXFER Procedure Do?*

The IMXFER procedure is used to transfer in-memory tables between two distributed SAS LASR Analytic Server instances. The procedure takes advantage of network topology and parallelism as much as possible.

The IMXFER procedure cannot be used with a non-distributed SAS LASR Analytic Server.

## Syntax: IMXFER Procedure

**PROC IMXFER** <*options*>;
  **SERVER** *server-name* <HOST=*host-name*> <PORT=*number*>;
  **TABLE** *export-server-name export-table-name*
    *import-server-name* <*import-table-name*> <*/ options*>;
**QUIT**;

## PROC IMXFER Statement

Transfers an in-memory table.

### Syntax

**PROC IMXFER** <*options*>;

#### *Optional Arguments*

**HOSTONLY**
specifies to transfer the tables through the root nodes on the two clusters. With this option, the data are collected by the exporting root node before sending them to the importing root node. The importing root node then distributes the data to its worker nodes before passing the data to the importing server instance.

Specify this option if you know in advance that the worker nodes of the server instances do not have network communication with each other. (Even if you do not specify this option when this network topology exists, the procedure detects the lack of communication, and routes the data this way automatically.) Specify the option so that time is not lost trying to establish network connections between the clusters.

**Alias** NOWORKER

**IMMEDIATE**
specifies that the procedure executes one statement at a time rather than accumulating statements in RUN blocks.

**Alias** SINGLESTEP

**LASRERROR**
specifies that the procedure terminate when an error message is received from one of the servers.

If you do not specify this option, the IMXFER procedure attempts to continue interactive processing of programming statements. For example, if you receive an error that a table with the same name already exists in the importing server instance, you might prefer to change the name and continue rather than end the procedure.

**NOPRINT**
This option suppresses the generation of ODS tables and other printed output in the IMXFER procedure.

**NOTIMINGMSG**
When an action completes successfully, the IMXFER procedure generates a SAS log message that contains the execution time of the request. Specify this option to suppress the message.

**Alias** NOTIME

**TIMEOUT=*n***
specifies the time in seconds that the worker nodes of the exporting server waits for network connections. When this interval of time has passed, the data transfer occurs through the root nodes only.

**Alias** CONNTIME=

**Default** 30 seconds

## SERVER Statement

The SERVER statement is used to specify a server instance to use in a transfer. In the statement, you assign a logical name to the server and you use that name subsequently to refer to the particular server instance. There is no limit to the number of SERVER statements. You can establish connections to more than two servers with the IMXFER procedure.

### Syntax

**SERVER** *server-name* <HOST=*host-name*> <PORT=*number*>;

#### *Required Argument*

**server-name**
specifies the name to use for referring to the server instance. The name is used in the TABLE statement to identify the exporting server and the importing server.

#### *SERVER Statement Options*

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. If this option is not specified, then the GRIDHOST environment variable is used.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server.

**Default** 10616

## TABLE Statement

The TABLE statement is used to specify the table to export from one server and import to another server.

### Syntax

**TABLE** *export-server-name export-table-name import-server-name* <*import-table-name*> </ *options*>;

#### *Required Arguments*

***export-server-name***
specifies the name to use for the server instance that is exporting the table.

***export-table-name***
specifies the in-memory table to export. The name is specified as *server-tag.member-name*.

***import-server-name***
specifies the name to use for the server instance that is importing the table.

### Optional Argument

***import-table-name***
specifies the name to use for the imported table.

If you do not specify a name, then the IMXFER procedure attempts to create a table with the same name as the exported table. If a table with the same name already exists in the importing server, then the transfer fails.

If you specify a table name, prefix the table name with the tag that you want to use for accessing the table. For example, **`HPS`** and **`USER.SASDEMO`** can be used as tags for a table name. For more information, see "Understanding Server Tags" on page 416.

### TABLE Statement Options

**DELETED= INCLUDE | INC | EXCLUDE**
specifies how rows that are marked for deletion are handled in the transfer. By default, DELETED=EXCLUDE, which implies that any row that has a deletion mark is not transferred.

If you specify DELETED=INCLUDE, the IMXFER procedure instructs the server to ignore the deletion marks. Any rows that are marked for purging are not transferred, regardless of the DELETED= option.

**Default** EXCLUDE

**WHERE="*where-expression*"**
specifies the WHERE clause to apply to the exported table. Only rows that meet the conditions of the WHERE expression are transferred.

**Alias** FILTER=

**PARTITION= NO | REMOVE | YES**
specifies how to handle partitioning (and ordering within the partitions) when a partitioned table is transferred. By default, PARTITION=YES, and implies that a partitioned table is transferred to the importing server and remains partitioned and ordered by the same variables. When the servers have different numbers of worker nodes, there is no guarantee that partitions end up on the same nodes. However, it is guaranteed that partitions appear together on a node in the importing server.

Partitioning incurs some overhead and if you transfer a table from a smaller to a larger number of nodes, you might not want to apply partitioning. (Removing the partitioning spreads the data out more evenly in the importing server.) Or, you might not want to maintain partitioning on transfer if the transfer is for archival purposes. In that case, specify PARTITION=NO or PARTITION=REMOVE. This transfers the table to the importing server without the partitioning information.

**Default** YES

**PERM=*mode***
specifies the permission setting for accessing the imported table. The mode value is specified as an integer value such as 755. The mode corresponds to the mode values that are used for UNIX file access permissions.

For Windows servers, the UNIX mode setting is not applicable. Access is controlled according to permissions that you set manually on the signature files directory.

| | |
|---|---|
| **Alias** | PERMISSION= |
| **Range** | 600 to 777 |

## QUIT Statement

This statement terminates the IMXFER procedure. When the QUIT statement is reached, all resources allocated by the procedure are released and all connections to servers are terminated.

**Interaction:** Using a DATA step or another procedure step is equivalent to issuing a QUIT statement. If there is an error during the procedure execution, it is also equivalent to issuing a QUIT statement.

### Syntax

**QUIT**;

# Examples: IMXFER Procedure

## Example 1: Copying Tables from One Server to Another

### Details

It might be necessary to copy tables from one SAS LASR Analytic Server instance to another.

### Program

```
proc imxfer;
    server s1 host="grid001.example.com" port=10031; 1
    server s2 host="grid001.example.com" port=10010;

    table s1 public.fact_table s2; 2
quit;
```

### Program Description

1. The first SERVER statement creates a reference to the server that is listening on port 10031. The second SERVER statement creates a reference to a server on the same host, but listening on port 10010.
2. The TABLE statement transfers the Hps.Fact_Table table from the server that is listening on port 10031 to the server that is listening on port 10010. Because no

*import-table-name* is specified, the table uses the name Hps.Fact_Table on the importing server.

## Example 2: Copying Tables from One Cluster to Another

### Details

It might be necessary to copy tables from one SAS LASR Analytic Server instance on one cluster to a server that is running on a different cluster. The clusters can have different numbers of machines.

When the number of machines is not the same, the IMXFER procedure automatically redistributes the rows of the table to provide the most even distribution possible. In most cases, equalizing the data distribution equalizes the work load and provides the best performance. By default, partitioned tables remain partitioned on the importing server. For more information, see the PARTITION= on page 378 option for the TABLE statement.

### Program

```
proc imxfer;
    server s1 host="grid001.example.com" port=10031; 1
    server s2 host="cluster2.example.com" port=10010;

    table s1 public.inventory s2 hps.inventory; 2
quit;


/* access the transferred table */
libname cluster2 sasiola host="cluster2.example.com" port=10010 tag="hps"; 3

proc imstat;
    table cluster2.inventory;
run;

    distributioninfo; 4
quit;
```

### Program Description

1. The first SERVER statement creates a reference to the server that is listening on port 10031. The second SERVER statement creates a reference to a server on a different cluster that is listening on port 10010.
2. The TABLE statement transfers the Public.Inventory table from the server that is listening on port 10031 to the server on the other cluster. The table is renamed to Hps.Inventory on the importing server.
3. To access the transferred table, the LIBNAME statement must use the value "hps" as the server tag.
4. The DISTRIBUTIONINFO statement for the IMSTAT procedure displays the number of rows that are used on each machine in the second cluster.

*Chapter 7*

# RECOMMEND Procedure



## Overview: RECOMMEND Procedure

### *Purpose of the RECOMMEND Procedure*

PROC RECOMMEND is an interactive procedure that executes the statements within RUN blocks, similar to the IMSTAT procedure. All tasks in a SAS LASR Analytic Server-based recommender system can be performed from the RECOMMEND procedure.

You use the RECOMMEND procedure to perform the following tasks:

- start a recommender system in a SAS LASR Analytic Server
- connect to an existing recommender system
- interact with one or more recommender systems in the same SAS LASR Analytic Server
- filter data using a WHERE clause
- remove a recommender system from a SAS LASR Analytic Server
- populate a recommender system with content-based data for use with the cluster-based method or with rating data for collaborative filtering

- add methods to a recommender system
- remove methods from a recommender system
- define and optimize ensemble methods
- hold a subset of the data to use for model validation
- obtain recommendations for users

### *Working with Recommender Systems*

Identify a recommender system in the SAS LASR Analytic Server by a two-level name, similar to a LIBNAME.MEMBER construct. The name, which is case-insensitive, must be unique among the recommender systems in the SAS LASR Analytic Server. It is possible for an in-memory table and a recommender system to have the same name, because the table and application name spaces are separate. However, as a best practice, avoid using the same names to prevent confusion for users.

You can specify the recommender system that you want to work with through options in the PROC RECOMMEND statement or through options on other statements. When you specify the recommender system in the PROC RECOMMEND statement, the procedure uses that system as the default during execution. You can explicitly call other recommender systems in statements within the RECOMMEND procedure. If no application name is specified, the default recommender system is RECOM.SYSTEM.

You add tables to a recommender system to identify the user content, item content, and ratings. Typically, you load a user-item-ratings table into the recommender system. The SAS LASR Analytic Server derives a number of internal tables from the tables that you provide. The subsequent operations of the recommender system use the internal tables only. Make sure the SAS LASR Analytic Server has sufficient resources to build the recommender system and its associated tables. You can drop the user-item-ratings table after the recommender system has been set up.

Access to a recommender system is governed by signature files in the same way that those files are used to control access to SAS LASR Analytic Server tables. However, a recommender system contains multiple derived tables. For example, granting Read access to a recommender system implies Read access to all associated tables for that system.

## Syntax: RECOMMEND Procedure

**PROC RECOMMEND** *<option(s)>*;

**ADD** *recommender-system <| option(s)>*;

**ADDTABLE** *table <| option(s)>*;

**INFO** *<| option(s)>*;

**METHOD** *method-name <| option(s)>*;

**PREDICT** / USERLIST=("*userID1*" <, "*userID2*" ...">) |
USERDATA=*libref.member-name <option(s)>*;

**REMOVE** *recommender-system <| option(s)>*;

| Statement | Task | Example |
|---|---|---|
| ADD | Add a recommender system. | Ex. 1, Ex. 2 |
| ADDTABLE | Add a table. | Ex. 1, Ex. 2 |
| INFO | Requests information. | Ex. 1 |
| METHOD | Add a method. | Ex. 1, Ex. 2 |
| PREDICT | Request recommendations. | Ex. 1, Ex. 2 |
| REMOVE | Remove a recommender system. | Ex. 1, Ex. 2 |

## PROC RECOMMEND Statement

Invokes the RECOMMEND procedure.

### Syntax

**PROC RECOMMEND** <*option(s)*>;

#### *Optional Arguments*

**HOST=*host-name***
specifies the host name of the SAS LASR Analytic Server.

**Default** GRIDHOST value or the machine name of the SAS session if GRIDHOST is not specified.

**IMMEDIATE**
specifies that the RECOMMEND procedure executes each statement individually rather than running all the procedure statements in a single block at the end of the procedure code.

**NOPRINT**
suppresses printed output and SAS ODS tables.

**PORT=*port-number***
specifies the number of the port that the SAS LASR Analytic Server uses to listen for requests.

**Default** LASRPORT macro variable if that variable is set

**SIGNER="*authorization-web-service-uri*"**
specifies the URI for the SAS LASR Authorization web service. The web service is provided by the SAS Visual Analytics software. For more information, see *SAS Visual Analytics: Administration Guide*.

**Example** SIGNER="https://server.example.com/SASLASRAuthorization"

**SYSTEM=*recommender-system***
specifies the name of the recommender system in the SAS LASR Analytic Server that the procedure works with. Specify a two-level name, similar to a LIBNAME.MEMBER construct.

To work with an existing recommender system, the name that you specify identifies the application in the SAS LASR Analytic Server. When you create a new recommender system, the specified value becomes the name of that new system.

**Alias** RECOM=

**Default** RECOM.SYSTEM

## ADD Statement

The ADD statement is used to add a recommender system to the SAS LASR Analytic Server. If you do not provide a system name, the recommender system for the PROC RECOMMEND statement is used.

**Requirement:** The recommender system that you specify must not already exist in the SAS LASR Analytic Server.

### Syntax

**ADD** *<recommender-system> </ option(s)>*;

#### *Optional Argument*

***recommender-system***
specifies the name of the recommender system to create.

**Default** Value provided for the SYSTEM= or RECOM= option in the PROC RECOMMEND statement. If no value is specified, the default value RECOM.SYSTEM is used.

#### *ADD Statement Options*

You can specify the following optional arguments after the slash (/) in the ADD statement. These options apply to the recommender system that is used by the ADD statement.

**DATAFILTER=*"expression"***
specifies an optional WHERE clause for the recommender system. All of the data is filtered by this WHERE clause.

**DESCENDING=*variable-name***
**DESCENDING=(*variable-list*)**
specifies which variables of the ORDERBY= list are used with descending sort order. Specifying the DESCENDING= option by itself has no effect. The option is specified in addition to the ORDERBY= option.

**Alias** DESC=

**ITEM=*column-name***
specifies the name of the column that contains item identification in the in-memory tables.

**LABEL=*'string'***
specifies a label that you can use to identify the recommender system. The label is returned in output from the SAS LASR Analytic Server.

**ORDERBY=(*variable-list*)**
specifies one or more variables to use for sorting ratings for a user or for an item in the derived tables. For example, you can specify a timestamp variable to arrange ratings in chronological order for each user or for each item. Separate multiple variables with a space.

**PERM=*mode***
specifies the permission setting for accessing the recommender system. The mode value is expressed as an integer, such as 755. The mode corresponds to the mode values that are used for UNIX file access permissions.

For Windows servers, the UNIX mode setting is not applicable. Access is controlled according to permissions that you set manually on the signature files directory.

**Alias** PERMISSION=

**Default** Permissions are set according to the UNIX file access permissions for the SAS LASR Analytic Server process.

**RATING=*column-name***
specifies the name of the column that contains ratings in the in-memory tables.

**Alias** RATE=*column-name*

**Note** The specified column must have numeric values.

**USER=*column-name***
specifies the name of the column that contains user identification in the in-memory tables.

## ADDTABLE Statement

The ADDTABLE statement specifies a table for a recommender system that the SAS LASR Analytic Server uses to derive internal tables. The ADDTABLE statement is used most often to identify a table that contains user-item-ratings information.

### Syntax

**ADDTABLE** *libref.member-name* <*/ option(s)*>;

#### *Required Argument*

***table***
specifies a source table that the SAS LASR Analytic Server uses to generate internal tables. Specify the table name in Lasrlib.Name format, where the Lasrlib value is the libref of the SAS LASR Analytic Server on which the recommender system is defined.

**Requirement** The table that you specify must already be in memory.

### ADDTABLE Statement Options

You can specify the following optional arguments after the slash (/) in the ADDTABLE statement. These options apply to the recommender system that is used by the ADDTABLE statement.

**SYSTEM=*recommender-system***
specifies the name of the recommender system in the SAS LASR Analytic Server that the procedure works with. Specify a two-level name, similar to a LIBNAME.MEMBER construct.

To work with an existing recommender system, the name that you specify identifies the application in the SAS LASR Analytic Server. When you create a new recommender system, the specified value becomes the name of that new system.

**Alias** RECOM=

**Default** RECOM.SYSTEM

**TYPE=ITEM**
**TYPE=USER**
**TYPE=RATING**
specifies what type of information the table contains. A table of TYPE=ITEM contains the item column that was specified in the ADD statement and content information for the items. A table of TYPE=USER contains the user column that was specified in the ADD statement and content information for the users. A table of TYPE=RATING contains user-item-ratings information.

**Default** RATING

**VARS=(*variable-list*)**
specifies a list of one or more variables to transfer to the internal tables that the SAS LASR Analytic Server derives from the in-memory table. For example, if you are adding a table of item content, then only a subset of the variables are useful to the recommender system. List the useful variables in the VARS= option.

**Default** Transfer all variables

## INFO Statement

The INFO statement requests information about one or more recommender applications in the SAS LASR Analytic Server.

### Syntax

**INFO** </ *option(s)*>;

### Optional Arguments

You can supply the following options in the INFO statement after the slash (/):

**ALL**
requests a list of all recommender systems in the SAS LASR Analytic Server. When you use the ALL option, an additional RECOM= or SYSTEM= option is ignored.

**METHODS=(*"method-name1" <, "method-name2" ...>*)**
requests information about one or more methods that are registered with the recommender system. Specify each method with a quoted string, and separate multiple methods with commas.

**SYSTEM=*recommender-system***
specifies the name of the recommender system in the SAS LASR Analytic Server that the procedure works with. Specify a two-level name, similar to a LIBNAME.MEMBER construct.

To work with an existing recommender system, the name that you specify identifies the application in the SAS LASR Analytic Server. When you create a new recommender system, the specified value becomes the name of that new system.

| | |
|---|---|
| **Alias** | RECOM= |
| **Default** | RECOM.SYSTEM |

## METHOD Statement

The METHOD statement adds a method for computing recommendations for a recommender system. You can use the METHOD statement to specify details for a method definition, rather than using the default settings for that method.

### Syntax

**METHOD** *method-name* <*/ option(s)*>;

#### *Required Argument*

***method-name***
specifies the name of the method to add to a recommender system. The method name can be one of the following values:

| | |
|---|---|
| AVERAGE \| AVE \| AVG | a default method that is used to produce recommendations for users that have insufficient information in the recommender system. For example, a method might require that at least two ratings are on record for a user. If that is not the case, a request for a recommendation is provided with the AVG method. |
| SLOPEONE \| SLOPE1 | a simple regression-based method. |
| NEAREST \| KNN | a *k*-nearest-neighbor method that is based on measures of association between items or users. This method is also called a collaborative filter. |
| SVD | a recommender method that is based on a singular-value decomposition of a user-item-ratings matrix. |
| ENSEMBLE | a collection of other methods that you specify. |
| ARM | a method that performs associative rule mining (ARM). |
| CLUSTER | a cluster-based method that uses item or user profiles. Items or users are clustered. Then the similarity information |

between items or users for each cluster is computed to make recommendations.

Any method that has not already been defined for the SAS LASR Analytic Server, except for the CLUSTER and ARM methods, is created when a prediction request is processed. For example, suppose that you request a prediction with the *k*-nearest-neighbor method, and that method has not been added to the recommender system. The SAS LASR Analytic Server creates that method with default parameters and adds the method to the recommender system at that time.

Use the METHOD statement to add a method with explicitly defined parameters if you do not want to use the defaults. You must use the METHOD statement to add either the ARM or CLUSTER method, because those methods cannot be created using default values. For more information, see "ARM Statement" on page 78 or "CLUSTER Statement" on page 100.

### Optional Arguments

**DATAFILTER=*"expression"***
specifies an optional WHERE clause for each method. All of the data is filtered by this WHERE clause.

**DETAILS**
requests that additional details are provided for the numerically intensive SVD and ensemble methods.

**LABEL='*string*'**
specifies a label by which the method can be identified. A label is important if you have multiple instances of a method definition (with different parameter values) in the recommender system.

**FCONV=*r***
specifies a relative function convergence criterion for the numerical optimization in SVD and ensemble methods.

**GCONV=*r***
specifies a relative gradient convergence criterion for the numerical optimization in SVD and ensemble methods.

**MAXITER=*n***
specifies the maximum number of iterations for the numerical optimization in SVD and ensemble methods.

**Default** 1 (a one-step update)

**MAXFEVAL=*n***
specifies the maximum number of function evaluations for the numerical optimization in SVD and ensemble methods.

**SEED=*n***
specifies the seed for random number generation in SVD and ensemble methods.

**SYSTEM=*recommender-system***
specifies the name of the recommender system in the SAS LASR Analytic Server that the procedure works with. Specify a two-level name, similar to a LIBNAME.MEMBER construct.

To work with an existing recommender system, the name that you specify identifies the application in the SAS LASR Analytic Server. When you create a new recommender system, the specified value becomes the name of that new system.

**Alias** RECOM=

**Default** RECOM.SYSTEM

### *Options for the SLOPEONE Method*

**HOLD=*n***
specifies the number of ratings to hold for users that are selected by the WITHHOLD= option. The specified number of ratings are selected at random to be held in a validation data set, which is a subset of the original data set.

Typically, you specify a positive number for the HOLD= option. However, you can specify a negative number, which indicates that all ratings should be held in the validation data set except for the specified number of ratings. For example, HOLD=–2 means that all ratings but two should be held in the validation data set.

**Default** 1

**Interaction** The HOLD= option is ignored if the WITHHOLD= option is not also specified.

**WITHHOLD=*r***
specifies a relative percentage of users whose ratings are included in a validation data set, which is a subset of the original data set. For example, WITHHOLD=0.1 indicates that 10% of users should be selected at random. A portion of the selected users' ratings are held in the validation data set. The number of ratings to select is specified by the HOLD= option.

**Range** 0–1, exclusive

### *Options for the KNN Method*

**HOLD=*n***
specifies the number of ratings to hold for users that are selected by the WITHHOLD= option. The specified number of ratings are selected at random to be held in a validation data set, which is a subset of the original data set.

Typically, you specify a positive number for the HOLD= option. However, you can specify a negative number, which indicates that all ratings should be held in the validation data set except for the specified number of ratings. For example, HOLD=–2 means that all ratings but two should be held in the validation data set.

**Default** 1

**Interaction** The HOLD= option is ignored if the WITHHOLD= option is not also specified.

**NEAREST=*k***
specifies the parameter *k* for a *k*-nearest-neighbor method. Only the *k* nearest neighbors are considered in deriving a recommendation for a particular user.

**Alias** K=

**NONNEGATIVE**
requests that only positive associations are used when computing a neighborhood in a *k*-nearest-neighbor method.

**Alias** POSITIVE

**PREFILTER=NONE**
**PREFILTER=TOP(*n*)**
**PREFILTER=THRESHOLD(*r*)**
specifies the type of prefiltering to apply when computing a neighborhood. If you specify PREFILTER=TOP(*n*), then a list of only the *n* nearest neighbors and their similarities are kept. If you specify PREFILTER=THRESHOLD(*r*), then the list of nearest neighbors includes items or users with similarities that exceed the threshold value *r*. If you specify PREFILTER=NONE, then neighborhoods are formed based on all similarities.

**Default** TOP(10)

**SIMILARITY=COSINE | COS | CV**
**SIMILARITY=CORR | PEARSON | PC**
**SIMILARITY=ADJCOS | AC**
specifies the similarity measure that is used in *k*-nearest-neighbor collaborative filtering. If you specify SIMILARITY=COSINE (or COS or CV), then the cosine measure is the similarity measure. If you specify SIMILARITY=CORR (or PEARSON or PC), then the Pearson's correlation coefficient, or product-moment correlation, is the similarity measure. If you specify SIMILARITY=ADJCOS (orAC), then the adjusted cosine measure is the similarity measure. For more information, see "How Similarity Measures Are Calculated" on page 394.

**WITHHOLD=*r***
specifies a relative percentage of users whose ratings are included in a validation data set, which is a subset of the original data set. For example, WITHHOLD=0.1 indicates that 10% of users should be selected at random. A portion of the selected users' ratings are held in the validation data set. The number of ratings to select is specified by the HOLD= option.

**Range** 0–1, exclusive

### *Options for the SVD Method*

**BINARY=*n***
specifies a rule to generate a binary rating. If a numeric rating exceeds *n*, then the binary rating is set to 1. Otherwise, the binary rating is set to 0.

**BINALPHA=*m***
specifies a weighting factor for the squared errors in the loss function of the matrix factorization.

**FACTORS=*n***
specifies the number of features for the user-item matrix. Values between 50 and 100 are typical. A value as low as 10 is useful for evaluating this option. Larger values increase the computational complexity and require more time to run.

**HOLD=*n***
specifies the number of ratings to hold for users that are selected by the WITHHOLD= option. The specified number of ratings are selected at random to be held in a validation data set, which is a subset of the original data set.

Typically, you specify a positive number for the HOLD= option. However, you can specify a negative number, which indicates that all ratings should be held in the validation data set except for the specified number of ratings. For example, HOLD=–2 means that all ratings but two should be held in the validation data set.

**Default** 1

**Interaction** The HOLD= option is ignored if the WITHHOLD= option is not also specified.

**LOSS=SE**
**LOSS=SEREG**
**LOSS=SEWREG**
**LOSS=KL | ENTROPY**
specifies the loss function for the matrix factorization. The LOSS=SE option indicates that the squared-error function is the loss function. The LOSS=SEREG and LOSS=SEWREG options are modifications of the squared-error loss function that include regularization terms in matrix norms or weighted matrix norms, respectively. Weighted regularization terms are weighted by $\lambda$, and you set the value of this parameter with the LAMBDA= option. The LOSS=KL (or ENTROPY) option indicates that the Kullback-Leibler divergence, or relative entropy, is the loss function.

**LAMBDA=*λ***
specifies the regularization factor for the loss functions.

**Applies to** LOSS=SEREG or LOSS=SEWREG

**TECHNIQUE=LBFGS**
**TECHNIQUE=ALS**
specifies the optimization method for the singular-value decomposition. The TECHNIQUE=LBFGS option indicates a limited-memory Broyden-Fletcher-Goldfarb-Shanno (BFGS) optimization method. This method is often used for solving neural network problems. The TECHNIQUE=ALS option indicates an alternating least squares optimization method.

**WITHHOLD=*r***
specifies a relative percentage of users whose ratings are included in a validation data set, which is a subset of the original data set. For example, WITHHOLD=0.1 indicates that 10% of users should be selected at random. A portion of the selected users' ratings are held in the validation data set. The number of ratings to select is specified by the HOLD= option.

**Range** 0–1, exclusive

### *Options for Ensemble Method*

**CONSTRAINT**
restricts the weights in the ensemble to lie between 0 and 1.

**HOLD=*n***
specifies the number of ratings to hold for users that are selected by the WITHHOLD= option. The specified number of ratings are selected at random to be held in a validation data set, which is a subset of the original data set.

Typically, you specify a positive number for the HOLD= option. However, you can specify a negative number, which indicates that all ratings should be held in the validation data set except for the specified number of ratings. For example, HOLD=–2 means that all ratings but two should be held in the validation data set.

**Default** 1

**Interaction** The HOLD= option is ignored if the WITHHOLD= option is not also specified.

**METHODS=(*"method1", "method2" <,"method3" ...>*)**
specifies the methods that participate in the ensemble. Enclose each method in quotation marks, and separate multiple values with a comma.

**Default** All methods except the AVERAGE method.

**Restriction** The AVERAGE method is not part of any ensemble.

**WITHHOLD=*r***
specifies a relative percentage of users whose ratings are included in a validation data set, which is a subset of the original data set. For example, WITHHOLD=0.1 indicates that 10% of users should be selected at random. A portion of the selected users' ratings are held in the validation data set. The number of ratings to select is specified by the HOLD= option.

**Range** 0–1, exclusive

### *Options for the Cluster Method*

**BUBMAXPTS=*n***
specifies the maximum number of points in each bubble. This number must exceed the value of the BUBMINPTS= option.

**BUBMINPTS=*n***
specifies the minimum number of points in each bubble.

**Default** 1

**CLUSTINFO**
generates the temporary table that contains the cluster results for each user or item.

**CLUSTVARS=(*variable-list*)**
lists the variables to use with the CLUSTER method.

**CLUSTERTECH=KMEANS**
**CLUSTERTECH=DBSCAN**
specifies the clustering technique.

**Default** KMEANS

**CONV=*c***
specifies the convergence criterion *c* for the *k*-means analysis. When the relative change in WCSS between successive iterations is less than *c*, the analysis is presumed to have converged.

**Default** 0.00001

**DIST=EUC | SQUAREDEUC | MANHATTAN | MAXIMUM | COSINE | JACCARD | HAMMING**
specifies the distance measure that is used in the clustering method. The *k*-means method uses DIST=EUC.

**Applies to** CLUSTERTECH=DBSCAN

**DMAX=*v***
specifies the maximum diameter of bubbles with the given distance measure.

**Default** 0

**EPS=*r***
specifies the distance value for neighborhood querying. For more information, see "CLUSTER Statement" on page 100.

**Applies to** CLUSTERTECH=DBSCAN

**INITMETHOD=FORGY | RAND | AVG**
specifies the method for obtaining the initial estimate of cluster assignment. For more information, see "CLUSTER Statement" on page 100.

**Alias** INIT=

**MINPTS=*n***
specifies the minimum number of points that are required in one cluster.

**Applies to** CLUSTERTECH=DBSCAN

**NOCASE**
specifies that the comparisons between terms and the values of character variables are case insensitive. By default, comparisons are case-sensitive.

**NOIDF**
specifies that only the term frequency is used to construct the vectors and that inverse document frequency is not used.

**NONORM**
specifies that the TF-IDF vectors are not normalized.

**NREP=*k***
specifies the number of representative points for each bubble.

**Default** 1

**NUMCLUSTERS=*k***
specifies the number of clusters for the *k*-means analysis.

**Alias** NUMCLUS=

**Default** 2

**SAVETERMS**
saves the TF-IDF vectors in the temporary table when the CLUSTINFO option is enabled.

**TERMS=(*"term1" <, "term2"...>*)**
specifies terms that are used to compute term frequency. Each string represents one term. For more information, see "CLUSTER Statement" on page 100.

**TERMDATA=*table-name***
specifies an in-memory table in the server that contains the term list. For more information, see "CLUSTER Statement" on page 100.

**TOKENS=(*"token1" <, "token2" ...>*)**
specifies the tokens that separate terms when scanning character variables. For more information, see "CLUSTER Statement" on page 100.

**TOKENDATA=*table-name***
specifies an in-memory table in the server that contains the tokens list.

**TYPE=ITEM | USER**
specifies which type of profile is used for the CLUSTER method. The CLUSTER method that uses a user profile table cannot be used in the ensemble model with other methods.

**Requirement** The user or item table must be added into the recommender system.

## Details

### *How Similarity Measures Are Calculated*

Similarity measures are used to determine the $k$ nearest neighbors of an item or a user for the KNN method. You can select cosine, adjusted cosine, or Pearson's correlation coefficient to measure the similarity between items or users.

Suppose that $r_{ui}$ is the rating of user $u$ for item $i$. Then the user-based similarity measures between users $u$ and $v$ are computed as follows:

$$Cosine(u, v) = \frac{\sum_{i \in I_{uv}}^{n} r_{ui} r_{vi}}{\sqrt{\sum_{i \in I_u} r_{ui}^2 \sum_{i \in I_v} r_{vi}^2}}$$

$$Corr(u, v) = \frac{\sum_{i \in I_{uv}} \left(r_{ui} - \bar{r}_{u\bullet}\right)\left(r_{vi} - \bar{r}_{v\bullet}\right)}{\sqrt{\sum_{i \in I_{uv}} \left(r_{ui} - \bar{r}_{u\bullet}\right)^2 \sum_{i \in I_{uv}} \left(r_{vi} - \bar{r}_{v\bullet}\right)^2}}$$

In these expressions, $I_{uv}$ denotes the set of items that have been rated by user $u$ and user $v$. The value $\bar{r}_{u\bullet}$ is the average rating by user $u$ across all items that she rated.

In an item-based recommender system, the similarity measures associate ratings for items across the set of users who rated items $i$ and $j$. Denote this set as $U_{ij}$. The Pearson correlation measure between items $i$ and $j$ are calculated as follows:

$$\mathrm{Corr}(i, j) = \frac{\sum_{u \in U_{ij}} \left(r_{ui} - \bar{r}_{\bullet i}\right)\left(r_{uj} - \bar{r}_{\bullet j}\right)}{\sqrt{\sum_{u \in U_{ij}} \left(r_{ui} - \bar{r}_{\bullet i}\right)^2 \sum_{u \in U_{ij}} \left(r_{uj} - \bar{r}_{\bullet j}\right)^2}}$$

$$AC(i, j) = \frac{\sum_{u \in U_{ij}} \left(r_{ui} - \bar{r}_{u\bullet}\right)\left(r_{uj} - \bar{r}_{u\bullet}\right)}{\sqrt{\sum_{u \in U_{ij}} \left(r_{ui} - \bar{r}_{u\bullet}\right)^2 \sum_{u \in U_{ij}} \left(r_{uj} - \bar{r}_{u\bullet}\right)^2}}$$

The equations for item similarity measures reflect that the differences in rating scales among users are typically more pronounced than the differences in ratings for individual items. That is, an item might get low scores from most users, but the ranges of scores among users often vary widely.

# PREDICT Statement

The PREDICT statement generates recommendations (predictions) for one or more users.

## Syntax

**PREDICT** / USERLIST=(*"userID1" <,"userID2" ...>*) *<option(s)>*;

**PREDICT** / USERDATA=*libref.member-name* *<option(s)>*;

### *Optional Arguments*

You can specify the following options in the PREDICT statement after the slash (/):

**DATAFILTER="*expression*"**
specifies the optional filter (WHERE clause) for this recommendation. The recommended items are filtered by the expression.

**ITEMDATA=*table-name***
specifies the in-memory table that contains the new transaction data for items. This table can be used when you specify METHOD=ARM to recommend new items based on users' new activities on items.

**LABEL='*string*'**
specifies the label that is assigned to the desired method of computing recommendations.

**METHOD=AVERAGE | AVE | AVG**
**METHOD=NEAREST | KNN**
**METHOD=SLOPEONE | SLOPE1**
**METHOD=SVD**
**METHOD=ENSEMBLE**
**METHOD=ARM**
**METHOD=CLUSTER**
specifies the method to use for computing recommendations. If the requested method is not yet defined for the recommender system, then the SAS LASR Analytic Server adds the method with default parameters and computes the recommendation. For more information, see "METHOD Statement" on page 387.

**NRECOMM=*n***
specifies the upper limit for the number of recommendations that are returned per user.

| | |
|---|---|
| **Alias** | NUM= |
| **Default** | 10 |

**OUT=*SAS-data-set***
specifies a SAS data set that stores recommendations.

| | |
|---|---|
| **Alias** | OUTDATA= |

**SYSTEM=*recommender-system***
specifies the name of the recommender system in the SAS LASR Analytic Server that the procedure works with. Specify a two-level name, similar to a LIBNAME.MEMBER construct.

To work with an existing recommender system, the name that you specify identifies the application in the SAS LASR Analytic Server. When you create a new recommender system, the specified value becomes the name of that new system.

**Alias** RECOM=

**Default** RECOM.SYSTEM

**TEMPTABLE**
specifies to store the recommendations in a temporary table in the server.

**USERDATA=*libref.member-name***
specifies an in-memory table that contains the user IDs for which you want recommendations. The variable with the user IDs must be named **`userid`**.

**USERLIST=(*"userID1" <,"userID2" ...>*)**
specifies the user IDs for which you want recommendations. This is a convenient format if you want recommendations for only a small number of users.

**Alias** USERS=

**Interaction** If a value is also specified for USERDATA, then only the USERDATA table is used.

## REMOVE Statement

Removes a recommender system from the SAS LASR Analytic Server, or removes a method from a recommender system.

### Syntax

**REMOVE** *<recommender-system> </ option(s)>*;

#### *Optional Argument*

***recommender-system***
specifies the name of the recommender system to remove from the SAS LASR Analytic Server. To remove the currently active recommender system, issue the REMOVE statement with no additional arguments or options.

**Default** Value provided for RECOM= or SYSTEM= in the PROC RECOMMEND statement. If no value was specified, the default value RECOM.SYSTEM is used.

#### *Options for the REMOVE Statement*

**METHOD=NEAREST | KNN**
**METHOD=SLOPEONE | SLOPE1**
**METHOD=SVD**
**METHOD=ENSEMBLE**
**METHOD=ARM**
**METHOD=CLUSTER**
specifies the method that you want to remove from a recommender system. If the method is included in an ensemble, then the ensemble is removed also.

**LABEL=*'string'***
specifies the label of the method to remove.

# Examples: RECOMMEND Procedure

## Example 1: Recommendations from Explicit Ratings

### Details

### Problem Description

This example draws on data that is derived from online movie viewing companies. A company wants to offer its customers recommendations of movies that they might like. These recommendations are based on ratings that are provided by users. The following table contains an example of a user-item-ratings matrix that online movie viewing companies might use.

***Table 7.1*** *Sample Movie Ratings from Customers*

| Customers | Movie 1 | Movie 2 | Movie 3 | Movie 4 | Movie 5 |
|---|---|---|---|---|---|
| User 1 | 4 | | 4 | | 4 |
| User 2 | | 1 | | 1 | |
| User 3 | 5 | 4 | | 5 | 4 |

In the sample data, customers rate the movies that they have seen using a 1–5 scale, where 1 is the lowest rating and 5 is the highest rating. This table represents a user-item-ratings matrix. Blank cells correspond to movies that a customer has not rated.

In practice, the matrix would be even more sparse than the sample data, as the typical customer rates only a very small fraction of all available movies. The goal of the recommender system is to predict ratings for all of the blank cells. Would User 1 like Movie 4? From the sample above, there is very little data available. However, with much more data, we might observe that User 1 and User 3 have similar taste in movies. Therefore, we could conclude that User 1 would also give Item 4 a high rating, based on the information that we already have from User 3. In a real-world situation, it is not necessary to predict every blank entry in a utility matrix. The system is required to supply only a few suggestions that a customer would rate highly.

### Example Data

This example uses the MovieLens data set (1M) that was developed by the GroupLens project at the University of Minnesota. The data that is displayed was downloaded in February 2014 from the GroupLens website.

*Note:* Per University of Minnesota guidelines, you cannot use this data for any commercial or revenue-bearing purpose without first obtaining permission from a faculty member of the GroupLens Research Project.

Before invoking the RECOMMEND procedure, we can print a part of the tables that are included in the recommender system by invoking the IMSTAT procedure.

```
proc imstat;
   table mylasr.movierating;
   fetch / format to=5;
run;

   table mylasr.movieprofile;
   fetch / format to=5;
run;

   table mylasr.userprofile;
   fetch / format to=5;
run;
quit;
```

The first FETCH statement prints a portion of the MovieRating table. The table contains rating information made by users (customers) about items (movies).

***Output 7.1*** *Sample Data from the MovieRatings Table*

| Selected Records from Table HPS.MOVIERATING | | | |
|---|---|---|---|
| **userID** | **itemID** | **rating** | **timeID** |
| 1 | 1193 | 5 | 978300760 |
| 1 | 661 | 3 | 978302109 |
| 1 | 914 | 3 | 978301968 |
| 1 | 3408 | 4 | 978300275 |
| 1 | 2355 | 5 | 978824291 |

The second and third FETCH statements print a portion of the MovieProfile and UserProfile tables. These tables contain information about each movie and user (customer), respectively.

***Output 7.2*** *Sample Profile Data*

| Selected Records from Table HPS.MOVIEPROFILE | | | |
|---|---|---|---|
| **itemID** | **title** | **genres** | **year** |
| 1 | Toy Story | Animation\|Children's\|Comedy | 1995 |
| 2 | Jumanji | Adventure\|Children's\|Fantasy | 1995 |
| 3 | Grumpier Old Men | Comedy\|Romance | 1995 |
| 4 | Waiting to Exhale | Comedy\|Drama | 1995 |
| 5 | Father of the Bride Part II | Comedy | 1995 |

| Selected Records from Table HPS.USERPROFILE | | | | |
|---|---|---|---|---|
| **userID** | **gender** | **age** | **occupation** | **zipcode** |
| 1 | F | 1 | 10 | 48067 |
| 2 | M | 56 | 16 | 70072 |
| 3 | M | 25 | 15 | 55117 |
| 4 | M | 45 | 7 | 02460 |
| 5 | M | 25 | 20 | 55455 |

### Program

```
proc recommend port=&portNumber recom = rs.movielens;
   add rs.movielens /item = itemid user = userid rating = rating; 1

   addtable mylasr.movierating  / recom = rs.movielens type = rating
                                  vars=(itemid userid rating); 2
   addtable mylasr.movieprofile  / recom = rs.movielens type = item;
   addtable mylasr.userprofile   / recom = rs.movielens type = user;
   run;

   method knn / label = "knn" k = 20 positive similarity = pc seed = 1234; 3
   run;

   method slope1 /label = "slope1";
   run;

   method svd / factors = 20 label = "svd" fconv = 1e-3 gconv = 1e-3
                maxiter = 100 seed = 1234 MAXFEVAL = 5000 function=L2
                lamda = 0.2 technique = lbfgs;
   run;

   method ensemble / methods =("svd","knn") label = "ensemble" details
                     MAXFEVAL=5000 maxiter=100 seed=1234 hold=2
                     withhold=0.1;
   run;

   predict / method = knn  label="knn" Num = 5
             users = ("1","33","478","2035"); 4
   run;

   info; 5
   run;

   remove rs.movielens; 6
run;
```

### Program Description

1. The ADD statement adds a recommender system, MovieLens, to the SAS LASR Analytic Server.
2. Three ADDTABLE statements add the MovieRating, MovieProfile, and UserProfile tables to the recommender system.

3. Each METHOD statement adds a method for computing recommendations to the recommender system. The methods KNN, SLOPE1, SVD, and ENSEMBLE were added with options specified for each method. For more information, see "METHOD Statement" on page 387.
4. The PREDICT statement generates five predictions for each specified user (1, 33, 478, and 2035).
5. The INFO statement requests information about all recommender systems on the SAS LASR Analytic Server. You can also specify a specific recommender system in the INFO statement to filter the results.
6. The REMOVE statement removes a recommender system from the server or removes a method.

### METHOD Statement Output

***Output 7.3*** *Output from the METHOD Statement Using the ENSEMBLE Option*

| Ensemble Optimal Coefficients for Recommender System RS.MOVIELENS | | | |
|---|---|---|---|
| Model ID | Method Type | Method Label | Coefficients |
| 0 | SVD | svd | 0.502078 |
| 1 | KNN | knn | 0.498489 |

***Output 7.4*** *Output from the METHOD Statement Using the Details Option*

| Loss Function Evaluation History for Recommender System RS.MOVIELENS | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of Evaluations | Loss Function | Root Mean Square Error | Mean Absolute Error | Root Mean Square Error for Holdout Sample | Mean Absolute Error for Holdout Sample | Precision for Holdout Sample | Recall for Holdout Sample | True Negative Rate for Holdout Sample | Accuracy for Holdout Sample |
| 0 | 1929347.43 | 1.389704 | 1.342099 | 1.425515 | 1.358544 | 0.569818 | 0.99848 | 0.098182 | 0.588576 |
| 1 | 1929347.43 | 1.389704 | 1.342099 | 1.425515 | 1.358544 | 0.569818 | 0.99848 | 0.098182 | 0.588576 |
| 2 | 1929347.43 | 1.389704 | 1.342099 | 1.425515 | 1.358544 | 0.569818 | 0.99848 | 0.098182 | 0.588576 |
| 3 | 1929347.43 | 1.389704 | 1.342099 | 1.425515 | 1.358544 | 0.569818 | 0.99848 | 0.098182 | 0.588576 |
| 4 | 1.21962E15 | 34940.49 | 219.5058 | 35715.73 | 222.2453 | . | 0 | 1 | 0.455298 |
| 5 | 1.21861E13 | 3492.886 | 69.40223 | 3570.38 | 70.26836 | . | 0 | 1 | 0.455298 |
| 6 | 1.21071E11 | 348.1259 | 21.91017 | 355.8457 | 22.18356 | . | 0 | 1 | 0.455298 |
| 7 | 1131419487 | 33.65339 | 6.811313 | 34.39585 | 6.896151 | . | 0 | 1 | 0.455298 |
| 8 | 5104630.6 | 2.260472 | 1.740098 | 2.307084 | 1.761131 | . | 0 | 1 | 0.455298 |
| 9 | 261062.243 | 0.511198 | 0.751175 | 0.525285 | 0.760341 | 0.780153 | 0.776596 | 0.738182 | 0.759106 |
| 10 | 261062.243 | 0.511198 | 0.751175 | 0.525285 | 0.760341 | 0.780153 | 0.776596 | 0.738182 | 0.759106 |
| 11 | 8.32875E10 | 288.74 | 19.95457 | 295.1857 | 20.20493 | 0.47947 | 0.560152 | 0.285455 | 0.429636 |
| 12 | 832579113 | 28.86887 | 6.309523 | 29.62376 | 6.389662 | 0.527704 | 0.607903 | 0.349091 | 0.490066 |
| 13 | 8528820.97 | 2.921874 | 1.993202 | 2.998835 | 2.021547 | 0.713306 | 0.790274 | 0.62 | 0.712748 |
| 14 | 338197.652 | 0.581838 | 0.80802 | 0.602849 | 0.823407 | 0.773273 | 0.782676 | 0.725455 | 0.756623 |

### PREDICT Statement Output

***Output 7.5*** *Output from the PREDICT Statement*

| Prediction from Recommender System RS.MOVIELENS | | | | | | |
|---|---|---|---|---|---|---|
| User | Rank | Rating | itemID | year | title | category |
| 1 | 1 | 5.3542 | 557.000000 | 1962.000000 | Mamma Roma | Drama |
| 1 | 2 | 5.0897 | 2503.000000 | 1998.000000 | Apple, The (Sib) | Drama |
| 1 | 3 | 5.0719 | 1178.000000 | 1957.000000 | Paths of Glory | Drama\|War |
| 1 | 4 | 5.0651 | 2360.000000 | 1998.000000 | Celebration, The (Festen) | Drama |
| 1 | 5 | 5.0172 | 3245.000000 | 1964.000000 | I Am Cuba (Soy Cuba/Ya Kuba) | Drama |
| 33 | 1 | 4.6101 | 2905.000000 | 1962.000000 | Sanjuro | Action\|Adventure |
| 33 | 2 | 4.5831 | 3897.000000 | 2000.000000 | Almost Famous | Comedy\|Drama |
| 33 | 3 | 4.5613 | 2503.000000 | 1998.000000 | Apple, The (Sib) | Drama |
| 33 | 4 | 4.5612 | 53.000000 | 1994.000000 | Lamerica | Drama |
| 33 | 5 | 4.5317 | 457.000000 | 1993.000000 | Fugitive, The | Action\|Thriller |

### INFO Statement Output

***Output 7.6*** *Output from the INFO Statement*

| Information for Recommender System RS.MOVIELENS | |
|---|---|
| Name | RS.MOVIELENS |
| Owner | sashli |
| Time Created | 19Nov2013:15:26:10 |
| Time Modified | 19Nov2013:15:48:22 |
| Item Variable | itemid |
| User Variable | userid |
| Rating Variable | rating |
| Rating Table (by item) | _T_ED56DB24_7FD07BCED270 |
| Number of ratings | 1000209 |
| Number of rated items | 3883 |
| Item Table | _T_ED56E3BC_7FD07A28E270 |
| Number of items | 3883 |
| User Table | _T_ED56EBD7_7FD07BCED270 |
| Number of users | 6040 |
| Rating Table (by user) | _T_ED5739F7_7FD07BCED220 |
| Number of ratings | 1000209 |
| Number of rating users | 6040 |
| Rating Sparsity | 0.957353202 |
| Number of Models | 5 |
| Number of KNN Models | 1 |
| Number of SlopeOne Models | 1 |
| Number of SVD Models | 1 |
| Number of Cluster Models | 1 |
| Number of Ensemble Models | 1 |

## Example 2: Recommendations from Implicit Information

### Details

In many situations, it is difficult to obtain explicit ratings that can be directly used to infer user preferences. Instead, we are provided with feedback such as purchase history,

browsing history, search patterns, or time spent on a website. This documentation presents two ways to infer user preferences from abundant implicit feedback to build a recommender system.

Let $U$ denote the number of users, and let $I$ denote the number of items. Let $f_u$ denote the frequency of user $u$ purchasing any item, and let $f_i$ denote the frequency of any user purchasing item $i$. Let $f_{ui}$ denote the frequency of item $i$ being purchased by user $u$. Similar to the term frequency-inverse document frequency weight ($tf - idf$) in text mining, there are two methods that we can use to convert frequency $f_{ui}$ to ratings $r_{ui}$:

$$r_{ui} = \log\left(f_{ui} + 1\right)\log\left(\frac{U}{f_i + 1}\right) \tag{1}$$

$$r_{ui} = \log\left(f_{ui} + 1\right)\log\left(\frac{I}{f_u + 1}\right) \tag{2}$$

Equation 1 places larger weight on items that are purchased less frequently. Equation 2 places larger weight on users who purchase items less frequently.

### Program: Transform a Transaction Table into a Rating Table

The following code provides an example of the syntax. The referenced table, salesfact, is not available for use. The table can be as simple as one column for a user ID and another column for the item ID.

```
proc imstat;

   table MYlasr.salesfact; 1
      tableinfo / save = tabinf;
      store tabinf(1,3) = nObs;
   run;
      distinct / save = dtab;
      store dtab(1,2) = numItem;
      store dtab(2,2) = numUser;
   run;

   table MYlasr.salesfact; 2
      compute joinkey "joinkey = &userID || &itemID;";
   run;

   table MYlasr.salesfact(tempnames=(t1));
      summary t1 / groupby=(&userID &itemID) temptable tn=t1 3
                      te="t1=1;" save=tabl;
      summary t1 / groupby=(&userID) temptable tn=t1
                      te="t1=1;" save=tab2;
      summary t1 / groupby=(&itemID) temptable tn=t1
                      te="t1=1;" save=tab3;
   run;
      store tab1(2,2) = freq_user_item; 4
      store tab2(2,2) = freq_user;
      store tab3(2,2) = freq_item;
   run;

   table MYlasr.&freq_user_item;
      schema &freq_user(&userID=&userID / prefix=UserTotal,_n_) 5
             &freq_item(&itemID=&itemID / prefix=ItemTotal,_n_) /
                mode=table;
```

```
run;

table MYlasr.&_templast_; 6
   compute joinkey "joinkey = &userID || &itemID;";
run;
table MYlasr.salesfact;
   schema &_templast_(joinkey=joinkey / prefix=r,_n_
                      UserTotal__N_ ItemTotal__N_);
run;
table MYlasr.&_templast_;
   compute Rating_iuf "Rating_iuf =
                       log10(r__N_+1)*log10(&nObs/(r_ItemTotal__n_+1));";
   run;
   compute Rating_iif "Rating_iif =
                       log10(r__N+1)*log10(&nObs/(r_UserTotal__n_+1));";
   run;
   compute Rating_simple "Rating_simple =
                          Round((r__N_/r_UserTotal__n_)*10+1,1);";
   run;

table MYlasr.&_templast_; 7
   save path="/hps/rating_tfidf" copies=1
        replace fullpath;
run;
```

### Program Description

1. The STORE statements assign the number of observations, the number of distinct users, and the number of distinct items to macro variables.
2. The COMPUTE statement adds a permanent column by concatenating the user ID and the item ID values.
3. The SUMMARY statements with the GROUPBY= option produce descriptive statistics in the temporary column (t1). The results of the SUMMARY statements are saved to temporary tables.
4. The STORE statements assign the names of the three temporary tables to three macro variables, Freq_User_Item, Freq_User, and Freq_Item.
5. The SCHEMA statement joins the star tables Freq_User_Item, Freq_User, and Freq_Item and creates a new temporary table.
6. Generate the ratings table. By using equations 1 and 2, as described in "Details" on page 402, convert the frequency counts into ratings.
7. The SAVE statement saves the rating table directly into HDFS as a SASHDAT table for future use.

### Program: Create a Recommender System with the Rating Table

The following code provides an example of the syntax. The referenced tables are not available for use.

```
proc recommend port=&lasrport recom = rs.DEPTSTORE;
   add rs.DEPTSTORE /item = item_sk user = household_sk 1
                    rating = &rating;

   addtable MYlasr.MBA_rating_tfidf  / recom = rs.DEPTSTORE 2
```

```
                 type = rating
                 vars=(item_sk household_sk &rating);
addtable MYlasr.household   / recom = rs.DEPTSTORE type = user;
addtable MYlasr.item        / recom = rs.DEPTSTORE type = item;
run;

method cluster / clusttech=kmeans numclus=50 dist=euc type=user 3
                  maxiter=10 label="clust" details
                  terms=("Convenience", "Occasional", "Unk")
                  tokens=(" ") noidf seed=1234 clustinfo
                  clustvars=(LIFESTYLE_SEGMENT FAMILY PET);
run;

predict / users=("11815911") method=cluster label="clust" 4
          Num = 5;
run;

method svd / factors = 20 label = "svd_1" fconv = 1e-3 gconv = 1e-3 5
          maxiter = 100  seed = 12314 MAXFEVAL = 5000
          function=L2 lamda = 0.2 technique = als;
run;

predict / method = svd label = "svd_1" Num = 5 6
          users = ("11815911");
run;

quit;
```

### Program Description

1. The ADD statement adds the Rs.DEPTSTORE system to the server.
2. The ADDTABLE statements add the tables to be analyzed to the system.
3. The METHOD statement using the CLUSTER method applies a model that first clusters users into several groups according to user profiles. Then, the method uses the nearest neighbor method to predict unknown ratings.
4. The PREDICT statement generates the top 5 ranked products for user 11815911 using the cluster method.
5. The next METHOD statement using the SVD method applies a model that is based on a singular-value decomposition of a user-item-ratings matrix.
6. The next PREDICT statement generates the top 5 ranked products for user 11815911 using the SVD method.

### PREDICT Statement Output

***Output 7.7*** *Output from the PREDICT Statement (Cluster Method)*

| User | Rank | Rating | Item_sk | Category | Desc |
|---|---|---|---|---|---|
| 11815911 | 1 | 4.98 | 1278 | produce | apples |
| 11815911 | 2 | 4.87 | 1189 | lifestyl | ice_crea |
| 11815911 | 3 | 4.77 | 1342 | produce | peppers |
| 11815911 | 4 | 4.72 | 5592 | butcher | steak |
| 11815911 | 5 | 4.64 | 1359 | canned | olives |

***Output 7.8*** *Output from the PREDICT Statement (SVD Method)*

| User | Rank | Rating | Item_sk | Category | Desc |
|---|---|---|---|---|---|
| 11815911 | 1 | 1.87 | 1342 | produce | peppers |
| 11815911 | 2 | 1.84 | 1278 | produce | apples |
| 11815911 | 3 | 1.79 | 1359 | canned | olives |
| 11815911 | 4 | 1.74 | 5592 | butcher | steak |
| 11815911 | 5 | 1.69 | 1189 | lifestyl | ice_crea |

*Chapter 8*

# VASMP Procedure



## Overview: VASMP Procedure

### *What Does the VASMP Procedure Do?*

The VASMP procedure is used to list in-memory tables and perform administration of Non-distributed SAS LASR Analytic Server instances.

## Syntax: VASMP Procedure

**PROC VASMP** <*options*>;
  **SERVERINFO** <*option*>;
  **SERVERPARM** <*option*>;
  **SERVERTERM** <*options*>;
  **SERVERWAIT** <*options*>;
  **TABLEINFO** <*/ options*>;
**QUIT**;

## PROC VASMP Statement

in a SAS LASR Analytic Server instance.

### Syntax

**PROC VASMP** <*options*>;

#### *Optional Arguments*

**DATA=*libref.member-name***
specifies the table to access from memory. The libref must be assigned from a SAS LASR Analytic Server engine LIBNAME statement.

**IMMEDIATE**
specifies that the procedure executes one statement at a time rather than accumulating statements in RUN blocks.

**Alias** SINGLESTEP

**NOPRINT**
This option suppresses the generation of ODS tables and other printed output in the VASMP procedure.

**NOTIMINGMSG**
When an action completes successfully, the VASMP procedure generates a SAS log message that contains the execution time of the request. Specify this option to suppress the message.

**Alias** NOTIME

## SERVERINFO Statement

The SERVERINFO statement returns information about the SAS LASR Analytic Server.

### Syntax

**SERVERINFO** </ *option*>;

#### *SERVERINFO Statement Options*

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option.

**NORANKS**
specifies to omit the list of host names for the worker nodes. This option reduces the output of the SERVERINFO option considerably for large environments.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server. If you do not specify a PORT= value, then behavior of the SERVERINFO statement depends on whether an

in-memory table is active. If there is no active table, then the procedure attempts to connect to the server using the LASRPORT macro variable. If a table is active, the information is gathered for the server that is implied by the libref of the active table.

## SERVERPARM Statement

The SERVERPARM statement enables you to change some global settings for the server if you have sufficient authorization. The user account that starts the server has privileges to modify server parameters.

### Syntax

**SERVERPARM** <*options*>;

#### *SERVERPARM Statement Options*

**CONCURRENT=*number***
specifies the number of concurrent requests that can execute in the server. Once the threshold is met, the requests are queued and then executed as the currently running requests complete.

**Alias** NACTIONS=

**Default** 20

**EXTERNALMEM=*pct***
specifies the percentage of memory that can be allocated before the server stops transferring data to external processes such as external actions and the SAS High-Performance Analytics procedures. If the percentage is exceeded, the server stops transferring data.

**Default** 75

**Tip** If the external processes are running on a separate cluster, you can increase this value to 100%.

**HADOOPHOME="*path*"**
specifies the path for the HADOOP_HOME environment variable. Changing this variable is useful for migrating SASHDAT files from one Hadoop installation to another.

Setting the HADOOP_HOME environment variable is a server-wide change. All requests, by all users, for reading files from HDFS and saving files, use the specified HADOOP_HOME. This can cause unexpected results if users are not aware of the change.

*Note:* If you are using this option to migrate SASHDAT files, then consider starting a server for that exclusive purpose.

**Alias** HADOOP=

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server. If you do not specify a PORT= value, then behavior of the SERVERPARM statement depends on whether an in-memory table is active. If there is no active table, then the procedure attempts to connect to the server using the LASRPORT macro variable. If a table is active, the information is gathered for the server that is implied by the libref of the active table.

**TABLECEILING=*n* M | G**
specifies a process virtual memory limit (in megabytes or gigabytes) for the server. After the limit is met, adding tables and appending rows to tables is rejected by the server. This option provides a soft limit by which the server can continue to function in a restricted way. Memory use might increase above the setting if the server requires memory to perform an analysis. However, new data, including temporary tables, cannot be added to the server. The limit applies to all virtual memory used by the server, not just the virtual memory that is used by tables. Setting the value to zero removes the limit.

This option has no effect for non-distributed servers. For non-distributed servers, you can specify a virtual memory size limit with the MEMSIZE system option.

| | |
|---|---|
| **Default** | Unlimited |
| **Applies to** | Distributed SAS LASR Analytic Server |

**TABLEMEM=*pct***
specifies the percentage of memory that can be allocated before the server rejects requests to add tables or append data. If the percentage is exceeded, adding a table or appending rows to tables fails. These operations continue to fail until the percentage is reset or the memory usage on the server drops below the threshold.

This option has no effect for non-distributed servers. For non-distributed servers, the memory limits can be controlled with the MEMSIZE system option.

*Note:* The specified *pct* value does not specify the percentage of memory allocated to in-memory tables. It is the percentage of all memory used by the entire machine that—if exceeded—prevents further addition of data to the server. The memory used is not measured at the process or user level, it is computed for the entire machine. In other words, if operating system processes allocate a lot of memory, then loading tables into the server might fail. The threshold is not affected by memory that is associated with SASHDAT tables that are loaded from HDFS.

| | |
|---|---|
| **Alias** | MEMLOAD= |
| **Default** | 75 |

## SERVERTERM Statement

The SERVERTERM statement sends a termination request to the server that is identified through the statement options. You must have sufficient authorization for this request to succeed.

### Syntax

**SERVERTERM** <*options*>;

### SERVERTERM Statement Options

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server.

## SERVERWAIT Statement

The SERVERWAIT statement suspends execution of the VASMP procedure until the server that it uses receives a termination request. This is useful for starting a non-distributed server from a batch program. This statement suspends the SAS session in which it is executed until the server stops or until an interrupt signal is received.

### Syntax

**SERVERWAIT** <*options*>;

### SERVERWAIT Statement Options

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server.

## TABLEINFO Statement

The TABLEINFO statement is used to return information about an in-memory table. This information includes the table name, label, number of rows and column, owner, encoding, and the time of table creation. If no table is in use, then information is returned for the in-memory tables for the server specified in the HOST= and PORT= options.

### Syntax

**TABLEINFO** </ *options*>;

### TABLEINFO Statement Options

**HOST="*host-name*"**
specifies the host name for the SAS LASR Analytic Server. Use this option with the PORT= option.

**PARTVARS**
specifies to include information about partition and orderby variables in the output of the TABLEINFO statement. This enables you to retrieve the names of those variables. If a table is not partitioned or ordered, "N/A" is displayed.

**PORT=*number***
specifies the port number for the SAS LASR Analytic Server. If you do not specify a PORT= value, then behavior of the TABLEINFO statement depends on whether an in-memory table is active. If there is no active table, then the procedure attempts to connect to the server using the LASRPORT macro variable. If a table is active, the information is gathered for the server that is implied by the libref of the active table.

## QUIT Statement

The QUIT statement is used to end the procedure execution. When the procedure reaches the QUIT statement, all resources allocated by the procedure are released. You can no longer execute procedure statements without invoking the procedure again. However, the connection to the server is not lost, because that connection was made through the SAS LASR Analytic Server engine. As a result, any subsequent invocation of the procedure that uses the same libref executes almost instantaneously because the engine is already connected to the server.

**Interaction:** Using a DATA step or another procedure step is equivalent to issuing a QUIT statement. If there is an error during the procedure execution, it is also equivalent to issuing a QUIT statement.

### Syntax

**QUIT**;

## Example: Copying Tables from One Hadoop Installation to Another

### Details

This example does not apply to a non-distributed SAS LASR Analytic Server. It might be necessary to work with more than one Hadoop installation so that you can copy SASHDAT files from one Hadoop installation to a newer version. The SAS LASR Analytic Server must be co-located with both Hadoop installations and both versions of Hadoop must be running.

*Note:* Using the HADOOPHOME= option to switch between Hadoop installations is a server-wide change. If users access the server while the setting is being switched, they might accidentally access the older Hadoop installation. Consider starting a server for the exclusive use of copying files.

### Program

```
proc lasr create port=12636 serverpermissions=700; 1
    performance host="grid001.example.com" install="/opt/TKGrid" nodes=all;
run;

libname private sasiola host="grid001.example.com" port=12636 tag='hps';

data private.iris; set sashelp.iris; run;  /* a table must be active */

proc VASMP data=private.iris; 2
```

```
    serverparm hadoophome="/olderhadoop/path"; 3
quit;

proc lasr add hdfs(path="/dept/sales/y2011" direct) port=12636; 4
    performance host="grid001.example.com";
run;

proc VASMP data=private.y2011(tag="dept.sales"); 5
    serverparm hadoophome="/newerhadoop/path"; 6
run;
    save path="/dept/sales/"; 7
quit;
```

### Program Description

1. Starting a server with SERVERPERMISSIONS=700 creates a single-user server. This is not required but can be used to prevent users from accessing the server while the HADOOP_HOME value is changed and accidentally accessing older or incorrect data.
2. You must have an active table. You can specify an active table with the DATA= option. Any table, such as the Iris data set can be used.
3. Use the SERVERPARM statement to specify the path to the older Hadoop installation with the HADOOPHOME= option. Specify the same path that is returned for the HADOOP_HOME environment variable for the older installation. Example: /hadoop/hadoop-0.21.
4. You must specify the DIRECT option. This statement loads table y2011 into memory from the /dept/sales directory in HDFS.
5. The TAG= option must be used to specify the in-memory table. The server tag matches the HDFS path to the table, but the slashes are replaced with periods (.). If the table was loaded from /, then specify TAG=HADOOP.
6. Use the SERVERPARM statement to specify the path to the newer Hadoop installation. Example: /hadoop-0.23/hadoop-0.23.1.
7. The SAVE statement writes the y2011 table to HDFS in the /dept/sales directory. The HDFS directory is in the newer Hadoop installation.

*Chapter 9*

# Using the SAS LASR Analytic Server Engine



## What Does the SAS LASR Analytic Server Engine Do?

The SAS LASR Analytic Server engine is used to add, remove, and access tables in a SAS LASR Analytic Server instance.

Typically, the tables that are loaded in memory are very large on a SAS LASR Analytic Server instance. The engine makes it possible to access a table and use procedures like the UNIVARIATE procedure. However, in this case, the entire table is transferred from the server instance to the SAS session and then the procedure is executed on the data. If the table is large, the data volume can overwhelm the SAS session.

The best performance for accessing the data through the engine is with a SAS High-Performance Analytics procedure. These procedures are designed to operate in a distributed computing environment and can read data in parallel from a SAS LASR Analytic Server instance.

## Understanding How the SAS LASR Analytic Server Engine Works

An engine is a component of SAS software that reads from or writes to a file. The SAS LASR Analytic Server engine provides Read and Write access for data and metadata

information such as variable attributes. Each engine enables SAS to access files that are in a particular format. There are several types of SAS engines.

You use the SAS LASR Analytic Server engine like other SAS data access engines. That is, you execute a LIBNAME statement to assign a libref and to specify the engine. You then use that libref throughout the SAS session where a libref is valid to access a SAS LASR Analytic Server instance.

## Understanding Server Tags

### *What is a Server Tag?*

A server tag is a text string that is associated with a table that is loaded into memory on a SAS LASR Analytic Server instance. The server tag is specified in the LIBNAME statement or as a data set option. The server tag and the table name are used together to match the name used for tables in the SAS LASR Analytic Server.

### *Why Use a Server Tag?*

The following list identifies some reasons for specifying a server tag:

- You must use a server tag in a LIBNAME statement or as a data set option to access tables that are loaded from HDFS.
- Different users can load tables with the same name, such as Forecast, into a server instance. You use a server tag and the Forecast table name to specify which table to access.
- Tables that are loaded into memory with the LASR procedure (but not from HDFS) use the libref as the server tag. In order to access these tables, you must specify the server tag.
- When you load a table into memory from HDFS with the LASR procedure, the table is assigned a server tag. The server tag represents the directory path from which the SASHDAT file was loaded. You need to use that server tag to access the table.

### *See Also*


- "Example 4: Accessing Tables Loaded with a DATA Step" on page 424
- "Example 5: Accessing Tables Loaded with the LASR Procedure" on page 425
- "Example 6: Accessing Tables That Are Loaded from HDFS" on page 426


## Comparing the SAS LASR Analytic Server Engine with the LASR Procedure

The engine and the LASR procedure are similar in that you can use them to load tables to memory in a SAS LASR Analytic Server instance. You can also use the engine and the procedure to unload tables from memory.

You can use the engine with the APPEND= data set option to add data to an existing table. The procedure cannot modify the data.

You cannot use the engine to load tables into memory from HDFS. Only the LASR procedure can be used to load tables into memory from HDFS.

You can use the LASR procedure to save in-memory tables to HDFS as SASHDAT tables. The procedure writes the data in parallel because the server instance uses HDFS as a co-located data provider.

You can use the engine to supply a libref to SAS procedures or DATA steps. However, be aware that if you use the engine as an input data source, the data volume can be large. Large data volumes can overwhelm the SAS session.

## What is Required to Use the SAS LASR Analytic Server Engine?

To use the SAS LASR Analytic Server engine, the following are required:

- access to the machines in the cluster where a SAS LASR Analytic Server is running. A server instance is started with the LASR procedure.
- an operating system user ID that is configured for passwordless secure shell (SSH) on the machines in the cluster

The requirement for passwordless SSH is not unique to using the engine. Passwordless SSH is used throughout SAS High-Performance Analytics. The SAS High-Performance Computing Management Console can be used to simplify configuring users for passwordless SSH.

## What is Supported?

The following list identifies some usage notes:

- The engine does not support views or BY-group processing.
- You cannot replace or overwrite tables in memory. You must unload the table and then load the new table.
- You cannot use the APPEND procedure. However, you can use an APPEND= data set option to achieve the same result.
- Loading tables into memory from HDFS is performed with the LASR procedure. You cannot load tables into memory from HDFS with the engine.
- The engine guarantees the data order for a particular configuration of worker nodes. If you load a table in to a server and you retrieve the data three times, the order of the data is the same. However, if you start another server and load the same table into a different number of worker nodes, then the order in which you retrieve the data is different. However, it is reproducible within fetches from a single server.
- Any order-dependent operation, such as the LAG or DIF functions, cannot rely on stability of results beyond that which can be guaranteed by the distribution model of the data.

When the engine is used to access a SAS LASR Analytic Server that is on the same machine, a network socket connection is used. This applies to Windows and Linux

operating environments. However, when you use the engine to connect to a server on a different host, the connection is based on passwordless SSH. As a result, the engine can connect to only remote hosts that use Linux.

*Chapter 10*

# LIBNAME Statement for the SAS LASR Analytic Server Engine



## Dictionary

### LIBNAME Statement Syntax

associates a SAS libref with tables on a SAS LASR Analytic Server.

**Valid in:** Anywhere

**Category:** Data Access

#### Syntax

**LIBNAME** *libref* **SASIOLA** <LASR= "*server-description-file*">
<HOST= "*grid-host*"> <PORT= *number*>
<TAG= *server-tag*> <FORMATEXPORT= DATA | NONE | ALL>
<NODEFAULTFORMAT= YES | NO>
<STARTSERVER <=(*non-distributed-server-options*)>>
<SIGNER= "*authorization-web-service-uri*">
<FORCESIGNER=YES | NO>
<VERBOSE= YES | NO>;

##### *Required Arguments*

***libref***
is a valid SAS name that serves as a shortcut name to associate with the tables on the SAS LASR Analytic Server. The name must conform to the rules for SAS names. A libref cannot exceed eight characters.

**SASIOLA**
is the engine name for the SAS LASR Analytic Server engine.

### Optional Arguments

**FORCESIGNER= YES | NO**
specifies whether to rely on the SAS LASR Authorization service or to create a signature file for data that is loaded into the server.

If you specify YES, then the server does not create a signature file for a table when the data is loaded into the server. This forces the server to rely on the SAS LASR Authorization service for data access decisions. This option enables sites that run the server as a service account and manage access controls in SAS metadata. The service account is identified as the table owner, although access control decisions are managed with SAS metadata.

**Default** NO

**FORMATEXPORT= DATA | NONE | ALL**
specifies how the engine interacts with user-defined formats when tables are added to the server instance. The default value is FORMATEXPORT=DATA. This option can be overridden in a data set option. This option has no effect for input data sets (data sets that are transferred from the server instance to the SAS client).

**DATA**
specifies that the definition of all user-defined formats associated with variables written to the server instance are transferred to the server. You can then use those formats when you access the table (from a client such as SAS Visual Analytics). The user-defined formats are transferred to the server only once. The formats are not transferred as XML streams on subsequent requests to the server.

**NONE**
specifies that user-defined formats are not transferred to the server.

**ALL**
specifies that all formats in the format catalog search path are converted and transferred to the server with the table. This option is useful if the catalog search path contains user-defined formats that are not associated with variables in the table, but you might want to use later. Considerable resources can be required to generate the XML representation of the formats for deployments that have large catalogs or a deep search path.

**HOST="*grid-host*"**
specifies the grid host that has a running server instance. Enclose the host name in quotation marks. If you do not specify the HOST= option, it is determined from the GRIDHOST= environment variable.

**Alias** SERVER=

**Interaction** If the LASR= option is specified, then the host name specified in the HOST= option is ignored.

**LASR="*server-description-file*"**
specifies the server to use. Provide the fully qualified path to the server description file.

**Interaction** If you specify the server description file to use, then you do not need to specify the HOST= or PORT= options.

**NODEFAULTFORMAT= YES | NO**
specifies whether a default format that is applied to a variable is reported by the engine.

If you do not specify a format for a variable when you add a table to the server, the engine adds a default format. The server applies BEST. for numeric variables and $. for character variables.

The engine displays this "forced" format in procedures that list variable attributes, such as the CONTENTS and DATASETS procedures. If you specify NODEFAULTFORMAT=YES, then the display of the "forced" format is suppressed.

*Note:* This setting does not control whether formats are applied to a variable.

**PORT=*number***
specifies the port number to use for connecting to the running server instance. If you use the PORT= option when you start a non-distributed server instance, then use this option to specify the network port number for the server.

**Interaction** The LASR procedure stores the port number of the last server instance that is started in the LASRPORT macro variable. You can specify PORT=&LASRPORT to use the macro variable.

**SIGNER="*authorization-web-service-uri*"**
specifies the URI for the SAS LASR Authorization web service. The web service is provided by the SAS Visual Analytics software. For more information, see *SAS Visual Analytics: Administration Guide*.

**Example** SIGNER="https://server.example.com/SASLASRAuthorization"

**STARTSERVER= YES | NO**
**STARTSERVER <=(*non-distributed-server-options*)>**
specifies to start a non-distributed server instance. Options are specified as name and value pairs. Separate each option with a space. The following options are available:

**AFFINITY= YES | NO**
requests that the concurrently executing threads of the server are associated with specific CPUs. When thread affinity is set to YES, a thread does not bounce between CPUs.

**Default** NO

**CLF= YES | NO**
specifies to use the common log format for log files. This format is a standardized text file format that is frequently analyzed by web analysis software. Specifying this option implies the LOGGING option.

**KEEPLOG= YES | NO**
specifies to keep the log files when the server exits instead of deleting them. By default, the log files are removed when the server exits. Specifying this option implies the LOGGING option.

**LOGGING= YES | NO**
specifies to enabling logging of server actions. The log file is stored with the signature files in the directory that is specified in the PATH= option. The log file is named in the pattern **`LASR.`*`timestamp`*`.0.saslasr.log`**.

**MAXLOGSIZE=*n***
specifies the maximum log file size, in megabytes, for a log file. When the log file reaches the specified size, the log file is rolled over and renamed with a sequentially assigned index number (for example, **`.log.1`**). The default value is 100 megabytes. Specifying this option implies the LOGGING option.

**TIP** Do not include an MB or M suffix when you specify the size.

**MAXLOGROLL=*n***
specifies the maximum number of log files to create. When the maximum has been reached, the server begins to overwrite existing log files. The oldest log file is overwritten first. The default value is 10. Specifying this option implies the LOGGING option.

**MERGELIMIT=*n***
specifies the limit for merging large result sets into smaller groups. The MERGEBINS= option specifies the size of the group. If MERGEBINS= is not specified, then *n* is the bin limit.

**MERGEBINS=*b***
specifies the number of bins that numeric variables are binned into when MERGELIMIT=*n* is reached.

**NOHOSTCHECK = YES | NO**
specifies that the server does not check that the host name specified in the HOST= option is the local host. This option can be useful with unusual network configurations.

**Interaction** When the SIGNER= option is also specified, the host name that is specified in the HOST= option is sent to the SAS LASR Authorization Service.

**NTHREADS=*n***
specifies the number of threads to use for the server. By default, *n* equals the number of CPU cores on the machine.

**PATH="*signature-file-path*"**
specifies the directory to use for storing the server and table signature files. The specified directory must already exist.

If you do not specify a value for PATH=, the signature files are stored in the default utility file directory of the SAS session.

**PERM=*mode***
specifies the permission setting for accessing the server instance. The mode value is specified as an integer value such as 755. The mode corresponds to the mode values that are used for UNIX file access permissions.

For Windows servers, the UNIX mode setting is not applicable. Access is controlled according to permissions that you set manually on the signature files directory.

**Alias** PERMISSION=

**Range** 600 to 777

**Alias** START=

**TAG=*server-tag***
specifies the tag to use for identifying the tables in the server instance. The value for *server-tag* cannot exceed 128 characters in length.

**VERBOSE= YES | NO**
specifies whether the engine accepts and reports extra messages from TKGrid. Specifying VERBOSE=YES can help diagnose problems with passwordless SSH setups, grid install locations, and so on.

The following message in the SAS log shows an example of a problem with passwordless SSH configuration.

```
ERROR: Failed to load the SAS LASR Analytic Server access extension
       in the distributed computing environment.

Server refused our key from:
/home/sasdemo/.ssh/id_rsa
Timeout waiting for Grid connection.
```

## Examples

### *Example 1: Submitting a LIBNAME Statement Using the Defaults*

#### Program

The following example shows the code for starting a server with the LASR procedure and then connecting to the same server with a LIBNAME statement:

```
option set=GRIDHOST="grid001.example.com"; 1
option set GRIDINSTALLLOC="/opt/TKGrid";

proc lasr
    create port=10010
    path="/tmp" noclass;

    performance nodes=all;
run;

libname salessvr sasiola; 2
```

```
NOTE: No tag was specified in the LIBNAME statement. The default tag (WORK) is
      used to name and identify tables in the LASR Analytic Server. You can
      specify a tag as a data set option.
NOTE: Libref SALESSVR was successfully assigned as follows:
      Engine:        SASIOLA
      Physical Name: SAS LASR Analytic Server engine on host
      'grid001.example.com', port 10010
```

#### Program Description

1. The grid host is specified in the GRIDHOST environment variable.
2. The default LIBNAME statement does not include the LASR=, HOST=, or PORT= options. The LIBNAME statement uses host name from the GRIDHOST environment variable and the LASRPORT macro variable and connect to server instance.

### *Example 2: Submitting a LIBNAME Statement Using the LASR= Option*

The following example shows a LIBNAME statement that uses the LASR= option to specify the server instance to use:

```
proc lasr
    create="/tmp/hrsvr" 1
    path="/opt/VADP/var/hr"
    noclass;

    performance host="grid001.example.com" install="/opt/TKGrid" nodes=all; 2
run;
```

```
libname hrsvr sasiola lasr="/tmp/hrsvr"; 3
```

### Program Description

1. A server instance is started with the CREATE= option. The server description file is /tmp/hrsvr.
2. The HOST= option is specified in the PERFORMANCE statement rather than specifying the GRIDHOST environment variable.
3. The LASR= option specifies the server description file that was created when the server instance started.

## *Example 3: Submitting a LIBNAME Statement Using the HOST= and PORT= Options*

The following example shows the code for starting a server with the LASR procedure and then submitting a LIBNAME statement to use the same server by specifying the HOST= and PORT= options.

```
proc lasr
    create port=10010
    path="/tmp"
    noclass;

    performance host="grid001.example.com" install="/opt/TKGrid" nodes=all;
run;
```

```
NOTE: The LASR procedure is executing in the distributed computing environment
      with 7 worker nodes.
NOTE: The server started on 'grid001.example.com' port 10010. 1
NOTE: The LASR Analytic Server port '12637' has been assigned to the macro
      variable "LASRPORT".
```

```
libname hrdata sasiola host="grid001.example.com" port=10010 tag='hr'; 2
```

```
NOTE: Libref hrdata was successfully assigned as follows:
      Engine:        SASIOLA
      Physical Name: SAS LASR Analytic Server engine on host
      'grid001.example.com', port 10010
```

### Program Description

1. When a server instance is started, the SAS log indicates the port number for the server instance.
2. The PORT= option in the LIBNAME statement references the port number. The value for the PORT= option can also be specified as PORT=&LASRPORT to use the port number for the most recently started server instance.

## *Example 4: Accessing Tables Loaded with a DATA Step*

The following example shows how to use the engine without a server tag in a DATA step.

```
libname sales sasiola port=10010;

data sales.prdsale;
```

```
    set sashelp.prdsale;
run;

proc datasets lib=sales;
quit;

* a server tag is not needed to access the data ;
proc print data=sales.prdsale(obs=5);
run;
```

When no server tag is specified, a default server tag that is named WORK is used.

***Output 10.1*** *DATASETS Procedure Output Showing the WORK Server Tag*

| Directory | |
|---|---|
| Libref | SALES |
| Engine | SASIOLA |
| Physical Name | SAS LASR Analytic Server engine on host 'grid001.example.com', port 10010 |

| # | Name | Member Type | Number of Rows | Number of Columns | Last Modified | Server Tag | Data Encoding | Owner |
|---|---|---|---|---|---|---|---|---|
| 1 | PRDSALE | DATA | 1440 | 10 | 29Nov12:20:53:18 | WORK | latin1 | sasdemo |

## *Example 5: Accessing Tables Loaded with the LASR Procedure*

When tables are loaded to memory on a server instance with the LASR procedure, the libref that is used with the procedure is set as the server tag. The following example shows how to add a table to a server instance and then access the table with a LIBNAME statement that includes a server tag.

```
proc lasr port=10010 add data=sashelp.prdsale noclass;
run;

libname lasr2 sasiola tag=sashelp;

proc datasets lib=lasr2;
run;

* a server tag is not needed to access the data              ;
* because a server tag is specified in the LIBNAME statement ;
proc print data=lasr2.prdsale(obs=5);
run;
```

By default, the libref is used as the server tag. The following display shows **`sashelp`** used as the server tag.

***Output 10.2*** *DATASETS Procedure Output Showing the SASHELP Server Tag*

| Directory | |
|---|---|
| Libref | LASR2 |
| Engine | SASIOLA |
| Physical Name | SAS LASR Analytic Server engine on host 'grid001.example.com', port 10010 |

| # | Name | Member Type | Number of Rows | Number of Columns | Last Modified | Server Tag | Data Encoding | Owner |
|---|---|---|---|---|---|---|---|---|
| 1 | PRDSALE | DATA | 1440 | 10 | 29Nov12:21:34:08 | SASHELP | latin1 | sasdemo |

### *Example 6: Accessing Tables That Are Loaded from HDFS*

When tables are loaded into memory on the server instance with the LASR procedure and the SASHDAT engine, the server tag is related to the HDFS directory name. The server tag is the same as the HDFS path to the SASHDAT file, but is delimited with periods (.) instead of slashes (/).

The following example shows how to add a table to a server instance from HDFS and then access the table with a LIBNAME statement that includes a server tag.

```
libname sales sashdat path="/dept/sales";

proc lasr port=10010 add data=sales.sales2012 noclass;
run;

libname lasr3 sasiola tag="dept.sales";

proc datasets lib=lasr3;
run;

* access the data with the "dept.sales" server tag;
proc print data=lasr3.sales2012(obs=5);
run;
```

***Output 10.3*** *DATASETS Procedure Output Showing the DEPT.SALES Server Tag*

| Directory | |
|---|---|
| Libref | LASR3 |
| Engine | SASIOLA |
| Physical Name | SAS LASR Analytic Server engine on host 'grid001.example.com', port 10010 |

| # | Name | Member Type | Number of Rows | Number of Columns | Last Modified | Server Tag | Data Encoding | Owner |
|---|---|---|---|---|---|---|---|---|
| 1 | SALES2012 | DATA | 452459 | 39 | 19Jul12:21:34:0 | DEPT.SALES | latin1 | sasdemo |

### *Example 7: Loading a Table and Partitioning*

Partitioning a table as it is loaded to memory can be a powerful feature for reducing processing times. For more information, see "Data Partitioning and Ordering" on page 25.

```
libname lasrlib sasiola host="grid001.example.com" port=10010 tag="sales";

data lasrlib.prdsale(partition=(country region) orderby=(descending year)); 1
    set sashelp.prdsale;
run;
```

#### Program Description

The Prdsale table is distributed to the machines in the cluster according to the PARTITION= data set option. The rows are distributed according to the unique combinations of the formatted values for the variables Country and Region. In addition, the ORDERBY= option is used to sort the rows in each partition by Year, in descending order.

### *Example 8: Creating an Empty Table*

Creating an empty table can be useful to seed the column information for a table that you later append to. The following statements create an empty table with two numeric and two character variables:

```
libname lasrlib sasiola host="grid001.example.com" port=10010 tag="sales";

data lasrlib.empty;
   length c1 $15;
   length c2 $12;
   x=1;
   y=1;
   c1="";
   c2="";
   delete;
run;
```

#### Program Description

The Empty table is added to the server.

*Chapter 11*

# Data Set Options for the SAS LASR Analytic Server Engine



## Dictionary

### APPEND= Data Set Option

specifies to append the data to an existing table in the server instance.

**Interaction:** You must use the NOCLASS option if you load the initial table with the LASR procedure.

#### Syntax

**APPEND=**YES | NO

## Details

By default, the SAS LASR Analytic Server engine does not permit appending observations to tables. The APPEND= data set option can be used to permit adding observations to an existing table.

***Example Code 11.1*** *Using the APPEND= Data Set Option*

```
proc lasr add data=grp1.sales noclass port=10010;
run;

libname grp1lasr host="grid001.example.com" port=10010 tag=grp1;

data grp1lasr.sales (append=yes);
    set yr2012.sales (keep=date location amount);
run;
```

As shown in the preceding example, the APPEND= data set option can be used to add observations to an existing table. The KEEP= option on the input data set specifies the variables from the input data to append. Any variables for which the input data set does not append data are set to missing. You cannot add new variables to the table.

The example also shows how to load the initial table to memory with the LASR procedure. The NOCLASS option must be specified if you use the LASR procedure. As an alternative, you can load the initial table to memory with the SAS LASR Analytic Server engine.

# ARRAY= Data Set Option

requests that the name space for the table on the SAS session is extended with names that are derived from a temporary numeric array.

## Syntax

**ARRAY=**(*array-name*, *n*)

## Details

The following example shows how to specify a temporary numeric array with variables named Temp1–Temp4:

```
proc imstat;
  table lasrlib.sales (array=(temp,4));
```

The variables in the temporary numeric array do not exist in the in-memory table, but the SAS session assumes that they are there. Using temporary names this way can be useful when your SAS program refers to calculated temporary columns that do not exist when table is opened for input. For example, this option can enable you to retrieve the results for calculated columns with the FETCH statement of the IMSTAT procedure.

This option is used for numeric variables only. If you want to refer to specific temporary variable names, you can also use the TEMPNAMES= option. The TEMPNAMES= option enables you to specify the variable type, and in the case of character variables, the length of the variable.

## AUTOCOMMIT= Data Set Option

specifies how rows are committed to an in-memory table during Append operations.

### Syntax

**AUTOCOMMIT=** *n*R

**AUTOCOMMIT=** *k*S

### Details

By default, rows are not committed to an in-memory table until the DATA step completes. That is, the rows are held in intermediate storage areas in the server and are not included in requests for data or computational results until the DATA step completes. If you specify AUTOCOMMIT=*n* or AUTOCOMMIT=*n*R, then the server commits the rows when at least *n* rows have been received.

If you specify AUTOCOMMIT=*k*S, the server commits any rows received within *k* seconds of the start of the Append operation or within *k* seconds of the previous commit.

## FORMATEXPORT= Data Set Option

specifies how the engine interacts with user-defined formats when tables are added to the server instance.

### Syntax

**FORMATEXPORT=**DATA | NONE | ALL

### Details

This option is used to override the FORMATEXPORT= option for the LIBNAME statement.

### See Also

FORMATEXPORT= option in the LIBNAME statement

## FULLCOPYTO= Data Set Option

specifies the number of full copies of a table to load into child non-distributed servers.

### Syntax

**FULLCOPYTO=***n*

### Details

Loading full copies of a table into child non-distributed servers is a strategy for enabling high-volume access to smaller tables. For more information, see "High Volume Access to Smaller Tables" on page 27.

### See Also

FULLCOPYTO= option in the LASR procedure

## HASH= Data Set Option

specifies that when partitioning data, the distribution of partitions is not determined by a tree, but by a hashing algorithm. As a result, the distribution of the partitions is not as evenly balanced, but it is effective when working with high-cardinality partition keys (in the order of millions of partitions).

### Syntax

PARTITION=(*variable-list*) **HASH=**YES | NO

### Example

```
data lasrlib.transactions(partition=(cust_id year) hash=yes);
    set somelib.sometable;
run;
```

## NODEFAULTFORMAT= Data Set Option

specifies whether a default format that is applied to a variable is reported by the server.

### Syntax

**NODEFAULTFORMAT=**YES | NO

### Details

This option is used to override the NODEFAULTFORMAT= option for the LIBNAME statement.

## ORDERBY= Data Set Option

specifies the variables by which to order the data within a partition.

**Example:** "Example 7: Loading a Table and Partitioning" on page 426

### Syntax

**ORDERBY=**(*variable-list*)

**ORDERBY=**(*variable-name* <DESCENDING> *variable-name*)

## Details

The variable names in the *variable-list* are separated by spaces.

The ordering is hierarchical. For example, ORDERBY=(A B) specifies ordering by the values of variable B within the ordered values of variable A. The specified variables must exist and cannot be specified as partitioning variables. The order is determined based on the raw value of the variables and uses locale-sensitive collation for character variables. By default, values are arranged in ascending order. You can specify descending order by preceding the variable name in the *variable-list* with the keyword DESCENDING.

## Example

The following code sample orders the data in the partitions by Year in ascending order and then by Quarter in descending order.

```
data lasrlib.prdsale (partition=(country region)
                        orderby=(year descending quarter));
    set sashelp.prdsale;
run;
```

# PARTITION= Data Set Option

specifies the list of partitioning variables to use for partitioning the table.

**Example:** "Example 7: Loading a Table and Partitioning" on page 426

## Syntax

**PARTITION=**(*variable-list*)

## Details

Partitioning is available only when you create tables. User-defined format definitions for partitioning variables are always transferred to the server, regardless of the FORMATEXPORT= option.

Partitioning by a variable that does not exist in the output table is an error. Partitioning by a variable listed in the ORDERBY= option is also an error. Partition keys are derived based on the formatted values in the order of the variable names in the *variable-list*.

Be aware that the key construction is not hierarchical. That is, PARTITION=(A B) specifies that any unique combination of formatted values for variables A and B defines a partition.

# PERM= Data Set Option

specify the permission setting for the table in the server.

**Alias:** PERMISSION=

### Syntax

**PERM=***mode*

### Details

The *mode* is specified as an integer (for example, PERM=755). The value is converted by the engine to a umask. If no permission is specified, the access permissions for the table are set according to the umask of user that loads the table.

For Windows servers, the UNIX mode setting is not applicable. Access is controlled according to permissions that you set manually on the signature files directory.

## SIGNER= Data Set Option

specifies the URI of the SAS LASR Authorization web service.

### Details

This option is used to override the SIGNER= option for the LIBNAME statement.

## SQUEEZE= Data Set Option

specifies to load the data as a compressed table.

### Syntax

**SQUEEZE**= YES | NO

## TAG= Data Set Option

specifies the tag to use for identifying the tables in the server instance.

### Syntax

**TAG=**'*server-tag*'

### Details

If no TAG= option is specified as a data set option, then the TAG= option from the LIBNAME statement is used. If the LIBNAME statement does not specify the TAG= option, then the name of the libref is used as the server tag.

## TEMPNAMES= Data Set Option

requests that the name space for the table on the SAS session is extended with the specified names.

## Syntax

**TEMPNAMES=**(*variable-name1* <, *variable-name2* …>)

## Details

The following example shows how to specify two temporary numeric variables named Difference and Ratio:

```
proc imstat;
    table lasrlib.prdsale (tempnames=(difference ratio));
run;

    fetch actual -- month predict ratio / tempnames=(difference ratio)
      tempexpress="difference = actual - predict; ratio = actual / predict";
  run;
```

By default, temporary variables that are specified through the TEMPNAMES= option are of numeric type. To define a temporary character variable, follow the variable name with a dollar sign ($) and an optional length.

The following example shows the variable Cust_Name as a character variable with a length of 20:

```
table lasrlib.accounts(tempnames=(total_deposits
                          cust_name $ 20
                          deposit_count
                          branch_count));
```

# UCA= Data Set Option

specifies that you want to use Unicode Collation Algorithms (UCA) to determine the ordering of character variables in the ORDERBY= option.

## Syntax

PARTITION=(*key*) ORDERBY=(*variable-list*) **UCA=**YES | NO

*Chapter 12*

# Using the SASHDAT Engine



## What Does the SASHDAT Engine Do?

The SASHDAT engine is used to distribute data in the Hadoop Distributed File System (HDFS) or an NFS-mounted file system such as that provided by the MapR NFS service. The engine enables you to distribute the data in a format that is designed for high-performance analytics. The block redundancy and distributed computing provided by HDFS is complemented by the block structure that is created with the engine.

The engine is designed to distribute data only. Because the data volumes in HDFS are typically very large, the engine is not designed to read SASHDAT files and transfer data back to the SAS client. For example, consider the case of reading several terabytes of data from a distributed computing environment, transferring that data back to a SAS session, and then using the UNIVARIATE or REG procedures on such a large volume of data. In contrast, the SAS High-Performance Analytics procedures are designed to operate in a distributed computing environment and to read data in parallel from a co-located data provider like HDFS.

## Understanding How the SASHDAT Engine Works

An engine is a component of SAS software that reads from or writes to a file. The SASHDAT engine is write-only for data and read-write for metadata information such as variable attributes. Each engine enables SAS to access files that are in a particular format. There are several types of SAS engines.

You use the SASHDAT engine like other SAS data access engines. That is, you execute a LIBNAME statement to assign a libref and to specify the engine. You then use that libref throughout the SAS session where a libref is valid to transfer data to the Hadoop Distributed File System (HDFS) or to retrieve information about a table in HDFS.

## What is Required to Use the SASHDAT Engine?

To use the SASHDAT engine, the following are required:

- access to the machines in the cluster where Hadoop is configured with the SAS Plug-ins for Hadoop, or where the machines use NFS to mount a distributed file system
- an operating system user ID that is configured for passwordless secure shell (SSH) on the machines in the cluster

The requirement for passwordless SSH is not unique to using the engine. Passwordless SSH is used throughout SAS High-Performance Analytics. The SAS High-Performance Computing Management Console can be used to simplify configuring users for passwordless SSH.

## What is Supported?

The SASHDAT engine is used with a Hadoop cluster that is configured to use the SAS Plug-ins for Hadoop, or with machines that use NFS to mount a distributed file system.

The engine is designed as a write-only engine for transferring data. However, SAS High-Performance Analytics procedures are designed to read data in parallel from a co-located data provider. The LASR procedure, and other procedures such as HPREG and HPLOGISTIC, can read data with the engine. The HPDS2 procedure is designed to read data and write data in parallel. The HPDS2 procedure can be used with the engine to read data and create new SASHDAT tables.

Whenever a SAS High-Performance Analytics procedure is used to create a SASHDAT table, the procedure creates the data with a default block size of 2 megabytes. This size can be overridden with the BLOCKSIZE= data set option.

The engine does not support views.

*Chapter 13*

# LIBNAME Statement for the SASHDAT Engine



## Dictionary

### LIBNAME Statement Syntax

Associates a SAS libref with SASHDAT tables stored in HDFS.

**Valid in:** Anywhere

**Category:** Data Access

#### Syntax

**LIBNAME** *libref* **SASHDAT**
<HOST="*grid-host*"> <INSTALL="*grid-install-location*">
<PATH="*HDFS-path*"> <COPIES=*n* > <INNAMEONLY=YES | NO>
<NODEFAULTFORMAT=YES | NO>
<SIGNER="*authorization-web-service-uri*">
<VERBOSE= YES | NO>;

##### *Required Arguments*

***libref***
is a valid SAS name that serves as a shortcut name to associate with the SASHDAT tables. The name must conform to the rules for SAS names. A libref cannot exceed eight characters.

**SASHDAT**
is the engine name for the SASHDAT engine.

### Optional Arguments

**COPIES=*n***
specifies the number of replications to make for the data set (beyond the original blocks). The default value is 2 when the INNAMEONLY option is specified and otherwise is 1. Replicated blocks are used to provide fault tolerance. If a machine in the cluster becomes unavailable, then the blocks needed for the SASHDAT file can be retrieved from replications on other machines. If you specify COPIES=0, then the original blocks are distributed, but no replications are made and there is no fault tolerance for the data.

**HOST="*grid-host*"**
specifies the grid host that has a running Hadoop NameNode. Enclose the host name in quotation marks. If you do not specify the HOST= option, it is determined from the GRIDHOST= environment variable.

**INNAMEONLY= YES | NO**
specifies that when data is added to HDFS, that it should be sent as a single block to the Hadoop NameNode for distribution. This option is appropriate for smaller data sets.

**Alias** NODIST

**INSTALL="*grid-install-location*"**
specifies the path to the TKGrid software on the grid host. If you do not specify this option, it is determined from the GRIDINSTALLLOC= environment variable.

**NODEFAULTFORMAT= YES | NO**
specifies whether a default format that is applied to a variable is reported by the engine.

If you do not specify a format for a variable when you add a table to HDFS, the engine adds a default format. The server applies BEST. for numeric variables and $. for character variables.

The engine displays this "forced" format in procedures that list variable attributes, such as the CONTENTS and DATASETS procedures. If you specify NODEFAULTFORMAT=YES, then the display of the "forced" format is suppressed.

*Note:* This setting does not control whether formats are applied to a variable.

**PATH="*HDFS-path*"**
specifies the fully qualified path to the directory to use for SASHDAT files. You do not need to specify this option in the LIBNAME statement because it can be specified as a data set option.

**SIGNER="*authorization-web-service-uri*"**
specifies the URI for the SAS LASR Authorization web service. This option enables the specified web service to supply the pass phrase for encryption. The HOST= and PATH= options must be associated with a Hadoop server that is registered in SAS metadata and configured to use encryption.

**Interaction** To use the encryption settings from SAS metadata, do not specify the ENCRYPT=AES or ENCRYPTKEY= data set options.

**Example** SIGNER="https://server.example.com/SASLASRAuthorization"

**VERBOSE= YES | NO**
specifies whether the engine accepts and reports extra messages from TKGrid. For more information, see the VERBOSE= option on page 422 for the SAS LASR Analytic Server engine.

## Examples

### *Example 1: Submitting a LIBNAME Statement Using the Defaults*

#### Program

The following example shows the code for connecting to a Hadoop NameNode with a LIBNAME statement:

```
option set=GRIDHOST="grid001.example.com"; 1
option set GRIDINSTALLLOC="/opt/TKGrid";

libname hdfs sashdat; 2
```

```
NOTE: Libref HDFS was successfully assigned as follows:
      Engine:        SASHDAT
      Physical Name: grid001.example.com
```

#### Program Description

1. The host name for the Hadoop NameNode is specified in the GRIDHOST environment variable.
2. The LIBNAME statement uses host name from the GRIDHOST environment variable and the path to TKGrid from the GRIDINSTALLLOC environment variable. The PATH= and COPIES= options can be specified as data set options.

### *Example 2: Submitting a LIBNAME Statement Using the HOST=, INSTALL=, and PATH= Options*

The following example shows the code for submitting a LIBNAME statement with the HOST=, INSTALL=, and PATH= options.

```
libname hdfs sashdat host="grid001.example.com" install="/opt/TKGrid"
    path="/user/sasdemo";
```

```
NOTE: Libref HDFS was successfully assigned as follows:
      Engine:        SASHDAT
      Physical Name: Directory '/user/sasdemo' of HDFS cluster on host
                     grid001.example.com
```

### *Example 3: Adding Tables to HDFS*

The following code sample demonstrates the LIBNAME statement and the REPLACE= and BLOCKSIZE= data set options. The LABEL= data set option is common to many engines.

```
libname arch "/data/archive";
libname hdfs sashdat host="grid001.example.com" install="/opt/TKGrid"
    path="/dept";

data hdfs.allyears(label="Sales records for previous years"
                   replace=yes blocksize=32m);
  set arch.sales2012
      arch.sales2011
      ...
  ;
run;
```

### Example 4: Adding a Table to HDFS with Partitioning

The following code sample demonstrates the PARTITION= and ORDERBY= data set options. The rows are partitioned according to the unique combinations of the formatted values for the Year and Month variables. Within each partition, the rows are sorted by descending values of the Prodtype variable. For more information, see "Data Partitioning and Ordering" on page 25.

```
libname hdfs sashdat host="grid001.example.com" install="/opt/TKGrid"
    path="/dept";

data hdfs.prdsale(partition=(year month) orderby=(descending prodtype));
    set sashelp.prdsale;
run;
```

### Example 5: Removing Tables from HDFS

Removing tables from HDFS can be performed with the DATASETS procedure.

```
libname hdfs sashdat host="grid001.example.com" install="/opt/TKGrid"
    path="/dept";

proc datasets lib=hdfs;
    delete allyears;
run;
```

```
NOTE: Deleting HDFS.ALLYEARS (memtype=DATA).
```

### Example 6: Creating a SASHDAT File from Another SASHDAT File

The following example shows copying a data set from HDFS, adding a calculated variable, and then writing the data to HDFS in the same library. The BLOCKSIZE= data set option is used to override the default 8-megabyte block size that is created by SAS High-Performance Analytics procedures. The COPIES=0 data set option is used to specify that no redundant blocks are created for the output SASHDAT file.

```
libname hdfs sashdat host="grid001.example.com" install="/opt/TKGrid"
    path="/dept";

proc hpds2
    in = hdfs.allyears(where=(region=212)) 1
    out = hdfs.avgsales(blocksize=32m copies=0); 2

    data DS2GTF.out;
        dcl double avgsales;
        method run();
        set DS2GTF.in;
            avgsales = avg(month1-month12);
        end;
    enddata;
run;
```

1 The WHERE clause is used to subset the data in the input SASHDAT file.

2 The BLOCKSIZE= and COPIES= options are used to override the default values.

## Example 7: Working with CSV Files

The comma-separated value (CSV) file format is a popular format for files stored in HDFS. The SASHDAT engine can read these files in parallel. The engine does not write CSV files.

### List the Variables in a CSV File

The following example shows how to access a CSV file in HDFS and use the CONTENTS procedure to list the variables in the file. For this example, the first line in the CSV file lists the variables names. The GETNAMES data set option is used to read them from the first line in the file.

```
libname csvfiles sashdat host="grid001.example.com" install="/opt/TKGrid"
    path="/user/sasdemo/csv";

proc contents data=csvfiles.rep(filetype=csv getnames=yes);
run;
```

***Output 13.1*** *List the Variables in a CSV File with the CONTENTS Procedure*

**The SAS System**

**The CONTENTS Procedure**

| | | | |
|---|---|---|---|
| **Data Set Name** | /user/sasdemo/csv/rep.csv | **Observations** | . |
| **Member Type** | DATA | **Variables** | 6 |
| **Engine** | SASHDAT | **Indexes** | 0 |
| **Created** | Tuesday, July 03, 2012 08:53:36 AM | **Observation Length** | 208 |
| **Last Modified** | Thursday, June 28, 2012 02:48:41 PM | **Deleted Observations** | 0 |
| **Protection** | | **Compressed** | NO |
| **Data Set Type** | | **Sorted** | NO |
| **Label** | | | |
| **Data Representation** | SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64 | | |
| **Encoding** | Default | | |

**Alphabetic List of Variables and Attributes**

| # | Variable | Type | Len |
|---|---|---|---|
| 1 | alias | Char | 13 |
| 6 | enddt | Char | 10 |
| 3 | path | Char | 140 |
| 4 | set | Char | 1 |
| 5 | startdt | Char | 10 |
| 2 | id | Char | 7 |

### Convert a CSV File to SASHDAT

The engine is not designed to transfer data from HDFS to a SAS client. As a consequence, the contents of a CSV file can be accessed only by a SAS High-Performance Analytics procedure that runs on the same cluster that is used for HDFS. The SAS High-Performance Analytics procedures can read the data because the procedures are designed to read data in parallel from a co-located data provider.

The following code sample shows how to convert a CSV file to a SASHDAT file with the HPDS2 procedure.

```
option set=GRIDHOST="grid001.example.com"; 1
option set=GRIDINSTALLLOC="/opt/TKGrid";

libname csvfiles sashdat path="/user/sasdemo/csv";

proc hpds2 in=csvfiles.rep(filetype=csv getnames=yes) 2
    out=csvfiles.rephdat(path="/user/sasdemo" copies=0 blocksize=32m); 3

    data DS2GTF.out;
        method run();
            set DS2GTF.in;
        end;
    enddata;
run;
```

1 The values for the GRIDHOST and GRIDINSTALLLOC environment variables are read by the SASHDAT engine in the LIBNAME statement and by the HPDS2 procedure.

2 The FILETYPE=CSV data set option enables the engine to read the CSV file. The GETNAMES= data set option is used to read the variable names from the first line in the CSV file.

3 The PATH= data set option is used to store the output as **/user/sasdemo/rephdat.sashdat**. The COPIES=0 data set option is used to specify that no redundant blocks are created for the rephdat.sashdat file.

*Chapter 14*

# Data Set Options for the SASHDAT Engine



## Dictionary

### BLOCKSIZE= Data Set Option

specifies the block size to use for distributing the data set.

**Valid in:** DATA Step

**Default:** 2 megabytes

**Example:** "Example 6: Creating a SASHDAT File from Another SASHDAT File" on page 442

#### Syntax

**BLOCKSIZE=**

## Details

By default, the SASHDAT engine distributes data in 2-megabyte blocks or the length of a record, which ever is greater. You can override this value by specifying the block size to use. Suffix values are B (bytes), K (kilobytes), M (megabytes), and G (gigabytes). The actual block size is slightly larger than the value that you specify. This occurs for any of the following reasons:

- to reach the record length. This occurs if the specified size is less than the record length.
- to align on a 512-byte boundary.
- to include a metadata header in HDFS for the SASHDAT file.

The following code shows an example of specifying the BLOCKSIZE= option.

***Example Code 14.1*** *Using the BLOCKSIZE= Data Set Option*

```
data hdfs.sales (blocksize=48M);
    set yr2012.sales;
run;
```

# COLUMNS= Data Set Option

specifies the variable names and types for a CSV file.

**Alias:** COLS=

**Applies to:** Reading CSV files

## Syntax

**COLUMNS**=(*column-specification* < …*column-specification*>);

### *Required Argument*

**column-specification**
is a name-value pair that specifies the column name and data type. For numeric data, specify **`double`** as the data type. For character data, specify **`'char(`*`length`*`)'`**.

**Default** Any variables that are not named are assigned the name VAR*n*.

**Example**
```
columns=(station='char(4)' obsdate='char(18)' tempf=double precip=double)
```

## Details

Numeric variables use eight bytes. For character variables, if the byte length is not specified, then the default action is to use eight bytes. If the variable in the CSV file uses fewer bytes than the specified length, then the variable is padded with spaces up to the specified length. If the variable in the CSV file uses more bytes than the specified length, then the variable is truncated to the specified length.

If the variable name is not specified, then the variable is named automatically. Automatically named variables are named **`VAR`*`n`***, starting at 1. If the data type is not specified and cannot be determined, the variable is assigned as **`char(8)`**.

**TIP** Do not use a comma between each column specification. Enclose **`'char(`*`n`*`)'`** in quotation marks.

## COPIES= Data Set Option

specifies the number of replications to make for the data set (beyond the original blocks).

**Default:** 1

### Syntax

**COPIES=***n*

### Details

The default value is 1. This default value creates one copy of each block, in addition to the original block. When the INNAMEONLY option is specified, the default is 2. Replicated blocks are used to provide fault tolerance for HDFS. If a machine in the cluster becomes unavailable, then the blocks needed for the SASHDAT file can be retrieved from replications on other machines.

You can specify COPIES=0 to avoid creating redundant blocks for the SASHDAT file. This option can be useful to preserve storage space when you have redundancy for the source data.

## ENCRYPT= Data Set Option

specifies the AES encryption algorithm.

### Syntax

**ENCRYPT**=AES

### Details

Specifies to encrypt the file by using the AES (Advanced Encryption Standard) algorithm. AES provides enhanced encryption. You must specify the ENCRYPTKEY= data set option when you are using ENCRYPT=AES.

***CAUTION:***

**Record all ENCRYPTKEY= values when you are using ENCRYPT=AES.** If you forget to record the ENCRYPTKEY= value, you lose your data. SAS cannot assist you in recovering the ENCRYPTKEY= value. The following note is written to the log:

```
Note: If you lose or forget the ENCRYPTKEY= value, there will
be no way to open the file or recover the data.
```

## ENCRYPTKEY= Data Set Option

specifies a key value for AES (Advanced Encryption Standard) encryption.

**Note:** You cannot change the key value on an AES-encrypted data set without re-creating the data set.

## Syntax

**ENCRYPTKEY=***key-value*

## Details

Specifies to assign an encryption key value. You must specify the ENCRYPTKEY= data set option when you are using ENCRYPT=AES. The key value can be up to 64 bytes long. To create an ENCRYPTKEY= key value with or without quotation marks, follow these rules:

**No quotation marks:**

- use alphanumeric characters and underscores only
- can be up to 64 bytes long
- use uppercase and lowercase letters
- must start with a letter
- cannot include blank spaces
- is not case sensitive

```
%let mykey=abcdefghi12;
encryptkey=&mykey
encryptkey=key_value
encryptkey=key_value1
```

**Single quotation marks:**

- use alphanumeric, special, and DBCS characters
- can be up to 64 bytes long
- use uppercase and lowercase letters
- can include blank spaces, but cannot contain all blanks
- is case sensitive

```
encryptkey='key_value'
encryptkey='1234*#mykey'
```

**Double quotation marks:**

- use alphanumeric, special, and DBCS characters
- can be up to 64 bytes long
- use uppercase and lowercase letters
- can include blank spaces, but cannot contain all blanks
- is case sensitive

```
encryptkey="key_value"
encryptkey="1234*#mykey"
%let mykey=Abcdefghi12;
encryptkey="&mykey"
```

When the ENCRYPTKEY= key value uses DBCS characters, the 64-byte limit applies to the character string after it has been transcoded to UTF-8 encoding. You can use the following DATA step to calculate the length in bytes of a key value in DBCS:

```
data _null_;
    key=length(unicodec('key-value','UTF8'));
    put 'key length=' key;
run;
```

## FILETYPE= Data Set Option

specifies whether to access a comma-separated value (CSV) file instead of a SASHDAT file.

**Applies to:** Reading CSV files

### Syntax

**FILETYPE**=CSV

### Details

The SASHDAT engine can be used to read CSV files. The engine does not write CSV files. Specify this option to use the file as input for a SAS High-Performance Analytics procedure or the SAS LASR Analytic Server.

The filename for CSV files in HDFS can be uppercase, mixed case, or lowercase. If more than one file in the directory has the same name (but with different casing), the engine does not read the file because the file reference is ambiguous.

### See Also

- COLUMNS= data set option
- GETNAMES data set option
- GUESSROWS= data set option

## GETNAMES= Data Set Option

specifies to read variable names from the first line in the CSV file.

**Applies to:** Reading CSV files

### Syntax

**GETNAMES=** YES | NO

### Details

Specify GETNAMES=YES if the first line of a CSV file contains the variable names for the file. Alternatively, you can specify the variable names in the COLUMNS= data set option, or you can use the default names that are provided by the SASHDAT engine.

## GETOBS= Data Set Option

specifies to retrieve the number of observations in SASHDAT files.

## Syntax

**GETOBS**= YES | NO

## Details

By default, the SASHDAT engine does not compute the number of observations in a SASHDAT file. This improves performance for SASHDAT files that are distributed among a large number of blocks, or for HDFS directories that have a large number of SASHDAT files. When you specify GETOBS=YES, the engine calculates the number of observations in a SASHDAT file.

```
ods select attributes;

proc datasets library=hdfs;
    contents data=sales2012(getobs=yes);
run;
```

**The SAS System**

**The DATASETS Procedure**

| | | | |
|---|---|---|---|
| **Data Set Name** | /user/sasdemo/sales2012.sashdat | **Observations** | 100000 |
| **Member Type** | DATA | **Variables** | 88 |
| **Engine** | SASHDAT | **Indexes** | 0 |
| **Created** | Thursday, June 28, 2012 09:47:45 AM | **Observation Length** | 704 |
| **Last Modified** | Wednesday, June 27, 2012 04:06:40 PM | **Deleted Observations** | 0 |
| **Protection** | | **Compressed** | NO |
| **Data Set Type** | | **Sorted** | NO |
| **Label** | | | |
| **Data Representation** | SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64 | | |
| **Encoding** | wlatin1 Western (Windows) | | |

# GUESSROWS= Data Set Option

specifies the number of lines in CSV file to scan for determining variable types and lengths.

**Default:** 20

**Applies to:** Reading CSV files

## Syntax

**GUESSROWS**=*n*

## Details

The SASHDAT engine scans the specified number of lines from the CSV file to determine the variable types and lengths. If the GETNAMES data set option is specified, then the engine begins scanning lines from the second line in the file.

## HASH= Data Set Option

specifies that when partitioning data, the distribution of partitions is not determined by a tree, but by a hashing algorithm. As a result, the distribution of the partitions is not as evenly balanced, but it is effective when working with high-cardinality partition keys (in the order of millions of partitions).

### Syntax

PARTITION=(*variable-list*) **HASH=** YES | NO

### Example

```
data hdfs.transactions(partition=(cust_id year) hash=yes);
    set somelib.sometable;
run;
```

## LOGUPDATE= Data Set Option

specifies to provide progress messages in the SAS log about the data transfer to the grid host.

### Syntax

**LOGUPDATE=** YES | NO

### Details

The data transfer size is not necessarily the same as the block size that is used to form blocks in HDFS. The data transfer size is selected to optimize network throughput. A message in the SAS log does not mean that a block was written to HDFS. The message indicates the transfer progress only.

```
data hdfs.sales2012(logupdate=yes);
    set saleslib.sales2012;
run;
```

```
NOTE: 4096 kBytes (5191 records) have been transmitted (1.91 MB/sec).
NOTE: 8192 kBytes (10382 records) have been transmitted (3.65 MB/sec).
NOTE: 12288 kBytes (15573 records) have been transmitted (5.19 MB/sec).
NOTE: 16384 kBytes (20764 records) have been transmitted (6.15 MB/sec).
NOTE: 20480 kBytes (25955 records) have been transmitted ( 7.3 MB/sec).
NOTE: 24576 kBytes (31146 records) have been transmitted (8.16 MB/sec).
NOTE: 28672 kBytes (36337 records) have been transmitted (8.83 MB/sec).
NOTE: 32768 kBytes (41528 records) have been transmitted (9.73 MB/sec).
NOTE: 36864 kBytes (46719 records) have been transmitted (10.3 MB/sec).
NOTE: 40960 kBytes (51910 records) have been transmitted (10.8 MB/sec).
NOTE: 45056 kBytes (57101 records) have been transmitted (11.6 MB/sec).
NOTE: 49152 kBytes (62292 records) have been transmitted (  12 MB/sec).
NOTE: 53248 kBytes (67483 records) have been transmitted (12.4 MB/sec).
NOTE: 57344 kBytes (72674 records) have been transmitted (12.9 MB/sec).
NOTE: 61440 kBytes (77865 records) have been transmitted (13.2 MB/sec).
NOTE: 65536 kBytes (83056 records) have been transmitted (13.5 MB/sec).
NOTE: 69632 kBytes (88247 records) have been transmitted (13.9 MB/sec).
NOTE: 73728 kBytes (93438 records) have been transmitted (14.1 MB/sec).
NOTE: 77824 kBytes (98629 records) have been transmitted (14.3 MB/sec).
NOTE: There were 100000 observations read from the data set SALESLIB.YEAR2012.
NOTE: The data set /user/sasdemo/sales2012 has 100000 observations and 86
variables.
NOTE: 78906 kBytes (100000 records) have been transmitted (14.3 MB/sec).
```

## ORDERBY= Data Set Option

specifies the variables by which to order the data within a partition.

**Example:** "Example 4: Adding a Table to HDFS with Partitioning" on page 442

### Syntax

**ORDERBY=**(*variable-list*)

**ORDERBY=**(*variable-name* <DESCENDING> *variable-name*)

### Details

The variable names in the *variable-list* are separated by spaces.

The ordering is hierarchical. For example, ORDERBY=(A B) specifies ordering by the values of variable B within the ordered values of variable A. The specified variables must exist and cannot be specified as partitioning variables. The order is determined based on the raw value of the variables and uses locale-sensitive collation for character variables. By default, values are arranged in ascending order. You can specify descending order by preceding the variable name in the *variable-list* with the keyword DESCENDING.

### Example

The following code sample orders the data in the partitions by Year in ascending order and then by Quarter in descending order.

```
data hdfs.prdsale (partition=(country region)
                   orderby=(year descending quarter));
    set sashelp.prdsale;
run;
```

## PARTITION= Data Set Option

specifies the list of partitioning variables to use for partitioning the table.

**Interaction:** If you specify the PARTITION= option and the BLOCKSIZE= option, but the block size is less than the calculated size that is needed for a block, the operation fails and the table is not added to HDFS. If you do not specify a block size, the size is calculated to accommodate the largest partition.

**Example:** "Example 4: Adding a Table to HDFS with Partitioning" on page 442

### Syntax

**PARTITION=**(*variable-list*)

### Details

Partitioning is available only when you add tables to HDFS. If you partition the table when you add it to HDFS, it becomes a partitioned in-memory table when you load it to SAS LASR Analytic Server. If you also specify the ORDERBY= option, then the ordering is preserved when the table is loaded to memory too.

Partition keys are derived based on the formatted values in the order of the variable names in the *variable-list*. All of the rows with the same partition key are stored in a single block. This ensures that all the data for a partition is loaded into memory on a single machine in the cluster. The blocks are replicated according to the default replication factor or the value that you specify for the COPIES= option.

If user-defined formats are used, then the format name is stored with the table, but not the format. The format for the variable must be available to the SAS LASR Analytic Server when the table is loaded into memory. This can be done by having the format in the format catalog search path for the SAS session.

Be aware that the key construction is not hierarchical. That is, PARTITION=(A B) specifies that any unique combination of formatted values for variables A and B defines a partition.

Partitioning by a variable that does not exist in the output table is an error. Partitioning by a variable listed in the ORDERBY= option is also an error.

## PATH= Data Set Option

specifies the fully qualified path to the HDFS directory to use for SASHDAT files.

### Syntax

PATH='*HDFS-path*'

### Details

This option overrides the PATH= option specified in the LIBNAME statement.

## PERM= Data Set Option

specifies how the engine sets the file access permissions on the SASHDAT file

**Alias:** PERMISSION=

### Syntax

**PERM=***mode*

### Details

The *mode* value is specified as an integer value such as 755. The mode corresponds to the mode values that are used for UNIX file access permissions.

Hadoop UMASK configuration settings can sometimes result in more restrictive access than the settings that you specified. You might need to review the permissions with HDFS utilities.

## REPLACE= Data Set Option

specifies whether to overwrite an existing SASHDAT file.

### Syntax

**REPLACE=**YES | NO

### Details

By default, the SASHDAT engine does not replace SASHDAT files. Specify REPLACE=YES as a data set option to replace a SASHDAT file by overwriting it.

## SIGNERFILEPOLICY Data Set Option

specifies to apply the encryption policies stored in metadata to the file.

### Syntax

SIGNERFILEPOLICY

### Details

When a table is written with the SASHDAT engine engine or saved with the SAVE statement in the IMSTAT procedure, the software does not check with metadata for the following:

- whether the Hadoop server and path are associated with a library
- whether a passphrase for encryption is associated with the library

If you specify SIGNERFILEPOLICY and the SIGNER= option, then the software checks whether the file should be encrypted. If it should be, the server encrypts the file with the passphrase that is stored in metadata.

## SQUEEZE= Data Set Option

specifies to add the table to HDFS in compressed form.

### Syntax

SQUEEZE= YES | NO

## UCA= Data Set Option

specifies that you want to use Unicode Collation Algorithms (UCA) to determine the ordering of character variables in the ORDERBY= option.

### Syntax

PARTITION=(*key*) ORDERBY=(*variable-list*) **UCA=** YES | NO

*Chapter 15*

# Programming with SAS LASR Analytic Server



## About Programming

When programming with SAS LASR Analytic Server, it is important to understand where the computation occurs and memory utilization.

- The IMSTAT procedure always performs the computation in the server, and the analysis is performed against the original in-memory table.
- Other procedures (for example, FREQ, UNIVARIATE, and RANK) transfer the in-memory table to the client machine. After the transfer, the session on the client machine performs the analysis on the copy of the data.
- Most DATA step programs operate by transferring the in-memory table to the client and then performing the computation. However, if a DATA step is written to use in-memory tables for input and output, the DATA step can run in-memory, with restrictions. The next section describes how to use this feature.

## DATA Step Programming for Scoring In SAS LASR Analytic Server

### *Scoring In-Memory Tables Using DATA Step Processing*

The DATA step can process in-memory tables for scoring with limitations:

- Only one input file and one output file is allowed.
- Only functions and formats that are supported by the DS2 language compile successfully.
- Some DATA step statements are not allowed, such as those pertaining to input and output.

For more information, see "Requirements for LASR Score Mode DATA Step Processing" on page 459 and "Restrictions in DATA Step Processing" on page 460.

To enable the DATA step to score in-memory SAS tables, set the system option DSACCEL=ANY.

If a SAS program does not meet the requirements for running in the SAS LASR Analytic Server, SAS writes informational or error messages to the log and executes the code in your Base SAS session. In this case, SAS reads and writes large tables over the network.

You can determine whether your code is compliant with the SAS LASR Analytic Server compiler by setting the system option MSGLEVEL= to I. When MSGLEVEL=I, SAS writes log messages that identify the non-compliant code.

### Example 1: A DATA Step Program For SAS LASR Analytic Server

This example demonstrates executing a DATA step program in the SAS LASR Analytic Server:

```
/* Enable DATA step parallel processing using the system option   */
/* and enable messages to view non-compliant code in the SAS log. */
options dsaccel=any msglevel=i;

/* Create a libref for in-memory tables. */
libname lasr sasiola host="grid001.example.com" port=10010 tag='hps';

/* Create a libref for the input data that is stored on disk. */
libname score '/myScoreData/';

/* Load the input table into memory */
data lasr.intr;
   set score.intrid;
run;

/* Execute the score code using the in-memory tables. */
/* Both tables must use the same libref.              */
data lasr.sumnormtable;
   set lasr.intr;

/* Execute the score code. */
if sum > 1000
  then score=1;

run;
```

### Example 2: Using User-Defined Formats with In-Memory Tables

You can use user-defined formats in a DATA step by using the CATALOG procedure to copy the format catalog to a library. This example copies the format library to Work:

```
/* Enable DATA step parallel processing using the system option   */
/* and enable messages to view non-compliant code in the SAS log. */

options dsaccel=any msglevel=i;

/* Create a libref for the in-memory tables. */
libname lasr sasiola host="grid001.example.com" port=10010 tag='hps';

/* Create a libref for the input data and format catalog that is  */
/* stored on disk.                                                */
libname score '/myScoreData/';

/* Copy the demx format catalog to the Work library */
proc catalog catalog=score.dmex;
   copy out=work.formats;
quit;

/* Enable in-memory processing (dsaccel) and load the input table */
/* into memory.                                                   */
data lasr.dmex;
   set score.dmex;
run;

/*  Enable in-memory processing (dsaccel) and execute the score code .*/
/*  using the in-memory tables.                                       */
data lasr.dmexout;
   set lasr.dmex;
   %inc "dmex.sas";
run;
```

SAS automatically searches the Work and Library libraries for a format catalog. If you copy the format library to a library other than Work or Library, then you must use the FMTSEARCH= system option to let SAS know the location of the format library.

```
options fmtsearch=(myFmtLib);
```

You must also specify the FMTSEARCH= system option if the format catalog name is not **format:**

```
options fmtsearch=(myFmtLib.myFmts);
```

### Requirements for LASR Score Mode DATA Step Processing

In order to score in-memory tables in SAS LASR Analytic Server, the following is required:

- The DSACCEL=ANY system option is set.
- The code must contain a LIBNAME statement using the SASIOLA engine.
- The input and output tables must use the same libref for the SASIOLA engine.
- The DATA statement must be followed immediately by the SET statement.

  This example demonstrates these requirements:

```
libname lasr sasiola;
data lasr.out;
```

```
    set lasr.in;
    /* DATA step code */
run;
```

### Restrictions in DATA Step Processing

Here are the restrictions for using the DATA step in SAS LASR Analytic Server:

- More than one SET statement is not supported. SET statement options are not allowed.
- The statements in the following table are not supported as part of a DATA step. The Alternative column provides possible alternatives, if one is available.

| DATA Step Statement | Alternative |
|---|---|
| BY (or FIRST. and LAST. variables) | See the "GROUPBY Statement" on page 160 or "Using Partitioning and Scoring" on page 335. |
| CARDS and CARDS4 | |
| CONTINUE | |
| DISPLAY | |
| ERROR | |
| FILE | |
| Subsetting IF | Two options are available. You can write a program that includes a subsetting IF and use the SCORE statement. Another option is use to use the BALANCE statement. See the example on page 462. |
| INFILE | |
| INPUT | |
| LEAVE | |
| LIST | |
| LOSTCARD | |
| MERGE | You might be able to use the "SCHEMA Statement" on page 329 to produce similar results. |
| MISSING | |
| MODIFY | |
| OUTPUT | You can use the **__lasr_output** variable in a program with the SCORE statement to control when to output an observation. You can write observations to a single output table only. |

| DATA Step Statement | Alternative |
|---|---|
| PUT | A PUT statement in a program that is used with the SCORE statement writes entries to a temporary table instead of to the SAS log. You can view the results of the PUT statements by specifying the PGMMSG option in the IMSTAT statement and then viewing the contents of the &_PGMMSG_ temporary table. For an example, see "Example 2: Using the RETAIN Statement" on page 339. |
| REMOVE | |
| RENAME | |
| REPLACE | |
| RETAIN | You can specify the DSRETAIN option to the SCORE statement and use the RETAIN statement in your program. See "Example 2: Using the RETAIN Statement" on page 339. |
| UPDATE | You can use the DATA= option with the UPDATE statement to make changes to a table. However, this statement updates the active table rather than creating a new table. See "Example 2: Update with a Data Set" on page 360. |
| WHERE | The statements in the IMSTAT procedure that operate on observations are subject to a WHERE statement. To create a new in-memory table from a subset of rows in the active table, you can specify a WHERE clause and then use the BALANCE statement. |
| WINDOW | |

- The ABORT statement has these restrictions:
  - The ABORT statement does not accept arguments.
  - The ABORT statement is not supported within functions. It is valid only in the main program.
- The DO statement does not accept a character index. For example, the following statement results in an error: **`do month='JAN','FEB','MAR';`**.
- These functions are not supported:
  - DIF
  - LAG
- The INPUT function does not support the question mark (?) and double question mark (??) modifiers.
- Some CALL routines are not supported. Routines are supported if there is an equivalent function.
- You can use only SAS formats and functions that are supported by the DS2 language. These formats and functions are documented in *SAS DS2 Language Reference*.
- Component objects are not supported.
- Scoring input variables cannot be modified.

# Which to Use? DATA Step or the SCORE Statement

## Advantages, Disadvantages, and Best Practices

The following list highlights some of the advantages and disadvantages for each approach:

- If you know that you want to run code in the server, then using the SCORE statement to the IMSTAT procedure provides assurance. Either the code runs in the server and the statement succeeds or the statement fails. Large tables are never transferred from the server to the client for processing.
- If you want to use the **__first_in_partition** or **__last_in_partition** variables, you must use the SCORE statement. Those variables can be used to provide some similarity to BY-group processing with FIRST. and LAST. variables.
- To use the DATA step and keep processing in the server, your program must use the DSACCEL=ANY system option and meet the requirements that are commonly associated with scoring. In this case, the DATA step determines that the requirements are met and performs the optimization of running in the server.
- Error messages are more helpful and troubleshooting is easier when you are using the DATA step than when you are running a program with the SCORE statement.

As a best practice, initial development with the DATA step provides rapid development. After the program is complete, or nearly so, switching to run the program with the SCORE statement provides assurance that the program runs in the server.

When you work with the SCORE action, use the IMSTAT procedure's PGMMSG option so that the server records any error information. You can read this information in the &_PGMMSG_ temporary table. PUT statements in your program are written to that table rather than the SAS log.

As a final consideration, before writing a DATA step program, check the list of alternatives in the preceding section to see whether a statement for the IMSTAT procedure can accomplish your goal. The GROUPBY, AGGREGATE, and TRANSFORM statements can be used to perform data preparation on in-memory tables.

## Using the BALANCE Statement to Subset a Table

All the statements for the IMSTAT procedure that operate on rows of data support WHERE expression processing. For example, the following code generates summary statistics for all cars with more than 300 horsepower:

```
libname example sasiola host="grid001.example.com" port=10010 tag="hps";

data example.cars; set sashelp.cars; run;

proc imstat;
    table example.cars;
    where horsepower > 300;
    summary;
run;
```

Using a WHERE expression to subset data is a best practice because it is efficient and avoids duplicating data in memory. However, if you must create a new table, you can use the BALANCE statement with a WHERE expression:

```
proc imstat;
    table example.cars;
    where horsepower > 300;
    summary horsepower cylinders invoice;  1
run;

    balance;  2
run;

    table example.&_TEMPLAST_;
    numrows;  3
run;

    promote cars_gt_300_hp;  4
run;

    table example.cars_gt_300_hp;
    summary horsepower cylinders invoice;
run;
```

1 The first SUMMARY statement is subject to the WHERE expression. This is the most efficient way to work with tables and should be used whenever possible. The results show that the minimum value is 302.

2 The BALANCE statement creates a new temporary table. The Cars table is still the active table and it is still subject to the WHERE expression. For distributed servers, the observations in the temporary table are distributed amongst the machines in the cluster evenly for an even workload.

3 The NUMROWS action is used in this example to demonstrate that the number of observations in the new temporary table matches the value of N from the first SUMMARY statement.

4 The PROMOTE statement makes the temporary table into a regular table and gives it a name. It must be set as the active table with the TABLE statement. The final SUMMARY statement demonstrates that the results are identical to the first SUMMARY statement.

| Summary Statistics for Table HPS.CARS | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Column | Min | Max | N | Sum | Mean | Std Dev. | Std Error | Coefficient of Variation | Number Missing |
| Horsepower | 302.00 | 500.00 | 50 | 17502 | 350.04 | 56.9271 | 8.0507 | 16.2630 | 0 |
| Cylinders | 6.0000 | 12.0000 | 50 | 400 | 8.0000 | 1.3401 | 0.1895 | 16.7515 | 0 |
| Invoice | 24926 | 173560 | 50 | 3027183 | 60544 | 26931 | 3808.69 | 44.4828 | 0 |

| Temporary Table Information for Table HPS.CARS | |
|---|---|
| Statement | BALANCE |
| Temporary Table | _T_6239F22E_7F80B30E2C28 |
| Table Type | BALANCE |

| Number of Rows Action for Table _T_6239F22E_7F80B30E2C28 |
|---|
| Number of Records |
| 50 |

| Summary Statistics for Table HPS.CARS_GT_300_HP | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Column | Min | Max | N | Sum | Mean | Std Dev. | Std Error | Coefficient of Variation | Number Missing |
| Horsepower | 302.00 | 500.00 | 50 | 17502 | 350.04 | 56.9271 | 8.0507 | 16.2630 | 0 |
| Cylinders | 6.0000 | 12.0000 | 50 | 400 | 8.0000 | 1.3401 | 0.1895 | 16.7515 | 0 |
| Invoice | 24926 | 173560 | 50 | 3027183 | 60544 | 26931 | 3808.69 | 44.4828 | 0 |

*Chapter 16*

# Text Analytics in SAS LASR Analytic Server



## SAS Linguistic Files

The text analytic capabilities of the server depend on accessing SAS linguistic files. The server uses these files to perform the parsing, term extraction, stemming, tagging of terms, and so on. The files must be made available to the server and the path must be specified in either the **TKTXTANIO_BINDAT_DIR** or **TKPARSE_BINDAT_DIR** environment variables. (See the examples.)

Information about installing the linguistic files for distributed deployments is provided in *SAS High-Performance Analytics Infrastructure: Installation and Configuration Guide*. A common location is **`/opt/TKTGDat`**, but the location can be customized at your site.

For non-distributed deployments, the files are installed with SAS as follows:

*Windows Specifics*

**`\SASFoundation\9.4\tktg\sasmisc`**

*UNIX Specifics*

```
/SASFoundation/9.4/misc/tktg
```

# Language Processing Concepts

## Stemming

Stemming identifies the possible root form of an inflected word. For example, the word talk is the stem of the words talk, talks, talking, and talked. In this case talk is the parent, and talk, talks, talking, and talked are its children.

## Tagging

Tagging disambiguates the grammatical category of a word by analyzing it in context. For example, consider the following sentence:

```
I like to bank at the local branch of my bank.
```

The first bank is tagged as a verb and the second bank is tagged as a noun. The possible speech tags that you might see are as follows:

| Tag | Description |
|---|---|
| ABBR | Abbreviation |
| ADJ | Adjective |
| ADV | Adverb |
| AUX | Auxiliary or modal term |
| CONJ | Conjunction |
| DET | Determiner |
| INTERJ | Interjection |
| NOUN | Noun |
| NOUN_GROUP | Compound noun |
| NUM | Number or numeric expression |
| PART | Infinitive marker, negative participle, or possessive marker |
| PREF | Prefix |
| PREP | Preposition |
| PROP | Proper noun |

| Tag | Description |
|---|---|
| PUNCT | Punctuation |
| VERB | Verb |
| VERBADJ | Verbal adjective |

## Noun Group Extraction

Noun groups provide more relevant information than simple nouns. A noun group is defined as a sequence of nouns and their modifiers. Noun group extraction uses part-of-speech tagging to identify nouns and their related words that together form a noun group. Examples of noun groups are "week-long cruises" and "Middle Eastern languages."

## Entity Identification

Entity identification uses SAS linguistic technologies to classify sequences of words into predefined classes. These classes are assigned as roles for the corresponding sequences. For example, "Person," "Location," "Company," and "Measurement" are identified as classes for "George W. Bush," "Boston," "SAS Institute," "2.5 inches," respectively. The following table lists the possible entities for English.

| Entity | Description |
|---|---|
| ADDRESS | Postal address or number and street name |
| COMPANY | Company name |
| CURRENCY | Currency or currency expression |
| INTERNET | Email address or URL |
| LOCATION | City, county, state, political or geographical place or region |
| MEASURE | Measurement or measurement expression |
| NOUN_GROUP | Phrases that contain multiple words |
| ORGANIZATION | Government, legal, or service agency |
| PERCENT | Percentage or percentage expression |
| PERSON | Person’s name |
| PHONE | Telephone number |
| PROP_MISC | Proper noun with an ambiguous classification |
| SSN | Social Security number |

| Entity | Description |
|---|---|
| TIME | Time or time expression |
| TIME_PERIOD | Measure of time expressions |
| TITLE | Person's title or position |
| VEHICLE | Motor vehicle, including color, year, make, and model |

# Data Preparation

### *Width Limitation for Text Fields*

Character variables in SAS data sets cannot exceed 32K. So, when using the TEXTPARSE statement in a SAS program, you are limited to 32K for a text fields.

### *Encoding*

The linguistic data files and libraries are sensitive to the NLS encoding. The encoding used in the TEXTPARSE statement is derived from the encoding of the active table that you parse.

### *A Document ID Variable Is Required*

The TEXTPARSE statement requires that each document is uniquely identified by a column in the table. This column essentially holds a record ID in the table, as each row is considered a separate document.

# Output Tables for the TEXTPARSE Statement

### *Sample Data and Program*

The TEXTPARSE statement generates a summary table and up to seven temporary tables. The following program provides sample data and statements for getting started and then showing the basic layout for each of the temporary tables.

```
data getstart;
infile cards delimiter='|' missover;
length text $150;
input text$ docid$;

cards;
High-performance analytics hold the key to |d01
unlocking the unprecedented business value of big data.|d02
Organizations looking for optimal ways to gain insights|d03
```

```
from big data in shorter reporting windows are turning to SAS.|d04
As the gold-standard leader in business analytics |d05
for more than 36 years,|d06
SAS frees enterprises from the limitations of |d07
traditional computing and enables them |d08
to draw instant benefits from big data.|d09
Faster Time to Insight.|d10
From banking to retail to health care to insurance, |d11
SAS is helping industries glean insights from data |d12
that once took days or weeks in just hours, minutes, or seconds.|d13
It's all about getting to and analyzing relevant data faster.|d14
Revealing previously unseen patterns, sentiments, and relationships.|d15
Identifying unknown risks.|d16
And speeding the time to insights.|d17
High-Performance Analytics from SAS Combining industry-leading |d18
analytics software with high-performance computing technologies|d19
produces fast and precise answers to unsolvable problems|d20
and enables our customers to gain greater competitive advantage.|d21
SAS In-Memory Analytics eliminate the need for disk-based processing|d22
allowing for much faster analysis.|d23
SAS In-Database executes analytic logic into the database itself |d24
for improved agility and governance.|d25
SAS Grid Computing creates a centrally managed,|d26
shared environment for processing large jobs|d27
and supporting a growing number of users efficiently.|d28
Together, the components of this integrated, |d29
supercharged platform are changing the decision-making landscape|d30
and redefining how the world solves big data business problems.|d31
Big data is a popular term used to describe the exponential growth,|d32
availability and use of information,|d33
both structured and unstructured.|d34
Much has been written on the big data trend and how it can |d35
serve as the basis for innovation, differentiation and growth.|d36
run;

options set=TKTXTANIO_BINDAT_DIR="/opt/TKTGDat";

libname example sasiola host="grid001.example.com" port=10010 tag=hps;

data example.getstart;
  set getstart;
run;

proc imstat data=example.getstart;
  textparse var=text docid=docid entities=std reducef=2
            select=(_all_) save=txtsummary;
run;
```

Computing singular value decomposition requires the input data to contain at least 25 documents and at least as many documents as there are machines in the cluster. By default, REDUCEF=4 but in this example is set to 2 to specify that a word only needs to appear twice to be kept for generating the term-by-document matrix. The default dimension for the singular-value decomposition is $k$=10 and the server generates ten topics.

The TEXTPARSE statement produces the following output. The names of the temporary tables are reported and begin with _T_. These names (and the rest of the output in the table) is stored in a temporary buffer that is named TXTSUMMARY.

**The IMSTAT Procedure**

| Text Parsing Summary | |
|---|---|
| Data Source | HPS.GETSTART |
| Document Variable | docid |
| Text Variable | text |
| Number of Documents | 36 |
| Number of Terms | 31 |
| Term Information | _T_DBDFC26_7F21541ABD38 |
| Transactional Terms (Bag of Words) | _T_DBDFC2B_7F21541ABE58 |
| V Matrix from SVD | _T_DBDFC78_7F21541ABBF8 |
| U Matrix from SVD | _T_DBDFC68_7F21541ABC08 |
| Topics | _T_DBDFC80_7F21541ABDC8 |
| Terms and Topics | _T_DBDFC88_7F21541ABDC8 |
| Document Projection | _T_DBDFC92_7F21541ABC98 |

Because the IMSTAT procedure was used in interactive mode with a RUN statement instead of QUIT, the STORE statement can be used to create macro variables for the temporary table names as follows:

```
    store txtsummary( 6,2) as Terms;
    store txtsummary( 7,2) as Parent;
    store txtsummary( 8,2) as V;
    store txtsummary( 9,2) as U;
    store txtsummary(10,2) as Topics;
    store txtsummary(11,2) as TermsByTopics;
    store txtsummary(12,2) as DocPro;
  run;
```

Each of the tables can then be accessed with a libref and the macro variable, such as **`example.&Terms.;`**. The following sections show how to access these tables.

## Terms Table

Based on the statements in the previous section, you can access the terms table as follows:

```
    table example.&Terms.;
    numrows;
    columninfo;
    where _ispar ne ".";
    fetch / format to=10;
  run;
```

*Note:* Filtering out the observations where _Ispar is missing results in showing only the terms that are used in the subsequent singular-value decomposition and topic generation.

The following output shows the number of observations in the Terms table, the data structure, and details about the terms that are identified by the TEXTPARSE statement.

| Number of Rows Action for Table _T_DBDFC26_7F21541ABD38 |
|---|
| Number of Records |
| 43 |

| Column Information for Table _T_DBDFC26_7F21541ABD38 | | | | |
|---|---|---|---|---|
| Id | Column | Type | Length | Format |
| 1 | Term | Char | 39 | $39. |
| 2 | Role | Char | 11 | $11. |
| 3 | Attribute | Char | 6 | $6. |
| 4 | Freq | Num | 8 | BEST12. |
| 5 | NumDocs | Num | 8 | BEST12. |
| 6 | _keep | Char | 1 | $1. |
| 7 | Key | Num | 8 | BEST12. |
| 8 | Parent | Num | 8 | BEST12. |
| 9 | Parent_id | Num | 8 | BEST12. |
| 10 | Weight | Num | 8 | BEST12. |
| 11 | _ispar | Char | 1 | $1. |

| Selected Records from Table _T_DBDFC26_7F21541ABD38 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Term | Role | Attribute | Freq | NumDocs | _keep | Key | Parent | Parent_id | Weight | _ispar |
| sas | COMPANY | Entity | 7 | 7 | Y | 1 | . | 1 | 0.4569834337 | |
| high-performance | Adj | Mixed | 3 | 3 | Y | 2 | . | 2 | 0.6934264036 | |
| big data | NOUN_GROUP | Alpha | 5 | 5 | Y | 3 | . | 3 | 0.5508777991 | |
| to | Prep | Alpha | 13 | 11 | Y | 4 | . | 4 | 0.3549850065 | |
| fast | Adj | Alpha | 4 | 4 | Y | 5 | 5 | 5 | 0.6131471928 | + |
| data | Noun | Alpha | 8 | 8 | Y | 6 | . | 6 | 0.4197207891 | |
| gain | Verb | Alpha | 2 | 2 | Y | 7 | . | 7 | 0.8065735964 | |
| big | Adj | Alpha | 6 | 6 | Y | 8 | . | 8 | 0.5 | |
| for | Prep | Alpha | 7 | 7 | Y | 9 | . | 9 | 0.4569834337 | |
| problem | Noun | Alpha | 2 | 2 | Y | 10 | 10 | 10 | 0.8065735964 | + |

## *Parent Table*

The Parent table is also known as the bag of words or as the term document. It is a sparse representation of the term by document weights. It is the input to the singular-value decomposition (SVD). The SVD then reduces the dimensionality of the problem

by focusing on the *k* dimensions with the largest singular values. It is essentially a dimension reduction technique.

```
    where; /* clear the _ispar filter that was in use */
    table example.&Parent.;
    numrows;
    columninfo;
    fetch / format to=10;
  run;
```

| Number of Rows Action for Table _T_DBDFC2B_7F21541ABE58 |
|---|
| Number of Records |
| 133 |

| Column Information for Table _T_DBDFC2B_7F21541ABE58 | | | | |
|---|---|---|---|---|
| Id | Column | Type | Length | Format |
| 1 | _termnum_ | Num | 8 | BEST12. |
| 2 | _id_ | Num | 8 | BEST12. |
| 3 | _count_ | Num | 8 | BEST12. |
| 4 | docid | Char | 8 | $8. |

| Selected Records from Table _T_DBDFC2B_7F21541ABE58 | | | |
|---|---|---|---|
| _termnum_ | _id_ | _count_ | docid |
| 2 | 1 | 0.6934264036 | d01 |
| 4 | 1 | 0.3549850065 | d01 |
| 14 | 1 | 0.2842372535 | d01 |
| 23 | 1 | 0.8065735964 | d01 |
| 31 | 1 | 0.6934264036 | d01 |
| 3 | 2 | 0.5508777991 | d32 |
| 4 | 2 | 0.3549850065 | d32 |
| 6 | 2 | 0.4197207891 | d32 |
| 8 | 2 | 0.5 | d32 |
| 14 | 2 | 0.2842372535 | d32 |

There are 133 nonzero entries in the term × document matrix. The full matrix would be a 43 × 36 matrix.

There are four columns in the parent table. The _termnum_, and _id_ column are the term number and an internal identifier. Values in the _termnum_ column correspond to values in the Key column in the Terms table. The _count_ column represents the term

weight. The last column is the document ID that corresponds to the DOCID= variable specified in the TEXTPARSE statement. In this example, it is named Docid.

### *SVD V Table*

```
    table example.&V.;
    numrows;
    columninfo;
    fetch / format to=10;
  run;
```

This table is one of the results of the singular-value decomposition of the sparse parent table (bag of words). The number of rows equals the number of documents in the input table. The number of columns equals the number of topics (the K= value of the SVD option) plus one for an ID variable.

| Number of Rows Action for Table _T_DBDFC78_7F21541ABBF8 |
|---|
| Number of Records |
| 35 |

| Column Information for Table _T_DBDFC78_7F21541ABBF8 | | | | |
|---|---|---|---|---|
| Id | Column | Type | Length | Format |
| 1 | docid | Char | 8 | $8. |
| 2 | Col1 | Num | 8 | BEST12. |
| 3 | Col2 | Num | 8 | BEST12. |
| 4 | Col3 | Num | 8 | BEST12. |
| 5 | Col4 | Num | 8 | BEST12. |
| 6 | Col5 | Num | 8 | BEST12. |
| 7 | Col6 | Num | 8 | BEST12. |
| 8 | Col7 | Num | 8 | BEST12. |
| 9 | Col8 | Num | 8 | BEST12. |
| 10 | Col9 | Num | 8 | BEST12. |
| 11 | Col10 | Num | 8 | BEST12. |

| Selected Records from Table _T_DBDFC78_7F21541ABBF8 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| docid | Col1 | Col2 | Col3 | Col4 | Col5 | Col6 | Col7 | Col8 | Col9 | Col10 |
| d01 | -0.001368478 | -0.031297635 | 0.0328846275 | 0.6594311396 | 0.159181427 | 0.0506019606 | 0.0817482516 | 0.0470631294 | 0.0077070783 | 0.1250205795 |
| d32 | 0.317352362 | -0.011440566 | 0.0078065057 | 0.0230516889 | 0.0507572793 | 0.7896425226 | 0.1325404705 | 0.0038832324 | 0.1406853732 | 0.0297125694 |
| d02 | 0.2817338798 | -0.021222863 | -0.000092079 | -0.01277088 | 0.1467155005 | 0.008094056 | -0.002375997 | -0.008860092 | 0.4960222023 | -0.013213389 |
| d33 | 0.0153020597 | 0.0064652449 | 7.4621768E-6 | -0.009665554 | -0.036833117 | 0.0166790643 | 0.0227357505 | 0.0602769036 | 0.2684938624 | 0.0263542808 |
| d03 | -0.015462638 | 0.1142344439 | 0.0363207085 | -0.057313647 | 0.0314318136 | 0.0488560586 | 0.0240009999 | 0.2903061608 | -0.062714708 | 0.6920315292 |
| d34 | 0.0092290728 | 0.0129301298 | -0.03050947 | -0.001446302 | -0.004478517 | -0.000490911 | 0.0305378609 | 0.0477870274 | 0.0258371312 | 0.0229022487 |
| d04 | 0.3722780404 | 0.0161497813 | 0.4959622064 | -0.03592384 | 0.2110774787 | -0.050539098 | 0.0603224206 | 0.0303069332 | -0.095980586 | 0.0594502866 |
| d35 | 0.5084605947 | 0.0703978264 | -0.228921014 | 0.0510203029 | -0.044055773 | -0.09187763 | 0.013191056 | 0.0301983933 | -0.198870376 | 0.107872319 |
| d05 | 0.0403041274 | -0.011473267 | 0.0009880843 | 0.1110836066 | 0.8213843715 | -0.06870313 | 0.0259236816 | -0.004809041 | 0.1522239959 | -0.023360794 |
| d36 | 0.0136850493 | 0.2201252425 | -0.160002869 | -0.026268715 | 0.4122500153 | 0.3607811028 | -0.043856913 | 0.0176890691 | 0.03975916 | 0.1893999968 |

### *SVD U Table*

```
    table example.&U.;
    numrows;
    columninfo;
    fetch / format to=10;
```

```
run;
```

This table contains the U matrix from the singular-value decomposition. The number of rows equals the number of terms in the SVD (31). The number of columns equals the number of topics (the K= value of the SVD option) plus an _ID_ column.

| Number of Rows Action for Table _T_DBDFC68_7F21541ABC08 |
|---|
| Number of Records |
| 31 |

| Column Information for Table _T_DBDFC68_7F21541ABC08 | | | | |
|---|---|---|---|---|
| Id | Column | Type | Length | Format |
| 1 | _ID_ | Num | 8 | BEST12. |
| 2 | Col1 | Num | 8 | BEST12. |
| 3 | Col2 | Num | 8 | BEST12. |
| 4 | Col3 | Num | 8 | BEST12. |
| 5 | Col4 | Num | 8 | BEST12. |
| 6 | Col5 | Num | 8 | BEST12. |
| 7 | Col6 | Num | 8 | BEST12. |
| 8 | Col7 | Num | 8 | BEST12. |
| 9 | Col8 | Num | 8 | BEST12. |
| 10 | Col9 | Num | 8 | BEST12. |
| 11 | Col10 | Num | 8 | BEST12. |

| Selected Records from Table _T_DBDFC68_7F21541ABC08 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| _ID_ | Col1 | Col2 | Col3 | Col4 | Col5 | Col6 | Col7 | Col8 | Col9 | Col10 |
| 1 | 0.0926249121 | 0.2982811527 | 0.3988844343 | 0.0712628886 | -0.063274612 | -0.056201672 | -0.040254213 | -0.037546795 | 0.1112823179 | 0.0121594364 |
| 2 | -0.021287172 | -0.011892781 | 0.0085110565 | 0.6192789695 | -0.014713345 | 0.0115912473 | -0.017054702 | 0.0595400924 | -0.02929076 | -0.010072058 |
| 3 | 0.4077750762 | -0.063169216 | 0.1228366162 | -0.041740868 | 0.02193976 | 0.155054316 | -0.059733899 | 0.0176902181 | -0.001059077 | -0.059086044 |
| 4 | 0.0511201072 | -0.117276972 | 0.2435562281 | 0.0474315768 | 0.0326127015 | 0.1500857485 | 0.418898615 | 0.1459186024 | -0.040054678 | 0.2571088793 |
| 5 | -0.063282625 | 0.0634577784 | 0.0162364969 | -0.036486961 | 0.0038752718 | -0.012592821 | 0.6780420968 | -0.05799108 | -0.070578597 | -0.061926195 |
| 6 | 0.4269469652 | -0.012566466 | 0.0290093669 | -0.014046398 | -0.01461274 | 0.0360037241 | 0.0710454019 | -0.009102881 | -0.021601423 | 0.0103605022 |
| 7 | -0.040272896 | 0.0444232713 | 0.0095648144 | -0.09301671 | -0.010808275 | 0.0081803714 | -0.045968777 | 0.4240911982 | -0.010528903 | 0.3037764771 |
| 8 | 0.4378253058 | -0.033744098 | -0.005996363 | -0.023041162 | 0.0186821655 | 0.0257166711 | -0.017257688 | 0.0084823412 | 0.1113913679 | -0.045559314 |
| 9 | -0.073371768 | 0.5333675817 | -0.092342132 | -0.107505238 | 0.1464327122 | 0.1048642894 | 0.012355482 | 0.072381089 | -0.07927017 | 0.1503637997 |
| 10 | 0.0732936568 | 0.0574874963 | -0.179768528 | -0.000526975 | -0.004888701 | -0.272860264 | 0.3405278339 | -0.005304161 | 0.2602985813 | -0.037304679 |

*Note:* The U matrix produced by the TEXTPARSE statement in the server does not match the U matrix that is generated by the HPTMINE procedure for SAS Text Miner. The SAS LASR Analytic Server performs a varimax rotation on the matrix. This step is done by SAS Text Miner on the HPTMINE output with the FACTOR procedure.

## Topics Table

```
  table example.&Topics;
  columninfo;
  fetch / format to=10;
run;
```

This table displays the topics identified by the server, the term weight cutoff for the topic, and a descriptive label formed from the terms in the topic with the highest weights. By default, five terms are used to label the topic. You can specify a different number with the NUMLABELS= option.

| Column Information for Table _T_DBDFC80_7F21541ABDC8 | | | | |
|---|---|---|---|---|
| Id | Column | Type | Length | Format |
| 1 | _topicid | Num | 8 | BEST12. |
| 2 | _termCutoff | Num | 8 | BEST12. |
| 3 | _name | Char | 76 | $76. |

| Selected Records from Table _T_DBDFC80_7F21541ABDC8 | | |
|---|---|---|
| _topicid | _termCutoff | _name |
| 1 | 0.256 | big, data, big data, +be, how |
| 2 | 0.253 | +process, for, analytics, sas, the |
| 3 | 0.256 | from, sas, in, to, analytics |
| 4 | 0.252 | high-performance, high-performance analytics, analytics, analytics, +compute |
| 5 | 0.253 | as, in, analytics, business, growth |
| 6 | 0.255 | growth, a, +be, big data, to |
| 7 | 0.254 | +fast, to, +problem, time, +be |
| 8 | 0.249 | +enable, +compute, gain, and, to |
| 9 | 0.255 | of, business, a, +problem, the |
| 10 | 0.255 | +insight, time, gain, to, the |

### *Terms by Topic Table*

```
  table example.&TermsByTopic;
  columninfo;
  fetch / format to=10;
run;
```

The terms-by-topic table is a sparse representation of the term-by-topic matrix.

| Column Information for Table _T_DBDFC88_7F21541ABDC8 | | | | |
|---|---|---|---|---|
| Id | Column | Type | Length | Format |
| 1 | _termid | Num | 8 | BEST12. |
| 2 | _topicid | Num | 8 | BEST12. |
| 3 | _weight | Num | 8 | BEST12. |

| Selected Records from Table _T_DBDFC88_7F21541ABDC8 | | |
|---|---|---|
| _termid | _topicid | _weight |
| 1 | 2 | 0.298 |
| 1 | 3 | 0.399 |
| 2 | 4 | 0.619 |
| 4 | 7 | 0.419 |
| 4 | 10 | 0.257 |
| 3 | 1 | 0.408 |
| 6 | 1 | 0.427 |
| 5 | 7 | 0.678 |
| 7 | 8 | 0.424 |
| 7 | 10 | 0.304 |

### *Document Projection Table*

```
  table example.&DocPro.;
  columninfo;
  fetch / format to=10;
run;
```

The projected document contains at a minimum the document ID variable from the input table and the projection onto the topic space. The number of rows equals the number of documents in the input table. The number of columns equals the number of topics (the K= value of the SVD option) plus the document ID column. Each of the Col1 to Col*k* columns are the projections.

| Column Information for Table _T_DBDFC92_7F21541ABC98 | | | | |
|---|---|---|---|---|
| Id | Column | Type | Length | Format |
| 1 | docid | Char | 8 | $8. |
| 2 | Col1 | Num | 8 | BEST12. |
| 3 | Col2 | Num | 8 | BEST12. |
| 4 | Col3 | Num | 8 | BEST12. |
| 5 | Col4 | Num | 8 | BEST12. |
| 6 | Col5 | Num | 8 | BEST12. |
| 7 | Col6 | Num | 8 | BEST12. |
| 8 | Col7 | Num | 8 | BEST12. |
| 9 | Col8 | Num | 8 | BEST12. |
| 10 | Col9 | Num | 8 | BEST12. |
| 11 | Col10 | Num | 8 | BEST12. |

| Selected Records from Table _T_DBDFC92_7F21541ABC98 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| docid | Col1 | Col2 | Col3 | Col4 | Col5 | Col6 | Col7 | Col8 | Col9 | Col10 |
| d01 | 0 | 0 | 0 | 0.961258621 | 0.2326513565 | 0 | 0.1260184292 | 0 | 0 | 0.0772953134 |
| d32 | 0.4826746909 | 0 | 0 | 0 | 0 | 0.8503620927 | 0.1152405201 | 0 | 0.1601023899 | 0.0706845195 |
| d02 | 0.6972672517 | 0 | 0 | 0 | 0.266983692 | 0 | 0 | 0 | 0.6652353628 | 0 |
| d33 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0.2030633291 | 0.9791656062 | 0 |
| d03 | 0 | 0.2640041438 | 0 | 0 | 0 | 0 | 0.1612156175 | 0.3706746922 | 0 | 0.8757348967 |
| d34 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| d04 | 0.7171513594 | 0.1165153689 | 0.5764525769 | 0 | 0.3085106863 | 0 | 0.1272595922 | 0 | -0.149512016 | 0.0780565995 |
| d35 | 0.9278694479 | 0 | -0.211032126 | 0 | 0 | 0 | 0 | 0.049446748 | -0.303444804 | 0 |
| d05 | 0 | 0 | 0.1650911085 | 0.2032653288 | 0.9481777394 | 0 | 0 | 0 | 0.1799641813 | 0 |
| d36 | 0 | 0.3645614055 | 0 | 0 | 0.6772513612 | 0.6301697157 | 0 | 0.1063564985 | 0 | 0 |

You can request that other variables, beside the document ID variable, are transferred to the projected document table. If you do not transfer variables, then the document projection table is a candidate for a join with the input table on the document ID in order to associate the documents with their topic weights.

*Appendix 1*

# SAS In-Memory Statistics



## About SAS In-Memory Statistics

SAS In-Memory Statistics is an offering that provides the data scientist or analytical expert with interactive programming access to in-memory data and integrates seamlessly with data in Hadoop or a Teradata Data Warehouse Appliance. This information applies to the SAS In-Memory Statistics 2.6 and later.

In order to use the offering, the following must be true:

- You are using a distributed SAS LASR Analytic Server only.
- The SAS LASR Analytic Server is co-located with Hadoop distribution that has been configured with the services from SAS Plug-ins for Hadoop. The services enable you to use the SASHDAT file format for storing tables in HDFS.
- **Data Access Requirement** At least one of the following:
  - For deployments that work with Hadoop data, SAS/ACCESS Interface to Hadoop must be configured on a client machine that you use for submitting SAS programs. Be sure to install the SAS Embedded Process on the machines in the Hadoop cluster. The SAS/ACCESS engine, the embedded process, and the HDMD procedure enable you to describe your data that is in Hadoop and access it directly without an intermediate metadata repository such as Hive.

- For deployments that work with Teradata, SAS/ACCESS Interface to Teradata must be configured on a client machine that you use for submitting SAS programs. Be sure to install the SAS Embedded Process on the machines in the Teradata appliance. The SAS/ACCESS engine and the embedded process work together to transfer data from Teradata into memory on the server.

- SAS Studio provides an interactive web-based development application that enables you to write and submit SAS programs. Make sure that your user ID is configured for passwordless SSH to the machines the cluster. Also make sure that you have passwordless SSH access from the machine that hosts SAS Studio to the machines in the cluster. For more information, see "Passwordless SSH" on page 16.

# Writing and Running SAS Programs

## Log On to SAS Studio

Start a web browser and direct it to a URL that is similar to the following example:

http://hostname.example.com/SASStudio

In addition to the host name being different, the protocol might be HTTPS and you might need to specify a port number.

**TIP** After you log on, click  in the toolbar to enable interactive mode. This mode is needed for working interactively with the IMSTAT procedure.

## Start a SAS LASR Analytic Server

You start a distributed server with the LASR procedure. See the following example:

```
options set=GRIDHOST="grid001.example.com";
options set=GRIDINSTALLLOC="/opt/TKGrid_REP"; 1

proc lasr create port=10011;
  performance nodes=all;
run;
```

1 Using a TKGrid_REP installation location enables reading data in parallel with SAS/ACCESS engines from distributed databases and Hadoop clusters.

## Load a Table to Memory

### Use the SAS LASR Analytic Server Engine

A common way to load tables into memory is to transfer them to the server using the SAS LASR Analytic Server engine. The following example transfers a table that is named Webscore to the server.

```
libname lasrlib sasiola host="grid001.example.com" port=10011;

data lasrlib.webscore;
    set somelib.webscore;
run;
```

### *Use the Hadoop Engine*

Your Hadoop cluster might already have data in HDFS that you want to analyze, such as DBMS tables that were imported with Sqoop or log files that were imported with Flume. In this case, you can use the SAS/ACCESS Interface to Hadoop to access that data.

```
options set=HADOOP_JAR_FILES_PATH="/opt/hadoopjars";

libname hdplib hadoop server="grid001.example.com"  1
        config="/home/sasdemo/config.xml"
        hdfs_metadir="/user/sasdemo/meta"
        hdfs_datadir="/user/sasdemo/data";

proc hdmd name=hdplib.webdata file_format=delimited  2
     encoding=utf8 sep="|" data_file="web-data.txt";
    column id int;
    column links varchar(256);
    column var1 double;
    column var2 double;
    column var3 double;
    column var4 double;
    column var5 double;
    column var6 double;
    column var7 double;
    column var8 double;
run;

proc lasr add data=hdplib.webdata port=10010;
    performance host="grid001.example.com";
run;
```

1 The LIBNAME statement with the Hadoop engine uses the HDFS_METADIR= option. This option enables working with XML-based table definitions called SASHDMD descriptors.

2 The HDMD procedure is used to create the SASHDMD descriptors from existing data in HDFS, which in this case is a delimited file.

For information about the Hadoop engine and the HDMD procedure, see *SAS/ACCESS for Relational Databases: Reference*.

### *Use the Teradata Engine*

You can use the SAS/ACCESS Interface to Teradata to read data from the database into memory on the server.

```
libname tdlib terdata server="dbccop1.example.com"  1
        database=hps
        user=dbc
        password=dbcpass;

proc lasr add data=tdlib.loans port=10010 noclass;  2
    performance host="grid001.example.com";
run;
```

1 The LIBNAME statement references the Teradata database.

2 The LASR procedure reads a table named Loans from the Teradata database and loads it into memory on the server.

For information about the Teradata engine, see *SAS/ACCESS for Relational Databases: Reference*.

## Sample the Data

The following statements show one way to sample data. A calculated column is created with a number that is uniformly distributed between 0 and 1. The sampling is done based on the value of the new column.

```
%let seed = 12345;

proc imstat;
    table lasrlib.webdata;  1
    tableinfo;
    columninfo;
    frequency goalVar;
run;

    compute sampkey "sampkey = ranuni(&seed.)";  2
run;

    table lasrlib.webdata;  3
    deleterows;
    where sampkey ge 0.31 and goalVar = 0;
run;
```

1 The TABLE statement specifies the Webdata table that was loaded to memory as the active table. The following three statements, TABLEINFO, COLUMNINFO, and FREQUENCY provide information about the table and a variable that is named Goalvar.

2 The COMPUTE statement creates a column that is named Sampkey. The column is permanent and is added to the table.

3 The TABLE statement is used again to reopen the table. This enables SAS to access the newly created column, Sampkey. The DELETEROWS statement is subject to the WHERE clause and marks 70% of the table for deletion where the goal was not met. Because the PURGE option is not used, the rows are not actually deleted. Instead, the rows are just disregarded in subsequent analyses that use the table.

## Create a Forest of Decision Trees, Assess, and Score

The following code sample demonstrates using the RANDOMWOODS statement with training and validation data.

```
table lasrlib.webdata;
 where sampkey ge 0.3;                        /* training set */
 randomwoods goalVar /
         input   = ( browser var1-var8 )
         nominal = ( browser )
         nbins=100 maxlevel=10 maxbranches=2  /* tree specs */
         greedy gain leafsize=50

         ntree=100 seed=1314 m=5              /* forest spec */
         treeinfo bootstrap=0.3
         temptable
 ;
```

```
run;
    table lasrlib.&_templast_;
    promote RF;                                /* promote the model into */
                                               /* a permanent table */
run;

    table lasrlib.webdata;
    where sampkey lt 0.3;                      /* validation set */
    randomwoods /
         lasrtree  = lasrlib.RF
         nominal   = ( browser )
         temptable
         assess
         vars      = ( userid goalVar )
    ;
run;

    table lasrlib.&_templast_;                 /* assess */
    where strip(_RF_Level_) eq '1';
    assess _RF_P_/ y = goalVar event = '1'
           nbins = 10 step = 0.001;
run;

    table lasrlib.webscore;                    /* score */
    compute goalVar "goalVar = 2";
    randomwoods /
             lasrtree  = lasrlib.RF
             nominal   = ( browser )
             temptable
             assess
             vars      = ( userid )
         ;
run;
    table lasrlib.&_templast_;
    promote scoreresult;
quit;
```

# Deploying SAS In-Memory Statistics

## *Installation Sequence*

If SAS In-Memory Statistics is installed along with a SAS solution such as SAS Visual Analytics, then follow the steps that are provided in the installation guide for the solution. The software is automatically installed when it is delivered with a SAS solution.

If you are not installing the software as part of a solution, then you are performing a "Basic" installation instead of a "Planned" installation. Use the documents in the following sections to install the software.

## Software for Your Analytics Cluster

### Basic Steps

Information about installing and configuring SAS High-Performance Analytics Environment, SAS Plug-ins for Hadoop, and SAS High-Performance Computing Management Console is available in the *SAS High-Performance Analytics Infrastructure: Installation and Configuration Guide*. This book is available at the following URL:

http://support.sas.com/documentation/solutions/hpainfrastructure/

*Note: SAS High-Performance Analytics Infrastructure: Installation and Configuration Guide* directs you to install the SAS Embedded Process. You can install it only after SAS Foundation and SAS/ACCESS are installed. This applies whether you use Hadoop or Teradata.

### How Is Hadoop Used?

The SAS Plug-ins for Hadoop provide services that run inside Hadoop to enable working with the SASHDAT format. This software is installed on the same machines in the analytics cluster (co-located) for the purpose of providing a physical location for staging data. By staging the data on the same machines, data access is more efficient.

Typically, the data to analyze are in a data store like Hadoop or Teradata that is remote from the analytics cluster. The LASR procedure can transfer the data from the remote data store and the SAVE statement in the IMSTAT procedure can persist the data in HDFS as a SASHDAT table.

For more information, see "Example 4: Deleting Rows and Saving a Table to HDFS" on page 368 or "SAVE Statement" on page 327.

## SAS Foundation and Related Software

### Install SAS Foundation

On the machine that you will use as the SAS client for writing and submitting SAS programs, run the SAS Deployment Wizard to install SAS Foundation. The type of SAS Studio that is installed depends on your host operating system. See the following sections for details.

### About SAS Studio Basic and SAS Studio - Single User

SAS Studio is a development application for writing SAS programs and submitting them. You can access SAS Studio through your web browser.

| | |
|---|---|
| SAS Studio Basic | is included with an order for SAS In-Memory Statistics for Hadoop on Linux for x64. |
| SAS Studio - Single User | is included with an order for SAS In-Memory Statistics for Hadoop on Windows. |

See *SAS Studio: Administrator's Guide* for information about installing SAS Studio and administration.

### *UNIX Hosts*

Use the SAS Deployment Wizard to install SAS. Refer to the documentation for UNIX hosts at the following URL:

http://support.sas.com/documentation/installcenter/94/unx/index.html

When you run the SAS Deployment Wizard, you can specify to install SAS Studio Basic. Refer to "SAS Studio Basic" in the *SAS Studio: Administrator's Guide*.

After the SAS Deployment Wizard installs the software, be sure to follow the instructions for configuring Hadoop JAR files. Refer to *Configuration Guide for SAS 9.4 Foundation for UNIX Environments* available at the preceding URL.

### *Windows Hosts*

Use the SAS Deployment Wizard to install SAS. Refer to the documentation for Windows hosts at the following URL:

http://support.sas.com/documentation/installcenter/94/win/index.html

After the SAS Deployment Wizard installs the software, be sure to follow the instructions for configuring Hadoop JAR files. Refer to *Configuration Guide for SAS 9.4 Foundation for Microsoft Windows for x64* available at the preceding URL.

## *SAS In-Database Products, SAS Embedded Process, and Remote Parallel Connections*

After SAS Foundation and SAS/ACCESS are installed, follow the instructions in the *SAS High-Performance Analytics Infrastructure: Installation and Configuration Guide* to install the SAS Embedded Process for Hadoop or Teradata on the remote Hadoop cluster or Teradata appliance. The instructions for Teradata are included in *SAS In-Database Products: Administrator's Guide*.

Make sure you follow the steps in section "Configuring the Analytics Environment for a Remote Parallel Connection" to ensure that the LASR procedure can load data in parallel from the data store to SAS LASR Analytic Server.

*Appendix 2*

# Removing a Machine from the Cluster



## About Removing a Machine from the Cluster

This information applies to deployments that use a distributed SAS LASR Analytic Server.

This information does not apply to non-distributed deployments.

For deployments that use Teradata EDW, Greenplum DCA, or a commercial distribution of Hadoop, such as Cloudera or Hortonworks, follow the product documentation for the appliance or cluster. If it is possible to run the appliance or cluster in a degraded state, you can follow the steps to remove the host name from the grid.hosts file.

## Which Servers Can I Leave Running When I Remove a Machine?

In most cases, all SAS LASR Analytic Server instances stop automatically if communication with any machine in the cluster fails. However, if any servers are still running, stop them.

For SAS Visual Analytics deployments, stop the SAS LASR Analytic Server Monitor.

## Remove the Host Name from the grid.hosts File

The host name for the machine to remove must be removed from the **`/opt/TKGrid/grid.hosts`** file. (The path might be different for your deployment.)

To remove the host name:

1. Log on to the root node for the deployment as the root user or the user ID that installed the High-Performance Analytics Environment.
2. Edit the **/opt/TKGrid/grid.hosts** file and delete the line that includes the host name to remove. Save and close the file.
3. Copy the updated **grid.hosts** file to the same location on all the machines in the cluster. You can use the **simcp** command to perform this task:

   ```
   /opt/TKGrid/bin/simcp /opt/TKGrid/grid.hosts /opt/TKGrid
   ```

If your deployment uses the SAS High-Performance Computing Management Console, remove the host name from the **/etc/gridhosts** file, too.

## Restart Servers

You can restart SAS LASR Analytic Server instances at this point. When you load tables from HDFS, data are loaded from redundant blocks on the remaining machines.

When the machine becomes available again (such as replacement hardware or new hardware), follow the steps in Appendix 3, “Adding Machines to the Cluster,”.

*Appendix 3*

# Adding Machines to the Cluster



## About Adding Machines to the Cluster

This information applies to deployments that use a distributed SAS LASR Analytic Server. These steps do not include information about licensing SAS software for additional CPUs. Contact your SAS representative for information about getting additional licensing and applying the license.

This information does not apply to non-distributed deployments.

For deployments that use Teradata EDW, Greenplum DCA, or a commercial distribution of Hadoop, follow the product documentation for information about expanding the appliance or cluster. After following those procedures, install the SAS High-Performance Analytics environment as described in *SAS High-Performance Analytics Infrastructure: Installation and Configuration Guide*.

## Which Servers Can I Leave Running When I Add a Machine?

It is best to stop all SAS LASR Analytic Server instances and services for any co-located Hadoop. For SAS Visual Analytics deployments, stop the SAS LASR Analytic Server Monitor, too.

If you prefer to deny access to the environment while performing this procedure, you can use the SAS High-Performance Computing Management Console to perform an SSH lockout. Be sure to permit access for root, the SAS installer account, the account that is used to run HDFS, and at least one administrator that can start and stop server instances.

You can ensure that saving files to HDFS is denied by using safe mode. The following command is an example:

```
$HADOOP_HOME/bin/hdfs dfsadmin -safemode enter
```

## Configure System Settings

Each of the additional machines must be configured identically to the existing machines with regard to operating system, drivers, and tuning settings. The high-level tasks are as follows:

- Configure passwordless SSH for the root user ID.
- Disable SELinux if it is disabled on the existing machines.
- Modify **/etc/ssh/sshd_config** with the following setting:

  ```
  MaxStartups 1000
  ```

- Modify **/etc/security/limits.conf** with the following settings:
  - `soft nproc 65536`
  - `hard nproc 65536`
  - `soft nofile 350000`
  - `hard nofile 350000`
- Modify **/etc/security/limits.d/90-nproc.conf** with the following setting:

  ```
  soft nproc 65536
  ```

- Modify **/etc/sysconfig/cpuspeed** with the following setting:

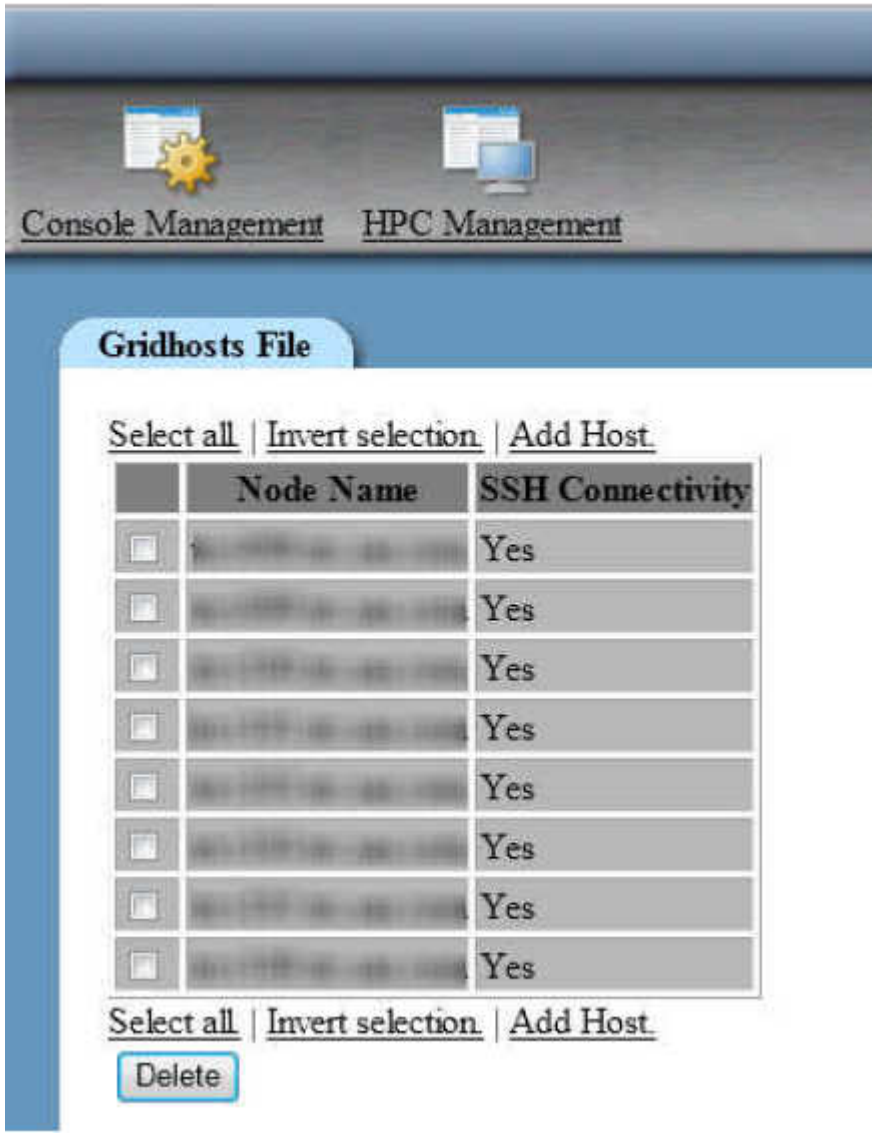
  ```
  GOVERNOR=performance
  ```

The previous settings are identical to the settings specified in the *SAS High-Performance Analytics Infrastructure: Installation and Configuration Guide*.

## Add Host Names to Gridhosts

If you use SAS High-Performance Computing Management Console to manage your cluster, you must add the host names for the additional machines to the **`/etc/gridhosts`** file. You can do this with a text editor, or you can use the Gridhosts File Management feature in the console. If you do not use the console for managing the cluster, then you do not need to perform this task.

If you use the console to add the hosts, make sure that the **SSH Connectivity** column indicates `Yes` for all the machines. The following display shows an example.




A `No` in the **SSH Connectivity** column indicates that passwordless SSH for the root user ID is not configured properly for the machine identified in the **Node Name** column.

# Propagate Operating System User IDs

## Which User IDs Must Be Propagated?

You must propagate the operating system user IDs that are used for Hadoop and for managing SAS LASR Analytic Server instances (starting, stopping, and loading tables).

## About Passwords

In most deployments, passwords are not used to log on to the machines in the cluster. Therefore, in most cases, it is not necessary to propagate passwords to the additional machines in the cluster.

However, if you want to preserve passwords and you use the SAS High-Performance Computing Management Console, you can view the value from the **`/etc/shadow`** file or within the console. When you add the user with the console, you can paste the value in the **Pre-encrypted password** field.

## Option 1: User Management Software

If your site uses account management software applications such as LDAP, NIS, or Active Directory on Linux for managing user accounts, then use that software to make the user accounts available on the additional machines.

## Option 2: Delete and Re-add Users

If you use SAS High-Performance Computing Management Console to manage the machines in the cluster, then you can delete each user and then add the user back to the system. This is an option when the number of operating system accounts is fairly low. When you add the user, the account is re-created on the original machines, and it is also added to the new machines.

*Note:* When using this method, be sure to note the UID and primary GID of the user before it is deleted and to reuse the same values when re-creating the user account.

*Note:* If you choose this option, be aware that using the **Generate and Propagate SSH Keys** option when you create the user account removes existing SSH keys. If you do not delete the home directory when you delete the user and you do not generate new SSH keys, then the existing keys can be reused.

## Option 3: Use Operating System Commands to Add Users

### About Using the Simultaneous Shell Command

You can view the existing user IDs and groups with the SAS High-Performance Computing Management Console, or from the **`/etc/passwd`** and **`/etc/group`** files. You can use operating system commands to add the groups and users to the addition machines.

The following sample commands use the **simsh** utility that is included with SAS High-Performance Computing Management Console. This utility attempts to configure users

and groups on the machines that already have the user accounts and groups. This results in an error message from those machines. The error message is harmless because the commands do not modify the existing configuration, but you might find them distracting. As an alternative, you can use the console to create a simultaneous utilities machine group that contains the host names for the new machines only. You can then specify the group name with the **simsh** command so that only the new machines are affected.

This document demonstrates using the **simsh** command with a machine group that is named **newnodes** for simplicity.

### *Add Groups*

Identify the groups to add to the new machine by viewing the console or looking at the **`/etc/group`** file. Make sure that you identify each group ID.

The following example shows how to add the group that is named **sasdemo** with a group number of 102 to the **newnodes** machine group :

```
/opt/webmin/utilbin/simsh -g newnodes "groupadd -g 102 sasdemo"
```

### *Add Users*

Identify the user IDs to add to the new machine by viewing the console or looking at the **`/etc/passwd`** file. Make sure that you identify each user ID.

The following example shows how to add a user:

```
/opt/webmin/utilbin/simsh -g newnodes
     "useradd -u 503 -g 102 -d /home/sasdemo -s /bin/bash sasdemo"
```

*Note:* The command must be entered on a single line.

### *Propagating Secure Shell Keys*

One way to propagate existing SSH keys to the new machines is to copy them to all the new machines in the cluster. The following example shows one way to perform this operation:

```
simcp /home/user/.ssh /home/user/
simsh chown -R user:group  /home/user/.ssh
```

You can use scripting to simplify this task and all the previous operating system commands, too. You also do not need to follow this strategy. Any method that is able to propagate the groups, user IDs, and SSH keys is acceptable.

## Configure SAS Plug-ins for Hadoop

### *Install Hadoop on the Additional Machines*

Install Hadoop on the additional machines according to the documentation that is available from the vendor. Afterward, install the SAS Plug-ins for Hadoop on the additional hosts according to the *SAS High-Performance Analytics Infrastructure: Installation and Configuration Guide*.

### (Optional) Copy a File to HDFS

If you put HDFS in safe mode at the beginning of this procedure, leave safe mode with a command that is similar to the following:

```
$HADOOP_HOME/bin/hdfs dfsadmin -safemode leave
```

To confirm that the additional machines are used, you can copy a file to HDFS and then list the locations of the blocks. Use a command that is similar to the following:

```
$HADOOP_HOME/bin/hadoop fs -D dfs.blocksize=512 -put /etc/fstab /hps
```

*Note:* The very small block size shown in the example is used to increase the number of blocks written and increase the likelihood that the new machines are used.

You can list the block locations with a command that is similar to the following:

```
$HADOOP_HOME/bin/hdfs fsck /hps/fstab -files -locations -blocks
```

Review the output to check for IP addresses for the new machines.

```
Connecting to namenode via http://0.0.0.0:50070
FSCK started by hdfs (auth:SIMPLE) from /192.168.9.156 for path /hps/fstab at
Wed Jan 30 09:45:24 EST 2013
/hps/fstab 2093 bytes, 5 block(s): OK
0. BP-1250061202-192.168.9.156-1358965928729:blk_-2796832940080983787_1074
len=512 repl=2 [192.168.8.217:50010, 192.168.8.230:50010]
1. BP-1250061202-192.168.9.156-1358965928729:blk_-7759726019690621913_1074
len=512 repl=2 [192.168.8.230:50010, 192.168.8.153:50010]
2. BP-1250061202-192.168.9.156-1358965928729:blk_-6783529658608270535_1074
len=512 repl=2 [192.168.9.159:50010, 192.168.9.158:50010]
3. BP-1250061202-192.168.9.156-1358965928729:blk_1083456124028341178_1074
len=512 repl=2 [192.168.9.158:50010, 192.168.9.160:50010]
4. BP-1250061202-192.168.9.156-1358965928729:blk_-4083651737452524600_1074
len=45 repl=2 [192.168.8.230:50010, 192.168.8.153:50010]
```

Delete the sample file:

```
$HADOOP_HOME/bin/hadoop fs -rm /hps/fstab
```

## Configure SAS High-Performance Analytics Infrastructure

### Strategies

The SAS High-Performance Analytics Infrastructure software must be installed on the new machines in the cluster. In addition, the existing installations must be updated so that the **grid.hosts** file includes the new host names.

The first option is to re-install the software. This adds the software to the new machines and updates the **grid.hosts** file. This option has the advantage of being very simple.

The second option is to copy the files to the new machines and then copy an updated **grid.hosts** file to all the machines.

**TIP** Installing the software to a new directory is not suggested because the path to the software might be specified in numerous server definitions that are registered in SAS metadata.

### *Option 1: Re-Install the Software*

The software is installed by running the **TKGrid_Linux_x86_64.sh** executable. For details about installing the software, see *SAS High-Performance Analytics Infrastructure: Installation and Configuration Guide*.

If you choose this option, stop all SAS LASR Analytic Server instances. Stopping the servers avoids the possibility of errors related to overwriting executables and libraries.

### *Option 2: Copy the Software*

Use a command that is similar to the following for copying the software to the new machines:

```
/opt/webmin/utilbin/simcp -g newnodes /opt/TKGrid/ /opt/TKGrid/
```

Modify the **`/opt/TKGrid/grid.hosts`** file and add the host names for the new machines. Then, copy the file to all machines, even the existing machines:

```
/opt/webmin/utilbin/simcp /opt/TKGrid/grid.hosts /opt/TKGrid/
```

### *Validate the Change*

Use the **mpirun** command to confirm that the new machines are accessible.

```
cd /opt/TKGrid/mpich2-install

./bin/mpirun -f ../grid.hosts hostname
```

The **hostname** command is run on each machine and the results are returned. Make sure that the response includes all the host names in the cluster.

```
grid098.example.com
grid103.example.com
grid106.example.com
grid100.example.com
grid105.example.com
grid101.example.com
grid104.example.com
grid099.example.com
```

## Restart SAS LASR Analytic Server Monitor

For SAS Visual Analytics deployments, restart the monitor:

```
cd SAS-config-dir/Levn/Applications/SASVisualAnalyticsX.X/
  HighPerformanceConfiguration/LASRMonitor.sh restart
```

Restarting the monitor causes an error for any users logged on to SAS Visual Analytics Administrator. Those users need to log off and log on again.

## Restart Servers and Redistribute HDFS Blocks

Log on to SAS Visual Analytics Administrator and perform the following steps:

1. Select **LASR** ⇨ **Monitor Resources** and sure that the additional machines appear in the Real-Time View.
2. Select **LASR** ⇨ **Manage Servers** and start each of the servers.
3. Select **Tools** ⇨ **Explore HDFS** and review the block distribution.

Any tables that are loaded from HDFS cannot initially use the additional hardware because the blocks have not been replicated to the additional machines. Within SAS Visual Analytic Administrator, you can view the block distribution from the **HDFS** tab to confirm that the blocks are on the original machines only. Check your vendor documentation for information about redistributing blocks.

As an alternative, if adding the data to HDFS again is possible, then you can delete files with SAS Visual Analytics Administrator and then add them back.

## View Explorations and Reports

To confirm that the additional machines are working as intended, view existing explorations and reports.

If you did not delete and then add the data back to HDFS, then make sure that you view explorations and reports that use data that is streamed to the server instance (instead of being loaded from HDFS). Or, make sure that new data sets are added to HDFS and create explorations and reports from the new data.

*Appendix 4*

# Using the Grid Monitor



## Starting the Grid Monitor

The grid monitor is used to monitor distributed SAS LASR Analytic Server and high-performance procedures. The monitor can be started from two locations:

- the machine that has SAS Foundation installed and that acts as a client to the server. This can be a Linux machine or a Windows machine.
- the machine that acts as the root node for the server.

For the machine that acts as a client to the server, the monitor program is available at the following location, depending on your operating environment:

*UNIX Specifics*
**SASHome/SASFoundation/*Version*/utilities/bin/tkgridmon**

*Windows Specifics*
**SASHome\SASFoundation*Version*\tkgridmon.exe**

You must specify values for **-gridhost** and **-gridinstall**. You must also specify the **-startui** option. See the following example:

```
./tkgridmon -gridhost grid001.example.com -gridinstall /opt/TKGrid -startui
```

The same options apply to the Windows operating environment.

The other way to start the monitor is to log on to the root node of the cluster and run it from that host. The monitor is available at the following location:

```
/opt/TKGrid/bin/gridmon.sh
```

The functionality of the monitor is identical, regardless of the location that you use to start it.

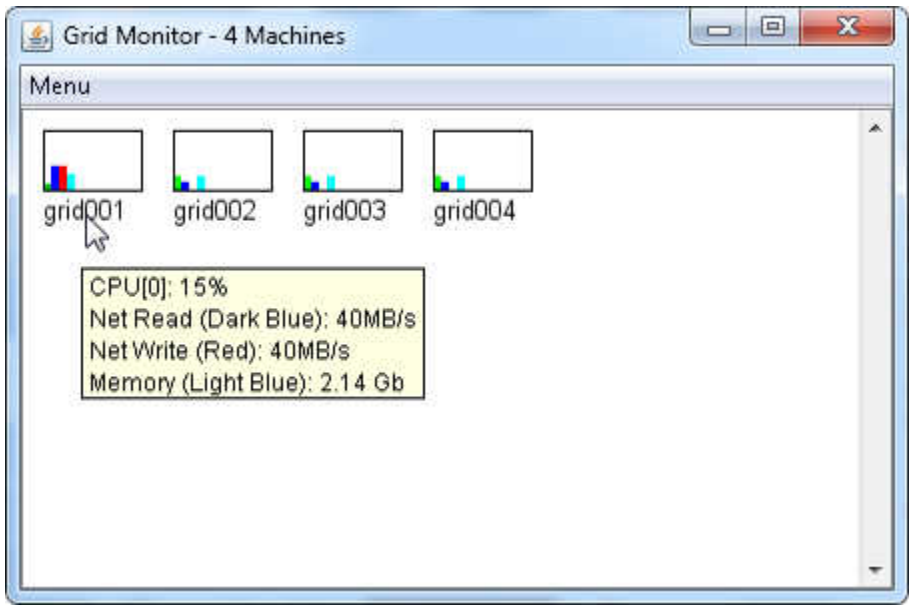
The initial display of the monitor is the node view, as show in Figure A4.1 on page 498.

# Monitoring Resources

## Grid Monitor Window

The Grid Monitor window view of the monitor shows one node for each machine in the cluster. When you place your pointer over a node icon, a tooltip shows the resource use on the machine.

***Figure A4.1*** *Grid Monitor Node View*

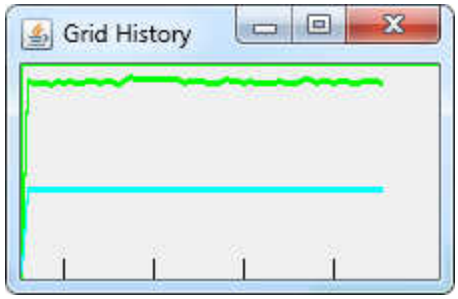



In the preceding display, the grid001 machine is distributing data to the other three machines in the cluster. The red line indicates that it is writing data on the network and the smaller blue lines on the other icons indicate that they are reading data. There is a **CPU[n]** display in the tooltip for each core on the machine. In most cases, there are 24 or more cores shown. Each of the cores is also represented by a green line on the node icon.

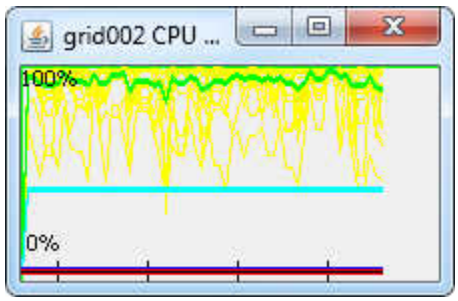
## Viewing History

The monitor offers two views of historic resource use. The first is a cumulative view of all the machines. The second view is for an individual machine. Both views show 60 seconds of historic activity.

The cumulative view for all the machines is shown after you select **Menu** ⇨ **Show Grid History**. The green line shows the average CPU utilization and the blue line shows the average memory use.




If you select a node icon in the node view, right-click, and select **Show History Graph**, then the monitor opens a window that shows the historic resource usage for that machine.



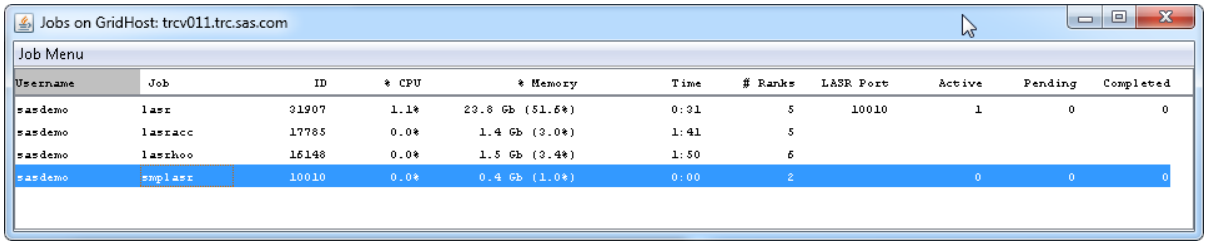

The indicators are as follows:

- **Green** Shows the average CPU usage for all the cores in the machine.
- **Yellow** Each line shows the CPU usage for one core.
- **Light blue** Shows the memory usage for the machine.
- **Dark blue** Shows the network read speed as a percent.
- **Red** Shows the network write speed as a percent.

For the network read and write percentages, when the monitor connects to the cluster, it determines the network interface speed, typically 1G or 10G Ethernet.

## Monitoring Jobs

If you select **Menu** ⇨ **Show Jobs on Grid** from the Grid Monitor window, then the Jobs window is displayed.




| Username | Job | ID | % CPU | % Memory | Time | # Ranks | LASR Port | Active | Pending | Completed |
|---|---|---|---|---|---|---|---|---|---|---|
| sasdemo | lasr | 31907 | 1.1% | 23.8 Gb (51.6%) | 0:31 | 5 | 10010 | 1 | 0 | 0 |
| sasdemo | lasracc | 17785 | 0.0% | 1.4 Gb (3.0%) | 1:41 | 5 | | | | |
| sasdemo | lasrhoo | 16148 | 0.0% | 1.5 Gb (3.4%) | 1:50 | 6 | | | | |
| sasdemo | smplasr | 10010 | 0.0% | 0.4 Gb (1.0%) | 0:00 | 2 | | 0 | 0 | 0 |

The Jobs window shows the following information:

| | |
|---|---|
| Username | The user ID that owns the job. |
| Job | The job name. Servers are listed as **`lasr`**. Connections with the SAS LASR Analytic Server engine are listed as **`lasracc`** and connections with the SASHDAT engine are listed as **`lasrhoo`**. Non-distributed servers that are started as children of a distributed server are listed as **`smplasr`**. Other processes that run on the cluster, such as high-performance procedures can appear in the list as well. |
| ID | The job ID. |
| % CPU | Total percentage of CPU that is used by the job on all machines. |
| % Memory | Actual amount and percentage of memory that is used by the job on all machines. |
| Time | Run time in days, hours, and minutes. If the job has not run for more than 24 hours, then the days field is not shown. |
| # Ranks | Number of machines used by the job. |
| LASR Port | Port number that the server listens on. |
| Active | Number of requests that the server is processing at the same time. |
| Pending | Number of requests that are waiting for processing. After the server reaches the threshold for the number of concurrent processes to run, the requests are queued. |
| Completed | Number of requests that were processed. |

TIP For more information about the **Active** and **Pending** fields, see "SERVERPARM Statement" on page 342.

If you right-click on a job, you can select the following items:

- **Kill Job** Kills the selected process. See "Stopping Processes" on page 502.
- **Kill all jobs with user *userid***
- **Show Ranks** Opens a new window that shows the processes on each machine that are related to the job. See "Monitoring Ranks across Machines" on page 501.
- **Copy to Clipboard**
- **Execute Commands on Rank 0** Used by SAS research and development. It is easier to use the Grid Monitor window to execute commands on a machine.
- **Execute Commands on Last Rank** Used by SAS research and development.

# Monitoring Ranks

## *What is a Rank?*

When a server starts, a software process starts on each machine in the cluster. In order to facilitate communication between the processes on different machines, each is assigned a rank. The rank is simply a number that begins at zero and increases for each additional machine.

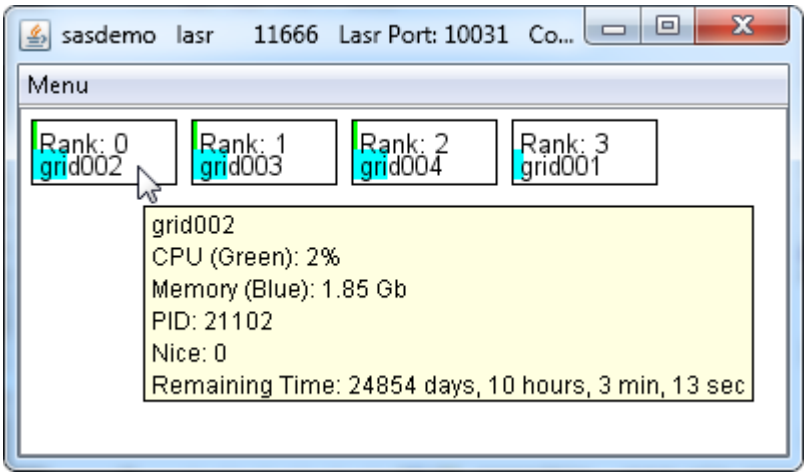
In terms of monitoring ranks, it's more important that you understand your choices for monitoring them:

- Do you want to monitor the processes for a server across the machines in the cluster?
- Do you want to monitor all the processes running on a single machine?

## *Monitoring Ranks across Machines*

From the Jobs window, select a job in the list, right-click, and select **Show Ranks**. A window that is similar to the following example appears.

***Figure A4.2*** *Ranks in a Server*



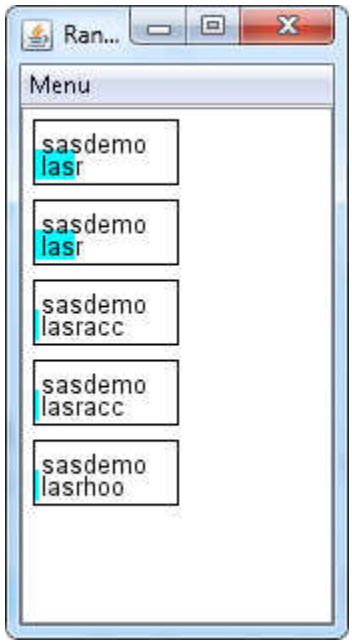

From this view, you can see that a server is using four machines. The tooltip shows the CPU and memory usage for the process on an individual machine. You can also see that grid001, the root node, is using less memory than the other three machines because the blue bar is shorter. This is normal because the root node of a server does not hold any rows of data from a table.

## *Monitoring Ranks on One Machine*

From the Grid Monitor window, select a node icon, right-click, and select **Show Ranks on Node**. A window that is similar to the following example appears. The following display was made narrow so that the processes would stack vertically. This is to

highlight that the processes are running on a single machine, similar to the list of jobs on the Jobs window.

***Figure A4.3*** *Ranks on One Machine*




In this view, the machine is used with two servers, two SAS LASR Analytic Server engine connections, and one SASHDAT engine connection.

### About Scaling

Both views of ranks, whether for process across machines, or all the processes on one machine offer a menu choice for scaling CPU usage. The settings are as follows:

One CPU
: With this setting, a green bar that is the full width of the node icon indicates that the CPU usage is greater than or equal to one CPU at maximum use.

Full Node
: With this setting, a green bar represents the CPU usage percentage of the entire machine.

For example, on a machine with 12 cores that use Intel Hyper-Threading Technology, a server starts 24 threads. If the server runs a single-threaded task on that machine, and the setting is **One CPU**, then the green bar extends the full width of the node icon and the machine appears to be fully utilized. However, if the setting is **Full Node**, then the green bar is 1/24th full and CPU usage might be difficult to notice.

## Stopping Processes

When you are using the Jobs window, you have a few options for stopping processes:

- Select a job, right-click, and select **Kill Job** or **Kill all jobs with user *userid***

- Select **Job Menu** ⇨ **Kill Old Processes** and then specify a threshold in days for identifying old processes.

When you are viewing ranks, you can right-click on a rank and the menu includes **Kill Rank**.

All these actions send the UNIX command **kill -2** to the processes. This gives the process a chance to exit gracefully. If the processes have not stopped in five seconds, then a **kill -9** signal is sent. Users are not notified of a shutdown, and in-memory tables are unloaded as the server stops. If you have **sudo** privilege, then you can kill other users' processes.

*Note:* It is preferable to stop a server with the TERM option to the LASR procedure, or with the SERVERTERM statement in the IMSTAT or VASMP procedures.

## Executing Commands

On most windows in the monitor, you can right-click on a job, rank, or node and select **Execute Commands**. This provides a convenient mechanism for running UNIX commands such as listing directories or copying files. You can also invoke commands like **hadoop fs -ls /hps**, if your user ID is configured to access Hadoop on the cluster.

*Appendix 5*

# Managing Resources



## About Managing Resources

For distributed SAS LASR Analytic Server, the server often needs to share resources with other processes, or other servers on the same cluster. At server start-up, the server reads the **`/opt/TKGrid/resource.settings`** file. This file is delivered with the server software and initially all the settings are disabled.

The following sections describe the basics for how you can modify the file to manage resources. The file that is delivered with the software includes examples of evaluating the following before allocating resources:

- the user ID that starts a server or job
- the application name

If you are unsure of the application names, you can use the grid monitor that is described in this book to see the application names of running servers and jobs.

*Note:* After you modify the file, copy it to all machines in the cluster, unless you performed a shared installation that accessible to all the machines.

***Table A5.1*** *Summary of Memory Management Controls*

| Control Parameter or Mechanism | Location | Distributed Server | Non-distributed server (Linux) | Non-distributed server (Windows) | Comments |
|---|---|---|---|---|---|
| resource.settings file<br>• TKMPI_MEMSIZE<br>• TKMPI_ULIMIT | TKGrid installation location, such as **`/opt/TKGrid`** | ✓ | | | This is the primary control for managing memory for distributed servers.<br>TKMPI_MEMSIZE<br>limit for in-memory tables and processing. Excludes SASHDAT tables.<br>TKMPI_ULIMIT<br>virtual memory limit for the server. Includes all tables and memory that is used for processing. |
| YARN — integrated with the resource.settings file | TKGrid installation directory | ✓ | | | • Requires YARN configuration and co-location with TKGrid.<br>• Verify that YARN does not automatically create cgroups that interfere with the server. |
| MEMSIZE= | SAS system option file, such as **`sasv9_usermods.cfg`** | | ✓ | ✓ | • This is the primary control for managing memory on non-distributed servers.<br>• Limits virtual memory size.<br>• If MAX or 0 is specified, the server uses 80% of physical RAM.<br>• On Linux, this option is subject to ulimit. |

| Control Parameter or Mechanism | Location | Distributed Server | Non-distributed server (Linux) | Non-distributed server (Windows) | Comments |
|---|---|---|---|---|---|
| TABLEMEM= | The LASR procedure or the advanced options for a server in SAS metadata | ✓ | | | • Tables cannot be added or appended after this limit is met.<br>• The default value is 75% of memory.<br>• Represents the average of memory use across all machines, including the root node.<br>• SASHDAT tables do not count toward the limit.<br>• The IMSTAT procedure can reset the limit while the server is running.<br>• Requires a restart of any web application server that includes SASLASRAuthorizationService. |
| EXTERNALMEM= | The LASR procedure or the advanced options for a server in SAS metadata | ✓ | | | External processes such as high-performance procedures, are prevented from retrieving data after this limit.<br>If the high-performance procedures (or other external process) are running on a separate cluster, you can set this threshold to 100. |
| Tables Limit | SAS Visual Analytics Administrator | ✓ | ✓ | ✓ | • Limits the amount of memory available for tables.<br>• Subject to MEMSIZE=, ulimit, and TKMPI_MEMSIZE settings. |
| cgroups | **`/etc/cgconfig.conf`** | | | | • Do not use cgroups to limit memory. Servers can be killed or become unresponsive.<br>• Limiting CPU use is OK. |
| ulimit | **`/etc/security/limits.conf`** | ✓ | ✓ | | • The address space (as) limit provides a virtual memory size limit that all other controls are subject to. Use **`ulimit -v`** to display the limit.<br>• Limits can be set for each user or group.<br>• The resident set size (rss) limit is ignored beginning with Linux kernel 2.4.30. |

## Using CGroups to Manage CPU

In terms of managing CPU usage, you can specify a TKMPI_CGROUP setting in the **`resource.settings`** file like the following:

```
# Cgroup to associate with TKGrid jobs.
export TKMPI_CGROUP="cgexec -g cpu:50"
```

By itself, specifying the option does not do anything. You need to create a cgroup, in this case, named **`50`** and assign CPU shares to the group. If you are not familiar with assigning CPU shares, you can use the SAS High-Performance Computing Management Console for managing the cgroups.

*Note:* Do not use cgroups to manage memory. If a server were to exceed a cgroup memory limit, the server is either killed or placed in a wait state and becomes unresponsive.

## Setting Memory Limits

There are two settings in the **`resource.settings`** file that are related to memory usage.

```
# VM limit (in KBytes). Default is unlimited
export TKMPI_ULIMIT="-v 50000000"

# Memory allocation limit (in MBytes). Excludes mmapped files.
# Default is unlimited.
export TKMPI_MEMSIZE=2097152
```

Some choices for using these settings are as follows:

TKMPI_ULIMIT
: sets a limit for the entire process. For SAS LASR Analytic Server, this includes the memory that is required to run the server itself, all tables, and memory that is used for processing actions. Specifically, the memory that is used when loading SASDHAT tables to memory is included in the limit.

TKMPI_MEMSIZE
: sets a limit to the size of in-memory tables and memory that is used for processing actions. SASHDAT tables that are loaded to memory are not included in the limit.

If you use these options for managing memory, be aware that the server uses memory for temporary operations like identifying distinct counts of values or grouping observations. If your goal is to limit the amount of memory that is used for in-memory tables, then include some overhead.

Also keep in mind that the default behavior for a server is to reject requests add tables or append data after 75% of memory is used on the machine. This percentage is configurable with the TABLEMEM= option to the SERVERPARMS statement for the IMSTAT and VASMP procedures. As with the TKMPI_MEMSIZE setting, the TABLEMEM= percentage does not apply to SASHDAT tables.

# Managing Resources with YARN

YARN (Yet Another Resource Negotiator) can manage Hadoop applications like MapReduce so that applications can reserve resources like CPU and memory so that resources are not denied to other applications. YARN applications request resources from a resource manager.

If YARN is already used on the cluster, then you can configure SAS LASR Analytic Server instances and high-performance procedures to participate in the resource accounting that YARN performs. This enables administrators to have a complete view of resource usage. Otherwise, having a mix of some applications accounting for their resources with YARN and others that are not essentially results in no management at all.

*Note:* Some YARN implementations create CGroups automatically, which can interfere with CGroups that you administer manually.

In order to integrate with YARN, the following settings in the **`resource.settings`** file are used.

```
# The number of cores to allocate to each host's container.
export TKMPI_YARN_CORES=1

# The amount of memory in megabytes to reserve.
export TKMPI_MEMSIZE=30000

# The priority of the application if scheduler uses it.
export TKMPI_YARN_PRIORITY=2

# Length of time TKGrid should wait for the resource reservation in seconds.
export TKMPI_YARN_TIMEOUT=3600

# The queue to submit the job to.
export TKMPI_YARN_QUEUE=default

# The next setting must be on one line in the resource.settings file,
# but is split for readability.
export TKMPI_RESOURCEMANAGER="java -cp \"`$HADOOP_HOME/bin/hadoop
 classpath`\" com.sas.grid.provider.yarn.tkgrid.JobLauncher
 --masterMem 2000 --javaMem 500 --hostlist \$TKMPI_YARN_HOSTS \
 --cores \$TKMPI_YARN_CORES --memory \$TKMPI_MEMSIZE \
 --priority \$TKMPI_YARN_PRIORITY --timeout \$TKMPI_YARN_TIMEOUT
 --jobname $TKMPI_APPNAME --queue $TKMPI_YARN_QUEUE"
```

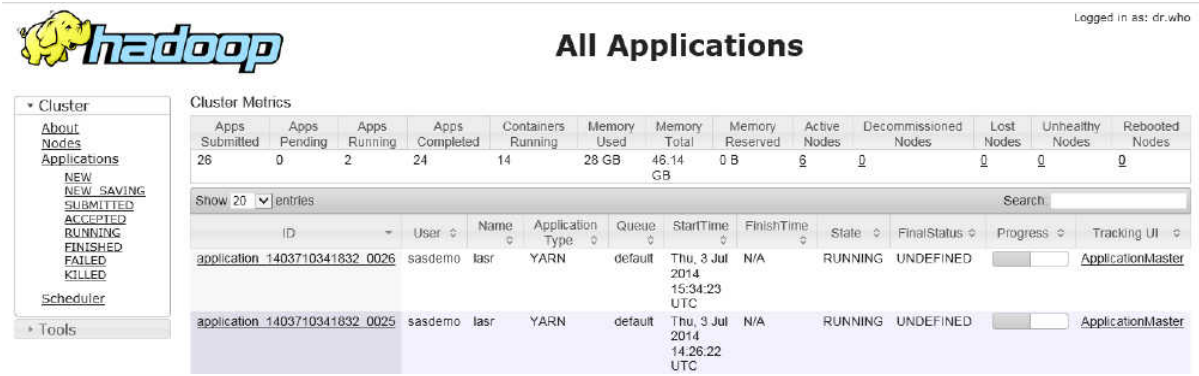
*Note:* The TKMPI_YARN_HOSTS and TKMPI_APPNAME variables are automatically set by SAS software.

The TKMPI_MEMSIZE variable specifies the amount of memory for YARN to reserve. SAS also uses the value to self-govern the memory allocations that are performed by the server or high-performance procedure.

YARN must be configured on the cluster. The **`JobLauncher`** class starts a YARN application to request the specified resources on each machine so that YARN knows it cannot allocate those resources to other applications. If YARN does not grant the resource in the time-out period (TKMPI_YARN_TIMEOUT), then the initializing of the server or job fails.

The servers and jobs are accounted for as YARN applications on the ResourceManager web user interface.

***Figure A5.1*** *Hadoop ResourceManager Web User Interface*




If the **State** field for an application indicates **`RUNNING`** and the **Progress** percentage is at 50%, then the resources are reserved and the application is running. This is the normal state.

If the **State** field indicates **`ACCEPTED`** and the **Tracking UI** field indicates **`UNASSIGNED`**, then the application is not running. Check if you are near or at capacity.

# Glossary

**Apache Hadoop**
a framework that allows for the distributed processing of large data sets across clusters of computers using a simple programming model.

**BY-group processing**
the process of using the BY statement to process observations that are ordered, grouped, or indexed according to the values of one or more variables. Many SAS procedures and the DATA step support BY-group processing. For example, you can use BY-group processing with the PRINT procedure to print separate reports for different groups of observations in a single SAS data set.

**co-located data provider**
a distributed data source, such as SAS Visual Analytics Hadoop or a third-party vendor database, that has SAS High-Performance Analytics software installed on the same machines. The SAS software on each machine processes the data that is local to the machine or that the data source makes available as the result of a query.

**grid host**
the machine to which the SAS client makes an initial connection in a SAS High-Performance Analytics application.

**Hadoop Distributed File System**
a framework for managing files as blocks of equal size, which are replicated across the machines in a Hadoop cluster to provide fault tolerance.

**HDFS**
See Hadoop Distributed File System

**Message Passing Interface**
is a message-passing library interface specification. SAS High-Performance Analytics applications implement MPI for use in high-performance computing environments.

**MPI**
See Message Passing Interface

**root node**
in a SAS High-Performance Analytics application, the role of the software that distributes and coordinates the workload of the worker nodes. In most deployments

the root node runs on the machine that is identified as the grid host. SAS High-Performance Analytics applications assign the highest MPI rank to the root node.

**SASHDAT file**
the data format used for tables that are added to HDFS by SAS. SASHDAT files are read in parallel by the server.

**server description file**
a file that is created by a SAS client when the LASR procedure executes to create a server. The file contains information about the machines that are used by the server. It also contains the name of the server signature file that controls access to the server.

**signature file**
small files that are created by the server to control access to the server and to the tables loaded in the server. There is one server signature file for each server instance. There is one table signature file for each table that is loaded into memory on a server instance.

**worker node**
in a SAS High-Performance Analytics application, the role of the software that receives the workload from the root node.

# Index

## T



## U