SAS/ACCESS® 9.1.3 for Relational Databases

Reference

Third Edition

The Power to Know®

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2006. *SAS/ACCESS® 9.1.3 for Relational Databases: Reference, Third Edition*. Cary, NC: SAS Institute Inc.

SAS/ACCESS® 9.1.3 for Relational Databases: Reference, Third Edition

ISBN-13: 978-1-59047-824-0
ISBN-10: 1-59047-824-X

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, February 2006

SAS Publishing provides a complete selection of books and electronic products to help customers use SAS software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit the SAS Publishing Web site at **support.sas.com/pubs** or call 1-800-727-3228.

Contents

What's New

Overview

- Beginning with SAS 9.0, threaded reads enable you to complete jobs in substantially less time than if each task is handled sequentially.
- The new CV2VIEW procedure converts SAS/ACCESS view descriptors into SQL views.
- Beginning with SAS 9.1.2, a SAS/ACCESS interface is added for MySQL databases.
- Beginning with SAS 9.1.3, several more hosts are supported for existing DBMSs. See the chapter, "SAS/ACCESS Features by Host" for more information.

Note: This section describes the features of SAS/ACCESS that are new or enhanced since SAS 8.2. △

All Supported SAS/ACCESS Interfaces to Relational Databases

- Threaded reads divide resource-intensive tasks into multiple independent units of work and execute those units in parallel.
- Temporary table support enables DBMS temporary tables to persist from one SAS step to the next. This support involves establishing a SAS connection to the DBMS that persists across SAS procedures and DATA steps.
- The new SQL options MULTI_DATASRC_OPT= and DBMASTER= optimize the performance of the SQL procedure. More detailed information is available about passing joins to the DBMS, determining when joins will fail, and optimizing WHERE clauses.
- The SASTRACE= system option now provides improved debugging capabilities.
- The CV2VIEW procedure converts SAS/ACCESS view and access descriptors to the SAS 9.0 format. It can also convert a view descriptor to a SAS 9.0 SQL view. As SAS/ACCESS moves forward with LIBNAME enhancements and tighter integration with the SAS Open Metadata Repository, SAS/ACCESS views will no longer be the method of choice.

- DBMS metadata can now be accurately maintained within the SAS Open Metadata Repository.
- The MULTI_DATASRC_OPT= option in the LIBNAME statement can be used in place of the DBKEY= option to improve performance when you are processing a join between two data sources.
- The DBMASTER= data set option designates which table is the master table when you are processing a join that involves tables from two different types of databases.
- The DIRECT_EXE= option in the LIBNAME statement enables you to pass an SQL statement directly to a database by using explicit pass-through when you are using PROC SQL with a libref.
- You now have the ability to encode the DBMS password that appears in SAS source code so that it does not appear as text in SAS programs.
- The CHANGE statement can be used to re-name SAS/ACCESS tables.
- Linux for Itanium-based Systems is available for DB2, Informix, Microsoft SQL Server, MySQL, ODBC, Oracle, and Sybase, beginning with SAS 9.1.3 Service Pack 1.
- Linux for Intel Architecture is now available for MySQL, beginning with SAS 9.1.3 Service Pack 1, and for Teradata, beginning with SAS 9.1.3.
- AIX (RS/6000) is available for MySQL, beginning with SAS 9.1.3 Service Pack 2.
- HP-UX for the Itanium Processor Family Architecture is available for Sybase, beginning with SAS 9.1.3 Service Pack 2, and for Teradata, beginning with Service Pack 3 in November 2005.
- 64-bit Windows is now available for Oracle and DB2 in addition to 64-bit UNIX, which was provided in SAS 8.2.
- Beginning with SAS 9.0, support is discontinued for the following:

  SAS/ACCESS Interface to CA-OpenIngres

  SAS/ACCESS Interface to Oracle Rdb under OpenVMS Alpha

  OS/2, OpenVMS VAX, MIPS ABI, Intel ABI, UNIX MP-RAS, and CMS operating environments

  CV2ODBC procedure.

SAS/ACCESS Interface to DB2 under z/OS

Note: z/OS is the successor to the OS/390 operating system. SAS/ACCESS 9.1 (and later) for z/OS is supported on both OS/390 and z/OS operating systems and, throughout this document, any reference to z/OS also applies to OS/390, unless otherwise stated. △

The SAS/ACCESS Interface to DB2 under z/OS features stored procedure support that includes passing input parameters, retrieving output parameters into SAS macro variables, and retrieving result sets into SAS tables.

The following options are new:

- The BL_DB2CURSOR= data set option specifies a string that contains a valid DB2 SELECT statement that points to either local or remote objects (tables or views). After your database administrator populates the communication database with the appropriate entries, you can select data from a remote location to load DB2 tables directly from other DB2 and non-DB2 objects.
- The BL_DB2LDCT3= data set option specifies a string in the LOAD utility control statement, following the field specification.
- The DBSLICE= data set option specifies user-supplied WHERE clauses to partition a DBMS query into component queries for threaded reads.

- The DBSLICEPARM= data set option and the DBSLICEPARM= LIBNAME statement option control the scope of DBMS threaded reads and the number of threads.
- The DEGREE= option in the LIBNAME statement determines whether DB2 uses parallelism.
- The REMOTE_DBTYPE= option in the LIBNAME statement ensures that the SQL that is used by some SAS procedures to access the DB2 catalog tables is generated properly, based on the database server type.
- The TRAP151= data set option enables columns that cannot be updated to be removed from a FOR UPDATE OF clause so that updating of columns can continue.

SAS/ACCESS Interface to DB2 under UNIX and PC Hosts

The following options are new:

- The DBSLICE= data set option specifies user-supplied WHERE clauses to partition a DBMS query into component queries for threaded reads.
- The DBSLICEPARM= data set option and the DBSLICEPARM= LIBNAME statement option control the scope of DBMS threaded reads and the number of threads.
- The SQL_FUNCTIONS= option in the LIBNAME statement specifies that the SQL functions that match the functions that are supported by SAS are passed to the DBMS for processing.
- The IGNORE_READ_ONLY_COLUMNS= data set option and option in the LIBNAME statement specifies whether to ignore or include columns whose data types are read-only when generating an SQL statement for inserts or updates.

SAS/ACCESS Interface to Informix

The following options are new:

- The DBSLICE= data set option specifies user-supplied WHERE clauses to partition a DBMS query into component queries for threaded reads.
- The DBSLICEPARM= data set option and the DBSLICEPARM= LIBNAME statement option control the scope of DBMS threaded reads and the number of threads.

SAS/ACCESS Interface to Microsoft SQL Server

The following options are new:

- The DBSLICE= data set option specifies user-supplied WHERE clauses to partition a DBMS query into component queries for threaded reads.
- The DBSLICEPARM= data set option and the DBSLICEPARM= LIBNAME statement option control the scope of DBMS threaded reads and the number of threads.
- The ERRLIMIT= option in the LIBNAME statement specifies the number of errors that are allowed while using the Fastload utility before SAS stops loading data to Teradata.
- The IGNORE_READ_ONLY_COLUMNS= data set option and option in the LIBNAME statement specifies whether to ignore or include columns whose data types are read-only when generating an SQL statement for inserts or updates.

SAS/ACCESS Interface to MySQL

Beginning with SAS 9.1.3 Service Pack 4, the INSERTBUFF= LIBNAME statement option and the INSERTBUFF= data set option specify the number of rows in a single insert operation.

Beginning with SAS 9.1.2, a SAS/ACCESS Interface to MySQL is available. MySQL software is an open-source SQL database server that runs on 32–bit Windows systems, 64–bit HP systems, and 64–bit Solaris systems. The SAS/ACCESS Interface to MySQL provides direct transparent access to MySQL databases through LIBNAME statements or the Pass-Through Facility. Various LIBNAME statement options and data set options, which are supported by the LIBNAME engine, enable you to control the data that is returned to SAS.

SAS/ACCESS Interface to ODBC

The following feature and options are new:

- ODBC 3.x standard API is supported.
- The DBSLICE= data set option specifies user-supplied WHERE clauses to partition a DBMS query into component queries for threaded reads.
- The DBSLICEPARM= data set option and the DBSLICEPARM= LIBNAME statement option control the scope of DBMS threaded reads and the number of threads.
- The SQL_FUNCTIONS= option in the LIBNAME statement specifies that the SQL functions that match the functions that are supported by SAS are passed to the DBMS.
- The IGNORE_READ_ONLY_COLUMNS= data set option and option in the LIBNAME statement specifies whether to ignore or include columns whose data types are read-only when generating an SQL statement for inserts or updates.

SAS/ACCESS Interface to OLE DB

The following options are new:

- The INSERTBUFF= option in the LIBNAME statement specifies the number of rows in a single insert operation.
- The SQL_FUNCTIONS= option in the LIBNAME statement specifies that the SQL functions that match the functions that are supported by SAS are passed to the DBMS.
- The IGNORE_READ_ONLY_COLUMNS= data set option and option in the LIBNAME statement specifies whether to ignore or include columns whose data types are read-only when generating an SQL statement for inserts or updates.

SAS/ACCESS Interface to Oracle

The following options are new:

- The BL_PARFILE= data set option creates a file that contains the SQL*Loader command line options.
- The DBSLICE= data set option specifies user-supplied WHERE clauses to partition a DBMS query into component queries for threaded reads.
- The DBSLICEPARM= data set option and the DBSLICEPARM= LIBNAME statement option control the scope of DBMS threaded reads and the number of threads.
- The OR_PARTITION= data set option enables you to read, update, and delete data from a specific partition in a partitioned table. It also enables you to insert and bulk-load data into a specific partition in a partitioned table. It also boosts performance.
- OR_UPD_NOWHERE= is now a data set option, in addition to being an option in the LIBNAME statement.
- The BL_INDEX_OPTIONS= data set option enables you to specify SQL*Loader Index options with bulk-loading. You can boost performance by specifying the SORTED INDEXES index option.
- The BL_RECOVERABLE= data set option enables you to specify whether the load process is recoverable. It enhances the performance of the bulk load.
- The BL_SUPPRESS_NULLIF= data set option enables you to specify whether the load process is recoverable. It enhances the performance of the bulk load.

SAS/ACCESS Interface to Sybase

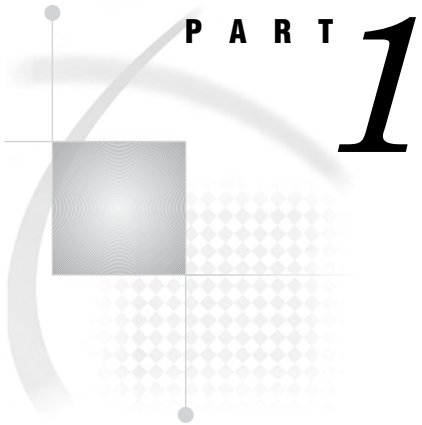
The following options are new:

- The DBSLICE= data set option specifies user-supplied WHERE clauses to partition a DBMS query into component queries for threaded reads.
- The DBSLICEPARM= data set option and the DBSLICEPARM= LIBNAME statement option control the scope of DBMS threaded reads and the number of threads.

SAS/ACCESS Interface to Teradata

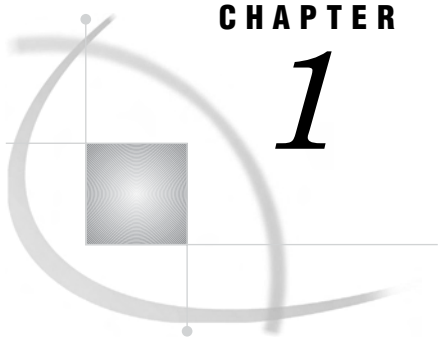
The following features and options are new:

- When processing WHERE statements that contain *literal* values for TIME or TIMESTAMP, the SAS engine passes the values to Teradata exactly as they were entered, without rounding or truncation. This behavior is new beginning with SAS 9.1.3 Service Pack 4.
- Beginning with SAS 9.1, specifying OBS=*n* causes the SAS/ACCESS Interface to Teradata to append SAMPLE *n* to the SQL that Teradata generates.
- The interface to Teradata's FastExport utility enables faster data extraction.
- The interface to Teradata supports a bulk-load capability called MultiLoad, which greatly accelerates insertion of data into both empty and existing Teradata tables. The SAS/ACCESS MultiLoad facility is similar to the native Teradata MultiLoad. You invoke the MultiLoad facility with the MULTILOAD= data set option. Note that MULTILOAD= works in conjunction with several other new MultiLoad data set options.
- The DBSLICE= data set option specifies user-supplied WHERE clauses to partition a DBMS query into component queries for threaded reads.
- The DBSLICEPARM= data set option and the DBSLICEPARM= LIBNAME statement option control the scope of DBMS threaded reads and the number of threads.
- The ERRLIMIT= option in the LIBNAME statement specifies the number of errors that are allowed while using the Fastload utility before SAS stops loading data to Teradata.
- The LOGDB= option in the LIBNAME statement specifies the name of an alternate database in which restart log tables from Teradata's FastExport utility are to be created.



PART 1

Concepts



CHAPTER

1

Overview of the SAS/ACCESS Interface to Relational Databases

About This Document

This document provides conceptual, reference, and usage information for the SAS/ACCESS Interface to relational database management systems (DBMSs). The information in this document applies generally to all of the relational database management systems that are supported by SAS/ACCESS software. *Because the availability and behavior of SAS/ACCESS features vary from one interface to another, you should use this general document in conjunction with the documentation for your SAS/ACCESS interface.* There is an individual document for each supported DBMS, and those documents are sold separately.

This document is intended for applications programmers and end users who meet the following conditions:

- familiar with the basics of their DBMS and its SQL (Structured Query Language)
- know how to use their operating environment
- can use basic SAS commands and statements.

Database administrators might also want to read this document to understand how the interface is implemented and administered.

Methods for Accessing Relational Database Data

The SAS/ACCESS interface to relational databases is a family of interfaces (each of which is licensed separately) that enable you to interact with data in other vendors' databases from within SAS. SAS/ACCESS provides the following methods for accessing relational DBMS data:

- The LIBNAME statement enables you to assign SAS librefs to DBMS objects such as schemas and databases. After a database is associated with a libref, you can

use a SAS two-level name to specify any table or view in the database and then work with the table or view as you would with a SAS data set.

- The Pass-Through Facility enables you to interact with a data source using its native SQL syntax without leaving your SAS session. The SQL statements are passed directly to the data source for processing.
- The ACCESS and DBLOAD procedures support indirect access to DBMS data. These procedures are no longer the recommended method for accessing DBMS data, but they continue to be supported for the database systems and environments on which they were available for SAS Version 6.

See "Selecting a SAS/ACCESS Method" on page 4 for information about when to use each method.

Note: Not all SAS/ACCESS interfaces support all of these features. See the section about features by host to determine which features are available in your environment. △

Selecting a SAS/ACCESS Method

Methods for Accessing DBMS Tables and Views

In SAS/ACCESS, there are often several ways to complete a task. For example, you can access DBMS tables and views by using the LIBNAME statement or the Pass-Through Facility. The advantages and limitations of these features are described below. Before processing complex or data-intensive operations, you might want to test several of these features to determine the most efficient feature for your particular task.

SAS/ACCESS LIBNAME Statement Advantages

It is generally recommended that you use the SAS/ACCESS LIBNAME statement to access your DBMS data because this is usually the fastest and most direct method. An exception to this is when you need to use non-ANSI standard SQL. ANSI standard SQL is required when you use the SAS/ACCESS library engine in the SQL procedure. The Pass-Through Facility, however, accepts all the extensions to SQL that are provided by your DBMS.

The SAS/ACCESS LIBNAME statement has the following advantages:

- Significantly fewer lines of SAS code are required to perform operations on your DBMS. For example, a single LIBNAME statement establishes a connection to your DBMS, enables you to specify how your data is processed, and enables you to easily view your DBMS tables in SAS.
- You do not need to know the SQL language of your DBMS in order to access and manipulate data on your DBMS. You can use SAS procedures, such as PROC SQL, or DATA step programming on any libref that references DBMS data. You can read, insert, update, delete, and append data, as well as create and drop DBMS tables by using SAS syntax.
- The LIBNAME statement provides more control over DBMS operations such as locking, spooling, and data type conversion through the use of LIBNAME and data set options.

- The engine can optimize the processing of joins and WHERE clauses by passing these operations directly to the DBMS. This takes advantage of your DBMS's indexing and other processing capabilities. For more information, see "Overview of Optimizing Your SQL Usage" on page 37.
- The engine can pass some functions directly to the DBMS for processing.

Pass-Through Facility Advantages

The Pass-Through Facility has the following advantages:

- Pass-Through Facility statements enable the DBMS to optimize queries, particularly when you join tables. The DBMS optimizer can take advantage of indexes on DBMS columns to process a query more quickly and efficiently.
- Pass-Through Facility statements enable the DBMS to optimize queries when the queries have summary functions (such as AVG and COUNT), GROUP BY clauses, or columns created by expressions (such as the COMPUTED function). The DBMS optimizer can use indexes on DBMS columns to process the queries more quickly.
- On some DBMSs, you can use Pass-Through Facility statements with SAS/AF applications to handle the transaction processing of the DBMS data. Using a SAS/AF application gives you complete control of COMMIT and ROLLBACK transactions. Pass-Through Facility statements give you better access to DBMS return codes.
- The Pass-Through Facility accepts all the extensions to ANSI SQL that are provided by your DBMS.

SAS/ACCESS Features for Common Tasks

The following table contains a list of tasks and the features that you can use to accomplish them.

Table 1.1 SAS/ACCESS Features for Common Tasks

Task	SAS/ACCESS Features
Read DBMS tables or views	LIBNAME statement*
	Pass-Through Facility
	View descriptors**
Create DBMS objects, such as tables	LIBNAME statement*
	DBLOAD procedure
	Pass-Through Facility's EXECUTE statement
Update, delete, or insert rows into DBMS tables	LIBNAME statement*
	View descriptors**
	Pass-Through Facility's EXECUTE statement
Append data to DBMS tables	DBLOAD procedure with APPEND option
	LIBNAME statement and APPEND procedure*
	Pass-Through Facility's EXECUTE statement

Task	SAS/ACCESS Features
List DBMS tables	LIBNAME statement and SAS Explorer window*
	LIBNAME statement and DATASETS procedure*
	LIBNAME statement and CONTENTS procedure*
	LIBNAME statement and SQL procedure dictionary tables*
Delete DBMS tables or views	LIBNAME statement and SQL procedure's DROP TABLE statement*
	LIBNAME statement and DATASETS procedure's DELETE statement*
	DBLOAD procedure with SQL DROP TABLE statement
	Pass-Through Facility's EXECUTE statement

* LIBNAME statement refers to the SAS/ACCESS LIBNAME statement.
** View descriptors refer to view descriptors that are created in the ACCESS procedure.

SAS Views of DBMS Data

SAS/ACCESS enables you to create a SAS view of data that exists in a relational database management system. A *SAS data view* defines a virtual data set that is named and stored for later use. A view contains no data, but rather describes data that is stored elsewhere. There are three types of SAS data views:

- *DATA step views* are stored, compiled DATA step programs.
- *SQL views* are stored query expressions that read data values from their underlying files, which can include SAS data files, SAS/ACCESS views, DATA step views, other SQL views, or relational database data.
- *SAS/ACCESS views* (also called view descriptors) describe data that is stored in DBMS tables. This is no longer a recommended method for accessing relational DBMS data. Use the CV2VIEW procedure to convert existing view descriptors into SQL views.

You can use all types of views as inputs into DATA steps and procedures. You can specify views in queries as if they were tables. A view derives its data from the tables or views that are listed in its FROM clause. The data accessed by a view is a subset or superset of the data in its underlying table(s) or view(s).

SQL views and SAS/ACCESS views can be used to update their underlying data if the view is based on only one DBMS table or is based on a DBMS view that is based on only one DBMS table, and if the view has no calculated fields. DATA step views cannot be used to update their underlying data; they can only read the data.

Your options for creating a SAS view of DBMS data are determined by the SAS/ACCESS feature that you are using to access the DBMS data. The following table lists the recommended methods for creating SAS views.

Table 1.2 Creating SAS Views

Feature You Use to Access DBMS Data	SAS View Technology You Can Use
SAS/ACCESS LIBNAME statement	SQL view or DATA step view of the DBMS table
SQL Procedure Pass-Through Facility	SQL view with CONNECTION TO component

CHAPTER 2

SAS Names and Support for DBMS Names

Introduction to SAS/ACCESS Naming

Because some DBMSs allow case-sensitive names and names with special characters, you must show special consideration when you use the names of DBMS objects (such as tables and columns) with SAS/ACCESS features. This section presents SAS/ACCESS naming conventions, default naming behaviors, options that can modify naming behavior, and usage examples. See the documentation for your SAS/ACCESS interface for information about how SAS handles your DBMS names.

SAS Naming Conventions

Length of Name

SAS naming conventions allow long names for SAS data sets and SAS variables. For example, MYDB.TEMP_EMPLOYEES_QTR4_2000 is a valid two-level SAS name for a data set.

The names of the following SAS language elements can be up to 32 characters in length:

- members of SAS libraries, including SAS data sets, data views, catalogs, catalog entries, and indexes
- variables in a SAS data set
- macros and macro variables.

The following SAS language elements have a maximum length of 8 characters:

- librefs and filerefs
- SAS engine names
- names of SAS/ACCESS access descriptors and view descriptors
- variable names in SAS/ACCESS access descriptors and view descriptors.

For a complete description of SAS naming conventions, see the *SAS Language Reference: Dictionary*.

Case-sensitivity

When SAS encounters mixed-case or case-sensitive names in SAS code, it stores and displays the names as they are specified. If the SAS variables Flight and dates are defined in mixed case, for example,

```
input Flight $3. +3 dates date9.;
```

then SAS displays the variable names as defined. Note how the column headings appear as defined:

Output 2.1 Mixed-Case Names Displayed in Output

```
          SAS System

   Obs    Flight        dates

    1      114       01MAR2000
    2      202       01MAR2000
    3      204       01MAR2000
```

Although SAS *stores* variable names as they are defined, it *recognizes* variables for processing without regard to case. For example, SAS processes these variables as FLIGHT and DATES. Likewise, renaming the Flight variable to "flight" or "FLIGHT" would result in the same processing.

SAS Name Literals

A SAS *name literal* is a name token that is expressed as a quoted string, followed by the letter **n**. Name literals enable you to use special characters or blanks that are not otherwise allowed in SAS names when you specify a SAS data set or variable. Name literals are especially useful for expressing database column and tables names that contain special characters.

Examples of name literals are

```
data mydblib.'My Staff Table'n;
```

and

```
data Budget_for_1999;

input '$ Amount Budgeted'n 'Amount Spent'n;
```

Name literals are subject to certain restrictions:

- You can use a name literal only for SAS variable and data set names, statement labels, and DBMS column and table names.
- You can use name literals only in a DATA step or in the SQL procedure.
- If a name literal contains any characters that are not allowed when VALIDVARNAME=V7, then you must set the system option to VALIDVARNAME=ANY. For details about using the VALIDVARNAME= system option, see "VALIDVARNAME= System Option" on page 275.

SAS/ACCESS Default Naming Behaviors

Modification and Truncation

When SAS/ACCESS reads DBMS column names that contain characters that are not standard in SAS names, the default behavior is to replace an unsupported character with an underscore (_). For example, the DBMS column name Amount Budgeted$ becomes the SAS variable name Amount_Budgeted_.

Note: Nonstandard names include those with blank spaces or special characters (such as @, #, %) that are not allowed in SAS names. △

When SAS/ACCESS encounters a DBMS name that exceeds 32 characters, it truncates the name.

After it has modified or truncated a DBMS column name, SAS appends a number to the variable name, if necessary, to preserve uniqueness. For example, DBMS column names MY$DEPT, My$Dept, and my$dept become SAS variable names MY_DEPT, MY_Dept0, and my_dept1.

ACCESS Procedure

If you attempt to use long names in the ACCESS procedure, you get an error message advising you that long names are not supported. Long member names, such as access descriptor and view descriptor names, are truncated to 8 characters. Long DBMS column names are truncated to 8-character SAS variable names within the SAS access descriptor. You can use the RENAME statement to specify 8-character SAS variable names, or you can accept the default truncated SAS variable names that are assigned by the ACCESS procedure.

The ACCESS procedure converts DBMS object names to uppercase characters unless they are enclosed in quotation marks. Any DBMS objects that are given lowercase names when they are created, or whose names contain special or national characters, must be enclosed in quotation marks.

DBLOAD Procedure

You can use long member names, such as the name of a SAS data set that you want to load into a DBMS table, in the DBLOAD procedure DATA= option. However, if you attempt to use long SAS variable names, you get an error message advising you that long variable names are not supported in the DBLOAD procedure. You can use the RENAME statement to rename the 8-character SAS variable names to long DBMS column names when you load the data into a DBMS table. You can also use the SAS data set option RENAME to rename the columns after they are loaded into the DBMS.

Most DBLOAD procedure statements convert lowercase characters in user-specified values and default values to uppercase. If your host or database is case-sensitive and you want to specify a value that includes lowercase alphabetic characters (for example, a user ID or password), enclose the entire value in quotation marks. You must also put quotation marks around any value that contains special characters or national characters.

The only exception is the DBLOAD SQL statement. The DBLOAD SQL statement is passed to the DBMS exactly as you type it, with case preserved.

Renaming DBMS Data

Renaming SAS/ACCESS Tables

You can rename DBMS tables and views using the CHANGE statement, as shown in the following example:

```
proc datasets lib=x;
  change oldtable=newtable;
quit;
```

You can rename tables using this method for the following engines:

DB2 z/OS

DB2 UNIX/PC

Informix

Microsoft SQL Server

ODBC

OLE DB

Oracle

Sybase

Teradata.

Note: If you change a table name, any view dependent on that table will no longer work, unless the view references the new table name. △

Renaming SAS/ACCESS Columns

You can use the RENAME statement to rename the 8-character default SAS variable names to long DBMS column names when you load the data into a DBMS table. You can also use the SAS data set option RENAME= to rename the columns after they are loaded into the DBMS.

Renaming SAS/ACCESS Variables

You can use the RENAME statement to specify 8-character SAS variable names such as access descriptors and view descriptors.

Options That Affect SAS/ACCESS Naming Behavior

To change how SAS handles case-sensitive or nonstandard DBMS table and column names, specify one or more of the following options.

PRESERVE_COL_NAMES=YES
is a SAS/ACCESS LIBNAME and data set option that applies *only* to creating DBMS tables. When set to YES, this option preserves spaces, special characters, and mixed case in DBMS column names. See "PRESERVE_COL_NAMES= LIBNAME Option" on page 128 for more information about this option.

PRESERVE_TAB_NAMES=YES
is a SAS/ACCESS LIBNAME option. When set to YES, this option preserves blank spaces, special characters, and mixed case in DBMS table names. See "PRESERVE_TAB_NAMES= LIBNAME Option" on page 130 for more information about this option.

Note: Specify the alias PRESERVE_NAMES=YES | NO if you plan to specify both the PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options in your LIBNAME statement. Using this alias saves time when you are coding. △

DQUOTE=ANSI
is a PROC SQL option. This option specifies whether PROC SQL treats values within *double* quotation marks as a character string or as a column name or table name. When you specify DQUOTE=ANSI, your SAS code can refer to DBMS names that contain characters and spaces that are not allowed by SAS naming conventions. Specifying DQUOTE=ANSI enables you to preserve special characters in table and column names in your SQL statements by enclosing the names in double quotation marks.

To preserve table names, you must also specify PRESERVE_TAB_NAMES=YES. To preserve column names when you create a table, you must also specify PRESERVE_COL_NAMES=YES.

VALIDVARNAME=ANY
is a global system option that can override the SAS naming conventions. See "VALIDVARNAME= System Option" on page 275 for information about this option.

The availability of these options and their default settings are DBMS-specific, so consult the SAS/ACCESS documentation for your DBMS to learn how the SAS/ACCESS engine for your DBMS processes names.

Naming Behavior When Retrieving DBMS Data

The following two tables illustrate how SAS/ACCESS processes DBMS names when retrieving data from a DBMS. This information applies generally to all the interfaces. In some cases, however, it is not necessary to specify these options because the option default values are DBMS-specific. See the documentation for your SAS/ACCESS interface for details.

Table 2.1 DBMS Column Names to SAS Variable Names When Reading DBMS Data

DBMS Column Name	Desired SAS Variable Name	Options
Case-sensitive DBMS column name, such as Flight	Case-sensitive SAS variable name, such as Flight	No options are necessary
DBMS column name with characters that are not valid in SAS names, such as My$Flight	Case-sensitive SAS variable name where an underscore replaces the invalid characters, such as My_Flight	No options are necessary
DBMS column name with characters that are not valid in SAS names, such as My$Flight	Nonstandard, case-sensitive SAS variable name, such as My$Flight	PROC SQL DQUOTE=ANSI or, in a DATA or PROC step, use a SAS name literal such as 'My$Flight'n and VALIDVARNAME=ANY

Table 2.2 DBMS Table Names to SAS Data Set Names When Reading DBMS Data

DBMS Table Name	Desired SAS Data Set Name	Options
Default DBMS table name, such as STAFF	Default SAS data set or member name (uppercase), such as STAFF	PRESERVE_TAB_NAMES=NO
Case-sensitive DBMS table name, such as Staff	Case-sensitive SAS data set, such as Staff	PRESERVE_TAB_NAMES=YES
DBMS table name with characters that are not valid in SAS names, such as All$Staff	Nonstandard, case-sensitive SAS data set name, such as All$Staff	PROC SQLDQUOTE=ANSI and PRESERVE_TAB_NAMES=YES or, in a DATA step or PROC, use a SAS name literal such as 'All$Staff'n and PRESERVE_TAB_NAMES=YES

Naming Behavior When Creating DBMS Objects

The following two tables illustrate how SAS/ACCESS handles variable names when creating DBMS objects such as tables and views. This information applies generally to all the interfaces. In some cases, however, it is not necessary to specify these options because the option default values are DBMS-specific. See the documentation for your DBMS for details.

Table 2.3 SAS Variable Names to DBMS Column Names When Creating Tables

SAS Variable Name as Input	Desired DBMS Column Name	Options
Any SAS variable name, such as Miles	Default DBMS column name (normalized to follow the DBMS's naming conventions), such as MILES	PRESERVE_COL_NAMES=NO
A case-sensitive SAS variable name, such as Miles	Case-sensitive DBMS column name, such as Miles	PRESERVE_COL_NAMES=YES
A SAS variable name with characters that are not valid in a normalized SAS name, such as Miles-to-Go	Case-sensitive DBMS column name that matches the SAS name, such as Miles-to-Go	PROC SQL DQUOTE=ANSI and PRESERVE_COL_NAMES=YES or, in a DATA or PROC step, use a SAS name literal and PRESERVE_COL_NAMES=YES and VALIDVARNAME=ANY

Table 2.4 SAS Data Set Names to DBMS Table Names

SAS Data Set Name as Input	Desired DBMS Table Name	Options
Any SAS data set name, such as Payroll	Default DBMS table name (normalized to follow the DBMS's naming conventions), such as PAYROLL	PRESERVE_TAB_NAMES=NO
A case-sensitive SAS data set name, such as Payroll	Case-sensitive DBMS table name, such as Payroll	PRESERVE_TAB_NAMES=YES
A case-sensitive SAS data set name with characters that are not valid in a normalized SAS name, such as Payroll-for-QC	Case-sensitive DBMS table name that matches the SAS name, such as Payroll-for-QC	PROC SQL DQUOTE=ANSI and PRESERVE_TAB_NAMES=YES or, in a DATA or PROC step, use a SAS name literal and PRESERVE_TAB_NAMES=YES

SAS/ACCESS Naming Examples

Replacing Unsupported Characters

In the following example, a view, myview, is created from the Oracle table mytable.

```
proc sql;
connect to oracle (user=testuser password=testpass);
create view myview as
   select * from connection to oracle
      (select "Amount Budgeted$", "Amount Spent$"
         from mytable);
quit;

proc contents data=myview;
run;
```

In the output produced by PROC CONTENTS, the Oracle column names (that were processed by the SQL view of MYTABLE) are renamed to different SAS variable names: Amount Budgeted$ becomes Amount_Budgeted_ and Amount Spent$ becomes Amount_Spent_.

Preserving Column Names

The following example uses the Oracle table PAYROLL to create a new Oracle table, PAY1, and then prints the table. Both the PRESERVE_COL_NAMES=YES and the PROC SQL DQUOTE=ANSI options are used to preserve the case and nonstandard characters in the column names. You do not need to quote the column aliases in order to preserve the mixed case. You only need double quotation marks when the column name has nonstandard characters or blanks.

By default, most SAS/ACCESS interfaces use DBMS-specific rules to set the case of table and column names. Therefore, even though the new Oracle table name pay1 is created in lowercase in this example, Oracle stores the name in uppercase as PAY1. If you want the table name to be stored as "pay1", you must set PRESERVE_TAB_NAMES=NO.

```
options linesize=120 pagesize=60 nodate;

libname mydblib oracle user=testuser password=testpass path='ora8_servr'
        schema=hrdept preserve_col_names=yes;

proc sql dquote=ansi;
create table mydblib.pay1 as
   select idnum as "ID #", sex, jobcode, salary,
          birth as BirthDate, hired as HiredDate
       from mydblib.payroll
   order by birth;

title "Payroll Table with Revised Column Names";
select * from mydblib.pay1;
quit;
```

SAS recognizes the JOBCODE, SEX, and SALARY column names, whether you specify them in your SAS code as lowercase, mixed case, or uppercase. In the Oracle table PAYROLL, the SEX, JOBCODE, and SALARY columns were created in uppercase; therefore, they retain this case in the new table (unless you rename them). A partial output from the example is shown:

Output 2.2 DBMS Table Created with Nonstandard and Standard Column Names

```
                Payroll Table with Revised Column Names

ID #  SEX  JOBCODE    SALARY             BirthDate             HiredDate
-------------------------------------------------------------------------
1118  M    PT3         11379    16JAN1944:00:00:00    18DEC1980:00:00:00
1065  M    ME2         35090    26JAN1944:00:00:00    07JAN1987:00:00:00
1409  M    ME3         41551    19APR1950:00:00:00    22OCT1981:00:00:00
1401  M    TA3         38822    13DEC1950:00:00:00    17NOV1985:00:00:00
1890  M    PT2         91908    20JUL1951:00:00:00    25NOV1979:00:00:00
```

Preserving Table Names

The following example uses PROC PRINT to print the DBMS table PAYROLL. The DBMS table was created in uppercase and since PRESERVE_TAB_NAMES=YES, the table name must be specified in uppercase. (If you set the PRESERVE_TAB_NAMES=NO, you can specify the DBMS table name in lowercase.) A partial output follows the example.

```
options nodate linesize=64;
libname mydblib oracle user=testuser password=testpass
        path='ora8_servr' preserve_tab_names=yes;

proc print data=mydblib.PAYROLL;
   title 'PAYROLL Table';
run;
```

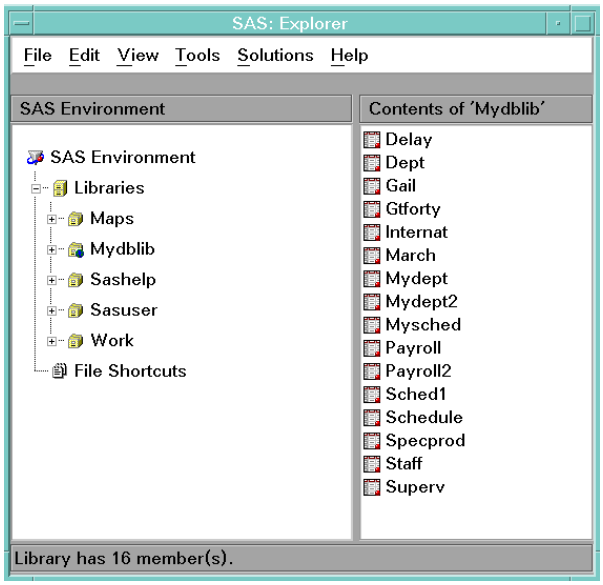
Output 2.3 DBMS Table with a Case-sensitive Name

```
                          PAYROLL Table
Obs   IDNUM     SEX     JOBCODE     SALARY                   BIRTH
1     1919       M        TA2         34376       12SEP1960:00:00:00
2     1653       F        ME2         35108       15OCT1964:00:00:00
3     1400       M        ME1         29769       05NOV1967:00:00:00
4     1350       F        FA3         32886       31AUG1965:00:00:00
5     1401       M        TA3         38822       13DEC1950:00:00:00
```

The following example submits a SAS/ACCESS LIBNAME statement and then opens the SAS Explorer window, which lists the Oracle tables and views that are referenced by the MYDBLIB libref. Notice that 16 members are listed and that all of the member names are in the case (initial capitalization) that is set by the Explorer window. The table names are capitalized because PRESERVE_TAB_NAMES= defaulted to NO.

```
libname mydblib oracle user=testuser pass=testpass;
```

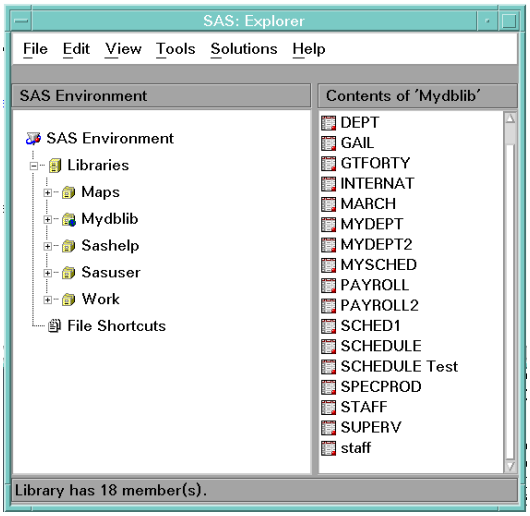
Display 2.1 SAS Explorer Window Listing DBMS Objects



If you submit a SAS/ACCESS LIBNAME statement with PRESERVE_TAB_NAMES=YES and then open the SAS Explorer window, you see a different listing of the Oracle tables and views referenced by the MYDBLIB libref.

```
libname mydblib oracle user=testuser password=testpass
        preserve_tab_names=yes;
```

Display 2.2 SAS Explorer Window Listing Case-Sensitive DBMS Objects



Notice that there are 18 members listed, including one that is in lowercase and one that has a name separated by a blank space. Because PRESERVE_TAB_NAMES=YES, SAS displays the tables names in the exact case in which they were created.

Using DQUOTE=ANSI

The following example creates a DBMS table with a blank space in its name. Double quotation marks are used to specify the table name, International Delays. Both of the preserve names LIBNAME options are also set by using the alias PRESERVE_NAMES=. Because PRESERVE_NAMES=YES, the schema airport is now case-sensitive for Oracle.

```
options linesize=64 nodate;

libname mydblib oracle user=testuser password=testpass path='airdata'
        schema=airport preserve_names=yes;

proc sql dquote=ansi;
create table mydblib."International Delays" as
   select int.flight as "FLIGHT NUMBER", int.dates,
          del.orig as ORIGIN,
          int.dest as DESTINATION, del.delay
      from mydblib.INTERNAT as int,
           mydblib.DELAY as del
      where int.dest=del.dest and int.dest='LON';
quit;

proc sql dquote=ansi outobs=10;
```

```
   title "International Delays";
select * from mydblib."International Delays";
```

Notice that you use single quotation marks to specify the data value for London (**int.dest='LON'**) in the WHERE clause. Because of the preserve name LIBNAME options, using double quotation marks would cause SAS to interpret this data value as a column name.

Output 2.4 DBMS Table with Nonstandard Column Names

```
                    International Delays

 FLIGHT
 NUMBER                  DATES  ORIGIN  DESTINATION     DELAY
 ------------------------------------------------------------
 219       01MAR1998:00:00:00  LGA     LON                18
 219       02MAR1998:00:00:00  LGA     LON                18
 219       03MAR1998:00:00:00  LGA     LON                18
 219       04MAR1998:00:00:00  LGA     LON                18
 219       05MAR1998:00:00:00  LGA     LON                18
 219       06MAR1998:00:00:00  LGA     LON                18
 219       07MAR1998:00:00:00  LGA     LON                18
 219       01MAR1998:00:00:00  LGA     LON                18
 219       02MAR1998:00:00:00  LGA     LON                18
 219       03MAR1998:00:00:00  LGA     LON                18
```

If you query a DBMS table and use a label to change the FLIGHT NUMBER column name to a standard SAS name (Flight_Number), a label (enclosed in single quotation marks) changes the name only in the output. Because this column name and the table name, International Delays, each have a space in their names, you have to enclose the names in double quotation marks. A partial output follows the example.

```
options linesize=64 nodate;

libname mydblib oracle user=testuser password=testpass path='airdata'
        schema=airport preserve_names=yes;

proc sql dquote=ansi outobs=5;
   title "Query from International Delays";
select "FLIGHT NUMBER" label='Flight_Number', dates, delay
   from mydblib."International Delays";
```

Output 2.5 Query Renaming a Nonstandard Column to a Standard SAS Name

```
    Query from International Delays

 Flight_
 Number                  DATES     DELAY
 ---------------------------------------
 219       01MAR1998:00:00:00        18
 219       02MAR1998:00:00:00        18
 219       03MAR1998:00:00:00        18
 219       04MAR1998:00:00:00        18
 219       05MAR1998:00:00:00        18
```

You can preserve special characters by specifying DQUOTE=ANSI and using double quotation marks around the SAS names in your SELECT statement.

```
proc sql dquote=ansi;
  connect to oracle (user=testuser password=testpass);
  create view myview as
    select "Amount Budgeted$", "Amount Spent$"
    from connection to oracle
      (select "Amount Budgeted$", "Amount Spent$"
        from mytable);
quit;
proc contents data=myview;
run;
```

Output from this example would show that Amount Budgeted$ remains Amount Budgeted$ and Amount Spent$ remains Amount Spent$.

Using Name Literals

The following example creates a table using name literals. You must specify the SAS option VALIDVARNAME=ANY in order to use name literals. Use PROC SQL to print the new DBMS table because name literals work only with PROC SQL and the DATA step. PRESERVE_COLUMN_NAMES=YES is required *only* because the table is being created with nonstandard SAS column names.

```
options ls=64 validvarname=any nodate;

libname mydblib oracle user=testuser password=testpass path='ora8servr'
preserve_col_names=yes preserve_tab_names=yes ;

data mydblib.'Sample Table'n;
   'EmpID#'n=12345;
   Lname='Chen';
   'Salary in $'n=63000;

proc sql;
   title "Sample Table";
   select * from mydblib.'Sample Table'n;
```

Output 2.6 DBMS Table to Test Column Names

```
          Sample Table

                      Salary
    EmpID#  Lname       in $
   -------------------------
      12345  Chen       63000
```

Using DBMS Data to Create a DBMS Table

The following example uses PROC SQL to create a DBMS table based on data from other DBMS tables. You preserve the case-sensitivity of the aliased column names by using PRESERVE_COL_NAMES=YES. A partial output is displayed after the code.

```
libname mydblib oracle user=testuser password=testpass
        path='hrdata99' schema=personnel preserve_col_names=yes;
```

```
proc sql;
create table mydblib.gtforty as
   select lname as LAST_NAME,
          fname as FIRST_NAME,
          salary as ANNUAL_SALARY
      from mydblib.staff a,
           mydblib.payroll b
   where (a.idnum eq b.idnum) and
          (salary gt 40000)
   order by lname;

proc print noobs;
   title 'Employees with Salaries over $40,000';
run;
```

Output 2.7 Updating DBMS Data

```
      Employees with Salaries over $40,000

                                          ANNUAL_
  LAST_NAME          FIRST_NAME          SALARY

  BANADYGA           JUSTIN               88606
  BAREFOOT           JOSEPH               43025
  BRADY              CHRISTINE            68767
  BRANCACCIO         JOSEPH               66517
  CARTER-COHEN       KAREN                40260
  CASTON             FRANKLIN             41690
  COHEN              LEE                  91376
  FERNANDEZ          KATRINA              51081
```

Using a SAS Data Set to Create a DBMS Table

The following example uses a SAS data step to create a DBMS table, College-Hires-1999, from a temporary SAS data set that has case-sensitive names. It creates the temporary data set and then defines the LIBNAME statement. Because it uses a DATA step to create the DBMS table, it must specify the table name as a name literal and specify the PRESERVE_TAB_NAMES= and PRESERVE_COL_NAMES= options (in this case, by using the alias PRESERVE_NAMES=).

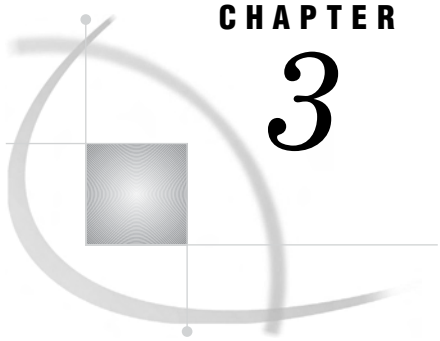
```
options validvarname=any nodate;

data College_Hires_1999;
   input IDnum $4. +3 Lastname $11. +2
         Firstname $10. +2 City $15. +2
         State $2.;
   datalines;
3413    Schwartz     Robert       New Canaan     CT
3523    Janssen      Heike        Stamford       CT
3565    Gomez        Luis         Darien         CT
;

libname mydblib oracle user=testuser password=testpass
        path='hrdata99' schema=hrdept preserve_names=yes;
```

```
data mydblib.'College-Hires-1999'n;
   set College_Hires_1999;

proc print;
   title 'College Hires in 1999';
run;
```

Output 2.8 DBMS Table with Case-Sensitive Table and Column Names

College Hires in 1999

Obs	IDnum	Lastname	Firstname	City	State
1	3413	Schwartz	Robert	New Canaan	CT
2	3523	Janssen	Heike	Stamford	CT
3	3565	Gomez	Luis	Darien	CT



CHAPTER

3 Data Integrity and Security

Introduction to Data Integrity and Security

This section briefly describes DBMS security issues and then presents measures you can take on the SAS side of the interface to help protect DBMS data from accidental update or deletion. This section also provides information about how SAS handles null values that will help you achieve consistent results.

DBMS Security

Privileges

The database administrator (DBA) controls who has privileges to access or update DBMS objects. The DBA also controls who can create objects, and creators of the objects control who can access the objects. A user cannot use DBMS facilities to access DBMS objects through SAS/ACCESS software unless the user has the appropriate DBMS privileges or authority on those objects. You can grant privileges on the DBMS side by using the Pass-Through Facility to EXECUTE an SQL statement, or by issuing a GRANT statement from the DBLOAD procedure SQL statement.

On the DBMS, you should give users only the privileges they must have. Privileges are granted on whole tables or views. A user must explicitly be granted privileges on the DBMS tables or views that underlie a view in order to use that view.

Note: See your DBMS documentation for more information about ensuring security on the DBMS side of the interface. △

Triggers

If your DBMS supports triggers, you can use them to enforce security authorizations or business-specific security considerations. When and how triggers are executed is determined by when the SQL statement is executed and how often the trigger is executed. Triggers can be executed before an SQL statement is executed, after an SQL statement is executed, or for each row of an SQL statement. Also, triggers can be defined for DELETE, INSERT, and UPDATE statement execution.

Enabling triggers can provide more specific security for delete, insert, and update operations. SAS/ACCESS abides by all constraints and actions that are specified by a trigger. For more information, see the documentation for your DBMS.

SAS Security

Securing Data

SAS preserves the data security provided by your DBMS and operating system; SAS/ACCESS does not override the security of your DBMS. To secure DBMS data from accidental update or deletion, you can take steps on the SAS side of the interface such as the following:

- specifying the SAS/ACCESS LIBNAME option DBPROMPT= to avoid saving connection information in your code
- creating SQL views and protecting them from unauthorized access by applying passwords.

These and other approaches are discussed in detail in the following sections.

Assigning SAS Passwords

By using SAS passwords, you can protect SQL views, SAS data sets, and descriptor files from unauthorized access. The following table summarizes the levels of protection that SAS passwords provide. Note that you can assign multiple levels of protection.

Table 3.1 Password Protection Levels and Their Effects

File Type	READ=	WRITE=	ALTER=
PROC SQL view of DBMS data	Protects the underlying data from being read or updated through the view; does not protect against replacement of the view	Protects the underlying data from being updated through the view; does not protect against replacement of the view	Protects the view from being modified, deleted, or replaced
Access descriptor	No effect on descriptor	No effect on descriptor	Protects the descriptor from being read or edited
View descriptor	Protects the underlying data from being read or updated through the view	Protects the underlying data from being updated through the view	Protects the descriptor from being read or edited

You can use the following methods to assign, change, or delete a SAS password:

- the global SETPASSWORD command, which opens a dialog box
- the DATASETS procedure's MODIFY statement.

The syntax for using PROC DATASETS to assign a password to an access descriptor, a view descriptor, or a SAS data file is as follows:

PROC DATASETS LIBRARY=*libref* MEMTYPE=*member-type*;
MODIFY *member-name* (*password-level* = *password-modification*);

RUN;

The *password-level* argument can have one or more of the following values: READ=, WRITE=, ALTER=, or PW=. PW= assigns read, write, and alter privileges to a descriptor or data file. The *password-modification* argument enables you to assign a new password or to change or delete an existing password. For example, this PROC DATASETS statement assigns the password MONEY with the ALTER level of protection to the access descriptor ADLIB.SALARIES:

```
proc datasets library=adlib memtype=access;
   modify salaries (alter=money);
run;
```

In this case, users are prompted for the password whenever they try to browse or update the access descriptor or try to create view descriptors that are based on ADLIB.SALARIES.

In the next example, the PROC DATASETS statement assigns the passwords MYPW and MYDEPT with READ and ALTER levels of protection to the view descriptor VLIB.JOBC204:

```
proc datasets library=vlib memtype=view;
   modify jobc204 (read=mypw alter=mydept);
run;
```

In this case, users are prompted for the SAS password when they try to read the DBMS data or try to browse or update the view descriptor VLIB.JOBC204. You need both levels to protect the data and descriptor from being read. However, a user could still update the data accessed by VLIB.JOBC204, for example, by using a PROC SQL UPDATE. Assign a WRITE level of protection to prevent data updates.

Note: When you assign multiple levels of passwords, use a different password for each level to ensure that you grant only the access privileges that you intend. △

To delete a password, put a slash after the password:

```
proc datasets library=vlib memtype=view;
   modify jobc204 (read=mypw/ alter=mydept/);
run;
```

Protecting Connection Information

In addition to directly controlling access to data, you can protect the data indirectly by protecting the connection information that SAS/ACCESS uses to reach the DBMS. Generally, this is achieved by not saving connection information in your code.

One way to protect connection information is by storing user name, password, and other connection options in a local environment variable. Access to the DBMS is denied unless the correct user and password information is stored in a local environment variable. See the documentation for your DBMS to determine whether this alternative is supported.

Another way to protect connection information is by requiring users to manually enter it at connection time. When you specify DBPROMPT=YES in a SAS/ACCESS LIBNAME statement, each user has to provide DBMS connection information in a dynamic, interactive manner. This is demonstrated in the following statement, which causes a dialog box to prompt the user to enter connection information, such as a username and password:

```
libname myoralib oracle dbprompt=yes defer=no;
```

The dialog box that appears contains the DBMS connection options that are valid for the SAS/ACCESS engine that is being used; in this case, Oracle.

Using the DBPROMPT= option on the LIBNAME statement offers several advantages. DBMS account passwords are protected because they do not need to be stored in a SAS program or descriptor file. Also, when a password or username changes, the SAS program does not need to be modified. Another advantage is that the same SAS program can be used by any valid username and password combination that is specified during execution. You can also use connection options in this interactive manner when you want to run a program on a production server instead of testing a server without making modifications to your code. By using the prompt window, the new server name can be specified dynamically.

Note: The DBPROMPT= option is not available in the SAS/ACCESS interface to DB2 under z/OS. △

Extracting DBMS Data to a SAS Data Set

If you are the owner of a DBMS table and do not want anyone else to read the data, you might want to extract the data (or a subset of the data) and not distribute information about either the access descriptor or view descriptor.

Note: You might need to take additional steps to restrict LIBNAME or Pass-Through access to the extracted data set. △

If you extract data from a view that has a SAS password assigned to it, the new SAS data file is automatically assigned the same password. If a view does not have a password, you can assign a password to the extracted SAS data file by using the MODIFY statement in the DATASETS procedure. See the *Base SAS Procedures Guide* for more information.

Defining Views and Schemas

If you want to provide access to some but not all fields in a DBMS table, you can create a SAS view that prohibits access to the sensitive data by specifying that particular columns be dropped. Columns that are dropped from views do not affect the underlying DBMS table and can be reselected for later use.

Some SAS/ACCESS engines support LIBNAME options that restrict or qualify the scope, or schema, of the tables in the libref. For example, the DB2 engine supports the AUTHID= and LOCATION= options, and the Oracle engine supports the SCHEMA= and DBLINK= options. See the SAS/ACCESS documentation for your DBMS to determine which options are available to you.

The following example uses the SAS/ACCESS interface to Oracle:

```
libname myoralib oracle user=testuser password=testpass
        path='myoraserver' schema=testgroup;

proc datasets lib=myoralib;
run;
```

In this example, the MYORALIB libref is associated with the Oracle schema named TESTGROUP. The DATASETS procedure lists only the tables and views that are accessible to the TESTGROUP schema. Any reference to a table that uses the libref MYORALIB is passed to the Oracle server as a qualified table name; for example, if the SAS program reads a table by specifying the SAS data set MYORALIB.TESTTABLE, the SAS/ACCESS engine passes the following query to the server:

```
select * from "testgroup.testtable"
```

Controlling DBMS Connections

Because the overhead of executing a connection to a DBMS server can be resource-intensive, SAS/ACCESS supports the CONNECTION= and DEFER= options to control when a DBMS connection is made, and how many connections are executed within the context of your SAS/ACCESS application. For most SAS/ACCESS engines, a connection to a DBMS begins one transaction, or work unit, and all statements issued in the connection execute within the context of the active transaction.

The CONNECTION= LIBNAME option enables you to specify how many connections are executed when the library is used and which operations on tables are shared within a connection. By default, the value is CONNECTION=SHAREDREAD, which means that a SAS/ACCESS engine executes a *shared read* DBMS connection when the library is assigned. Every time a table in the library is read, the read-only connection is used. However, if an application attempts to update data using the libref, a separate connection is issued, and the update occurs in the new connection. As a result, there is one connection for read-only transactions and a separate connection for each update transaction.

In the following example, the SAS/ACCESS engine issues a connection to the DBMS when the libref is assigned. The PRINT procedure reads the table by using the first connection. When the PROC SQL updates the table, the update is performed with a second connection to the DBMS.

```
libname myoralib oracle user=testuser password=testpass
        path='myoraserver';

proc print data=myoralib.mytable;
run;
```

```
proc sql;
   update myoralib.mytable set acctnum=123
      where acctnum=456;
quit;
```

The following example uses the SAS/ACCESS interface to DB2 under z/OS. The LIBNAME statement executes a connection by way of the DB2 Call Attach Facility to the DB2 DBMS server:

```
libname mydb2lib db2 authid=testuser;
```

If you want to assign more than one SAS libref to your DBMS server, and if you do not plan to update the DBMS tables, SAS/ACCESS enables you to optimize the way in which the engine performs connections. Your SAS librefs can share a single read-only connection to the DBMS if you use the CONNECTION=GLOBALREAD option. The following example shows you how to use the CONNECTION= option with the ACCESS= option to control your connection and to specify read-only data access.

```
libname mydblib1 db2 authid=testuser
   connection=globalread access=readonly;
```

If you do not want the connection to occur when the library is assigned, you can delay the connection to the DBMS by using the DEFER= option. When you specify DEFER=YES on the LIBNAME statement, for example,

```
libname mydb2lib db2 authid=testuser defer=yes;
```

the SAS/ACCESS engine connects to the DBMS the first time a DBMS object is referenced in a SAS program.

Note: If you use DEFER=YES to assign librefs to your DBMS tables and views in an AUTOEXEC program, the processing of the AUTOEXEC file is faster because the connections to the DBMS are not made every time SAS is invoked. △

Locking, Transactions, and Currency Control

SAS/ACCESS provides options that enable you to control some of the row, page, or table locking operations that are performed by the DBMS and the SAS/ACCESS engine as your programs are executed. For example, by default, the SAS/ACCESS Oracle engine does not lock any data when it reads rows from Oracle tables. However, you can override this behavior by using the locking options that are supported in the SAS/ ACCESS interface to Oracle.

If you want to lock the data pages of a table while SAS is reading the data to prevent other processes from updating the table, you can use the READLOCK_TYPE= option, as in the following example:

```
libname myoralib oracle user=testuser pass=testpass
        path='myoraserver' readlock_type=table;

data work.mydata;
   set myoralib.mytable(where=(colnum > 123));
run;
```

In this example, the SAS/ACCESS Oracle engine obtains a TABLE SHARE lock on the table so that the data cannot be updated by other processes while your SAS program is reading it.

In the following example, Oracle acquires row-level locks on rows read for update in the tables in the libref.

```
libname myoralib oracle user=testuser password=testpass
   path='myoraserver' updatelock_type=row;
```

Note: Each SAS/ACCESS interface supports specific options; see the SAS/ACCESS documentation for your DBMS to determine which options it supports. △

Customizing DBMS Connect and Disconnect Exits

You can specify DBMS commands or stored procedures to be executed immediately after a DBMS connection or before a DBMS disconnect by using the LIBNAME options DBCONINIT= and DBCONTERM=, as in the following example:

```
libname myoralib oracle user=testuser password=testpass
        path='myoraserver' dbconinit="EXEC MY_PROCEDURE";

proc sql;
   update myoralib.mytable set acctnum=123
      where acctnum=567;
quit;
```

When the libref is assigned, the SAS/ACCESS engine connects to the DBMS and passes a command to the DBMS to execute the stored procedure MY_PROCEDURE. By default, a new connection to the DBMS is made for every table that is opened for updating, so MY_PROCEDURE is executed a second time after a connection is made to update the table MYTABLE.

To execute a DBMS command or stored procedure *only* after the first connection in a library assignment, you can use the DBLIBINIT= option. Similarly, the DBLIBTERM= option enables you to specify a command to be executed prior to the disconnection of only the first library connection, as in the following example:

```
libname myoralib oracle user=testuser password=testpass
        dblibinit="EXEC MY_INIT" dblibterm="EXEC MY_TERM";
```

Potential Result Set Differences When Processing Null Data

When your data contains null values or when internal processing generates intermediate data sets that contain null values, you might get different result sets depending on whether the processing is done by SAS or by the DBMS. Although in many cases this does not present a problem, it is important to understand how these differences occur.

Most relational database systems have a special value called null, which means an absence of information and is analogous to a SAS missing value. SAS/ACCESS translates SAS missing values to DBMS null values when creating DBMS tables from within SAS and, conversely, translates DBMS null values to SAS missing values when reading DBMS data into SAS.

There is, however, an important difference in the behavior of DBMS null values and SAS missing values:

- A DBMS null value is interpreted as the absence of data, so you cannot sort a DBMS null value or evaluate it with standard comparison operators.
- A SAS missing value is interpreted as its internal floating point representation because SAS supports 28 missing values (where a period (.) is the most common missing value). Because SAS supports multiple missing values, you can sort a SAS missing value and evaluate it with standard comparison operators.

This means that SAS and the DBMS interpret null values differently, which has significant implications when SAS/ACCESS passes queries to a DBMS for processing. This can be an issue in the following situations:

- when filtering data (for example, in a WHERE clause, a HAVING clause, or an outer join ON clause). SAS interprets null values as missing; many DBMS exclude null values from consideration. For example, if you have null values in a DBMS column that is used in a WHERE clause, your results might differ depending on whether the WHERE clause is processed in SAS or is passed to the DBMS for processing. This is because the DBMS removes null values from consideration in a WHERE clause, but SAS does not.
- when using certain functions. For example, if you use the MIN aggregate function on a DBMS column that contains null values, the DBMS does not consider the null values, but SAS interprets the null values as missing, which affects the result.
- when submitting outer joins where internal processing generates nulls for intermediate result sets.
- when sorting data. SAS sorts null values low; most DBMSs sort null values high. (See “Sorting DBMS Data” on page 33 for more information.)

For example, create a simple data set that consists of one observation and one variable:

```
libname myoralib oracle user=testuser password=testpass;
data myoralib.table;
x=.;       /*create a missing value */
run;
```

Then, print the data set using a WHERE clause, which SAS/ACCESS passes to the DBMS for processing:

```
proc print data=myoralib.table;
   where x<0;
run;
```

The log indicates that no observations were selected by the WHERE clause, because Oracle interprets the missing value as the absence of data, and does not evaluate it with the less-than (<) comparison operator.

When there is the potential for inconsistency, consider using one of the following strategies:

- Use the LIBNAME option DIRECT_SQL= to control whether the processing is done by SAS or by the DBMS.
- Use the Pass-Through Facility to ensure that the processing is done by the DBMS.
- Add the "is not null" expression to WHERE clauses and ON clauses to ensure that you will get the same result regardless of whether SAS or the DBMS does the processing.

Note: Use the data set option NULLCHAR= to specify how the DBMS interprets missing SAS character values when updating DBMS data or inserting rows into a DBMS table. △

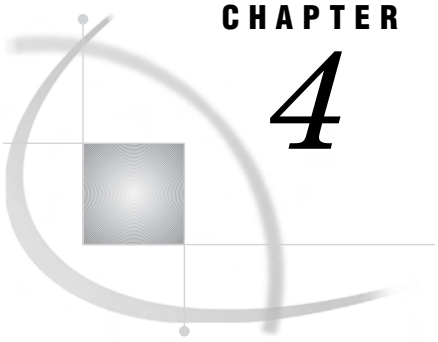
You can use the first of these strategies to force SAS to process the data in the example below:

```
libname myoralib oracle user=testuser password=testpass
        direct_sql=nowhere;   /* forces SAS to process WHERE clauses */
data myoralib.table;
x=.;       /*create a missing value */
run;
```

Then, print the data set using a WHERE clause:

```
proc print data=myoralib.table;
   where x<0;
run;
```

This time, the log indicates that one observation was read from the data set, because SAS evaluates the missing value as satisfying the less-than-zero condition in the WHERE clause.



CHAPTER

4

Performance Considerations

Increasing Throughput of the SAS Server

When you invoke SAS as a server that responds to multiple clients, you can use the DBSRVTP= system option to improve the performance of the clients. The DBSRVTP= option tells the SAS server whether to put a hold (or block) on the originating client while making performance-critical calls to the database. By holding or blocking the originating client, the SAS/ACCESS server remains available for other clients; they do not have to wait for the originating client to complete its call to the database.

Limiting Retrieval

Column Selection

Limiting the number of rows that are returned to SAS from the DBMS is an extremely important performance consideration, because the less data that is requested by the SAS job, the faster the job runs.

Where possible, specify selection criteria in order to limit the number of rows that the DBMS returns to SAS. Use the SAS WHERE clause to retrieve a subset of the DBMS data.

Likewise, select only the DBMS columns that your program needs. Selecting unnecessary columns slows your job.

The KEEP= and DROP= Options

Just as with a SAS data set you can use the DROP= and KEEP= data set options to prevent retrieving unneeded columns from your DBMS table.

In this example, the KEEP= data set option causes the SAS/ACCESS engine to select only the SALARY and DEPT columns when it reads the MYDBLIB.EMPLOYEES table.

```
libname mydblib db2 user=testid password=testpass database=testdb;

proc sql;
select *
   from mydblib.employees(keep=salary dept)
   where dept='ACC024';
quit;
```

The generated SQL that is processed by the DBMS will be similar to the following:

```
SELECT "SALARY", "DEPT" FROM EMPLOYEES
   WHERE(DEPT="ACC024")
```

Without the KEEP option, the SQL processed by the DBMS would be similar to the following:

```
SELECT * FROM EMPLOYEES    WHERE(DEPT="ACC024")
```

This would result in all of the columns from the EMPLOYEES table being read in to SAS.

The DROP= data set option is a parallel option that specifies columns to omit from the output table. Keep in mind that the DROP= and KEEP= data set options are not interchangeable with the DROP and KEEP statements. Use of the DROP and KEEP statements when selecting data from a DBMS can result in retrieval of all column into SAS, which can seriously impact performance.

For example, the following would result in all of the columns from the EMPLOYEES table being retrieved into SAS. The KEEP statement would be applied when creating the output data set.

```
libname mydblib db2 user=testid password=testpass database=testdb;

data temp;
   set mydblib.employees;
   keep salary;
run;
```

The following is an example of how to use the KEEP data set option to retrieve only the SALARY column:

```
data temp;
   set mydblib.employees(keep=salary);
run;
```

Repeatedly Accessing Data

CAUTION:
If you need to access the most current DBMS data, then access it directly from the database every time. Do not follow the extraction suggestions that are presented in this section. △

It is sometimes more efficient to extract (copy) DBMS data to a SAS data file than to repeatedly read the data by using a SAS view. SAS data files are organized to provide optimal performance with PROC and DATA steps. Programs that use SAS data files are often more efficient than SAS programs that read DBMS data directly.

Consider extracting data when you are working with a large DBMS table and you plan to use the same DBMS data in several procedures or DATA steps during the same SAS session.

Note: You can extract DBMS data to a SAS data file by using the OUT= option, a DATA step, or ACCESS procedures. △

Sorting DBMS Data

Sorting DBMS data can be resource intensive, whether you use the SORT procedure, a BY statement, or an ORDER BY clause on a DBMS data source or in the SQL procedure's SELECT statement. Sort data only when it is needed for your program. The following list contains guidelines for sorting data:

- If you specify a BY statement in a DATA or PROC step that references a DBMS data source, it is recommended for performance reasons that you associate the BY variable (in a DATA or PROC step) with an indexed DBMS column. If you reference DBMS data in a SAS program and the program includes a BY statement for a variable that corresponds to a column in the DBMS table, the SAS/ACCESS LIBNAME engine automatically generates an ORDER BY clause for that variable. The ORDER BY clause causes the DBMS to sort the data before the DATA or PROC step uses the data in a SAS program. If the DBMS table is very large, this sorting can adversely affect your performance. Use a BY variable that is based on an indexed DBMS column in order to reduce this negative impact.
- The outermost BY or ORDER BY clause overrides any embedded BY or ORDER BY clauses, including those specified by the DBCONDITION= option, those specified in a WHERE clause, and those in the selection criteria in a view descriptor. In the following example, the EXEC_EMPLOYEES data set includes a BY statement that sorts the data by the variable SENIORITY. However, when that data set is used in the following PROC SQL query, the data is ordered by the SALARY column and not by SENIORITY.

```
libname mydblib oracle user=testuser password=testpass;
data exec_employees;
   set mydblib.staff (keep=lname fname idnum);
   by seniority;
   where salary >= 150000;
run;

proc sql;
select * from exec_employees
   order by salary;
```

- Do not use PROC SORT to sort data from SAS back into the DBMS, because this impedes performance and has no effect on the order of the data.
- The database does not guarantee sort stability when you use PROC SORT. Sort stability means that the ordering of the observations in the BY statement is exactly the same every time the sort is run against static data. If you absolutely require sort stability, you must place your database data into a SAS data set, and then use PROC SORT.
- When you use PROC SORT, be aware that the sort rules for SAS and for your DBMS might be different. Use the Base SAS system option SORTPGM to specify which rules (host, SAS, or DBMS) are applied:

  SORTPGM=BEST
  sorts data according to the DBMS sort rules, then the host sort rules, and then the SAS sort rules. (Sorting uses the first available and pertinent sorting algorithm in this list.) This is the default.

  SORTPGM=HOST
  sorts data according to host rules and then SAS rules. (Sorting uses the first available and pertinent sorting algorithm in this list.)

  SORTPGM=SAS
  sorts data by SAS rules.

Temporary Table Support for SAS/ACCESS

Overview

DBMS temporary table support in SAS consists of the ability to retain DBMS temporary tables from one SAS step to the next. This ability is a result of establishing a SAS connection to the DBMS that persists across multiple SAS procedures and DATA steps.

Temporary table support for SAS 9.1 is available on the following DBMSs:

DB2 z/OS

DB2 UNIX/PC

Informix

ODBC

OLE DB

Oracle

Sybase

Teradata

The value of DBMS temporary table support in SAS is increased performance potential. By pushing processing to the DBMS in certain situations, an overall performance gain can be achieved. The following processes outline, in general, how to use DBMS temporary tables in SAS 9.1.

General Temporary Table Use

To use temporary tables on the DBMS, complete the following steps:

1. Establish a global connection to the DBMS that will persist across SAS procedure and DATA step boundaries.
2. Create a DBMS temporary table and load it with data.
3. Use the DBMS temporary table with SAS.

Note: Closing the global connection causes the DBMS temporary table to close as well. △

Pushing Heterogeneous Joins

To push heterogeneous joins to the DBMS, complete the following steps:

1. Establish a global connection to the DBMS that will persist across SAS procedure and DATA step boundaries.
2. Create a DBMS temporary table and load it with data.
3. Perform a join on the DBMS using the DBMS temporary and DBMS permanent tables.
4. Process the result of the join with SAS.

Pushing Updates

To push updates (process transactions) to the DBMS, complete the following steps:

1. Establish a global connection to the DBMS that will persist across SAS procedure and DATA step boundaries.
2. Create a DBMS temporary table and load it with data.
3. Issue SQL that uses values in the temporary table to process against the production table.
4. Process the updated DBMS tables with SAS.

Note: These processing scenarios are purposefully generic; however they apply to each DBMS that supports temporary tables. Refer to the SAS/ACCESS section for your database for complete details. △

CHAPTER

5

Optimizing Your SQL Usage

Overview of Optimizing Your SQL Usage

SAS/ACCESS takes advantage of DBMS capabilities by passing certain SQL operations to the DBMS whenever possible. This can reduce data movement, which can improve performance. The performance impact can be significant when you are accessing large DBMS tables and the SQL that is passed to the DBMS subsets the table to reduce the amount of rows. SAS/ACCESS sends operations to the DBMS for processing in the following situations:

- When operations use the Pass-Through Facility. When you use the Pass-Through Facility, you submit DBMS-specific SQL statements that are sent directly to the DBMS for execution. For example, when you submit Transact-SQL statements to be passed to a Sybase database.
- When SAS/ACCESS can translate the operations into the SQL of the DBMS. When you use the SAS/ACCESS LIBNAME statement and PROC SQL, you submit SAS statements that SAS/ACCESS can often translate into the SQL of the DBMS and then pass to the DBMS for processing.

By using the automatic translation abilities, you can often achieve the performance benefits of the Pass-Through Facility without needing to write DBMS-specific SQL code. The following sections describe the SAS SQL operations that SAS/ACCESS can pass to the DBMS for processing. See “Optimizing the Passing of WHERE Clauses to the DBMS” on page 42 for information about passing WHERE clauses to the DBMS.

Note: There are certain conditions that prevent operations from being passed to the DBMS. For example, when you use an INTO clause or any data set option, operations are processed in SAS instead of being passed to the DBMS. Re-merges, union joins, and truncated comparisons also prevent operations from being passed to the DBMS.

Additionally, it is important to note that when you join tables across multiple tables, implicit passthrough utilizes the first connection. Consequently, LIBNAME options from subsequent connections are ignored.

You can use the SASTRACE= system option to determine whether an operation is processed by SAS or is passed to the DBMS for processing. △

To prevent operations from being passed to the DBMS, use the LIBNAME option DIRECT_SQL=.

Passing Functions to the DBMS Using PROC SQL

When you use the SAS/ACCESS LIBNAME statement, the SAS SQL aggregate functions MIN, MAX, AVG, MEAN, FREQ, N, SUM, and COUNT are passed to the DBMS because they are SQL ANSI-defined aggregate functions.

For example, the following query of the Oracle table EMP is passed to the DBMS for processing:

```
libname myoralib oracle user=testuser password=testpass;
proc sql;
   select count(*) from myoralib.emp;
quit;
```

This code causes Oracle to process the following query:

```
select COUNT(*) from EMP
```

SAS/ACCESS can also translate other SAS functions into DBMS-specific functions so they can be passed to the DBMS.

In the following example, the SAS function UPCASE is translated into the Oracle function UPPER:

```
libname myoralib oracle user=testuser password=testpass;
proc sql;
   select customer from myoralib.customers
   where upcase(country)="USA";
quit;
```

The translated query that is processed in Oracle is

```
select customer from customers where upper(country)='USA'
```

The functions that are passed are different for each DBMS. See the documentation for your SAS/ACCESS interface to determine which functions it translates.

Passing Joins to the DBMS

When you perform a join across tables in a single DBMS, SAS/ACCESS can often pass the join to the DBMS for processing. Before implementing a join, PROC SQL checks to see whether the DBMS can do the join. A comparison is made using the SAS/ACCESS LIBNAME statement for the tables. Certain criteria must be met for the join to proceed.

If it can, PROC SQL passes the join to the DBMS, which performs the join and returns only the results to SAS. If the DBMS cannot do the join, PROC SQL processes the join.

The following types of joins are eligible for passing to the DBMS:

- for all DBMSs, inner joins between two or more tables.
- for DBMSs that support ANSI outer join syntax, outer joins between two or more DBMS tables.
- for ODBC and Microsoft SQL Server, outer joins between two or more tables. However, the outer joins must not be mixed with inner joins in a query.
- For DBMSs that support nonstandard outer join syntax (ORACLE, Sybase and INFORMIX), outer joins between two or more tables, with the following restrictions:

  Full outer joins are not supported.

  Only a comparison operator is allowed in an ON clause. For Sybase, the only valid comparison operator is '='.

  For ORACLE and Sybase, both operands in an ON clause must reference a column name. If an operand is a literal it may not be passed to the DBMS. Since these DBMSs do not support this, all ON clauses are transformed into WHERE clauses before attempting to pass the join to the DBMS. This can result in queries not being passed to the DBMS if they include additional WHERE clauses or contain complex join conditions.

  For INFORMIX, the outer joins cannot consist of more than two tables and cannot contain a WHERE clause.

  For Sybase: If there are multiple joins, or joins with additional WHERE clauses, it may be advisable to have the join processed internally by PROC SQL rather than passing it to the DBMS. This is because Sybase evaluates multi-joins with WHERE clauses differently than SAS. To allow PROC SQL to process the join internally use the SAS/Access DIRECT_SQL= option.

  Note: If PROC SQL cannot successfully pass down a complete query to the DBMS, it may re-attempt passing down a subquery. The SQL that is passed to the DBMS can be analyzed by turning on SAS tracing options. The SAS trace information will display the exact queries being passed to the DBMS for processing. △

In the following example, two large DBMS tables named TABLE1 and TABLE2 have a column named DeptNo, and you want to retrieve the rows from an inner join of these tables where the DeptNo value in TABLE1 is equal to the DeptNo value in TABLE2. The join between two tables in the DBLIB library (which references an Oracle database) is detected by PROC SQL and passed by SAS/ACCESS directly to the DBMS. The DBMS processes the inner join between the two tables and returns only the resulting rows to SAS.

```
libname dblib oracle user=testuser password=testpass;
proc sql;
   select tab1.deptno, tab1.dname from
      dblib.table1 tab1,
      dblib.table2 tab2
      where tab1.deptno = tab2.deptno;
quit;
```

The query is passed to the DBMS, generating the following Oracle code:

```
select table1."deptno", table1."dname" from TABLE1, TABLE2
   where TABLE1."deptno" = TABLE2."deptno"
```

In the following example, an outer join between two Oracle tables, TABLE1 and TABLE2, is passed to the DBMS for processing.

```
libname myoralib oracle user=testuser password=testpass;
proc sql;
   select * from myoralib.table1 right join myoralib.table2
      on table1.x = table2.x
      where table2.x > 1;
quit;
```

The query is passed to the DBMS, generating the following Oracle code:

```
select table1."X", table2."X" from TABLE1, TABLE2
   where TABLE1."X" (+)= TABLE2."X"
   and (TABLE2."X" > 1)
```

Passing the DELETE Statement to Empty a Table

When you use the SAS/ACCESS LIBNAME statement with the DIRECT_EXE option set to DELETE, the SAS SQL DELETE statement gets passed to the DBMS for execution as long as it does not contain a WHERE clause. The DBMS deletes all the rows, but does not delete the table itself.

The following example illustrates how a DELETE statement gets passed to Oracle for the table EMP to be emptied:

```
libname myoralib oracle user=testuser password=testpass direct_exe=delete;
proc sql;
   delete from myoralib.emp;
quit;
```

This code causes Oracle to execute the following:

```
delete from emp
```

When Passing Joins to the DBMS Will Fail

SAS/ACCESS will, by default, attempt to pass certain types of SQL statements directly to the DBMS for processing. Most notable are SQL join statements that otherwise would be processed as individual queries to each data source that belonged to the join. In that instance, the join would then be performed internally by PROC SQL. Passing the join to the DBMS for direct processing can result in significant performance gains.

However, there are several reasons why a join statement under PROC SQL might not be passed to the DBMS for processing. In general, the success of the join depends upon the nature of the SQL that was coded and the DBMS's acceptance of the generated syntax. It is also greatly influenced by the use of option settings. The following are the primary reasons why join statements might fail to be passed:

- The generated SQL syntax is not accepted by the DBMS.

  PROC SQL attempts to pass the SQL join query directly to the DBMS for processing. The DBMS can reject the syntax for any number of reasons. In this event, PROC SQL attempts to open both tables individually and perform the join internally.

- The SQL query involves multiple librefs that do not share connection characteristics.

If the librefs are specified using different servers, user IDs, or any other connection options, PROC SQL will not attempt to pass the statement to the DBMS for direct processing.

- The use of data set options in the query.

  The specification of any data set option on a table that is referenced in the SQL query will prohibit the statement from successfully passing to the DBMS for direct processing.

- The use of certain LIBNAME options.

  The specification of LIBNAME options that request member level controls, such as table locks ("READ_LOCK_TYPE= LIBNAME Option" on page 136 or "UPDATE_LOCK_TYPE= LIBNAME Option" on page 148), will prohibit the statement from successfully passing to the DBMS for direct processing.

- The "DIRECT_SQL= LIBNAME Option" on page 113 option setting.

  The DIRECT_SQL= option default setting is YES. PROC SQL attempts to pass SQL joins directly to the DBMS for processing. Other setting for the DIRECT_SQL= option influence the nature of the SQL statements that PROC SQL will attempt to pass down to the DBMS, or even if it will attempt to pass anything at all.

  DIRECT_SQL=YES
  : PROC SQL automatically attempts to pass the SQL join query to the DBMS. This is the default setting for this option. The join attempt could fail due to a DBMS return code. If this happens, PROC SQL attempts to open both tables individually and perform the join internally.

  DIRECT_SQL=NO
  : PROC SQL does not attempt to pass SQL join queries to the DBMS. Other SQL statements can be passed, however. If the "MULTI_DATASRC_OPT= LIBNAME Option" on page 125 is in effect, the generated SQL can also be passed.

  DIRECT_SQL=NONE
  : PROC SQL does not attempt to pass any SQL directly to the DBMS for processing.

  DIRECT_SQL=NOWHERE
  : PROC SQL attempts to pass SQL to the DBMS including SQL joins. However, it does not pass any WHERE clauses associated with the SQL statement. This causes any join that is attempted with direct processing to fail.

  DIRECT_SQL=NOFUNCTIONS
  : PROC SQL does not pass any statements in which any function is present to the DBMS. Normally PROC SQL attempts to pass down any functions coded in the SQL to the DBMS, provided the DBMS supports the given function.

  DIRECT_SQL=NOGENSQL
  : PROC SQL does not attempt to pass SQL join queries to the DBMS. Other SQL statements can be passed down, however. If the MULTI_DATASRC_OPT= option is in effect, the generated SQL can be passed.

  DIRECT_SQL=NOMULTOUTJOINS
  : PROC SQL does not attempt to pass any multiple outer joins to the DBMS for direct processing. Other SQL statements can be passed, however, including portions of a multiple outer join.

- The use of SAS functions on the SELECT clause can prevent joins from being passed.

Passing DISTINCT and UNION Processing to the DBMS

When you use the SAS/ACCESS LIBNAME statement to access DBMS data, the DISTINCT and UNION operators are processed in the DBMS rather than in SAS. For example, when PROC SQL detects a DISTINCT operator, it passes the operator to the DBMS to check for duplicate rows. The DBMS then returns only the unique rows to SAS.

In the following example, the Oracle table CUSTBASE is queried for unique values in the STATE column.

```
libname myoralib oracle user=testuser password=testpass;
proc sql;
   select distinct state from myoralib.custbase;
quit;
```

The DISTINCT operator is passed to Oracle, generating the following Oracle code:

```
select distinct custbase."STATE" from CUSTBASE
```

Oracle then passes the results from this query back to SAS.

Optimizing the Passing of WHERE Clauses to the DBMS

Use the following general guidelines for writing efficient WHERE clauses:

- Avoid the NOT operator if you can use an equivalent form.
  Inefficient: **where zipcode not>8000**
  Efficient: **where zipcode<=8000**
- Avoid the >= and <= operators if you can use the BETWEEN predicate.
  Inefficient: **where ZIPCODE>=70000 and ZIPCODE<=80000**
  Efficient: **where ZIPCODE between 70000 and 80000**
- Avoid LIKE predicates that begin with % or _ .
  Inefficient: **where COUNTRY like '%INA'**
  Efficient: **where COUNTRY like 'A%INA'**
- Avoid arithmetic expressions in a predicate.
  Inefficient: **where SALARY>12*4000.00**
  Efficient: **where SALARY>48000.00**
- Use DBKEY=, DBINDEX=, and MULTI_DATASRC_OPT= when appropriate. See "Using the DBINDEX=, DBKEY=, and MULTI_DATASRC_OPT= Options" on page 43 for details about these options.

Whenever possible, SAS/ACCESS passes WHERE clauses to the DBMS, because the DBMS processes them more efficiently than SAS does. SAS translates the WHERE clauses into generated SQL code. The performance impact can be particularly significant when you are accessing large DBMS tables. The following section describes how and when functions are passed to the DBMS. For information about passing processing to the DBMS when you are using PROC SQL, see "Overview of Optimizing Your SQL Usage" on page 37.

Note: If you have NULL values in a DBMS column that is used in a WHERE clause, be aware that your results might differ depending on whether the WHERE clause is processed in SAS or is passed to the DBMS for processing. This is because DBMSs tend to remove NULL values from consideration in a WHERE clause, while SAS does not. △

To prevent WHERE clauses from being passed to the DBMS, use the LIBNAME option DIRECT_SQL= NOWHERE.

Passing Functions to the DBMS Using WHERE Clauses

When you use the SAS/ACCESS LIBNAME statement, SAS/ACCESS translates several SAS functions in WHERE clauses into DBMS-specific functions so they can be passed to the DBMS.

In the following SAS code, SAS can translate the FLOOR function into a DBMS function and pass the WHERE clause to the DBMS.

```
libname myoralib oracle user=testuser password=testpass;
proc print data=myoralib.personnel;
   where floor(hourlywage)+floor(tips)<10;
run;
```

The generated SQL that is processed by the DBMS would be similar to

```
SELECT "HOURLYWAGE", "TIPS" FROM PERSONNEL
   WHERE ((FLOOR("HOURLYWAGE") + FLOOR("TIPS")) < 10)
```

If the WHERE clause contains a function that SAS cannot translate into a DBMS function, SAS retrieves all the rows from the DBMS and then applies the WHERE clause.

The functions that are passed are different for each DBMS. See the documentation for your SAS/ACCESS interface to determine which functions it translates.

Using the DBINDEX=, DBKEY=, and MULTI_DATASRC_OPT= Options

When you code a join operation in SAS, and the join cannot be passed directly to a DBMS for processing, the join will be performed by SAS. Normally this processing will involve individual queries to each data source that belonged to the join, and the join being performed internally by SAS. When you join a large DBMS table and a small SAS data set or DBMS table, using the DBKEY= , DBINDEX=, and MULTI_DATASRC_OPT= options might enhance performance. These options enable you to retrieve a subset of the DBMS data into SAS for the join.

When MULTI_DATASRC_OPT=IN_CLAUSE is specified for DBMS data sources in a PROC SQL join operation, the procedure will retrieve the unique values of the join column from the smaller table to construct an IN clause. This IN clause will be used when SAS is retrieving the data from the larger DBMS table. The join is performed in SAS. If a SAS data set is used, no matter how large, it is always in the IN_CLAUSE. For better performance, it is recommended that the SAS data set be smaller than the DBMS table, otherwise processing can be extremely slow.

MULTI_DATASRC_OPT= generates a SELECT COUNT to determine the size of data sets that are not SAS data sets. If you know the size of your data set, you can use DBMASTER to designate the larger table.

MULTI_DATASRC_OPT= might provide performance improvements over DBKEY=. If both options are specified, DBKEY= overrides MULTI_DATASRC_OPT=.

MULTI_DATASRC_OPT= is only used when SAS is processing a join with PROC SQL. It will not be used for SAS datastep processing. For certain joins operations, such as those involving additional subsetting applying to the query, PROC SQL might determine that it is more efficient to process the join internally. In these situations it will not use the MULTI_DATASRC_OPT= optimization even when specified. If PROC

SQL determines it can pass the join directly to the DBMS it will also not use this option even though it is specified.

In this example, the MULTI_DATASRC_OPT= option is used to improve the performance of a SQL join statement. MULTI_DATASRC_OPT= instructs PROC SQL to pass the WHERE clause to the SAS/ACCESS engine with an IN clause built from the SAS table. The engine then passes this optimized query to the DBMS server. The IN clause is built from the unique values of the SAS DeptNo variable. As a result, only rows that match the WHERE clause are retrieved from the DBMS. Without this option, PROC SQL retrieves all the rows from the Dept table and applies the WHERE clause during PROC SQL processing in SAS. Processing can be both CPU-intensive and I/O-intensive if the Oracle Dept table is large.

```
data keyvalues;
   deptno=30;
   output;
   deptno=10;
   output;
run;

libname dblib oracle user=testuser password=testpass
     path='myorapath' multi_datasrc_opt=in_clause;

proc sql;
   select bigtab.deptno, bigtab.loc
     from dblib.dept bigtab,
         keyvalues smallds
     where bigtab.deptno=smallds.deptno;
quit;
```

The SQL statement that is created by SAS/ACCESS and passed to the DBMS is similar to the following

```
SELECT "DEPTNO", "LOC" FROM DEPT WHERE (("DEPTNO" IN (10,30)))
```

Using DBKEY or DBINDEX decreases performance when the SAS data set is too large. These options cause each value in the transaction data set to generate a new result set (or open cursor) from the DBMS table. For example, if your SAS data set has 100 observations with unique key values, you request 100 result sets from the DBMS, which might be very expensive. You must determine whether use of these options is appropriate, or whether you can achieve better performance by reading the entire DBMS table (or by creating a subset of the table).

DBINDEX= and DBKEY= are mutually exclusive. If you specify them together, DBKEY= overrides DBINDEX=. Both of these options are ignored if you specify the SAS/ACCESS data set option DBCONDITION= or the SAS data set option WHERE=.

DBKEY= does not require that any database indexes be defined; nor does it check the DBMS system tables. This option instructs SAS to use the specified DBMS column name or names in the WHERE clause that is passed to the DBMS in the join.

The DBKEY= option can also be used in a SAS DATA step, with the KEY= option in the SET statement, to improve the performance of joins. You specify a value of KEY=DBKEY in this situation. The following DATA step creates a new data file by joining the data file KEYVALUES with the DBMS table MYTABLE. The variable DEPTNO is used with the DBKEY= option to cause a WHERE clause to be issued by SAS/ACCESS.

```
data sasuser.new;
   set sasuser.keyvalues;
   set dblib.mytable(dbkey=deptno) key=dbkey;
```

```
run;
```

Note: When you use DBKEY= with the DATA step MODIFY statement, there is no implied ordering of the data that is returned from the database. If the master DBMS table contains records with duplicate key values, using DBKEY= can alter the outcome of the DATA step. Because SAS regenerates result sets (open cursors) during transaction processing, the changes you make during processing have an impact on the results of subsequent queries. Therefore, before you use DBKEY= in this context, determine whether your master DBMS file has duplicate values for keys (remembering that the REPLACE, OUTPUT, and REMOVE statements can cause duplicate values to appear in the master table). △

The DBKEY= option does not require or check for the existence of indexes created on the DBMS table. Therefore, the DBMS system tables are not accessed when you use this option. The DBKEY= option is preferred over the DBINDEX= option for this reason. If you perform a join and use PROC SQL, you *must* ensure that the columns that are specified through the DBKEY= option match the columns that are specified in the SAS data set.

CAUTION:

Before you use the DBINDEX= option, take extreme care to evaluate some characteristics of the DBMS data. The number of rows in the table, the number of rows returned in the query, and the distribution of the index values in the table are among the factors to take into consideration. Some experimentation might be necessary to discover the optimum settings. △

You can use the DBINDEX= option instead of the DBKEY= option if you know that the DBMS table has one or more indexes that use the column(s) on which the join is being performed. Use DBINDEX=*index-name* if you know the name of the index, or use DBINDEX=YES if you do not know the name of the index. Use this option as a data set option, and *not* a LIBNAME option, because index lookup can potentially be an expensive operation.

DBINDEX= requires that the join table *must* have a database index that is defined on the columns involved in the join. If there is no index, then all processing of the join takes place in SAS, where all rows from each table are read into SAS and SAS performs the join.

Note: The data set options NULLCHAR= and NULLCHARVAL= determine how SAS missing character values are handled during DBINDEX= and DBKEY= processing. △

CHAPTER

6

Threaded Reads

Overview of Threaded Reads in SAS/ACCESS

In Version 8 and earlier, SAS opened a single connection to the DBMS to read a table. SAS statements requesting data were converted to an SQL statement and passed to the DBMS. The DBMS processed the SQL statement, produced a result set consisting of table rows and columns, and transferred the result set back to SAS on the single connection.

With a threaded read, the table read time can be reduced by retrieving the result set on multiple connections between SAS and the DBMS. SAS is able to create multiple threads, and a read connection is established between the DBMS and each SAS thread. The result set is partitioned across the connections, and rows are passed to SAS simultaneously (in parallel) across the connections, improving performance.

Underlying Technology of Threaded Reads

To perform a threaded read, SAS first creates threads, which are standard operating system tasks controlled by SAS, within the SAS session. Next, SAS establishes a DBMS connection on each thread. SAS then causes the DBMS to partition the result set and reads one partition per thread. To cause the partitioning, SAS appends a WHERE clause to the SQL so that a single SQL statement becomes multiple SQL statements, one for each thread.

For example:

```
proc reg SIMPLE
data=dblib.salesdata (keep=salesnumber maxsales);


var _ALL_;
run;
```

Previous versions of SAS opened a single connection and issued:

```
SELECT salesnumber,maxsales FROM SALESDATA;
```

Assuming SalesData has an integer column EmployeeNum, SAS 9.1, *might* open two connections, issuing the following statements:

```
SELECT salesnumber,maxsales FROM salesdata WHERE (EMPLOYEENUM mod 2)=0;
```

and

```
SELECT salesnumber,maxsales FROM SALESDATA WHERE (EMPLOYEENUM mod 2)=1;
```

For more information about MOD, see "Autopartitioning Techniques in SAS/ ACCESS" on page 53

Note: Might is an important word here. Most, but not all, SAS/ACCESS interfaces support threaded reads in SAS 9.1. The partitioning WHERE clauses that SAS generates vary. SAS cannot always generate partitioning WHERE clauses, in which case the SAS user can supply them. There also might be other means used to partition the data, instead of the WHERE clauses. △

SAS/ACCESS Interfaces and Threaded Reads

The SAS/ACCESS interfaces that support threaded reads for SAS 9.1 are listed here. More interfaces are expected to support threaded reads in future releases.

- Oracle*
- DB2 under z/OS
- DB2 under UNIX and PC hosts
- ODBC
- Informix
- Sybase
- Teradata**

*Threaded reads for the Oracle engine are not supported under MVS (z/OS).

**Teradata on OS/390, z/OS, and Unix supports only FastExport threaded reads; see the Teradata specific documentation for details.

Threaded reads work across all UNIX and Windows platforms where you run SAS. There are special considerations for z/OS; see the Oracle-specific and Teradata-specific sections about threaded reads for details.

Scope of Threaded Reads

SAS steps called threaded applications are automatically eligible for a threaded read. Threaded applications are bottom-to-top fully threaded SAS procedures that perform

data reads, numerical algorithms, and data analysis in threads. Only some SAS procedures are threaded applications. Here is a basic example of PROC REG, a SAS threaded application:

```
libname lib oracle user=scott password=tiger;
proc reg simple
data=lib.salesdata (keep=salesnumber maxsales);
var _all_;
run;
```

For DBMSs, many more SAS steps can become eligible for a threaded read, specifically, steps with a read-only table. A libref has the form Lib.DbTable, where Lib is a SAS libref that "points" to DBMS data, and DbTable is a DBMS table. Here are sample read-only tables for which threaded reads can be turned on:

```
libname lib oracle user=scott password=tiger;
proc print data=lib.dbtable;
run;

data local;
set lib.families;
where gender="F";
run;
```

An eligible SAS step can require user assistance in order to actually perform threaded reads. If SAS is unable to automatically generate a partitioning WHERE clause or to otherwise perform threaded reads, then the user can code an option that supplies partitioning. To determine if SAS can automatically generate a partitioning WHERE clause, use “SASTRACE= System Option” on page 264 and “SASTRACELOC= System Option” on page 274.

Threaded reads can be turned off altogether. This eliminates additional DBMS activity associated with SAS threaded reads, such as additional DBMS connections and multiple SQL statements.

Threaded reads are not supported for the Pass-Through Facility, in which you code your own DBMS-specific SQL that is passed directly to the DBMS for processing.

Options That Affect Threaded Reads

SAS/ACCESS provides two options precisely for threaded reads from DBMSs: “DBSLICE= Data Set Option” on page 211 and “DBSLICEPARM= Data Set Option” on page 212.

DBSLICE= is a data set option, applicable only to a table reference. It permits you to code your own WHERE clauses to partition table data across threads, and is useful when you are familiar with your table data. For instance, if your DBMS table has a CHAR(1) column Gender, and your clients are approximately half female, Gender equally partitions the table into two parts. Therefore, an example DBSLICE= might be:

```
proc print data=lib.dbtable (dbslice=("gender='f'" "gender='m'"));
where dbcol>1000;
run;
```

SAS creates two threads and about half of the data is delivered in parallel on each connection.

When applying DBSLICEPARM=ALL instead of DBSLICE=, SAS attempts to "autopartition" the table for you.With the default DBSLICEPARM=THREADED_APPS setting, threaded reads are automatically attempted only for SAS threaded applications

(SAS procedures that thread I/O and numeric operations). DBSLICEPARM=ALL extends threaded reads to more SAS procedures, specifically steps that only read tables. Or, DBSLICEPARM=NONE turns it off entirely. It can be specified as a data set option, LIBNAME option, or as a global SAS option.

The first argument to DBSLICEPARM= is required and extends or restricts threaded reads. The second, optional argument is not commonly used and limits the number of DBMS connections. The following examples demonstrate the different uses of DBSLICEPARM=:

- UNIX or Windows SAS invocation option that turns on threaded reads for all read-only libref.

```
--dbsliceparm ALL
```

- Global SAS option that turns off threaded reads.

```
option dbsliceparm=NONE;
```

- LIBNAME option that restricts threaded reads to just SAS threaded applications.

```
libname lib oracle user=scott password=tiger dbsliceparm=THREADED_APPS;
```

- Table option that turns on threaded reads (with a maximum of three connections, in the example below)

```
proc print data=lib.dbtable(dbsliceparm=(ALL,3));
where dbcol>1000;
run;
```

DBSLICE= and DBSLICEPARM= apply only to DBMS table reads. THREADS= and CPUCOUNT= are additional SAS options that apply to threaded applications. For more information about these options, see the *SAS Language Reference: Dictionary*.

Generating Trace Information for Threaded Reads

A threaded read is a complex feature. A SAS step can be eligible for a threaded read, but not have it applied. Performance effect is not always easy to predict. Use the SASTRACE option to see if threaded reads occurred and to help assess performance. The following examples demonstrate usage scenarios with SAS/ACCESS to Oracle. Keep in mind that trace output is in English only and changes from release to release.

```
/*Turn on SAS tracing */
options sastrace=',,t,' sastraceloc=saslog nostsuffix;


/* Run a SAS job */

data work.locemp;
set trlib.MYEMPS(DBBSLICEPARM=(ALL,3));
where STATE in ('GA', 'SC', 'NC') and ISTENURE=0;
run;
```

The above job produces the following trace messages:

```
406  data work.locemp;
407  set trlib.MYEMPS(DBSLICEPARM=(ALL, 3));
408  where STATE in ('GA', 'SC', 'NC') and ISTENURE=0;
```

```
409  run;

ORACLE:  DBSLICEPARM option set and 3 threads were requested
ORACLE:  No application input on number of threads.

ORACLE:  Thread 2 contains 47619 obs.
ORACLE:  Thread 3 contains 47619 obs.
ORACLE:  Thread 1 contains 47619 obs.
ORACLE:  Threaded read enabled. Number of threads created: 3
```

If you want to see the SQL that is executed during the threaded read, you can set tracing to sastrace=',,t,d' and run the job again. This time the output will contain the threading information as well as all of the SQL statements processed by Oracle:

```
ORACLE_9: Prepared:
SELECT * FROM MYEMPS  418  data work.locemp;

419  set trlib.MYEMPS(DBSLICEPARM=(ALL, 3));
420  where STATE in ('GA', 'SC', 'NC') and ISTENURE=0;
421  run;

ORACLE:  DBSLICEPARM option set and 3 threads were requested
ORACLE:  No application input on number of threads.

ORACLE_10: Executed:
SELECT  "HIREDATE", "SALARY", "GENDER", "ISTENURE", "STATE", "EMPNUM", "NUMCLASSES"
     FROM MYEMPS WHERE  ( ( ("STATE" IN  ( 'GA' , 'NC' , 'SC' ) ) ) AND
     ("ISTENURE" = 0 ) ) AND ABS(MOD("EMPNUM",3))=0

ORACLE_11: Executed:
SELECT  "HIREDATE", "SALARY", "GENDER", "ISTENURE", "STATE", "EMPNUM", "NUMCLASSES"
     FROM MYEMPS WHERE  ( ( ("STATE" IN  ( 'GA' , 'NC' , 'SC' ) ) ) AND
     ("ISTENURE" = 0 ) ) AND ABS(MOD("EMPNUM",3))=1

ORACLE_12: Executed:
SELECT  "HIREDATE", "SALARY", "GENDER", "ISTENURE", "STATE", "EMPNUM", "NUMCLASSES"
     FROM MYEMPS WHERE  ( ( ("STATE" IN  ( 'GA' , 'NC' , 'SC' ) ) ) AND
     ("ISTENURE" = 0 ) ) AND (ABS(MOD("EMPNUM",3))=2 OR "EMPNUM" IS NULL)

ORACLE:  Thread 2 contains 47619 obs.
ORACLE:  Thread 1 contains 47619 obs.
ORACLE:  Thread 3 contains 47619 obs.
ORACLE:  Threaded read enabled. Number of threads created: 3
```

Notice that the Oracle engine used the EMPNUM column as a partitioning column.

If a threaded read cannot be done either because all of the candidates for autopartitioning are in the WHERE clause, or because the table does not contain a column that fits the criteria, you will see a warning in your log. For example, the data set below uses a WHERE clause that contains all of the possible autopartitioning columns:

```
data work.locemp;
set trlib.MYEMPS (DBLISCEPARM=ALL);
where EMPNUM<=30 and ISTENURE=0 and SALARY<=35000 and NUMCLASSES>2;
run;
```

You receive the following warnings:

```
ORACLE:  No application input on number of threads.
ORACLE:  WARNING: Unable to find a partition column for use w/ MOD()
ORACLE:  The engine cannot automatically generate the partitioning WHERE clauses.
ORACLE:  Using only one read connection.
ORACLE:  Threading is disabled due to an error. Application reverts to non-threading
         I/O's.
```

If the SAS job contains any options that are invalid when the engine tries to perform threading, you also receive a warning.

```
libname trlib oracle user=orauser pw=orapw path=oraserver DBSLICEPARM=(ALL);

proc print data=trlib.MYEMPS (OBS=10);
where EMPNUM<=30;
run;
```

This produces the following message:

```
ORACLE:  Threading is disabled due to the ORDER BY clause or the FIRSTOBS/OBS option.
ORACLE:  Using only one read connection.
```

To produce timing information, add an 's' in the last slot of sastrace. For example:

```
options sastrace=',,t,s' sastraceloc=saslog nostsuffix;

data work.locemp;
set trlib.MYEMPS (DBSLICEPARM=ALL);
where EMPNUM<=10000;
run;
```

This produces the following:

```
ORACLE:  No application input on number of threads.
ORACLE:  Thread 1 contains 5000 obs.
ORACLE:  Thread 2 contains 5000 obs.

Thread 0 fetched 5000 rows
DBMS Threaded Read Total Time: 1234 mS
DBMS Threaded Read User CPU:   46 mS
DBMS Threaded Read System CPU: 0 mS

Thread 1 fetched 5000 rows
DBMS Threaded Read Total Time: 469 mS
DBMS Threaded Read User CPU:   15 mS
DBMS Threaded Read System CPU: 15 mS
ORACLE:  Threaded read enabled. Number of threads created: 2
NOTE: There were 10000 observations read from the data set TRLIB.MYEMPS.
     WHERE EMPNUM<=10000;

Summary Statistics for ORACLE are: Total SQL prepare seconds were:          0.001675
Total seconds used by the ORACLE ACCESS engine were      7.545805
```

For more information regarding tracing, please see the SASTRACE documentation.

Performance Impact of Threaded Reads

Threaded reads only increase performance when the DBMS result set is large. Performance is optimal when the partitions are similar in size. Threaded reads should reduce the elapsed time of your SAS step, but unusual cases can slow the SAS step. They generally increase the workload on your DBMS.

For instance, threaded reads for DB2 under z/OS involve a tradeoff, generally reducing job elapsed time but increasing DB2 workload and CPU utilization. See the auto partitioning documentation for DB2 under z/OS for details.

SAS automatically tries to autopartition table references for SAS in threaded applications. To determine whether autopartitioning is occurring and to assess its performance, complete the following tasks:

- Turn on SAS tracing to see if SAS is autopartitioning and to view the SQL associated with each thread.
- Know your DBMS algorithm for autopartitioning.
- Turn threaded reads on and off, and compare the elapsed times.

To optimally tune threaded reads, follow these guidelines:

- Use it only when pulling large result sets into SAS from the DBMS.
- Use DBSLICE= to partition if SAS autopartitioning does not occur.
- Override autopartitioning with DBSLICE= if you can manually provide substantially better partitioning. The best partitioning equally distributes the result set across the threads.
- Consult the DBMS-specific section of this documentation for information and tips concerning your specific DBMS.

Threaded reads are most effective on new, faster computer hardware running SAS, and with a powerful parallel edition of the DBMS. For example, if SAS runs on a fast uniprocessor or on a multiprocessor machine and your DBMS runs on a high-end SMP server, you will receive substantial performance gains. However you receive minimal gains or even performance degradation when running SAS on an old desktop model with a nonparallel DBMS edition running on old hardware.

Autopartitioning Techniques in SAS/ACCESS

SAS/ACCESS products share an autopartitioning scheme based on the MOD function. Some products support additional techniques. For example, if your Oracle tables are physically partitioned in the DBMS, the SAS/ACCESS interface to Oracle automatically partitions in accordance with Oracle physical partitions rather than using MOD. The SAS/ACCESS interface to Teradata uses FastExporting, if available, which enables the FastExport utility to direct the partitioning.

MOD is a mathematical function that produces the remainder of a division operation. Your DBMS table must contain a column to which SAS can apply the MOD function — a numeric column constrained to integral values. DBMS integer and small integer columns suit this purpose. Integral decimal (numeric) type columns can work as well. On each thread, SAS appends a WHERE clause to your SQL that uses the MOD function with the numeric column to create a subset of the result set. Combined, these subsets add up to exactly the result set for your original single SQL statement.

For example, assume your original SAS-produced SQL is **SELECT CHR1, CHR2 FROM DBTAB** and that table Dbtab contains integer column IntCol. SAS creates two threads and issues:

```
SELECT CHR1, CHR2 FROM DBTAB WHERE (MOD(INTCOL,2)=0)
```

and

```
SELECT CHR1, CHR2 FROM DBTAB WHERE (MOD(INTCOL,2)=1)
```

Rows with an even value for IntCol are retrieved by the first thread. Rows with an odd value for IntCol are retrieved by the second thread. Distribution of rows across the two threads is optimal if IntCol has a 50/50 distribution of even and odd values.

SAS modifies the SQL for columns containing negative integers, for nullable columns, and to combine SAS WHERE clauses with the partitioning WHERE clauses. SAS can also run more than two threads. You use the second parameter of the DBSLICEPARM= option to increase the number of threads.

The success of this technique depends on the distribution of the values in the chosen integral column. Without knowledge of the distribution, your SAS/ACCESS product attempts to pick the best possible column. For example, indexed columns are given preference for some DBMSs. However, column selection is more or less a guess, and the SAS guess might cause poor distribution of the result set across the threads. If no suitable numeric column is found, MOD cannot be used at all, and threaded reads will not occur if your SAS/ACCESS product has no other partitioning technique. For these reasons, you should explore autopartitioning particulars for your DBMS and judiciously utilize DBSLICE= to augment autopartitioning. For particulars on autopartitioning, see the documentation for your DBMS.

Data Ordering in SAS/ACCESS

The order in which table rows are delivered to SAS varies each time a step is rerun with threaded reads. Most DBMS editions, especially increasingly popular parallel editions, do not guarantee consistent ordering.

Two-Pass Processing for SAS Threaded Applications

Two-pass processing occurs when a SAS threaded application“Scope of Threaded Reads” on page 48 requests that data be made available for multiple pass reading (that is, more than one pass through the data set). In the context of DBMS engines, this requires that as the data is read from the database, temporary spool files are written containing the read data. There is one temporary spool file per thread, and each spool file will contain all the data read on that thread. If three threads are specified for threaded reads, then three temporary spool files are written.

As the application requests subsequent passes of the data, the data is read from the temporary spool files, not re-read from the database. The temporary spool files can be written on different disks, reducing any disk read contention, and enhancing performance. To accomplish this, the SAS option UTILLOC= is used to define different disk devices and directory paths when creating temporary spool files. There are several different ways to specify this option:

- In the SAS config file, add the line:

```
--utilloc("C:\path" "D:\path" "E:\path")
```

- Specify the UTILLOC= option on the SAS command line:

  on Windows:

  ```
  sas --utilloc(c:\path d:\path e:\path)
  ```

  on UNIX:

  ```
  sas --utilloc '(\path \path2 \path3)'
  ```

For more information about the UTILLOC= SAS option, see the *SAS Language Reference: Dictionary*.

When Threaded Reads Will Not Occur

Threading will not occur

- when a BY statement is used in a PROC or DATA step
- when the OBS or the FIRSTOBS option is in a PROC or DATA step
- when the KEY or the DBKEY option is used PROC or DATA step
- if no modable column is in the table For more information, see Autopartioning Techniques in SAS"Autopartitioning Techniques in SAS/ACCESS" on page 53.
- if all modable columns within a table are in WHERE clauses

  For more information, see Autopartioning Techniques in SAS"Autopartitioning Techniques in SAS/ACCESS" on page 53.
- if the NOTHREADS system option is set
- if DBSLICEPARM=NONE
- if no modable column is in the table

Summary of Threaded Reads

For large reads of table data, SAS 9.1 threaded reads can speed up SAS jobs. They are particularly useful when you understand the autopartitioning technique specific to your DBMS and use DBSLICE= to manually partition only when appropriate. Look for enhancements in future SAS releases.

CHAPTER 7

How SAS/ACCESS Works

Introduction to How SAS/ACCESS Works

Note: Not all features are supported by all SAS/ACCESS interfaces. See the documentation for your SAS/ACCESS interface to determine which features are supported in your environment. △

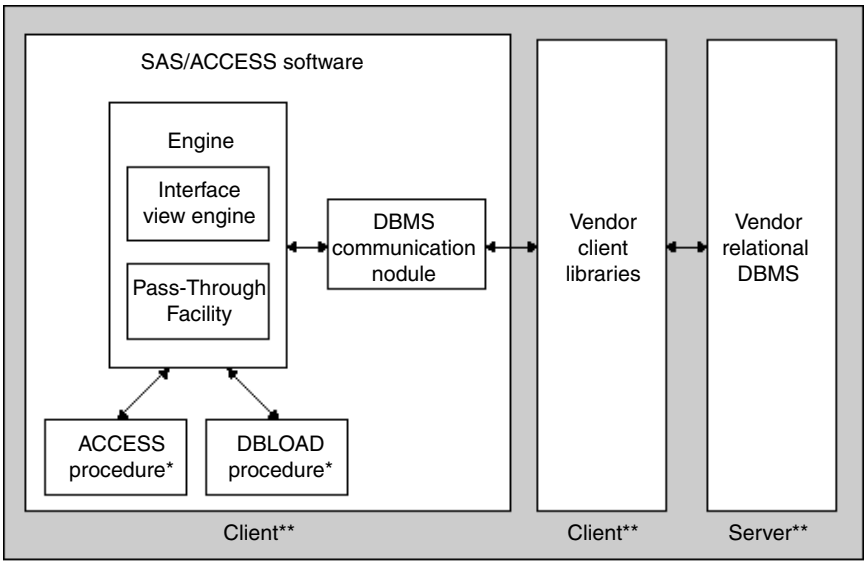
Installation Requirements

Before you use any SAS/ACCESS features, you must install Base SAS, the SAS/ACCESS interface for the DBMS that you are accessing, and any required DBMS client software. See the SAS installation instructions and DBMS client installation instructions for more information.

SAS/ACCESS Interfaces

Each SAS/ACCESS interface consists of one or more data access engines that translate read and write requests from SAS into appropriate calls for a specific DBMS. The following image depicts the relationship between a SAS/ACCESS interface and a relational DBMS.

Figure 7.1 How SAS Connects to the DBMS



* The ACCESS procedure and the DBLOAD procedure are not supported by all SAS/ACCESS interfaces.
** In some cases, both client and server software can reside on the same machine.

You can invoke a SAS/ACCESS relational DBMS interface by using either a LIBNAME statement or a PROC SQL statement. (You can also use the ACCESS and DBLOAD procedures with some of the SAS/ACCESS relational interfaces. However, these procedures are no longer the recommended way to access relational database data.)

How the SAS/ACCESS LIBNAME Statement Works

Accessing Data From a DBMS Object

SAS/ACCESS enables you to read, update, insert, and delete data from a DBMS object as if it were a SAS data set. The process is as follows:

1. You invoke a SAS/ACCESS interface by specifying a DBMS engine name and the appropriate connection options in a LIBNAME statement.
2. You enter SAS requests as you would when accessing a SAS data set.
3. SAS/ACCESS generates DBMS-specific SQL statements that are equivalent to the SAS requests that you enter.
4. SAS/ACCESS submits the generated SQL to the DBMS.

The SAS/ACCESS engine defines which operations are supported on a table and calls code that translates database operations such as open, get, put, or delete into DBMS-specific SQL syntax. SAS/ACCESS engines use an established set of routines with calls that are tailored to each DBMS.

Processing Queries, Joins, and Data Functions

To enhance performance, SAS/ACCESS can also transparently pass a growing number of queries, joins, and data functions to the DBMS for processing (instead of

retrieving the data from the DBMS and then doing the processing in SAS). For example, an important use of this feature is the handling of PROC SQL queries that access DBMS data. Here's how it works:

1. PROC SQL examines each query to determine whether it might be profitable to send all or part of the query to the DBMS for processing.
2. A special query textualizer in PROC SQL translates queries (or query fragments) into DBMS-specific SQL syntax.
3. The query textualizer submits the translated query to the SAS/ACCESS engine for approval.
4. If SAS/ACCESS approves the translation, it sends an approval message to PROC SQL and the query (or query fragment) gets processed by the DBMS, which returns the results to SAS. Any queries or query fragments that can not be passed to the DBMS are processed in SAS.

See the chapter on performance considerations for detailed information about tasks that SAS/ACCESS can pass to the DBMS.

How the Pass-Through Facility Works

When you read and update DBMS data with the Pass-Through Facility, SAS/ACCESS passes SQL statements directly to the DBMS for processing. Here are the steps:

1. Invoke PROC SQL and submit a PROC SQL CONNECT statement that includes a DBMS name and the appropriate connection options to establish a connection with a specified database.
2. Use a CONNECTION TO component in a PROC SQL SELECT statement to read data from a DBMS table or view. In the SELECT statement (that is, the PROC SQL query) that you write, use the SQL that is native to your DBMS. SAS/ACCESS passes the SQL statements directly to the DBMS for processing. If the SQL syntax that you enter is correct, the DBMS processes the statement and returns any results to SAS. If the DBMS does not recognize the syntax that you enter, it returns an error that appears in the SAS log. The SELECT statement can be stored as a PROC SQL view. For example:

```
proc sql;
connect to oracle (user=scott password=tiger);
create view budget2000 as select amount_b,amount_s
   from connection to oracle
   (select Budgeted, Spent from annual_budget);
quit;
```

3. Use a PROC SQL EXECUTE statement to pass any dynamic, non-query SQL statements (such as INSERT, DELETE, and UPDATE) to the database. As with the CONNECTION TO component, all EXECUTE statements are passed to the DBMS exactly as you submit them. INSERT statements must contain literal values. For example:

```
proc sql;
   connect to oracle(user=scott password=tiger);
   execute (create view whotookorders as select ordernum, takenby,
     firstname, lastname,phone from orders, employees
     where orders.takenby=employees.empid) by oracle;
   execute (grant select on whotookorders to testuser) by oracle;
```

```
disconnect from oracle;
quit;
```

4 Terminate the connection with the DISCONNECT statement.

See Chapter 12, "The Pass-Through Facility for Relational Databases," on page 277 for more details.

How the ACCESS Procedure Works

Overview of the ACCESS Procedure

When you use the ACCESS procedure to create an access descriptor, the SAS/ACCESS interface view engine requests the DBMS to execute a SQL SELECT statement to the data dictionary tables in your DBMS dynamically (by using DBMS-specific call routines or interface software). The ACCESS procedure then issues the equivalent of a DESCRIBE statement to gather information about the columns in the specified table. The access descriptor's information about the table and its columns is then copied into the view descriptor when it is created. Therefore, it is not necessary for SAS to call the DBMS when it creates a view descriptor.

The process is as follows:

1 When you supply the connection information to PROC ACCESS, the SAS/ACCESS interface calls the DBMS to connect to the database.
2 SAS constructs a SELECT * FROM *table-name* statement and passes it to the DBMS to retrieve information about the table from the DBMS data dictionary. This SELECT statement is based on the information you supplied to PROC ACCESS. It enables SAS to determine whether the table exists and can be accessed.
3 The SAS/ACCESS interface calls the DBMS to get table description information, such as the column names, data types (including width, precision, and scale), and whether the columns accept null values.
4 SAS closes the connection with the DBMS.

Reading Data

When you use a view descriptor in a DATA step or procedure to read DBMS data, the SAS/ACCESS interface view engine requests the DBMS to execute a SQL SELECT statement. The interface view engine follows these steps:

1 Using the connection information that is contained in the created view descriptor, the SAS/ACCESS interface calls the DBMS to connect to the database.
2 SAS constructs a SELECT statement that is based on the information stored in the view descriptor (table name and selected columns and their characteristics) and passes this information to the DBMS.
3 SAS retrieves the data from the DBMS table and passes it back to the SAS procedures as if it were observations in a SAS data set.
4 SAS closes the connection with the DBMS.

For example, if you execute the following SAS program using a view descriptor, the previous steps are executed once for the PRINT procedure and then a second time for the GCHART procedure. (The data used for the two procedures is not necessarily the same because the table might have been updated by another user between procedure executions.)

```
proc print data=vlib.allemp;
run;

proc gchart data=vlib.allemp;
   vbar jobcode;
run;
```

Updating Data

You use a view descriptor, DATA step, or procedure to update DBMS data in much the same way as when reading data. Any of the following steps might also occur:

- Using the connection information that is contained in the specified access descriptor, the SAS/ACCESS interface calls the DBMS to connect to the database.
- When rows are added to a table, SAS constructs a SQL INSERT statement and passes it to the DBMS. When you reference a view descriptor, you can use the ADD command in FSEDIT and FSVIEW, the APPEND procedure, or an INSERT statement in PROC SQL to add data to a DBMS table. (You can also use the Pass-Through Facility's EXECUTE statement to add, delete, or modify DBMS data directly. Literal values must be used when inserting data with the Pass-Through Facility.)
- When rows are deleted from a DBMS table, SAS constructs a SQL DELETE statement and passes it to the DBMS. When you reference a view descriptor, you can use the DELETE command in FSEDIT and FSVIEW or a DELETE statement in PROC SQL to delete rows from a DBMS table.
- When data in the rows is modified, SAS constructs a SQL UPDATE statement and passes it to the DBMS. When you reference a view descriptor, you can use FSEDIT, the MODIFY command in FSVIEW, or an INSERT statement in PROC SQL to update data in a DBMS table. You can also reference a view descriptor in the DATA step's UPDATE, MODIFY, and REPLACE statements.
- SAS closes the connection with the DBMS.

How the DBLOAD Procedure Works

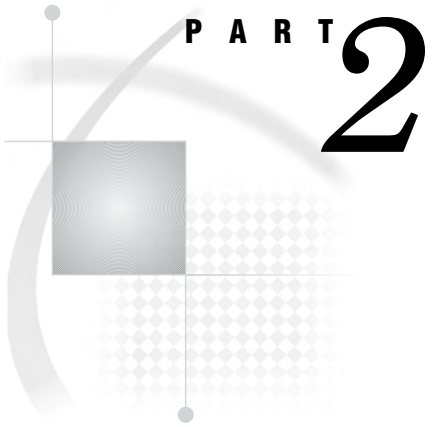
When you use the DBLOAD procedure to create a DBMS table, the procedure issues dynamic SQL statements to create the table and insert data from a SAS data file, DATA step view, PROC SQL view, or view descriptor into the table.

The SAS/ACCESS interface view engine completes the following steps:

1. When you supply the connection information to PROC DBLOAD, the SAS/ACCESS interface calls the DBMS to connect to the database.
2. SAS uses the information that is provided by the DBLOAD procedure to construct a SELECT * FROM *table-name* statement, and passes the information to the DBMS to determine if the table already exists. PROC DBLOAD continues only if a table with that name does not exist, unless you use the DBLOAD APPEND option.
3. SAS uses the information that is provided by the DBLOAD procedure to construct a SQL CREATE TABLE statement and passes it to the DBMS.
4. SAS constructs a SQL INSERT statement for the current observation and passes it to the DBMS. New INSERT statements are constructed and then executed repeatedly until all of the observations from the input SAS data set are passed to the DBMS. Some DBMSs have a bulkcopy capability that allows a group of observations to be inserted at once. See your DBMS documentation to determine if your DBMS has this capability.

5 Additional nonquery SQL statements specified in the DBLOAD procedure are executed as submitted by the user. The DBMS returns an error message if a statement does not execute successfully.

6 SAS closes the connection with the DBMS.



PART 2

General Reference

CHAPTER 8

SAS/ACCESS Features by Host

Introduction

This section lists by host environment the features that are supported in each SAS/ACCESS relational interface.

SAS/ACCESS Interface to DB2 under UNIX and PC Hosts: Supported Features

The following table lists the features that are supported in the SAS/ACCESS Interface to DB2 under UNIX and PC hosts. To find out which versions of your DBMS are supported, refer to your System Requirements manual.

Table 8.1 Features by Host Environment for DB2 under UNIX and PC Hosts

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
Windows NT & 2000	X	X		X	X
HP-UX	X	X		X	X
Solaris 2	X	X		X	X
Linux for Intel Architecture	X	X		X	X

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
Linux for Itanium-based Systems	X	X		X	X
AIX (RS/6000)	X	X		X	X

SAS/ACCESS Interface to DB2 under z/OS: Supported Features

The following table lists the features that are supported in the SAS/ACCESS Interface to DB2 under z/OS. To find out which versions of your DBMS are supported, refer to your System Requirements manual.

Table 8.2 Features by Host Environment for DB2 under z/OS

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
z/OS	X	X	X	X	X

SAS/ACCESS Interface to Informix: Supported Features

The following table lists the features that are supported in the SAS/ACCESS Interface to Informix. To find out which versions of your DBMS are supported, refer to your System Requirements manual.

Table 8.3 Features by Host Environment for Informix

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
HP-UX	X	X			
HP-UX for the Itanium Processor Family Architecture	X	X			
Solaris 2	X	X			
Compaq Tru64*	X	X			

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
Linux for Itanium-based Systems	X	X			
AIX (RS/6000)	X	X			

* formerly Digital UNIX (AXP/OSF1)

SAS/ACCESS Interface to Microsoft SQL Server: Supported Features

The following table lists the features that are supported in the SAS/ACCESS Interface to Microsoft SQL Server. To find out which versions of your DBMS are supported, refer to your System Requirements manual.

Table 8.4 Features by Host Environment for Microsoft SQL Server

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	Bulk load support
HP-UX	X	X		X	
Solaris 2	X	X		X	
Linux for Itanium-based Systems	X	X		X	
AIX (RS/6000)	X	X		X	

SAS/ACCESS Interface to MySQL: Supported Features

The following table lists the features that are supported in the SAS/ACCESS Interface to MySQL. To find out which versions of your DBMS are supported, refer to your System Requirements manual.

Table 8.5 Features by Host Environment for MySQL

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
Windows NT and 2000	X	X			
Solaris 2	X	X			

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
AIX (RS/6000)	X	X			
Linux for Intel Architecture	X	X			
Linux for Itanium-based Systems	X	X			
HP-UX	X	X			

SAS/ACCESS Interface to ODBC: Supported Features

The following table lists the features that are supported in the SAS/ACCESS Interface to ODBC. To find out which versions of your DBMS are supported, refer to your System Requirements manual.

Table 8.6 Features by Host Environment for ODBC

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
Windows NT & 2000	X	X		X	X**
HP-UX	X	X		X	
HP-UX for the Itanium Processor Family Architecture	X	X		X	
Solaris 2	X	X		X	
Linux for Intel Architecture	X	X		X	
Linux for Itanium-based Systems	X	X		X	

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
AIX (RS/6000)	X	X		X	
Compaq Tru64*	X	X		X	

* formerly Digital UNIX (AXP/OSF1)

** Bulk load support is available only with the SQL Server driver on Windows platforms.

SAS/ACCESS Interface to OLE DB: Supported Features

The following table lists the features that are supported in the SAS/ACCESS Interface to OLE DB. To find out which versions of your DBMS are supported, refer to your System Requirements manual.

Table 8.7 Features by Host Environment for OLE DB

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
Windows NT & 2000	X	X			X

SAS/ACCESS Interface to Oracle: Supported Features

The following table lists the features that are supported in the SAS/ACCESS Interface to Oracle. To find out which versions of your DBMS are supported, refer to your System Requirements manual.

Table 8.8 Features by Host Environment for Oracle

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
Windows NT & 2000	X	X	X	X	X
HP-UX	X	X	X	X	X
Solaris 2	X	X	X	X	X
Linux for Intel Architecture	X	X	X	X	X
Linux for Itanium-based Systems	X	X	X	X	X

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
Compaq Tru64*	X	X	X	X	X
z/OS	X	X	X	X	X
OpenVMS Alpha	X	X	X	X	X

* formerly Digital UNIX (AXP/OSF1)

SAS/ACCESS Interface to Sybase: Supported Features

The following table lists the features that are supported in the SAS/ACCESS Interface to Sybase. To find out which versions of your DBMS are supported, refer to your System Requirements manual.

Table 8.9 Features by Host Environment for Sybase

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
Windows NT & 2000	X	X	X	X	X
HP-UX	X	X	X	X	X
HP-UX for the Itanium Processor Family Architecture	X	X	X	X	X
Solaris 2	X	X	X	X	X
Linux for Intel Architecture	X	X	X	X	X
Linux for Itanium-based Systems	X	X	X	X	X
AIX (RS/6000)	X	X	X	X	X
Compaq Tru64*	X	X	X	X	X

* formerly Digital UNIX (AXP/OSF1)

SAS/ACCESS Interface to Teradata: Supported Features

The following table lists the features that are supported in the SAS/ACCESS Interface to Teradata. To find out which versions of your DBMS are supported, refer to your System Requirements manual.

Table 8.10 Features by Host Environment for Teradata

	SAS/ACCESS LIBNAME statement	Pass-Through Facility	ACCESS procedure	DBLOAD procedure	bulk load support
Windows NT & 2000	X	X			X
z/OS	X	X			X
HP-UX	X	X			X
HP-UX for the Itanium Processor Family Architecture	X	X			X
Solaris 2	X	X			X
AIX (RS/6000)	X	X			X
Linux for Intel Architecture	X	X			X

CHAPTER 9

The LIBNAME Statement for Relational Databases

Overview of the LIBNAME Statement for Relational Databases

Assigning Librefs

The SAS/ACCESS LIBNAME statement extends the SAS global LIBNAME statement to enable you to assign a libref to a relational DBMS. This feature lets you reference a DBMS object directly in a DATA step or SAS procedure, enabling you to read from and write to a DBMS object as though it were a SAS data set. You can associate a SAS libref with a relational DBMS database, schema, server, or group of tables and views.

Sorting Data

When you use the SAS/ACCESS LIBNAME statement to associate a libref with relational DBMS data, you might observe some behavior that differs from that of normal SAS librefs. Because these librefs refer to database objects, such as tables and views, they are stored in the format of your DBMS, which differs from the format of normal SAS data sets. This is helpful to remember when you access and work with DBMS data.

For example, you can sort the observations in a normal SAS data set and store the output to another data set. However, in a relational DBMS, sorting data often has no effect on how it is stored. Because you cannot depend on your data to be sorted in the DBMS, you must sort the data at the time of query. Furthermore, when you sort DBMS data, the results might vary depending on whether your DBMS places data with NULL values (which are translated in SAS to missing values) at the beginning or the end of the result set.

Using SAS Functions

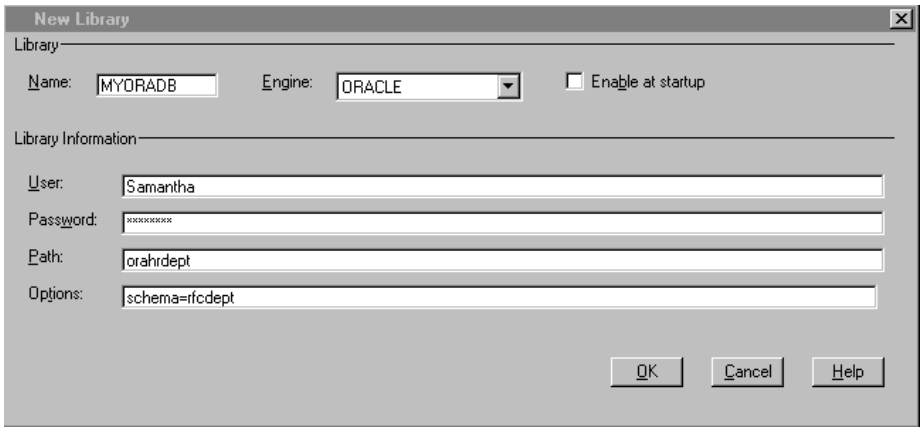
When you use librefs that refer to DBMS data with SAS functions, some functions might return a value that differs from what is returned when you use the functions with normal SAS data sets. For example, the PATHNAME function might return a blank value. For a normal SAS libref, a blank value means that the libref is not valid. However, for a libref associated with a DBMS object, a blank value means only that there is no pathname associated with the libref.

Usage of some functions might also vary. For example, the LIBNAME function can accept an optional *SAS-data-library* argument. When you use the LIBNAME function to assign or de-assign a libref that refers to DBMS data, you omit this argument. For full details about how to use SAS functions, see the *SAS Language Reference: Dictionary*.

Assigning a Libref Interactively

An easy way to associate a libref with a relational DBMS is to use the New Library window. One method to open this window is to issue the DMLIBASSIGN command from your SAS session's command box or command line. The window can also be opened by clicking the file cabinet icon in the SAS Explorer toolbar. In the display below, the user Samantha assigns a libref MYORADB to an Oracle database referred to by the SQL*Net alias ORAHRDEPT. The LIBNAME option, SCHEMA=, enables Samantha to access database objects that are owned by another user.

Display 9.1 New Library Window



The following list describes how to use the features of the New Library window:

- **Name**: enter the libref that you want to assign to a SAS library or a relational DBMS.
- **Engine**: click the down arrow to select a name from the pull-down listing.
- **Enable at startup**: click this if you want the specified libref to be assigned automatically when you open a SAS session.
- **Library Information**: these fields represent the SAS/ACCESS connection options and vary according to the SAS/ACCESS engine that you specify. Enter the appropriate information for your site's DBMS. The **Options** field enables you to enter SAS/ACCESS LIBNAME options. Use blanks to separate multiple options.
- OK: click this button to assign the libref, or click Cancel to exit the window without assigning a libref.

LIBNAME Statement Syntax for Relational Databases

Associates a SAS libref with a DBMS database, schema, server, or group of tables and views

Valid: Anywhere

Syntax

❶ **LIBNAME** *libref engine-name*
<*SAS/ACCESS-connection-options*>
<*SAS/ACCESS-LIBNAME-options*>;

❷ **LIBNAME** *libref* CLEAR|_ALL_ CLEAR;

❸ **LIBNAME** *libref* LIST|_ALL_ LIST;

Arguments

The SAS/ACCESS LIBNAME statement takes the following arguments:

libref
is any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views. Like the global SAS LIBNAME statement, the SAS/ACCESS LIBNAME statement creates shortcuts or nicknames for data storage locations. While a *SAS* libref is an alias for a virtual or physical directory, a *SAS/ACCESS* libref is an alias for the DBMS database, schema, or server where your tables and views are stored.

engine-name
is the SAS/ACCESS engine name for your DBMS, such as **`oracle`** or **`db2`**. The engine name is required. Because the SAS/ACCESS LIBNAME statement associates a libref with a SAS/ACCESS engine that supports connections to a particular DBMS, it requires a DBMS-specific engine name.

See the documentation for your SAS/ACCESS interface to find your engine's name.

SAS/ACCESS-connection-options
provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS; these arguments are different for each database. For example, to connect to an Oracle database, your connection options are USER=, PASSWORD=, and PATH=:

```
libname myoralib oracle user=testuser password=testpass path='voyager';
```

If the connection options contain characters that are not allowed in SAS names, enclose the values of the arguments in quotation marks. On some DBMSs, if you specify the appropriate system options or environment variables for your database, you can omit the connection options.

See the documentation for your SAS/ACCESS interface for detailed information about your connection options.

SAS/ACCESS-LIBNAME-options
define how DBMS objects are processed by SAS. Some LIBNAME options can enhance performance; others determine locking or naming behavior. For example, the PRESERVE_COL_NAMES= option enables you to specify whether to preserve spaces, special characters, and mixed case in DBMS column names when creating tables. The availability and default behavior of many of these options are DBMS-specific.

See the documentation for your SAS/ACCESS interface for a list of the LIBNAME options that are available for your DBMS.

See "LIBNAME Options for Relational Databases" on page 78 for detailed information about all of the LIBNAME options.

CLEAR
disassociates one or more currently assigned librefs.

Specify *libref* to disassociate a single libref. Specify _ALL_ to disassociate all currently assigned librefs.

ALL
specifies that the CLEAR or LIST argument applies to all currently-assigned librefs.

LIST
writes the attributes of one or more SAS/ACCESS libraries or SAS data libraries to the SAS log.

Specify *libref* to list the attributes of a single SAS/ACCESS library or SAS library. Specify _ALL_ to list the attributes of all libraries that have librefs in your current session.

Details

❶ Using Data from a DBMS You can use a LIBNAME statement to read from and write to a DBMS table or view as though it were a SAS data set.

For example, in MYDBLIB.EMPLOYEES_Q2, MYDBLIB is a SAS libref that points to a particular group of DBMS objects, and EMPLOYEES_Q2 is a DBMS table name. When you specify MYDBLIB.EMPLOYEES_Q2 in a DATA step or procedure, you dynamically access the DBMS table. SAS supports reading, updating, creating, and deleting DBMS tables dynamically.

❷ Disassociating a Libref from a SAS Library To disassociate or clear a libref from a DBMS, use a LIBNAME statement, specifying the libref (for example, MYDBLIB) and the CLEAR option as follows:

```
libname mydblib CLEAR;
```

You can clear a single specified libref or all current librefs.

The database engine disconnects from the database and closes any free threads or resources that are associated with that libref's connection.

❸ Writing SAS Library Attributes to the SAS Log Use a LIBNAME statement to write the attributes of one or more SAS/ACCESS libraries or SAS data libraries to the SAS log. Specify *libref* to list the attributes of a single SAS/ACCESS library or SAS library, as follows:

```
libname mydblib LIST;
```

Specify _ALL_ to list the attributes of all libraries that have librefs in your current session.

SQL Views with Embedded LIBNAME Statements

With SAS software, you can embed LIBNAME statements in the definition of an SQL view. This means that you can store, in an SQL view, a LIBNAME statement that contains all the information required to connect to a DBMS. Whenever the SQL view is read, PROC SQL uses the embedded LIBNAME statement to assign a libref. After the view has been processed, PROC SQL de-assigns the libref.

In the following example, an SQL view of the Oracle table DEPT is created. Whenever this view is used in a SAS program, the library ORALIB is assigned, using the connection information (user name, password, and data source) that is provided in the embedded LIBNAME statement.

```
proc sql;
    create view sasuser.myview as
       select dname from oralib.dept
          using libname oralib oracle
                user=scott pw=tiger datasrc=orsrv;
quit;
```

Note: The USING LIBNAME syntax is used to embed LIBNAME statements in SQL views. For more information about the USING LIBNAME syntax, see the PROC SQL topic in the *Base SAS Procedures Guide*. △

Assigning a Libref with a SAS/ACCESS LIBNAME Statement

The following statement creates a libref, MYDBLIB, that uses the SAS/ACCESS interface for DB2:

```
libname mydblib db2 ssid=db2a authid=testid server=os390svr;
```

The following statement associates the SAS libref MYDBLIB with an Oracle database that uses the SQL*Net alias AIRDB_REMOTE. You specify the SCHEMA= option on the SAS/ACCESS LIBNAME statement to connect to the Oracle schema in which the database resides. In this example, Oracle schemas reside in a database.

```
libname mydblib oracle user=testuser password=testpass
        path=airdb_remote schema=hrdept;
```

The AIRDB_REMOTE database contains a number of DBMS objects, including several tables, such as STAFF. After you assign the libref, you can reference the Oracle table like a SAS data set and use it as a data source in any DATA step or SAS procedure. In the following SQL procedure statement, MYDBLIB.STAFF is the two-level SAS name for the STAFF table in the Oracle database AIRDB_REMOTE:

```
proc sql;
   select idnum, lname
      from mydblib.staff
      where state='NY'
      order by lname;
quit;
```

You can use the DBMS data to create a SAS data set:

```
data newds;
   set mydblib.staff(keep=idnum lname fname);
run;
```

You can also use the libref and data set with any other SAS procedure. This statement prints the information in the STAFF table:

```
proc print data=mydblib.staff;
run;
```

This statement lists the database objects in the MYDBLIB library:

```
proc datasets library=mydblib;
quit;
```

Using the Prompting Window When Specifying LIBNAME Options

The following statement uses the DBPROMPT= option to cause the DBMS connection prompting window to appear and prompt you for connection information:

```
libname mydblib oracle dbprompt=yes;
```

When you use this option, you enter connection information into the fields in the prompting window rather than on the LIBNAME statement.

You can add the DEFER=NO option to make the prompting window appear at the time that the libref is assigned rather than when the table is opened:

```
libname mydblib oracle dbprompt=yes defer=no;
```

Assigning a Libref to a Remote DBMS

SAS/CONNECT (single-user) and SAS/SHARE (multiple user) software give you access to data by means of *remote library services* (RLS). RLS enables you to access your data on a remote machine as if it were local. For example, it permits a graphical interface to reside on the local machine while the data remains on the remote machine.

This access is given to data stored in many kinds of SAS files, such as external databases (through the SAS/ACCESS LIBNAME statement and views created with it) and SAS data views (views created with PROC SQL, the DATA step, and SAS/ACCESS software). RLS enables you to access SAS data sets, SAS views, and relational DBMS data that are defined by SAS/ACCESS LIBNAME statements. For more information, see the discussion about remote library services in the *SAS/SHARE User's Guide*.

You can use RLS to update relational DBMS tables that are referenced with the SAS/ACCESS LIBNAME statement.

In the following example, the SAS/ACCESS LIBNAME statement makes a connection to a DB2 database that resides on the remote SAS/SHARE server REMOS390. This LIBNAME statement is submitted in a local SAS session. The SAS/ACCESS engine name is specified in the remote option RENGINE=. The DB2 connection option and any LIBNAME options are specified in the remote option ROPTIONS=. Options are separated by a blank space. RLSDB2.EMPLOYEES is a SAS data set that references the DB2 table EMPLOYEES.

```
libname rlsdb2 rengine=db2 server=remos390
               roptions="ssid=db2a authid=testid";

proc print data=rlsdb2.employees;
run;
```

See Also

"Overview of the LIBNAME Statement for Relational Databases" on page 73

LIBNAME Options for Relational Databases

When you specify an option in the LIBNAME statement, it applies to all objects (such as tables and views) in the database that the libref represents. For information

about options that you specify on individual SAS data sets, see the chapter about data set options.

Many LIBNAME options are also available for use with the Pass-Through Facility. See the section on the Pass-Through Facility in the documentation for your SAS/ACCESS interface to determine which LIBNAME options are available in the Pass-Through Facility for your DBMS.

For a list of the LIBNAME options available in your SAS/ACCESS interface, see the documentation for your SAS/ACCESS interface.

Note: When a like-named option is specified in both the LIBNAME statement and after a data set name, SAS uses the value that is specified on the data set name. △

ACCESS= LIBNAME Option

Determines the access level with which a libref connection is opened

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: none

Syntax

ACCESS=READONLY

Syntax Description

READONLY
specifies that tables and views can be read but not updated.

Details

Using this option prevents writing to the DBMS. If this option is omitted, tables and views can be read and updated if you have the necessary DBMS privileges.

AUTHID= LIBNAME Option

Enables you to qualify your table names with an authorization ID, user ID, or group ID

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS

Default value: none

Syntax

AUTHID=*authorization-ID*

Syntax Description

authorization-ID
cannot exceed 8 characters.

Details

When you specify the AUTHID= option, every table that is referenced by the libref is qualified as *authid.tablename* before any SQL code is passed to the DBMS. If you do not specify a value for AUTHID=, the table name is not qualified before it is passed to the DBMS. After the DBMS receives the table name, it automatically qualifies it with your user ID. You can override the LIBNAME AUTHID= option by using the AUTHID= data set option. This option is not validated until you access a table.

SCHEMA= is an alias for this option.

See Also

To apply this option to an individual data set, see the data set option "AUTHID= Data Set Option" on page 156.

AUTOCOMMIT= LIBNAME Option

Indicates whether updates are committed immediately after they are submitted

Valid in: the SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. Please refer to your DBMS for details.

DBMS support: DB2 UNIX/PC, ODBC, OLE DB, Microsoft SQL Server, MySQL, Sybase

Default value: DBMS-specific, see details below

Syntax

AUTOCOMMIT= YES | NO

Syntax Description

YES
specifies that all updates, deletes, and inserts are committed (that is, saved to a table) immediately after they are submitted, and no rollback is possible.

NO
specifies that the SAS/ACCESS engine automatically performs the commit when it reaches the DBCOMMIT= value, or the default number of rows if DBCOMMIT is not set.

Details

If you are using the SAS/ACCESS LIBNAME statement, the default is NO if the data source provider supports transactions and the connection is used for updating data. For

MySQL, the default is YES. For read-only connections and the Pass-Through Facility, the default is YES.

See Also

To apply this option to an individual data set, see the data set option "AUTOCOMMIT= Data Set Option" on page 156.

BL_KEEPIDENTITY= LIBNAME Option

Determines whether the identity column that is created during bulk loading is populated with values generated by Microsoft SQL Server or with values provided by the user

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: OLE DB

Default value: NO

Syntax

BL_KEEPIDENTITY= YES | NO

Syntax Description

YES
specifies that the user must provide values for the identity column.

NO
specifies that Microsoft SQL Server generates values for an identity column in the table.

Details

This option is only valid when you use the Microsoft SQL Server provider.

See Also

To apply this option to an individual data set, see the data set option "BL_KEEPIDENTITY= Data Set Option" on page 175.

BL_KEEPNULLS= LIBNAME Option

Indicates how NULL values in Microsoft SQL Server columns that accept NULL are handled during bulk loading

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: OLE DB

Default value: YES

Syntax

BL_KEEPNULLS= YES | NO

Syntax Description

YES
specifies that Microsoft SQL Server preserves NULL values inserted by the OLE DB interface.

NO
specifies that Microsoft SQL Server replaces NULL values that are inserted by the OLE DB interface with a default value (as specified in the DEFAULT constraint).

Details

This option only affects values in Microsoft SQL Server columns that accept NULL and have a DEFAULT constraint.

See Also

To apply this option to an individual data set, see the data set option "BL_KEEPNULLS= Data Set Option" on page 176.

BL_LOG= LIBNAME Option

Specifies the name of the error file to which all errors are written when BULKLOAD=YES

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: ODBC, Microsoft SQL Server
Default value: none

Syntax

BL_LOG=*filename*

Details

This option is only valid for connections to Microsoft SQL Server. If BL_LOG= is not specified, errors are not recorded during bulk loading.

See Also

To apply this option to an individual data set, see the data set option "BL_LOG= Data Set Option" on page 178.

BL_OPTIONS= LIBNAME Option

Passes options to the DBMS bulk load facility, affecting how it loads and processes data

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: ODBC, OLE DB
Default value: not specified

Syntax

BL_OPTIONS=*'option <..., option>'*

Details

BL_OPTIONS= enables you to pass options to the DBMS bulk load facility when it is invoked, thereby affecting how data is loaded and processed. You must separate multiple options with commas and enclose the entire string of options in quotation marks.

By default, no options are specified. This option takes the same values as the *-h* HINT option of the Microsoft BCP utility. Refer to the Microsoft SQL Server documentation for more information about bulk copy options.

This option is only valid when you are using the Microsoft SQL Server driver or the Microsoft SQL Server provider on Windows platforms.

ODBC Details: The supported hints are ORDER, ROWS_PER_BATCH, KILOBYTES_PER_BATCH, TABLOCK, and CHECK_CONSTRAINTS. If you specify UPDATE_LOCK_TYPE=TABLE, the TABLOCK hint is automatically added.

See Also

To apply this option to an individual data set, see the data set option "BL_OPTIONS= Data Set Option" on page 179.

BULKLOAD= LIBNAME Option

Determines whether SAS uses a DBMS facility to insert data into a DBMS table

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: ODBC, OLE DB, Teradata
Default value: NO

Syntax

BULKLOAD=YES | NO

Syntax Description

YES
calls a DBMS-specific bulk load facility in order to insert or append rows to a DBMS table.

NO
does not call the DBMS bulk load facility.

Details

See the SAS/ACCESS documentation for your DBMS for additional, DBMS-specific details.

See Also

To apply this option to an individual data set, see the data set option "BULKLOAD= Data Set Option" on page 189.

CAST= LIBNAME Option

Specifies whether data conversions should be performed on the Teradata DBMS server or by SAS

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: Teradata
Default value: none

Syntax

CAST=YES | NO

Syntax Description

YES
forces data conversions (casting) to be done on the Teradata DBMS server and overrides any data overhead percentage limit.

NO

forces data conversions to be done by SAS and overrides any data overhead percentage limit.

Details

Internally, SAS numbers and dates are floating point values. Teradata has varying formats for numbers, including integers, floating point values, and decimal values. Number conversion must occur when you are reading Teradata numbers that are not floating point (Teradata FLOAT). SAS/ACCESS can use the Teradata CAST= function to cause Teradata to perform numeric conversions. The parallelism of Teradata makes it well suited to perform this work. This is especially true when running SAS on z/OS (MVS) where CPU activity can be costly.

CAST= can cause more data to be transferred from Teradata to SAS, as a result of the option forcing the Teradata type into a larger SAS type. For example, the CAST= transfer of a Teradata BYTEINT to SAS floating point adds seven overhead bytes to each row transferred.

The following Teradata types are candidates for casting:

- INTEGER
- BYTEINT
- SMALLINT
- DECIMAL
- DATE.

SAS/ACCESS limits data expansion for CAST= to 20 percent in order to trade rapid data conversion by Teradata for extra data transmission. If casting does not exceed a 20 percent data increase, all candidate columns are cast. If the increase exceeds this limit, then SAS attempts to cast Teradata DECIMAL types only. If casting only DECIMAL types still exceeds the increase limit, data conversions are done by SAS.

You can alter the casting rules by using either the CAST= or "CAST_OVERHEAD_MAXPERCENT= LIBNAME Option" on page 86 option. With CAST_OVERHEAD_MAXPERCENT=, you can change the 20 percent overhead limit. With CAST=, you can override the percentage rules:

- CAST=YES forces Teradata to cast all candidate columns
- CAST=NO cancels all Teradata casting

CAST= only applies when you are reading Teradata tables into SAS. It does not apply when you are writing Teradata tables from SAS.

Also, CAST= only applies to SQL that SAS generates for you. If you supply your own SQL with the explicit SQL feature of PROC SQL, you must code your own casting clauses to force data conversions to occur in Teradata instead of SAS.

Examples

The following example demonstrates the use of the CAST= option in a LIBNAME statement to force casting for all tables referenced:

```
libname mydblib teradata user=testuser pw=testpass cast=yes;
proc print data=mydblib.emp;
where empno<1000;
run;

proc print data=mydblib.sal;
where salary>50000;
run;
```

The following example demonstrates the use of the CAST= option in a table reference in order to turn off casting for that table:

```
proc print data=mydblib.emp (cast=no);
where empno<1000;
run;
```

See Also

"CAST= Data Set Option" on page 189

CAST_OVERHEAD_MAXPERCENT= LIBNAME Option

Specifies the overhead limit for data conversions to be performed in Teradata instead of SAS

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: Teradata
Default value: 20 percent

Syntax

CAST_OVERHEAD_MAXPERCENT=<*n*>

Syntax Description

<n>
Any positive numeric value. The engine default is 20.

Details

Teradata INTEGER, BYTEINT, SMALLINT, and DATE columns require conversion when read in to SAS. Conversions can be performed either by Teradata or by SAS. When performed in Teradata, using Teradata's CAST operator, the row size transmitted to SAS can increase. CAST_OVERHEAD_MAXPERCENT= limits the allowable increase, also called conversion overhead.

Examples

The following example demonstrates the use of CAST_OVERHEAD_MAXPERCENT= to increase the allowable overhead to 40 percent:

```
proc print data=mydblib.emp (cast_overhead_maxpercent=40);
where empno<1000;
run;
```

See Also

"CAST= LIBNAME Option" on page 84 for more information about conversions, conversion overhead, and casting.

CELLPROP= LIBNAME Option

Modifies the metadata and content of a result data set that is defined through an MDX command

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: OLE DB
Default value: VALUE

Syntax

CELLPROP=VALUE | FORMATTED_VALUE

Syntax Description

VALUE
specifies that the SAS/ACCESS engine tries to return actual data values. If all of the values in a column are numeric, then that column is defined as NUMERIC.

FORMATTED_VALUE
specifies that the SAS/ACCESS engine returns formatted data values. All of the columns are defined as CHARACTER.

Details

When an MDX command is issued, the resulting data set might have columns that contain one or more types of data values — the actual value of the cell or the formatted value of the cell.

For example, if you issue an MDX command and the resulting data set contains a column named SALARY, the column could contain data values of two types. It could contain numeric values, such as **50000**, or it could contain formatted values, such as **$50,000**. Setting the CELLPROP= option determines how the values are defined and the value of the column.

It is possible for a column in a result set to contain both NUMERIC and CHARACTER data values. For example, a data set might return the data values of **50000**, **60000**, and **UNKNOWN**. SAS data sets cannot contain both types of data. In this situation, even if you specify CELLPROP=VALUE, the SAS/ACCESS engine defines the column as CHARACTER and returns formatted values for that column.

For more information about MDX commands, see the SAS/ACCESS documentation for OLE DB.

COMMAND_TIMEOUT= LIBNAME Option

Specifies the number of seconds to wait before a data source command times out

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: OLE DB
Default value: 0

Syntax

COMMAND_TIMEOUT=*number-of-seconds*

Syntax Description

number-of-seconds
is an integer greater than or equal to 0.

Details

The default value is 0, which means there is no time-out.

See Also

To apply this option to an individual data set, see the data set option "COMMAND_TIMEOUT= Data Set Option" on page 192.

CONNECTION= LIBNAME Option

Specifies whether operations on a single libref share a connection to the DBMS, and whether operations on multiple librefs share a connection to the DBMS

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

CONNECTION= SHAREDREAD | UNIQUE | SHARED | GLOBALREAD | GLOBAL

Syntax Description

Not all values are valid for all SAS/ACCESS interfaces. See details below.

SHAREDREAD
specifies that all READ operations that access DBMS tables *in a single libref* share a single connection. A separate connection is established for every table that is opened for update or output operations.

Where available, this is usually the default value because it offers the best performance and it guarantees data integrity.

UNIQUE
specifies that a separate connection is established every time a DBMS table is accessed by your SAS application.

Use UNIQUE if you want each use of a table to have its own connection.

SHARED (This value is valid in the interfaces to DB2 under z/OS, DB2 under UNIX and PC hosts, ODBC, Oracle, and Microsoft SQL Server.)
specifies that *all* operations that access DBMS tables *in a single libref* share a single connection.

Use this option with caution. When a single SHARED connection is used for multiple table opens, a commit or rollback performed on one table being updated also applies to all other tables opened for update. Even if a table is just opened for READ, its READ cursor might get resynchronized as a result of this commit or rollback. If the cursor is resynchronized, there is no guarantee that the new solution table will match the original solution table that was being read.

Use SHARED to eliminate the deadlock that can occur when you create and load a DBMS table from an existing table that resides in the same database or tablespace. This only happens in certain output processing situations and is the only recommended use for CONNECTION=SHARED.

Note: The CONNECTION= option only influences the connections used for opening tables with a libref. Setting CONNECTION=SHARED has no influence on utility connections or explicit passthrough connections. △

GLOBALREAD
specifies that all READ operations that access DBMS tables *in multiple librefs* share a single connection if the following is true:

- the participating librefs are created by LIBNAME statements that specify identical values for the CONNECTION=, CONNECTION_GROUP=, DBCONINIT=, DBCONTERM=, DBLIBINIT=, and DBLIBTERM= options
- the participating librefs are created by LIBNAME statements that specify identical values for any DBMS connection options.

A separate connection is established for each table that is opened for update or output operations.

GLOBAL (This value is valid in the interfaces to DB2 under z/OS, DB2 under UNIX and PC hosts, ODBC, Oracle, and Microsoft SQL Server.)
specifies that *all* operations that access DBMS tables *in multiple librefs* share a single connection if the following is true:

- all of the participating librefs are created by LIBNAME statements that specify identical values for the CONNECTION=, CONNECTION_GROUP=, DBCONINIT=, DBCONTERM=, DBLIBINIT=, and DBLIBTERM= options
- all of the participating librefs are created by LIBNAME statements that specify identical values for any DBMS connection options.

One connection is shared for all tables that are referenced by any of the librefs for which CONNECTION=GLOBAL is specified.

Use this option with caution. When a GLOBAL connection is used for multiple table opens, a commit/rollback performed on one table being updated also applies to all other tables opened for update. Even if a table is just opened for READ, its READ cursor might get resynchronized as a result of this commit/rollback. If the cursor is resynchronized, there is no guarantee that the new solution table will match the original solution table that was being read.

Details

For most SAS/ACCESS interfaces, there must be a connection, also known as an *attach*, to the DBMS server before any data can be accessed. Typically, each DBMS connection has one transaction, or work unit, that is active in the connection. This transaction is affected by any SQL commits or rollbacks that the engine performs within the connection while executing the SAS application.

The CONNECTION= option enables you to control the number of connections, and therefore transactions, that your SAS/ACCESS interface executes and supports for each LIBNAME statement.

GLOBALREAD is the default value for CONNECTION= when you specify CONNECTION_GROUP=.

This option is supported by the SAS/ACCESS interfaces that support single connections or multiple, simultaneous connections to the DBMS.

MySQL Details: The default value is UNIQUE.

ODBC and Microsoft SQL Server Details: If the data source supports only one active open cursor per connection, the default value is CONNECTION=UNIQUE; otherwise, the default value is CONNECTION=SHAREDREAD.

Teradata Details: for channel-attached systems (MVS), the default is SHAREDREAD; for network attached systems (UNIX and PC platforms), the default is UNIQUE

Examples

In the following SHAREDREAD example, MYDBLIB makes the first connection to the DBMS. This connection is used to print the data from MYDBLIB.TAB. MYDBLIB2 makes the second connection to the DBMS. A third connection is used to update MYDBLIB.TAB. The third connection is closed at the end of the PROC SQL UPDATE statement. The first and second connections are closed with the CLEAR option.

```
libname mydblib oracle user=testuser  /* connection 1 */
        pw=testpass path='myorapath'
        connection=sharedread;

libname mydblib2 oracle user=testuser /* connection 2 */
        pw=testpass path='myorapath'
        connection=sharedread;

proc print data=mydblib.tab ...
proc sql;                            /* connection 3 */
   update mydblib.tab ...

libname mydblib clear;
libname mydblib2 clear;
```

In the following GLOBALREAD example, the two librefs, MYDBLIB and MYDBLIB2, share the same connection for read access because CONNECTION=GLOBALREAD and the connection options are identical. The first connection is used to print the data from MYDBLIB.TAB while a second connection is made for updating MYDBLIB.TAB. The second connection is closed at the end of the step. Note that the first connection is closed with the final LIBNAME statement.

```
libname mydblib oracle user=testuser /* connection 1 */
        pw=testpass path='myorapath'
        connection=globalread;

libname mydblib2 oracle user=testuser
        pw=testpass path='myorapath'
        connection=globalread;

proc print data=mydblib.tab ...
proc sql;                          /* connection 2 */
```

```
   update mydblib.tab ...

libname mydblib clear;          /* does not close connection 1 */
libname mydblib2 clear;         /* closes connection 1 */
```

In the following UNIQUE example, the libref, MYDBLIB, does not establish a connection. A connection is established in order to print the data from MYDBLIB.TAB. That connection is closed at the end of the print procedure. Another connection is established to updated MYDBLIB.TAB. That connection is closed at the end of the PROC SQL. The CLEAR option in the LIBNAME statement at the end of this example does not close any connections.

```
libname mydblib oracle user=testuser
        pw=testpass path='myorapath'
        connection=unique;

proc print data=mydblib.tab ...
proc sql;
   update mydblib.tab ...

libname mydblib clear;
```

In the following GLOBAL example for DB2 under z/OS, both PROC DATASETS invocations appropriately report "no members in directory" because SESSION.B, as a temporary table, has no entry in the system catalog SYSIBM.SYSTABLES. However, the DATA _NULL_ step and SELECT * from PROC SQL step both return the expected rows. For DB2 under z/OS, when SCHEMA=SESSION the database first looks for a temporary table before attempting to access any physical schema named SESSION.

```
libname x db2 connection=global schema=SESSION;
proc datasets lib=x;
quit;

/*
 *  DBMS-specific code to create a temporary table impervious
 *  to commits, and then populate the table directly in the
 *  DBMS from another table.
 */
proc sql;
connect to db2(connection=global schema=SESSION);
execute ( DECLARE GLOBAL TEMPORARY TABLE SESSION.B LIKE SASDXS.A
          ON COMMIT PRESERVE ROWS
         ) by db2;
execute ( insert into SESSION.B select * from SASDXS.A
         ) by db2;
quit;

/*
 *  Get at the temp table through the global libref.
 */
data _null_;
set x.b;
put _all_;
run;

/*
```

```
 *  Get at the temp table through the global connection.
 */
proc sql;
connect to db2 (connection=global schema=SESSION);
select * from connection to db2
( select * from SESSION.B  );
quit;

proc datasets lib=x;
quit;
```

In the following SHARED example, DB2DATA.NEW is created in the database TEST. Because the table DB2DATA.OLD exists in the same database, the option CONNECTION=SHARED enables the DB2 engine to share the connection both for reading the old table and for creating and loading the new table.

```
libname db2data db2 connection=shared;
data db2data.new (in = 'database test');
   set db2data.old;
run;
```

In the following GLOBAL example, two different librefs share one connection.

```
libname db2lib db2 connection=global;
libname db2data db2 connection=global;
data db2lib.new(in='database test');
   set db2data.old;
run;
```

If you did not use the CONNECTION= option in the above two examples, you would deadlock in DB2 and get the following error:

```
ERROR:  Error attempting to CREATE a DBMS table.
ERROR:  DB2 execute error DSNT408I SQLCODE = --911,
ERROR:  THE CURRENT UNIT OF WORK HAS BEEN ROLLED
        BACK DUE TO DEADLOCK.
```

See Also

"DEFER= LIBNAME Option" on page 109
"ACCESS= LIBNAME Option" on page 79
"CONNECTION_GROUP= LIBNAME Option" on page 92

CONNECTION_GROUP= LIBNAME Option

Causes operations on multiple librefs to share a connection to the DBMS. Also causes operations on multiple Pass-Through Facility CONNECT statements to share a connection to the DBMS

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: none

Syntax

CONNECTION_GROUP= *connection-group-name*

Syntax Description

connection-group-name
is the name of a connection group.

Details

This option causes a DBMS connection to be shared by all READ operations on multiple librefs if the following is true:

- all of the participating librefs are created by LIBNAME statements that specify the same value for CONNECTION_GROUP=
- all of the participating librefs are created by LIBNAME statements that specify identical DBMS connection options.

To share a connection for *all* operations against multiple librefs, specify CONNECTION= GLOBAL on all participating LIBNAME statements. Not all SAS/ACCESS interfaces support CONNECTION=GLOBAL.

Note: If you specify CONNECTION=GLOBAL or CONNECTION=GLOBALREAD, operations on multiple librefs can share a connection even if you omit CONNECTION_GROUP=. △

For Informix users, the CONNECTION_GROUP option enables multiple librefs or multiple Pass-Through Facility CONNECT statements to share a connection to the DBMS. This overcomes the Release 8.2 limitation in which users were unable to access scratch tables across step boundaries as a result of new connections being established with every procedure.

Example

In the following example, the MYDBLIB libref shares a connection with MYDBLIB2 by specifying CONNECTION_GROUP=MYGROUP and by specifying identical connection options. The libref MYDBLIB3 makes a second connection to another connection group called ABC. The first connection is used to print the data from MYDBLIB.TAB, and is also used for updating MYDBLIB.TAB. The third connection is closed at the end of the step. Note that the first connection is closed by the final LIBNAME statement for that connection. Similarly, the second connection is closed by the final LIBNAME statement for that connection.

```
libname mydblib oracle user=testuser    /* connection 1 */
        pw=testpass
        connection_group=mygroup;

libname mydblib2 oracle user=testuser
        pw=testpass
        connection_group=mygroup;
```

```
libname mydblib3 oracle user=testuser   /* connection 2 */
        pw=testpass
        connection_group=abc;

proc print data=mydblib.tab ...
proc sql;                               /* connection 1 */
   update mydblib.tab ...

libname mydblib clear;     /* does not close connection 1*/
libname mydblib2 clear;    /* closes connection 1 */
libname mydblib3 clear;    /* closes connection 2 */
```

CURSOR_TYPE= LIBNAME Option

Specifies the cursor type for read-only and updatable cursors

Valid in: the SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. Please refer to your DBMS for details.
DBMS support: DB2 UNIX/PC, ODBC, OLE DB, Microsoft SQL Server
Default value: DBMS-specific

Syntax

CURSOR_TYPE= DYNAMIC | FORWARD_ONLY | KEYSET_DRIVEN | STATIC

Syntax Description

DYNAMIC
specifies that the cursor reflects all of the changes that are made to the rows in a result set as you move the cursor. The data values and the membership of rows in the cursor can change dynamically on each fetch. This is the default for the DB2 UNIX/PC, ODBC, and Microsoft SQL Server interfaces.

FORWARD_ONLY
specifies that the cursor functions like a DYNAMIC cursor except that it only supports fetching the rows sequentially. (This value is not valid in OLE DB.)

KEYSET_DRIVEN
specifies that the cursor determines which rows belong to the result set when the cursor is opened. However, changes that are made to these rows are reflected as you scroll around the cursor.

STATIC
specifies that the cursor builds the complete result set when the cursor is opened. No changes that are made to the rows in the result set after the cursor is opened are reflected in the cursor. Static cursors are read-only.

Details

Not all drivers support all cursor types. An error is returned if the specified cursor type is not supported.

The driver is allowed to modify the default without an error.

OLE DB Details: By default, this option is not set and the provider uses a default. See your provider documentation for more information. See OLE DB programmer reference documentation for details about these properties. The OLE DB properties applied to an open row set are as follows:

CURSOR_TYPE=	OLE DB Properties Applied
DYNAMIC	DBPROP_OTHERINSERT=TRUE, DBPROP_OTHERUPDATEDELETE=TRUE
KEYSET_DRIVEN	DBPROP_OTHERINSERT=FALSE, DBPROP_OTHERUPDATEDELET=TRUE
STATIC	DBPROP_OTHERINSERT=FALSE, DBPROP_OTHERUPDATEDELETE=FALSE

See Also

To apply this option to an individual data set, see the data set option "CURSOR_TYPE= Data Set Option" on page 192.

DBCOMMIT= LIBNAME Option

Causes an automatic COMMIT (a permanent writing of data to the DBMS) after a specified number of rows have been processed

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: 1000 when inserting rows into a DBMS table; 0 when updating a DBMS table

Syntax

DBCOMMIT=*n*

Syntax Description

n
is an integer greater than or equal to 0.

Details

DBCOMMIT= affects update, delete, and insert processing. The number of rows that are processed includes rows that are not processed successfully. If you set DBCOMMIT=0, a commit is issued only once (after the procedure or DATA step completes).

If the DBCOMMIT= option is explicitly set, SAS/ACCESS fails any update that has a WHERE clause.

Note: If you specify both DBCOMMIT= and ERRLIMIT=, and these options collide during processing, then the COMMIT is issued first and the ROLLBACK is issued second. Because the COMMIT (caused by the DBCOMMIT= option) is issued prior to the ROLLBACK (caused by the ERRLIMIT= option), the DBCOMMIT= option is said to override the ERRLIMIT= option in this situation. △

DB2 under UNIX and PC Hosts Details: When BULKLOAD=YES, the default is 10000.

See Also

To apply this option to an individual data set, see the data set option "DBCOMMIT= Data Set Option" on page 193.

DBCONINIT= LIBNAME Option

Specifies a user-defined initialization command to be executed immediately after every connection to the DBMS that is within the scope of the LIBNAME statement or libref

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: none

Syntax

DBCONINIT=<'>*DBMS-user-command*<'>

Syntax Description

DBMS-user-command
is any valid command that can be executed by the SAS/ACCESS engine and that does not return a result set or output parameters.

Details

The initialization command that you select can be a stored procedure or any DBMS SQL statement that might provide additional control over the interaction between your SAS/ACCESS interface and the DBMS.

The command executes immediately after each DBMS connection is successfully established. If the command fails, then a disconnect occurs and the libref is not assigned. You must specify the command as a single, quoted string.

Note: The initialization command might execute more than once, because one LIBNAME statement might have multiple connections; for example, one for reading and one for updating. △

Examples

In the following example, the DBCONINIT= option causes the DBMS to apply the SET statement to every connection that uses the MYDBLIB libref.

```
libname mydblib db2
        dbconinit="SET CURRENT SQLID='myauthid'";

proc sql;
   select * from mydblib.customers;

   insert into mydblib.customers
     values('33129804', 'VA', '22809', 'USA',
            '540/545-1400', 'BENNETT SUPPLIES', 'M. JONES',
            '2199 LAUREL ST', 'ELKTON', '22APR97'd);

   update mydblib.invoices
    set amtbill = amtbill*1.10
    where country = 'USA';

quit;
```

In the following example, a stored procedure is passed to DBCONINIT=.

```
libname mydblib oracle user=testuser pass=testpass
        dbconinit="begin dept_test(1001,25)";
end;
```

The SAS/ACCESS engine retrieves the stored procedure and executes it.

See Also

"DBCONTERM= LIBNAME Option" on page 97

DBCONTERM= LIBNAME Option

Specifies a user-defined termination command to be executed before every disconnect from the DBMS that is within the scope of the LIBNAME statement or libref

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: none

Syntax

DBCONTERM=<'>*DBMS-user-command*<'>

Syntax Description

DBMS-user-command
is any valid command that can be executed by the SAS/ACCESS engine and that does not return a result set or output parameters.

Details

The termination command that you select can be a stored procedure or any DBMS SQL statement that might provide additional control over the interaction between the SAS/ACCESS engine and the DBMS. The command executes immediately before SAS terminates each connection to the DBMS. If the command fails, then SAS provides a warning message but the library deassignment and disconnect still occur. You must specify the command as a single, quoted string.

Note: The termination command might execute more than once, because one LIBNAME statement might have multiple connections; for example, one for reading and one for updating. △

Examples

In the following example, the DBMS drops the Q1_SALES table before SAS disconnects from the DBMS.

```
libname mydblib db2 user=testuser using=testpass
        db=invoice bconterm='drop table q1_sales';
```

In the following example, the stored procedure, SALESTAB_STORED_PROC, is executed each time SAS connects to the DBMS, and the BONUSES table is dropped when SAS terminates each connection.

```
libname mydblib db2 user=testuser
        using=testpass db=sales
        dbconinit='exec salestab_stored_proc'
        dbconterm='drop table bonuses';
```

See Also

"DBCONINIT= LIBNAME Option" on page 96

DBCREATE_TABLE_OPTS= LIBNAME Option

Specifies DBMS-specific syntax to be added to the CREATE TABLE statement

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata
Default value: none

Syntax

DBCREATE_TABLE_OPTS=*'DBMS-SQL-clauses'*

DBMS-SQL-clauses
are one or more DBMS-specific clauses that can be appended to the end of an SQL CREATE TABLE statement.

Details

DBCREATE_TABLE_OPTS= enables you to add DBMS-specific clauses to the end of the SQL CREATE TABLE statement. The SAS/ACCESS engine passes the SQL CREATE TABLE statement and its clauses to the DBMS, which executes the statement and creates the DBMS table. DBCREATE_TABLE_OPTS= applies only when you are creating a DBMS table by specifying a libref associated with DBMS data.

See Also

To apply this option to an individual data set, see the data set option "DBCREATE_TABLE_OPTS= Data Set Option" on page 195.

DBGEN_NAME= LIBNAME Option

Specifies how SAS automatically renames DBMS columns that contain characters that SAS does not allow, such as $, to valid SAS variable names

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS

Syntax

DBGEN_NAME= DBMS | SAS

Syntax Description

DBMS
specifies that the DBMS columns are renamed to valid SAS variable names. Disallowed characters are converted to underscores. If a column is converted to a name that already exists, then a sequence number is appended to the end of the new name.

SAS
specifies that DBMS columns are renamed to the format _COL*n*, where *n* is the column number (starting with zero).

Details

SAS retains column names when it reads data from tables, unless a column name contains characters that SAS does not allow, such as **$**. SAS allows alphanumeric characters and the underscore (_).

For example, if you specify DBGEN_NAME=SAS, a DBMS column named **Dept$Amt** is renamed to **_COL*n***. If you specify DBGEN_NAME=DBMS, the **Dept$Amt** column is renamed to **Dept_Amt**.

This option is intended primarily for National Language Support, notably for the conversion of Kanji to English characters. English characters that are converted from Kanji are often those that are not allowed in SAS.

Note: The various SAS/ACCESS interfaces each handled name collisions differently in SAS Version 6. Some interfaces appended to the end of the name; other interfaces replaced the last character(s) in the name. Some interfaces used a single sequence number, other interfaces used unique counters. If you specify VALIDVARNAME=V6 ; then name collisions must be handled the same as they were in SAS Version 6. △

See Also

To apply this option to an individual data set, see the data set option "DBGEN_NAME= Data Set Option" on page 197

DBINDEX= LIBNAME Option

Improves performance when processing a join that involves a large DBMS table and a small SAS data set

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

DBINDEX= YES | NO

Syntax Description

YES
specifies that SAS uses columns in the WHERE clause that have defined DBMS indexes.

NO
specifies that SAS does not use indexes that are defined on DBMS columns.

Details

When you are processing a join that involves a large DBMS table and a relatively small SAS data set, you might be able to use DBINDEX= to improve performance.

CAUTION:
Improper use of this option can degrade performance. See "Using the DBINDEX=, DBKEY=, and MULTI_DATASRC_OPT= Options" on page 43 for detailed information about using this option. △

See Also

To apply this option to an individual data set, see the data set option "DBINDEX= Data Set Option" on page 199.

DBLIBINIT= LIBNAME Option

Specifies a user-defined initialization command to be executed once within the scope of the LIBNAME statement or libref that established the first connection to the DBMS

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: none

Syntax

DBLIBINIT= <'>*DBMS-user-command*<'>

Syntax Description

DBMS-user-command
is any DBMS command that can be executed by the SAS/ACCESS engine and that does not return a result set or output parameters.

Details

The initialization command that you select can be a script, stored procedure, or any DBMS SQL statement that might provide additional control over the interaction between your SAS/ACCESS interface and the DBMS.

The command executes immediately after the first DBMS connection is successfully established. If the command fails, then a disconnect occurs and the libref is not assigned. You must specify the command as a single, quoted string, unless it is an environment variable.

DBLIBINIT= fails if either CONNECTION=UNIQUE or DEFER=YES, or if both of these LIBNAME options are specified.

When multiple LIBNAME statements share a connection, the initialization command executes only for the first LIBNAME statement, immediately after the DBMS connection is established. (Multiple LIBNAME statements that use CONNECTION=GLOBALREAD and identical values for CONNECTION_GROUP=, DBCONINIT=, DBCONTERM=, DBLIBINIT=, and DBLIBTERM= options and any DBMS connection options can share the same connection to the DBMS.)

Example

In the following example, CONNECTION=GLOBALREAD is specified in both LIBNAME statements, but the DBLIBINIT commands are different. Therefore, the second LIBNAME statement fails to share the same physical connection.

```
libname mydblib oracle user=testuser pass=testpass
        connection=globalread dblibinit='Test';

libname mydblib2 oracle user=testuser pass=testpass
        connection=globalread dblibinit='NoTest';
```

See Also

"DBLIBTERM= LIBNAME Option" on page 102

DBLIBTERM= LIBNAME Option

Specifies a user-defined termination command to be executed once, before the DBMS that is associated with the first connection made by the LIBNAME statement or libref disconnects

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: none

Syntax

DBLIBTERM= <'>*DBMS-user-command*<'>

Syntax Description

DBMS-user-command
is any DBMS command that can be executed by the SAS/ACCESS engine and that does not return a result set or output parameters.

Details

The termination command that you select can be a script, stored procedure, or any DBMS SQL statement that might provide additional control over the interaction between the SAS/ACCESS engine and the DBMS. The command executes immediately before SAS terminates the last connection to the DBMS. If the command fails, then SAS provides a warning message but the library deassignment and disconnect still occurs. You must specify the command as a single, quoted string.

DBLIBTERM= fails if either CONNECTION=UNIQUE or DEFER=YES or both of these LIBNAME options are specified.

When two LIBNAME statements share the same physical connection, the termination command is executed only once. (Multiple LIBNAME statements that use CONNECTION=GLOBALREAD and identical values for CONNECTION_GROUP=, DBCONINIT=, DBCONTERM=, DBLIBINIT=, and DBLIBTERM= options and any DBMS connection options can share the same connection to the DBMS.)

Example

In the following example, CONNECTION=GLOBALREAD is specified on both LIBNAME statements, but the DBLIBTERM commands are different. Therefore, the second LIBNAME statement fails to share the same physical connection.

```
libname mydblib oracle user=testuser pass=testpass
   connection=globalread dblibterm='Test';

libname mydblib2 oracle user=testuser pass=testpass
   connection=globalread dblibterm='NoTest';
```

See Also

"DBLIBINIT= LIBNAME Option" on page 101

DBLINK= LIBNAME Option

Specifies a link from your local database to database objects on another server (in the Oracle interface); specifies a link from your default database to another database on the server to which you are connected (in the Sybase interface)

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: Oracle, Sybase

Default value: none

Syntax

DBLINK=*database-link*

Details

Oracle Details: A link is a database object that is used to identify an object stored in a remote database. A link contains stored path information and may also contain user name and password information for connecting to the remote database. If you specify a link, SAS uses the link to access remote objects. If you omit DBLINK=, SAS accesses objects in the local database.

Sybase Details: This option enables you to link to another database within the same server to which you are connected. If you omit DBLINK=, SAS can only access objects in your default database.

See Also

To apply this option to an individual data set, see the data set option "DBMASTER= Data Set Option" on page 203.

DBMAX_TEXT= LIBNAME Option

Determines the length of any very long DBMS character data type that is read into SAS or written from SAS when using a SAS/ACCESS engine

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 UNIX/PC, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase

Default value: 1024

Syntax

DBMAX_TEXT=<*integer*>

Syntax Description

integer
is an integer between 1 and 32,767.

Details

This option applies to reading, appending, and updating rows in an existing table. It does not apply when you are creating a table.

Examples of a DBMS data type are the SYBASE TEXT data type or the Oracle LONG RAW data type.

See Also

To apply this option to an individual data set, see the data set option "DBMAX_TEXT= Data Set Option" on page 204.

DBNULLKEYS= LIBNAME Option

Controls the format of the WHERE clause when you use the DBKEY= data set option

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, ODBC, OLE DB, Oracle, Microsoft SQL Server

Default value: DBMS-specific

Syntax

DBNULLKEYS= YES | NO

Details

If there might be NULL values in the transaction table or the master table for the columns that you specify in the DBKEY= option, use DBNULLKEYS=YES. This is the default for most interfaces. When you specify DBNULLKEYS=YES and specify a column that is not defined as NOT NULL in the DBKEY= data set option, SAS generates a WHERE clause that can find NULL values. For example, if you specify DBKEY=COLUMN and COLUMN is not defined as NOT NULL, SAS generates a WHERE clause with the following syntax:

```
WHERE ((COLUMN = ?) OR ((COLUMN IS NULL) AND (? IS NULL)))
```

This syntax enables SAS to prepare the statement once and use it for any value (NULL or NOT NULL) in the column. Note that this syntax has the potential to be much less efficient than the shorter form of the WHERE clause (presented below). When you specify DBNULLKEYS=NO or specify a column that is defined as NOT NULL in the DBKEY= option, SAS generates a simple WHERE clause.

If you know that there are no NULL values in the transaction table or the master table for the columns that you specify in the DBKEY= option, then you can use DBNULLKEYS=NO. This is the default for the interface to Informix. If you specify DBNULLKEYS=NO and specify DBKEY=COLUMN, SAS generates a shorter form of the WHERE clause (regardless of whether or not the column specified in DBKEY= is defined as NOT NULL):

```
WHERE (COLUMN = ?)
```

See Also

To apply this option to an individual data set, see the data set option "DBNULLKEYS= Data Set Option" on page 206.

DBPROMPT= LIBNAME Option

Specifies whether SAS displays a window that prompts the user to enter DBMS connection information prior to connecting to the DBMS in interactive mode

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 UNIX/PC, Informix, MySQL, ODBC, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: NO

Syntax

DBPROMPT=YES | NO

Syntax Description

YES
specifies that SAS displays a window that interactively prompts you for the DBMS connection options the first time the libref is used.

NO
specifies that SAS does not display the prompting window.

Details

If you specify DBPROMPT=YES, it is not necessary to provide connection options with the LIBNAME statement. If you do specify connection options with the LIBNAME statement and you specify DBPROMPT=YES, then the connection option values are displayed in the window (except for the password value, which appears as a series of asterisks). All of these values can be overridden interactively.

The DBPROMPT= option interacts with the DEFER= option to determine when the prompt window appears. If DEFER=NO, the DBPROMPT window opens when the LIBNAME statement is executed. If DEFER=YES, the DBPROMPT window opens the first time a table or view is opened. The DEFER= option normally defaults to NO but defaults to YES if DBPROMPT=YES. You can override this default by explicitly setting DEFER=NO.

The DBPROMPT window usually opens only once for each time that the LIBNAME statement is specified. It might open multiple times if DEFER=YES and the connection fails when SAS tries to open a table. In these cases, the DBPROMPT window opens until a successful connection occurs or you click Cancel.

The maximum password length for most of the SAS/ACCESS LIBNAME interfaces is 32 characters.

Oracle Details: You can enter 30 characters for the USERNAME and PASSWORD and up to 70 characters for the PATH, depending on your platform.

Teradata Details: You can enter up to 30 characters for the USERNAME and PASSWORD.

Examples

In the following example, the DBPROMPT window does not open when the LIBNAME statement is submitted because DEFER=YES. The DBPROMPT window opens when the PRINT procedure is processed, a connection is made, and the table is opened.

```
libname mydblib oracle dbprompt=yes
   defer=yes;

proc print data=mydblib.staff;
run;
```

In the following example, the DBPROMPT window opens while the LIBNAME statement is processing. The DBPROMPT window does not open in subsequent statements because the DBPROMPT window opens only once per LIBNAME statement.

```
libname mydblib oracle dbprompt=yes
   defer=no;
```

In the following example, values provided in the LIBNAME statement are pulled into the DBPROMPT window. The values **testuser** and **ABC_server** appear in the DBPROMPT window and can be edited and confirmed by the user. The password value appears in the DBPROMPT window as a series of asterisks; it can also be edited by the user.

```
libname mydblib oracle
        user=testuser pw=testpass
```

```
path='ABC_server' dbprompt=yes defer=no;
```

See Also

To apply this option to a view descriptor, see the data set option "DBPROMPT= Data Set Option" on page 207.

DBSASLABEL= LIBNAME Option

Specifies the column labels an engine uses

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: COMPAT

Syntax

DBSASLABEL=COMPAT | NONE

Syntax Description

COMPAT
specifies that the labels returned should be compatible with what the application normally receives. In otherwords, engines exhibit their normal behavior.

NONE
specifies that the engine does not return a column label. The engine will return blanks for the column labels.

Details

By default, the SAS/ACCESS interface for your DBMS generates column labels from the column names, rather than from the real column labels.

This option enables the user to override this default behavior. It is useful in the PROC SQL context where column labels instead of column aliases are used as headers.

Examples

The following example demonstrates how DBSASLABEL= is used as a LIBNAME option to return blank column labels so that PROC SQL can use the column aliases as the column headers.

```
libname x oracle user=scott pw=tiger;
proc sql;
     select deptno as Department ID, loc as Location from mylib.dept(dbsaslabel=none);
```

Without the DBSASLABEL= option set to NONE, the aliases would be ignored, and DEPTNO and LOC would be used as column headers in the result set.

DBSLICEPARM= LIBNAME Option

Controls the scope of DBMS threaded reads and the number of threads

Valid in: the SAS/ACCESS LIBNAME statement (Also available as a SAS configuration option, SAS invocation option, global SAS option, or data set option)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, Microsoft SQL Server, ODBC, Oracle, Sybase, Teradata

Default value: THREADED_APPS,2 (DB2 z/OS, Oracle,Teradata,) THREADED_APPS,2 *or* 3 (DB2 UNIX/PC, Informix, Microsoft SQL Server, ODBC, Sybase)

Syntax

DBSLICEPARM=NONE | THREADED_APPS |ALL

DBSLICEPARM=(NONE | THREADED_APPS |ALL <, *max-threads*>)

Syntax Description

Two syntax diagrams are shown here in order to highlight the simpler version. In most cases, the simpler version suffices.

NONE
disables DBMS threaded read. SAS reads tables on a single DBMS connection, as it did with SAS Version 8 and earlier.

THREADED_APPS
Makes fully threaded SAS procedures (threaded applications) eligible for threaded reads.

ALL
Makes all read-only librefs eligible for threaded reads. This includes SAS threaded applications, as well as the SAS DATA step and numerous SAS procedures.

max-threads
positive integer value that specifies the maximum number of connections per table read. A partition or portion of the data is read on each connection. The combined rows across all partitions are the same irrespective of the number of connections. Changes to the number of connections do not change the result set. Increasing the number of connections instead redistributes the same result set across more connections.

There are diminishing returns when increasing the number of connections. With each additional connection, more burden is placed on the DBMS, and a smaller percentage of time saved on the SAS step. Therefore, you should consult your DBMS-specific documentation concerning partitioned reads before using this parameter.

Details

DBSLICEPARM= can be used in numerous locations, and the usual rules of option precedence apply. Table option has the highest precedence, then LIBNAME option, and so on. SAS configuration file option has the lowest precedence as DBSLICEPARM= in any of the other locations overrides that configuration setting.

DBSLICEPARM=ALL and DBSLICEPARM=THREADED_APPS make SAS programs eligible for threaded reads. To see if threaded reads are actually generated, turn on SAS tracing and run a program, as shown in the following example:

```
options sastrace=',,t' sastraceloc=saslog nostsuffix;
proc print data=lib.dbtable(dbsliceparm=(ALL));
   where dbcol>1000;
run;
```

If you want to directly control the threading behavior, use the DBSLICE= data set option.

For DB2 UNIX/PC, Informix, Microsoft SQL Server, ODBC, and Sybase, the default thread number is dependent on whether an application passes in the number of threads (CPUCOUNT=) and whether the data type of the column selected for the data partitioning purpose is binary.

Examples

The following code demonstrates how to use DBSLICEPARM= in a PC SAS configuration file entry to turn off threaded reads for all SAS users:

```
--dbsliceparm NONE
```

The following code demonstrates how to use DBSLICEPARM= as an z/OS invocation option to turn on threaded reads for read-only references to DBMS tables throughout a SAS job:

```
sas o(dbsliceparm=ALL)
```

The following code demonstrates how to use DBSLICEPARM= as a SAS global option, most likely as one of the first statements in your SAS code, to increase maximum threads to three for SAS threaded apps:

```
option dbsliceparm=(threaded_apps,3);
```

The following code demonstrates how to use DBSLICEPARM= as a LIBNAME option to turn on threaded reads for read-only table references that use this particular libref:

```
libname dblib oracle user=scott password=tiger dbsliceparm=ALL;
```

The following code demonstrates how to use DBSLICEPARM= as a table level option to turn on threaded reads for this particular table, requesting up to four connections:

```
proc reg SIMPLE;
   data=dblib.customers (dbsliceparm=(all,4));
   var age weight;
     where years_active>1;
run;
```

See Also

"DBSLICEPARM= Data Set Option" on page 212

DEFER= LIBNAME Option

Specifies when the connection to the DBMS occurs

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: NO

Syntax

DEFER= NO | YES

Syntax Description

NO
specifies that the connection to the DBMS occurs when the libref is assigned by a LIBNAME statement.

YES
specifies that the connection to the DBMS occurs when a table in the DBMS is opened.

Details

The default value of NO is overridden if DBPROMPT=YES.

The DEFER= option is ignored when CONNECTION=UNIQUE, because a connection is performed every time a table is opened.

ODBC and Microsoft SQL Server Details: When you set DEFER=YES, you must also set the PRESERVE_TAB_NAMES= and PRESERVE_COL_NAMES= options to their desired values. Normally, SAS queries the data source to determine the correct defaults for these options during LIBNAME assignment, but setting DEFER=YES postpones the connection. Because these values must be set at the time of LIBNAME assignment, you must assign them explicitly when you set DEFER=YES.

See Also

"CONNECTION= LIBNAME Option" on page 88

DEGREE= LIBNAME Option

Determines whether DB2 uses parallelism

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS

Default value: ANY

Syntax

DEGREE= ANY | 1

Syntax Description

ANY
enables DB2 to use parallelism, and issues the SET CURRENT DEGREE ='xxx' for all DB2 threads that use that libref.

1
explicitly disables the use of parallelism.

Details

When DEGREE=ANY, DB2 has the option of using parallelism, when it is appropriate.

Setting DEGREE=1 prevents DB2 from performing parallel operations. Instead, DB2 is restricted to performing one task that, while perhaps slower, uses less system resources.

DELETE_MULT_ROWS= LIBNAME Option

Indicates whether to allow SAS to delete multiple rows from a data source, such as a DBMS table

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: ODBC, OLE DB, Microsoft SQL Server
Default value: NO

Syntax

DELETE_MULT_ROWS=YES | NO

Syntax Description

YES
specifies that SAS/ACCESS processing continues if multiple rows are deleted. This might produce unexpected results.

NO
specifies that SAS/ACCESS processing does not continue if multiple rows are deleted.

Details

Some providers do not handle the following DBMS SQL statement well and, therefore, delete more than the current row:

```
DELETE ... WHERE CURRENT OF CURSOR
```

DIRECT_EXE= LIBNAME Option

Enables you to pass an SQL delete statement directly to a DBMS via implicit pass-through.

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: none

Syntax

DIRECT_EXE=DELETE

Syntax Description

DELETE
specifies that an SQL delete statement is passed directly to the DBMS for processing.

Details

Performance is greatly increase by using DIRECT_EXE=, because the SQL delete statement is passed directly to the DBMS, instead of SAS reading the entire result set and deleting one row at a time.

Examples

The following example demonstrates the use of DIRECT_EXE= to empty a table from a database.

```
libname x oracle user=scott password=tiger
   path=oraclev8 schema=dbitest
direct_exe=delete; /* create an oracle table of 5 rows */data x.dbi_dft;
do col1=1 to 5;
output;
end;
run;

options sastrace=",,,d" sastraceloc=saslog nostsuffix;
proc sql;
delete * from x.dbi_dft;
quit;
```

By turning trace on, you should see something similar to the following:

Output 9.1 SAS Log Output

```
ORACLE_9: Executed:
delete from dbi_dft
```

DIRECT_SQL= LIBNAME Option

Enables you to specify whether generated SQL is passed to the DBMS for processing

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: YES

Syntax

DIRECT_SQL= YES | NO | NONE | NOGENSQL | NOWHERE |NOFUNCTIONS |NOMULTOUTJOINS

Syntax Description

YES
specifies that generated SQL from PROC SQL is passed directly to the DBMS for processing.

NO
specifies that generated SQL from PROC SQL is not passed to the DBMS for processing. This is the same as specifying the value NOGENSQL.

NONE
specifies that generated SQL is not passed to the DBMS for processing. This includes SQL that is generated from PROC SQL, SAS functions that can be converted into DBMS functions, joins, and WHERE clauses.

NOGENSQL
prevents PROC SQL from generating SQL to be passed to the DBMS for processing.

NOWHERE
prevents WHERE clauses from being passed to the DBMS for processing. This includes SAS WHERE clauses and PROC SQL generated or PROC SQL specified WHERE clauses.

NOFUNCTIONS
prevents SQL statements from being passed to the DBMS for processing when they contain functions.

NOMULTOUTJOINS
specifies that PROC SQL will not attempt to pass any multiple outer joins to the DBMS for processing. Other join statements may be passed down however, including portions of a multiple outer join.

Details

By default, processing is passed to the DBMS whenever possible, because the database might be able to process the functionality more efficiently than SAS does. In some instances, however, you might not want the DBMS to process the SQL. For example, the presence of null values in the DBMS data might cause different results depending on whether the processing takes place in SAS or in the DBMS. If you do not want the

DBMS to handle the SQL, use DIRECT_SQL= to force SAS to handle some or all of the SQL processing.

If you specify DIRECT_SQL=NOGENSQL, then PROC SQL does not generate DBMS SQL. This means that SAS functions, joins, and DISTINCT processing that occur *within* PROC SQL are not passed to the DBMS for processing. (SAS functions *outside* PROC SQL can still be passed to the DBMS.) However, if PROC SQL contains a WHERE clause, the WHERE clause *is* passed to the DBMS, if possible. Unless you specify DIRECT_SQL=NOWHERE, SAS attempts to pass all WHERE clauses to the DBMS.

If you specify more than one value for this option, separate the values with spaces and enclose the list of values in parentheses. For example, you could specify DIRECT_SQL=(NOFUNCTIONS, NOWHERE).

DIRECT_SQL= overrides the LIBNAME option "SQL_FUNCTIONS= LIBNAME Option" on page 143. If you specify SQL_FUNCTIONS=ALL and DIRECT_SQL=NONE, no functions are passed.

Examples

The following example prevents a join between two tables from being processed by the DBMS, by setting DIRECT_SQL=NOGENSQL. Instead, SAS processes the join.

```
proc sql;
create view work.v as
   select tab1.deptno, dname from
          mydblib.table1 tab1,
          mydblib.table2 tab2
   where tab1.deptno=tab2.deptno
   using libname mydblib oracle user=testuser
          password=testpass path=myserver direct_sql=nogensql;
```

The following example prevents a SAS function from being processed by the DBMS.

```
libname mydblib oracle user=testuser password=testpass direct_sql=nofunctions;
proc print data=mydblib.tab1;
   where lastname=soundex ('Paul');
```

ENABLE_BULK= LIBNAME Option

Enables the connection to process bulk copy when you load data into a Sybase table

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: Sybase

Default value: YES

Syntax

ENABLE_BULK=YES | NO

Syntax Description

NO
disables the bulk copy ability for the libref.

YES
enables the connection to perform a bulk copy of SAS data into Sybase.

Details

Bulk copy groups rows so that they are inserted as a unit into the Sybase table. Using bulk copy can improve performance.

If you use both the LIBNAME option, ENABLE_BULK=, and the data set option, BULKLOAD=, the values of the two options must be the same or an error is returned. However, since the default value of ENABLE_BULK= is YES, you do not have to specify ENABLE_BULK= in order to use the BULKLOAD= data set option.

Note: In Version 7 and previous releases, this option was called BULKCOPY=. In Version 8 and later, an error is returned if you specify BULKCOPY=. △

ERRLIMIT= LIBNAME Option

Specifies the number of errors that are allowed while using the Fastload utility before SAS stops loading data to Teradata.

Valid in: DATA and PROC steps (wherever Fastload is used)
DBMS support: Teradata
Default value: 1 million

Syntax

ERRLIMIT=*integer*

Syntax Description

integer
positive integer that represents the number of errors after which SAS stops loading data.

Details

SAS stops loading data when the specified number of errors is reached and Fastload is paused. When Fastload is paused, the table being loaded cannot be used. Since restart capability for Fastload is not yet supported, the error tables must be manually deleted before the table can be loaded again.

Example

In the following example, SAS stops processing and pauses Fastload at the occurrence of the tenth error.

```
libname mydblib teradata user=terauser pw=XXXXXX ERRLIMIT=10;

data mydblib.trfload(bulkload=yes dbtype=(i='int check (i > 11)') );
     do
         i=1 to 50000;output;
     end;
run;
```

IGNORE_ READ_ONLY_COLUMNS= LIBNAME Option

Specifies whether to ignore or include columns whose data types are read-only when generating an SQL statement for inserts or updates

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 UNIX/PC, ODBC, OLE DB, Microsoft SQL Server

Default value: NO

Syntax

IGNORE_READ_ONLY_COLUMNS=YES | NO

Syntax Description

YES
specifies that the SAS/ACCESS engine ignores columns whose data types are read-only when you are generating insert and update SQL statements

NO
specifies that the SAS/ACCESS engine does not ignore columns whose data types are read-only when you are generating insert and update SQL statements

Details

Several databases include data types that can be read-only, such as Microsoft SQL Server's timestamp data type. Several databases also have properties that allow certain data types to be read-only, such as Microsoft SQL Server's identity property.

When the IGNORE_READ_ONLY_COLUMNS option is set to NO (the default), and a DBMS table contains a column that is read-only, an error is returned indicating that the data could not be modified for that column.

Example

For the following example, a database that contains the table Products is created with two columns: ID and PRODUCT_NAME. The ID column is defined by a read-only data type and PRODUCT_NAME is a character column.

```
CREATE TABLE products (id int IDENTITY PRIMARY KEY, product_name varchar(40))
```

Assume you have a SAS data set that contains the name of your products, and you would like to insert the data into the Products table:

```
data work.products;
   id=1;
   product_name='screwdriver';
   output;
   id=2;
   product_name='hammer';
   output;
   id=3;
   product_name='saw';
   output;
   id=4;
   product_name='shovel';
   output;
run;
```

With IGNORE_READ_ONLY_COLUMNS=NO (the default), an error is returned by the database because in this example, the ID column cannot be updated. However, if you set the option to YES and execute a PROC APPEND, the append succeeds, and the SQL statement that is generated does not contain the ID column.

```
libname x odbc uid=dbitest pwd=dbigrp1 dsn=lupinss
              ignore_read_only_columns=yes;
options sastrace=',,,d' sastraceloc=saslog nostsuffix;
proc append base=x.PRODUCTS data=work.products;
run;
```

See Also

To apply this option to an individual data set, see the data set option "IGNORE_ READ_ONLY_COLUMNS= Data Set Option" on page 217.

IN= LIBNAME Option

Enables you to specify the database and tablespace in which you want to create a new table

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC

Default value: none

Syntax

IN=*'database-name.tablespace-name'* | 'DATABASE *database-name*'

Syntax Description

database-name.tablespace-name
specifies the names of the database and tablespace, which are separated by a period. Enclose the entire specification in single quotation marks.

DATABASE *database-name*
specifies only the database name. Specify the word DATABASE, then a space, then the database name. Enclose the entire specification in single quotation marks.

Details

The IN= option is relevant only when you are creating a new table. If you omit this option, the default is to create the table in the default database, implicitly creating a simple tablespace.

See Also

To apply this option to an individual data set, see the data set option "IN= Data Set Option" on page 218.

INSERT_SQL= LIBNAME Option

Determines the method that is used to insert rows into a data source

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: ODBC, OLE DB, Microsoft SQL Server
Default value: DBMS-specific, see details below

Syntax

INSERT_SQL=YES | NO

Syntax Description

YES
specifies that SAS/ACCESS uses the data source's SQL insert method to insert new rows into a table.

NO
specifies that SAS/ACCESS uses an alternate (DBMS-specific) method to insert new rows into a table.

Details

Flat file databases (such as dBASE, FoxPro, and text files) generally have improved insert performance when INSERT_SQL=NO, but other databases might have inferior insert performance (or might fail) with this setting, so you should experiment to determine the optimal setting for your situation.

ODBC Details: The ODBC default is YES, except for Microsoft Access, which has a default of NO. When INSERT_SQL=NO, the SQLSetPos (SQL_ADD) function inserts rows in groups that are the size of the INSERTBUFF= option value. The SQLSetPos (SQL_ADD) function does not work unless it is supported by your driver.

OLE DB Details: By default, the OLE DB interface attempts to use the most efficient row insertion method for each data source. You can use the INSERT_SQL option to

override the default in the event that it is not optimal for your situation. The OLE DB alternate method (used when this option is set to NO) uses the OLE DB IRowsetChange interface.

Microsoft SQL Server Details: The Microsoft SQL Server default is YES. When INSERT_SQL=NO, the SQLSetPos (SQL_ADD) function inserts rows in groups that are the size of the INSERTBUFF= option value. The SQLSetPos (SQL_ADD) function does not work unless it is supported by your driver.

See Also

To apply this option to an individual data set, see the data set option "INSERT_SQL= Data Set Option" on page 219.

INSERTBUFF= LIBNAME Option

Specifies the number of rows in a single insert operation

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 UNIX/PC, ODBC, OLE DB, Oracle, Microsoft SQL Server, MySQL

Default value: DBMS-specific

Syntax

INSERTBUFF=*positive-integer*

Syntax Description

positive-integer
specifies the number of rows to insert. SAS allows the maximum that is allowed by the DBMS.

Details

The optimal value for this option varies with factors such as network type and available memory. You might need to experiment with different values in order to determine the best value for your site.

The SAS application messages that indicate the success or failure of an insert operation represent information for only a single insert, even when multiple inserts are performed. Therefore, when you assign a value that is greater than INSERTBUFF=1, these messages might be incorrect.

If you specify the DBCOMMIT= option with a value that is less than the value of INSERTBUFF=, then DBCOMMIT= overrides INSERTBUFF=.

Note: When you insert rows by using the VIEWTABLE window or the FSVIEW or FSEDIT procedure, use INSERTBUFF=1 to prevent the DBMS interface from trying to insert multiple rows. These features do not support inserting more than one row at a time. △

Note: Additional driver-specific restrictions might apply. △

DB2 UNIX/PC Details: You must specify INSERT_SQL=YES in order to use this option. If one row in the insert buffer fails, all rows in the insert buffer fail. The default is calculated based upon the row length of your data.

ODBC Details: The default is 1.

OLE DB Details: The default is 1.

Oracle Details: The default is 10.

Microsoft SQL Server Details: You must specify INSERT_SQL=YES in order to use this option. The default is 1.

MySQL Details: The default is 0. Any value greater than 0 turns on the INSERTBUFF= option. The engine then calculates how many rows it can insert at one time, based on the row size. If one row in the insert buffer fails, all rows in the insert buffer might fail, depending on your storage type.

See Also

To apply this option to an individual data set, see the data set option "INSERTBUFF= Data Set Option" on page 220.

INTERFACE= LIBNAME Option

Specifies the name and location of the interfaces file that is searched when you connect to the Sybase server

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: Sybase

Default value: none

Syntax

INTERFACE=<'>*filename*<'>

Details

The interfaces file contains names and access information for the available servers on the network. If you omit a filename, the default action for your operating system occurs. INTERFACE= is not used in some operating environments. Contact your database administrator to see whether this statement applies to your computing environment.

KEYSET_SIZE= LIBNAME Option

Specifies the number of rows that are keyset driven

Valid in: the SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. Please refer to your DBMS for details.

DBMS support: ODBC, Microsoft SQL Server

Default value: 0

Syntax

KEYSET_SIZE=*number-of-rows*

Syntax Description

number-of-rows
is an integer with a value between 0 and the number of rows in the cursor.

Details

This option is valid only when CURSOR_TYPE=KEYSET_DRIVEN. See "CURSOR_TYPE= LIBNAME Option" on page 94 for more information about keyset-driven cursors.

If KEYSET_SIZE=0, then the entire cursor is keyset driven. If a value greater than 0 is specified for KEYSET_SIZE=, then the value chosen indicates the number of rows within the cursor that will functions as a keyset-driven cursor. When you scroll beyond the bounds that are specified by KEYSET_SIZE=, then the cursor becomes dynamic and new rows might be included in the cursor. This becomes the new keyset and the cursor functions as a keyset-driven cursor again. Whenever the value that is specified is between 1 and the number of rows in the cursor, the cursor is considered to be a mixed cursor because part of the cursor functions as a keyset-driven cursor and part of the cursor functions as a dynamic cursor.

See Also

To apply this option to an individual data set, see the data set option "KEYSET_SIZE= Data Set Option" on page 221.

LOCATION= LIBNAME Option

Enables you to further qualify exactly where a table resides

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: DB2 z/OS
Default value: none

Syntax

LOCATION=*location*

Details

The location name maps to the location in the SYSIBM.SYSLOCATIONS catalog in the communication database.

In the DB2 interface, the location is converted to the first level of a three-level table name: *location.authid.table*. The connection to the remote DB2 subsystem is done implicitly by DB2's Distributed Data Facility (DDF) when DB2 receives a three-level table name in an SQL statement.

If you omit this option, SAS accesses the data from the local DB2 database unless you have specified a value for the SERVER= option. This option is not validated until you access a DB2 table. If you specify LOCATION=, you must also specify the AUTHID= option.

See Also

- To apply this option to an individual data set, see the data set option "LOCATION= Data Set Option" on page 222
- For information about accessing a database server on Linux, UNIX, or Windows using a libref, see the REMOTE_DBTYPE= LIBNAME option.

.

LOCKTABLE= LIBNAME Option

Places exclusive or shared locks on tables

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: Informix
Default value: no locking

Syntax

LOCKTABLE= EXCLUSIVE | SHARE

Syntax Description

EXCLUSIVE
specifies that other users are prevented from accessing each table that you open in the libref.

SHARE
specifies that other users or processes can read data from the tables, but they cannot update the data.

Details

You may lock tables only if you are the owner or have been granted the necessary privilege.

See Also

To apply this option to an individual data set, see the data set option "LOCKTABLE= Data Set Option" on page 222.

LOCKTIME= LIBNAME Option

Specifies the number of seconds to wait until rows are available for locking

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: Informix

Default value: none

Syntax

LOCKTIME=*positive-integer*

Details

You must specify LOCKWAIT=YES for LOCKTIME= to have an effect. If you omit the LOCKTIME= option and use LOCKWAIT=YES, SAS suspends your process indefinitely until a lock can be obtained.

See Also

"LOCKWAIT= LIBNAME Option" on page 123

LOCKWAIT= LIBNAME Option

Specifies whether to wait indefinitely until rows are available for locking

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: Informix, Oracle

Default value: DBMS-specific

Syntax

LOCKWAIT= YES | NO

Syntax Description

YES
specifies that SAS waits until rows are available for locking.

NO
specifies that SAS does not wait and returns an error to indicate that the lock is not available.

LOGDB= LIBNAME Option

Redirects the log tables created by Teradata's FastExport utility to an alternate database

Valid in: DATA and PROC steps (wherever Fastload is used)
DBMS support: Teradata
Default value: Default Teradata database for the libref

Syntax

LOGDB=<*database-name*>

Syntax Description

database-name
the name of the Teradata database.

Details

This option enables the restart log tables that are used by Teradata's FastExport utility to be created in an alternate database. You must have necessary permissions to create tables in the specified database. Only restart tables are created in the specified database. Restart capability of FastExport is not yet supported.

Example

In the following example, PROC PRINT invokes Teradata's FastExport utility if it is installed. The LOGDB= option in the LIBNAME statement causes the restart log tables that are created by Teradata's FastExport to be created in the database ALTDB.

```
libname mydblib teradata user=testuser pw=testpass logdb=altdb;
proc print data=mydblib.mytable(dbsliceparm=all);
run;
```

MAX_CONNECTS= LIBNAME Option

Specifies the maximum number of simultaneous connections that Sybase allows

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: Sybase
Default value: 25

Syntax

MAX_CONNECTS=*numeric-value*

Details

If you omit MAX_CONNECTS=, the default for the maximum number of connections is 25. Note that increasing the number of connections has a direct impact on memory.

MULTI_DATASRC_OPT= LIBNAME Option

Used in place of DBKEY to improve performance when processing a join between two data sources

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: NONE

Syntax

MULTI_DATASRC_OPT=NONE |IN_CLAUSE

Syntax Description

NONE
turns off the functionality of the option.

IN_CLAUSE
specifies that an IN clause containing the values read from a smaller table will be used to retrieve the matching values in a larger table based on a key column designated in an equi-join.

Details

When processing a join between a SAS data set and a DBMS table, the SAS data set should be smaller than the DBMS table for optimal performance. However, in the event that the SAS data set is *larger* than that DBMS table, the SAS data set will still be used in the IN clause.

When SAS is processing a join between two DBMS tables, SELECT COUNT (*) is issued to determine which table is smaller and if it qualifies for an IN clause. You can use "DBMASTER= Data Set Option" on page 203 to prevent the SELECT COUNT (*) from being issued.

Currently, the IN clause has a limit of 4,500 unique values.

Oracle Details: Oracle can handle an IN clause of only 1,000 values. Therefore, it divides larger IN clauses into multiple, smaller IN clauses. The results are combined into a single result set. For example if an IN clause contained 4,000 values, Oracle will produce 4 IN clauses that each contain 1,000 values. A single result will be produced, as if all 4,000 values were processed as a whole.

OLE DB Details: OLE DB restricts the number of values allowed in an IN clause to 255.

Setting DBKEY= automatically overrides MULTI_DATASRC_OPT=.

DIRECT_SQL= can impact this option as well. If DIRECT_SQL=NONE or NOWHERE, the IN clause cannot be built and passed to the DBMS, regardless of the

value of MULTI_DATASRC_OPT=. These setting for DIRECT_SQL= prevent a WHERE clause from being passed.

Examples

The following example builds and passes an IN clause from the SAS table to the DBMS table, retrieving only the necessary data to process the join:

```
proc sql;
create view work.v as
select tab2.deptno, tab2.dname from
work.sastable tab1, dblib.table2 tab2
where tab12.deptno = tab2.deptno
using libname dblib oracle user=testuser password=testpass
  multi_datasrc_opt=in_clause;
quit;
```

The following example prevents the building and passing of the IN clause to the DBMS, requiring all rows from the DBMS table to be brought into SAS for processing the join:

```
 libname dblib oracle user=testuser password=testpass multi_datasrc_opt=none;
proc sql;
   select tab2.deptno, tab2.dname from
     work.table1 tab1,
     dblib.table2 tab2
   where tab1.deptno=tab2.deptno;
quit;
```

See Also

"DBMASTER= Data Set Option" on page 203

OR_UPD_NOWHERE= LIBNAME Option

Specifies whether SAS uses an extra WHERE clause when updating rows with no locking

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: Oracle
Default value: YES

Syntax

OR_UPD_NOWHERE= YES | NO

Syntax Description

YES
specifies that SAS does not use an additional WHERE clause to determine whether each row has changed since it was read. Instead, SAS uses the SERIALIZABLE

isolation level (available with Oracle 7.3 and above) for update locking. If a row changes after the serializable transaction starts, the update on that row fails.

NO
specifies that SAS uses an additional WHERE clause to determine whether each row has changed since it was read. If a row has changed since being read, the update fails.

Details

Use this option when you are updating rows without locking (UPDATE_LOCK_TYPE=NOLOCK).

By default (OR_UPD_NOWHERE=YES), updates are performed in serializable transactions. This enables you to avoid extra WHERE clause processing and potential WHERE clause floating point precision problems.

ORACLE_73_OR_ABOVE= is an alias for this option.

Note: Due to the published Oracle bug 440366, sometimes an update on a row fails even if the row has not changed. Oracle offers the following solution: When creating a table, increase the number of INITRANS to at least 3 for the table. △

See Also

To apply this option to an individual data set or a view descriptor, see the data set option "OR_UPD_NOWHERE= Data Set Option" on page 241.

PACKETSIZE= LIBNAME Option

Enables you to specify the packet size for Sybase to use

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: Sybase

Default value: current server setting

Syntax

PACKETSIZE=*numeric-value*

Syntax Description

numeric-value
is any multiple of 512, up to the limit of the maximum network packet size setting on your server.

Details

If you omit PACKETSIZE=, the default is the current server setting. You can query the default network packet value in ISQL by using the Sybase **`sp_configure`** command.

PREFETCH= LIBNAME Option

Enables the PreFetch facility on tables that are accessed by the libref defined with the LIBNAME statement

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: Teradata
Default value: not enabled

Syntax

PREFETCH='*unique_storename*, [#*sessions*,*algorithm*]'

Syntax Description

unique_storename
is a unique name that you specify. This value names the Teradata macro that PreFetch creates to store selected SQL statements in the first run of a job. During subsequent runs of the job, SAS/ACCESS presubmits the stored SQL statements in parallel to the Teradata DBMS.

#sessions
controls the number of statements that PreFetch submits in parallel to Teradata. A valid value is 1 through 9. If you do not specify a *#sessions* value, the default is 3.

algorithm
specifies the algorithm that PreFetch uses to order the selected SQL statements. Currently, the only valid value is SEQUENTIAL.

Details

Before using PreFetch, see the PreFetch description in the SAS/ACCESS documentation for Teradata for a complete discussion, including when and how the option enhances read performance of a job that is run more than once.

PRESERVE_COL_NAMES= LIBNAME Option

Preserves spaces, special characters, and case-sensitivity in DBMS column names when you create DBMS tables

Valid in: the SAS/ACCESS LIBNAME statement (when you create DBMS tables)
DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Teradata
Default value: DBMS-specific

Syntax

PRESERVE_COL_NAMES= NO | YES

Syntax Description

NO
specifies that column names that are used in DBMS table creation are derived from SAS variable names by using the SAS variable name normalization rules (see the VALIDVARNAME system option for more information). However, the database will apply its DBMS-specific normalization rules to the SAS variable names when creating the DBMS column names.

The use of N-Literals to create column names that use database keywords or special symbols other than the underscore character might be illegal when DBMS normalization rules are applied. To include nonstandard SAS symbols or database keywords, specify PRESERVE_COL_NAMES=YES.

NO is the default for most DBMS interfaces.

YES
specifies that column names that are used in table creation are passed to the DBMS with special characters and the exact, case-sensitive spelling of the name preserved.

Details

This option applies only when you use SAS/ACCESS to create a new DBMS table. When you create a table, you assign the column names by using one of the following methods:

- To control the case of the DBMS column names, specify variables using the desired case and set PRESERVE_COL_NAMES=YES. If you use special symbols or blanks, you must set VALIDVARNAME= to ANY and use N-Literals. See the section about names in *SAS/ACCESS for Relational Databases: Reference* and the section about system options in *SAS Language Reference: Dictionary* for further information.
- To enable the DBMS to normalize the column names according to its naming conventions, specify variables using any case and set PRESERVE_COLUMN_NAMES= NO.

Note: When you use SAS/ACCESS to read from, insert rows into, or modify data in an existing DBMS table, SAS identifies the database column names by their spelling. Therefore, when the database column exists, the case of the variable does not matter. △

See the topic about naming in the documentation for your SAS/ACCESS interface for additional details.

Specify the alias PRESERVE_NAMES= if you plan to specify both the PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options in your LIBNAME statement. Using this alias saves you some time when coding.

To use column names in your SAS program that are not valid SAS names, you must use one of the following techniques:

- Use the DQUOTE= option in PROC SQL and then reference your columns using double quotation marks. For example:

```
proc sql dquote=ansi;
   select "Total$Cost" from mydblib.mytable;
```

- Specify the global system option VALIDVARNAME=ANY and use name literals in the SAS language. For example:

```
proc print data=mydblib.mytable;
   format 'Total$Cost'n 22.2;
```

Note that if you are *creating* a table in PROC SQL, you must also include the PRESERVE_COL_NAMES=YES option in your LIBNAME statement. For example:

```
libname mydblib oracle user=testuser password=testpass
        preserve_col_names=yes;
   proc sql dquote=ansi;
      create table mydblib.mytable ("my$column" int);
```

PRESERVE_COL_NAMES= does not apply to the Pass-Through Facility.

See Also

To apply this option to an individual data set, see the data set option "PRESERVE_COL_NAMES= Data Set Option" on page 243

PRESERVE_TAB_NAMES= LIBNAME Option

Preserves spaces, special characters, and case-sensitivity in DBMS table names

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Teradata

Default value: DBMS-specific

Syntax

PRESERVE_TAB_NAMES= NO | YES

Syntax Description

NO

specifies that when you create DBMS tables or refer to an existing table, the table names are derived from SAS member names by using SAS member name normalization. However, the database applies DBMS-specific normalization rules to the SAS member names. Therefore, the table names are created or referenced in the database following the DBMS-specific normalization rules.

When you use SAS to read a list of table names (for example, in the SAS Explorer window), the tables whose names do not conform to the SAS member name normalization rules do not appear in the output. In SAS line mode, the number of tables that do not display from PROC DATASET due to this restriction is noted as a note:

Note: "Due to the PRESERVE_TAB_NAMES=NO LIBNAME option setting, 12 table(s) have not been displayed." △

You will not get this warning when using SAS Explorer.

SAS Explorer displays DBMS table names in capitalized form when PRESERVE_TAB_NAMES=NO. This is now how the tables are represented in the DBMS.

NO is the default for most DBMS interfaces.

YES

specifies that table names are read from and passed to the DBMS with special characters, and the exact, case-sensitive spelling of the name is preserved.

Details

See the topic on naming in the documentation for your SAS/ACCESS interface for additional details.

To use table names in your SAS program that are not valid SAS names, use one of the following techniques:

- Use the PROC SQL option DQUOTE= and place double quotation marks around the table name. The libref must specify PRESERVE_TAB_NAMES=YES. For example:

```
libname mydblib oracle user=testuser password=testpass
        preserve_tab_names=yes;
proc sql dquote=ansi;
   select * from mydblib."my table";
```

- Use name literals in the SAS language. The libref must specify PRESERVE_TAB_NAMES=YES. For example:

```
libname mydblib oracle user=testuser password=testpass preserve_tab_names=yes;
proc print data=mydblib.'my table'n;
run;
```

Specify the alias PRESERVE_NAMES= to save time if you are specifying both PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= in your LIBNAME statement.

Oracle Details: Unless you specify PRESERVE_TAB_NAMES=YES, the table name that you enter for SCHEMA= or for the DBINDEX= data set option is converted to uppercase.

Example

If you use PROC DATASETS to read the table names in an Oracle database that contains three tables, My_Table, MY_TABLE, and MY TABLE. The results differ depending on the setting of PRESERVE_TAB_NAMES.

If the libref specifies PRESERVE_TAB_NAMES=NO, then the PROC DATASETS output is one table name, MY_TABLE. This is the only table name that is in Oracle normalized form (uppercase letters and a valid symbol, the underscore). My_Table is not displayed because it is not in Oracle normalized form, and MY TABLE is not displayed because it is not in SAS member normalized form (the embedded space is a nonstandard SAS character).

If the libref specifies PRESERVE_TAB_NAMES=YES, then the PROC DATASETS output includes all three table names, My_Table, MY_TABLE, and MY TABLE.

See Also

"PRESERVE_COL_NAMES= LIBNAME Option" on page 128

QUALIFIER= LIBNAME Option

Enables you to identify database objects, such as tables and views, using the specified qualifier

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: MySQL, ODBC, OLE DB, Microsoft SQL Server
Default value: none

Syntax

QUALIFIER=<*qualifier-name*>

Details

If this option is omitted, the default is the default DBMS qualifier name, if any. QUALIFIER= can be used for any DBMS that allows three-part identifier names, such as *qualifier.schema.object*.

MySQL Details: The MySQL interface does not support three-part identifier names, so a two-part name is used (such as *qualifier.object*).

Examples

In the following LIBNAME statement, the QUALIFIER= option causes any reference in SAS to mydblib.employee to be interpreted by ODBC as mydept.scott.employee.

```
libname mydblib odbc dsn=myoracle
        password=testpass schema=scott
        qualifier=mydept;
```

In the following example, the QUALIFIER= option causes any reference in SAS to mydblib.employee to be interpreted by OLE DB as pcdivision.raoul.employee.

```
libname mydblib oledb provider=SQLOLEDB
        properties=("user id"=dbajorge "data source"=SQLSERVR)
        schema=raoul qualifier=pcdivision;
proc print data=mydblib.employee;
run;
```

See Also

To apply this option to an individual data set, see the data set option "QUALIFIER= Data Set Option" on page 245.

QUALIFY_ROWS= LIBNAME Option

Uniquely qualifies all member values in a result set

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: OLE DB
Default value: NO

Syntax

QUALIFY_ROWS= YES | NO

Syntax Description

YES
specifies that when the OLE DB interface flattens the result set of an MDX command, the values in each column are uniquely identified using a hierarchical naming scheme.

NO
specifies that when the OLE DB interface flattens the result set of an MDX command, the values in each column are not qualified, which means they might not be unique.

Details

For example, when this option is set to NO, a GEOGRAPHY column might have a value of **`PORTLAND`** for **`Portland, Oregon`**, and the same value of **`PORTLAND`** for **`Portland, Maine`**. When you set this option to YES, the two values might become **`[USA].[Oregon].[Portland]`** and **`[USA].[Maine].[Portland]`**, respectively.

Note: Depending on the size of the result set, QUALIFY_ROWS=YES can have a significant, negative impact on performance, because it forces the OLE DB interface to search through various schemas to gather the information needed to created unique qualified names. △

For more information about MDX commands, see the SAS/ACCESS documentation for OLE DB.

QUERY_TIMEOUT= LIBNAME Option

Specifies the number of seconds of inactivity to wait before canceling a query

Valid in: the SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. Please refer to your DBMS for details.

DBMS support: DB2 UNIX/PC, ODBC, Microsoft SQL Server

Default value: 0

Syntax

QUERY_TIMEOUT=*number-of-seconds*

Details

The default value of 0 indicates that there is no time limit for a query. This option is useful when you are testing a query or if you suspect that a query might contain an endless loop.

See Also

To apply this option to an individual data set, see the data set option “QUERY_TIMEOUT= Data Set Option” on page 246.

QUOTE_CHAR= LIBNAME Option

Specifies which quotation mark character to use when delimiting identifiers

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: ODBC, OLE DB, Microsoft SQL Server
Default value: none

Syntax

QUOTE_CHAR=*character*

Syntax Description

character
is the quotation mark character to use when delimiting identifiers, such as the double quotation mark (").

Details

The provider usually specifies the delimiting character. However, when there is a difference between what the provider allows for this character and what the DBMS allows, the QUOTE_CHAR= option overrides the character returned by the provider.

ODBC Details: This option is mainly for the ODBC interface to Sybase and should be used in conjunction with the DBCONINIT and DBLIBINIT LIBNAME options.

QUOTE_CHAR= overrides the ODBC default because some drivers return a blank for the identifier delimiter even though the DBMS uses a quotation mark (for example, ODBC to Sybase).

Microsoft SQL Server Details: QUOTE_CHAR= overrides the Microsoft SQL Server default.

Examples

If you would like your quotation character to be a single quotation mark, then specify the following:

```
libname x odbc dsn=mydsn pwd=mypassword quote_char="'";
```

If you would like your quotation character to be a double quotation mark, then specify the following:

```
libname x odbc dsn=mydsn pwd=mypassword quote_char='"';
```

QUOTED_IDENTIFIER= LIBNAME Option

Enables you to specify table and column names with embedded spaces and special characters

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: Sybase

Default value: NO

Syntax

QUOTED_IDENTIFIER=YES | NO

Details

This option is used in place of the PRESERVE_TAB_NAMES and PRESERVE_COL_NAMES options, which have no effect on the Sybase interface, due to Sybase's default to case-sensitivity.

READ_ISOLATION_LEVEL= LIBNAME Option

Defines the degree of isolation of the current application process from other concurrently running application processes

Valid in: the SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. Please refer to your DBMS for details.

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

READ_ISOLATION_LEVEL=*DBMS-specific value*

Syntax Description

See the documentation for your SAS/ACCESS interface for the values for your DBMS.

Details

The degree of isolation defines

- the degree to which rows that are read and updated by the current application are available to other concurrently executing applications
- the degree to which update activity of other concurrently executing application processes can affect the current application.

In the interfaces to ODBC and DB2 under UNIX and PC hosts, this option is ignored if READ_LOCK_TYPE= is not set to ROW.

See Also

To apply this option to an individual data set, see the data set option "READ_ISOLATION_LEVEL= Data Set Option" on page 246.

READ_LOCK_TYPE= LIBNAME Option

Specifies how data in a DBMS table is locked during a READ transaction

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

READ_LOCK_TYPE= ROW | PAGE | TABLE | NOLOCK |VIEW

Syntax Description

ROW
locks a row if any of its columns are accessed. If you are using the interface to ODBC or DB2 under UNIX and PC hosts, READ_LOCK_TYPE=ROW indicates that locking is based on the READ_ISOLATION_LEVEL= option. (This value is valid in the DB2 under UNIX and PC hosts, Microsoft SQL Server, ODBC, and Oracle interfaces.)

PAGE
locks a page of data, which is a DBMS-specific number of bytes. (This value is valid in the Sybase interface.)

TABLE
locks the entire DBMS table. If you specify READ_LOCK_TYPE=TABLE, you must also specify CONNECTION= UNIQUE, or you receive an error message. Setting CONNECTION=UNIQUE ensures that your table lock is not lost, for example, due to another table closing and committing rows in the same connection. (This value is valid in the DB2 under z/OS, DB2 under UNIX and PC hosts, ODBC, Oracle, Microsoft SQL Server, and Teradata interfaces.)

NOLOCK
does not lock the DBMS table, pages, or rows during a read transaction. (This value is valid in the Oracle and Sybase interfaces, and in the ODBC and Microsoft SQL Server interfaces when using the Microsoft SQL Server driver.)

VIEW
locks the entire DBMS view. (This value is valid in the Teradata interface.)

Details

If you omit READ_LOCK_TYPE=, the default is the DBMS' default action. You can set a lock for one DBMS table by using the data set option or for a group of DBMS tables by using the LIBNAME option.

See the documentation for your SAS/ACCESS interface for additional details.

Example

In the following example, the libref MYDBLIB uses the SAS/ACCESS interface to Oracle to connect to an Oracle database. USER=, PASSWORD=, and PATH= are

SAS/ACCESS connection options. The LIBNAME options specify that row-level locking is used when data is read or updated:

```
libname mydblib oracle user=testuser password=testpass
        path=myorapth read_lock_type=row update_lock_type=row;
```

See Also

To apply this option to an individual data set, see the data set option "READ_LOCK_TYPE= Data Set Option" on page 247.

READ_MODE_WAIT= LIBNAME Option

Specifies during SAS/ACCESS read operations whether Teradata should wait to acquire a lock or should fail the request when the DBMS resource is already locked by a different user

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: Teradata
Default value: none

Syntax

READ_MODE_WAIT= YES | NO

Syntax Description

YES
specifies for Teradata to wait to acquire the lock, so SAS/ACCESS waits indefinitely until it can acquire the lock.

NO
specifies for Teradata to fail the lock request if the specified DBMS resource is locked.

Details

If you specify READ_MODE_WAIT=NO and if a different user holds a restrictive lock, then the executing SAS step fails. SAS/ACCESS continues processing the job by executing the next step. For more information, see the SAS/ACCESS documentation for Teradata.

If you specify READ_MODE_WAIT=YES, SAS/ACCESS waits indefinitely until it can acquire the lock.

A *restrictive* lock means that another user is holding a lock that prevents you from obtaining your desired lock. Until the other user releases the restrictive lock, you cannot obtain your lock. For example, another user's table level WRITE lock prevents you from obtaining a READ lock on the table.

See Also

To apply this option to an individual data set, see the data set option "READ_MODE_WAIT= Data Set Option" on page 248.

READBUFF= LIBNAME Option

Specifies the number of rows of DBMS data to read into the buffer

Valid in: the SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. Please refer to your DBMS for details.

DBMS support: DB2 UNIX/PC, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase

Default value: DBMS-specific

Syntax

READBUFF=*integer*

Syntax Description

integer
is the positive number of rows to hold in memory. SAS allows the maximum number that is allowed by the DBMS.

Details

This option improves performance by specifying a number of rows that can be held in memory for input into SAS. Buffering data reads can decrease network activities and increase performance. However, because SAS stores the rows in memory, higher values for READBUFF= use more memory. In addition, if too many rows are selected at once, then the rows that are returned to the SAS application might be out of date. For example, if someone else modifies the rows, you do not see the changes.

When READBUFF=1, only one row is retrieved at a time. The higher the value for READBUFF=, the more rows the DBMS engine retrieves in one fetch operation.

DB2 UNIX/PC Details: If you do not specify this option, the buffer size is automatically calculated based upon the row length of your data and the SQLExtendedFetch API call is used (this is the default). ROWSET_SIZE= is an alias for this option.

ODBC, and Microsoft SQL Server Details: If you do not specify this option, the SQLFetch API call is used and no internal SAS buffering is performed (this is the default). Setting READBUFF=1 or greater causes the SQLExtendedFetch API call to be used. ROWSET_SIZE= is an alias for this option.

OLE DB Details: The default is 1. ROWSET_SIZE= is an alias for this option.

Oracle Details: The default is 250.

Sybase Details: The default is 100.

See Also

To apply this option to an individual data set, see the data set option “READBUFF= Data Set Option” on page 249.

REMOTE_DBTYPE= LIBNAME Option

Specifies whether the libref points to a database server on z/OS or to one on Linux, UNIX, or Windows (LUW)

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS

Default value: ZOS

Syntax

REMOTE_DBTYPE= LUW | ZOS

Syntax Description

LUW
specifies that the database server that is accessed through the libref resides on Linux, UNIX, or Windows.

ZOS
specifies that the database server that is accessed through the libref runs on z/OS (default).

Details

Specifying REMOTE_DBTYPE= in the LIBNAME statement ensures that the SQL that is used by some SAS procedures to access the DB2 catalog tables is generated properly, and that it is based upon the database server type.

This option also enables special catalog calls (such as DBMS::Indexes) to function properly when the target database does not reside on a mainframe computer.

REMOTE_DBTYPE= is used in conjunction with the SERVER= CONNECT statement option or the LOCATION= LIBNAME option. If neither option is used, then REMOTE_DBTYPE= is ignored.

Example

The following is an example of using REMOTE_DBTYPE= with the SERVER= option.

```
libname mylib db2 ssid=db2a server=db2_udb remote_dbtype=luw;
proc datasets lib=mylib;

quit;
```

This SAS code, by specifying REMOTE_DBTYPE=LUW, enables the catalog call to work properly for this remote connection.

```
proc sql;
 connect to db2 (ssid=db2a server=db2_udb remote_dbtype=luw);

 select * from connection to db2
 select * from connection to db2
```

```
      (DBMS::PrimaryKeys ("", "JOSMITH", ""));
quit;
```

See Also

For more information about the other options that work in conjunction with REMOTE_DBTYPE=, see the LOCATION= LIBNAME option and the SERVER= CONNECT statement option.

REREAD_EXPOSURE= LIBNAME Option

Specifies whether the SAS/ACCESS engine functions like a random access engine for the scope of the LIBNAME statement

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata
Default value: NO

Syntax

REREAD_EXPOSURE= NO | YES

Syntax Description

NO
specifies that the SAS/ACCESS engine functions as an RMOD engine, which means that your data is protected by the normal data protection that SAS provides.

YES
specifies that the SAS/ACCESS engine functions like a random access engine when rereading a row so that you cannot guarantee that the same row is returned. For example, if you read row 5 and someone else deletes it, then the next time you read row 5, you will read a different row. You have the potential for data integrity exposures within the scope of your SAS session.

Details

CAUTION:
Using REREAD_EXPOSURE= could cause data integrity exposures △

Oracle Details: To avoid data integrity problems, it is advisable to set UPDATE_LOCK_TYPE= TABLE if you set REREAD_EXPOSURE=YES.

ODBC and OLE DB Details: To avoid data integrity problems, it is advisable to set UPDATE_ISOLATION_LEVEL= S (serializable) if you set REREAD_EXPOSURE=YES.

SCHEMA= LIBNAME Option

Enables you to read database objects, such as tables and views, in the specified schema

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

SCHEMA=*schema-name*

Details

If this option is omitted, you connect to the default schema for your DBMS.

The values for SCHEMA= are usually case-sensitive, so use care when you specify this option.

Oracle Details: Specify a schema name to be used when referring to database objects. SAS can access another user's database objects by using a specified schema name. If PRESERVE_TAB_NAMES=NO, SAS converts the SCHEMA= value to uppercase because all values in the Oracle data dictionary are uppercase unless quoted.

Teradata Details: If you omit this option, a libref points to your default Teradata database, which often has the same name as your user name. You can use this option to point to a different database. This option enables you to view or modify a different user's DBMS tables or views if you have the required Teradata privileges. (For example, to read another user's tables, you must have the Teradata privilege SELECT for that user's tables.) The Teradata alias for SCHEMA= is DATABASE=. For more information about changing the default database, see the DATABASE statement in your Teradata documentation.

Examples

In the following LIBNAME statement example, the SCHEMA= option causes any reference in SAS to mydb.employee to be interpreted by DB2 as scott.employee.

```
libname mydb db2 SCHEMA=SCOTT;
```

To access an Oracle object in another schema, use the SCHEMA= option as in the following example. The schema name is typically a person's user name or ID.

```
libname mydblib oracle user=testuser
        password=testpass path='hrdept_002' schema=john;
```

In the following example, the Oracle SCHEDULE table resides in the AIRPORTS schema, and is specified as AIRPORTS.SCHEDULE. To access this table in PROC PRINT and still use the libref (CARGO) in the SAS/ACCESS LIBNAME statement, you specify the schema in the SCHEMA= option. Then you put the *libref.table* in the procedure's DATA statement.

```
libname cargo oracle schema=airports user=testuser password=testpass
        path="myorapath";
proc print data=cargo.schedule;
run;
```

In the Teradata interface example that follows, the user testuser prints the emp table, which is located in the otheruser database.

```
libname mydblib teradata user=testuser pw=testpass schema=otheruser;
proc print data=mydblib.emp;
run;
```

See Also

To apply this option to an individual data set, see the data set option "SCHEMA= Data Set Option" on page 251.

SHOW_SYNONYMS= LIBNAME Option

Specifies whether PROC DATASETS shows only tables and views for the current user (or schema if the SCHEMA= option is specified)

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: Oracle

Default value: NO

Syntax

SHOW_SYNONYMS= <YES | NO>

Syntax Description

YES
specifies that PROC DATASETS shows only the synonyms that represent the tables and views for the current user (or schema if the SCHEMA= option is specified).

NO
specifies that PROC DATASETS shows only tables and views for the current user (or schema if the SCHEMA= option is specified).

Details

Instead of submitting PROC DATASETS, you can click the libref for the SAS Explorer window to get this same information.

SPOOL= LIBNAME Option

Specifies whether SAS creates a utility spool file during read transactions that read data more than once

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: YES

Syntax

SPOOL= YES | NO | DBMS

Syntax Description

YES
specifies that SAS creates a utility spool file into which it writes the rows that are read the first time. For subsequent passes through the data, the rows are read from the utility spool file rather than being re-read from the DBMS table. This guarantees that the row set is the same for every pass through the data.

NO
specifies that the required rows for all passes of the data are read from the DBMS table. No spool file is written. There is no guarantee that the row set is the same for each pass through the data.

DBMS
is valid for Oracle only. The required rows for all passes of the data are read from the DBMS table but additional enforcements are made on the DBMS server side to ensure the row set is the same for every pass through the data. This setting causes the SAS/ACCESS interface to Oracle to satisfy the two-pass requirement by starting a read-only transaction. SPOOL=YES and SPOOL=DBMS have comparable performance results for Oracle; however, SPOOL=DBMS does not use any disk space. When SPOOL is set to DBMS, the CONNECTION option must be set to UNIQUE. If not, an error occurs.

Details

In some cases, SAS processes data in more than one pass through the same set of rows. Spooling is the process of writing rows that have been retrieved during the first pass of a data read to a spool file. In the second pass, rows can be reread without performing I/O to the DBMS a second time. When data must be read more than once, spooling improves performance. Spooling also guarantees that the data remains the same between passes, as most SAS/ACCESS interfaces do not support member-level locking.

Teradata Details: SPOOL=NO requires SAS/ACCESS to issue identical SELECT statements to Teradata twice. Additionally, because the Teradata table can be modified between passes, SPOOL=NO can cause data integrity problems. Use SPOOL=NO with discretion.

MySQL Details: Do not use SPOOL=NO with the MySQL interface.

SQL_FUNCTIONS= LIBNAME Option

Specifies that the functions that match those supported by SAS should be passed to the DBMS

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: DB2 UNIX/PC, ODBC, OLE DB, MySQL
Default value: NONE

Syntax

SQL_FUNCTIONS=ALL

Syntax Description

ALL
specifies that functions that match those that are supported by SAS should be passed to the DBMS.

Details

DB2 UNIX/PC, ODBC Details: When SQL_FUNCTIONS= is set to ALL, only the functions that are supported by the DBMS drivers are passed. Only a fraction of the functions might be available, based on your provider:

DATE

DATEPART

DATETIME

TIME

TIMEPART

TODAY

DAY

HOUR

MINUTE

MONTH

QRT

SECOND

WEEKDAY

YEAR

BYTE

COMPRESS

INDEX

LENGTH

REPEAT

SOUNDEX

SUBSTR

TRANWRD

TRIMN

MOD

OLE DB Details: When SQL_FUNCTIONS= is set to ALL, the following functions are passed to the DBMS, regardless of whether they are supported by the driver:

LOWCASE
UPCASE
ABS
ARCOS
ARSIN
ATAN
CEIL
COS
EXP
FLOOR
LOG
LOG10
SIGN
SIN
SQRT
TAN
DATE
DATEPART
DATETIME
TIME
TIMEPART
TODAY
DAY
HOUR
MINUTE
MONTH
QRT
SECOND
WEEKDAY
YEAR
BYTE
COMPRESS
INDEX
LENGTH
REPEAT
SOUNDEX
SUBSTR

TRANWRD

TRIMN

MOD

Use of this option can cause unexpected results, especially if used for NULL processing and date/time/timestamp handling. For example, the following SAS code executed without SQL_FUNCTIONS= enabled returns the SAS date 15308:

```
proc sql;
   select distinct DATE () from x.test;
quit;
```

However, the same code with SQL_FUNCTIONS=ALL, returns 2001-1-29, which is an ODBC date format. Care should be exercised when using this option.

STRINGDATES= LIBNAME Option

Specifies whether to read date and time values from the database as character strings or as numeric date values

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 UNIX/PC, ODBC, OLE DB, Microsoft SQL Server

Default value: NO

Syntax

STRINGDATES= YES | NO

Syntax Description

YES
specifies that SAS reads date and time values as character strings.

NO
specifies that SAS reads date and time values as numeric date values.

Details

STRINGDATES=NO is used for Version 6 compatibility.

TRACE= LIBNAME Option

Specifies whether to turn on tracing information that is used in debugging

Valid in: the SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. Please refer to your DBMS for details.

DBMS support: ODBC, Microsoft SQL Server

Default value: NO

Syntax

TRACE= YES | NO

Syntax Description

YES
specifies that tracing is turned on, and the DBMS driver manager writes each function call to the trace file that is specified by TRACEFILE=.

NO
specifies that tracing is not turned on.

Details

This option is not supported on UNIX platforms.

See Also

"TRACEFILE= LIBNAME Option" on page 147

TRACEFILE= LIBNAME Option

Specifies the filename to which the DBMS driver manager writes trace information

Valid in: the SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. Please refer to your DBMS for details.

DBMS support: ODBC, Microsoft SQL Server

Default value: none

Syntax

TRACEFILE= *filename* | <'>*path-and-filename*<'>

Details

TRACEFILE= is used only when TRACE=YES. If you specify a filename without a path, the SAS trace file is stored with your data files. If you specify a directory, enclose the fully qualified filename in single quotation marks.

If you do not specify the TRACEFILE= option, output is directed to a default file.

This option is not supported on UNIX platforms.

See Also

"TRACE= LIBNAME Option" on page 146

UPDATE_ISOLATION_LEVEL= LIBNAME Option

Defines the degree of isolation of the current application process from other concurrently running application processes

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

UPDATE_ISOLATION_LEVEL= *DBMS-specific-value*

Syntax Description

The values for this option are DBMS-specific. See the documentation for your SAS/ACCESS interface.

Details

The degree of isolation defines the following

- the degree to which rows that are read and updated by the current application are available to other concurrently executing applications
- the degree to which update activity of other concurrently executing application processes can affect the current application.

In the interfaces to ODBC and DB2 under UNIX and PC hosts, this option is ignored if UPDATE_LOCK_TYPE= is not set to ROW.

See Also

To apply this option to an individual data set, see the data set option "UPDATE_ISOLATION_LEVEL= Data Set Option" on page 256.

UPDATE_LOCK_TYPE= LIBNAME Option

Specifies how data in a DBMS table is locked during an update transaction

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: DB2 z/OS, DB2 UNIX/PC, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

UPDATE_LOCK_TYPE= ROW | PAGE | TABLE | NOLOCK |VIEW

Syntax Description

ROW
locks a row if any of its columns are going to be updated. (This value is valid in the DB2 under UNIX and PC hosts, Microsoft SQL Server, ODBC, OLE DB, and Oracle interfaces.)

PAGE
locks a page of data, which is a DBMS-specific number of bytes. (This value is valid in the Sybase interface.)

TABLE
locks the entire DBMS table. (This value is valid in the DB2 under z/OS, DB2 under UNIX and PC hosts, ODBC, Oracle, Microsoft SQL Server, and Teradata interfaces.)

NOLOCK
does not lock the DBMS table, page, or any rows when reading them for update. (This value is valid in the Microsoft SQL Server, ODBC, Oracle and Sybase interfaces.)

VIEW
locks the entire DBMS view. (This value is valid in the Teradata interface.)

Details

You can set a lock for one DBMS table by using the data set option or for a group of DBMS tables by using the LIBNAME option.

See the documentation for your SAS/ACCESS interface for additional details.

See Also

To apply this option to an individual data set, see the data set option "UPDATE_LOCK_TYPE= Data Set Option" on page 257.

UPDATE_MODE_WAIT= LIBNAME Option

Specifies during SAS/ACCESS update operations whether Teradata should wait to acquire a lock or fail the request when the DBMS resource is locked by a different user

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: Teradata

Default value: none

Syntax

UPDATE_MODE_WAIT= YES | NO

Syntax Description

YES
specifies for Teradata to wait to acquire the lock, so SAS/ACCESS waits indefinitely until it can acquire the lock.

NO
specifies for Teradata to fail the lock request if the specified DBMS resource is locked.

Details

If you specify UPDATE_MODE_WAIT=NO and if a different user holds a restrictive lock, then the executing SAS step fails. SAS/ACCESS continues processing the job by executing the next step.

A *restrictive* lock means that a different user is holding a lock that prevents you from obtaining your desired lock. Until the other user releases the restrictive lock, you cannot obtain your lock. For example, another user's table level WRITE lock prevents you from obtaining a READ lock on the table.

Use SAS/ACCESS locking options only when Teradata's standard locking is undesirable.

For more information, see the documentation for the Teradata interface.

See Also

To apply this option to an individual data set, see the data set option "UPDATE_MODE_WAIT= Data Set Option" on page 258.

UPDATE_MULT_ROWS= LIBNAME Option

Indicates whether to allow SAS to update multiple rows from a data source, such as a DBMS table

Valid in: the SAS/ACCESS LIBNAME statement
DBMS support: ODBC, OLE DB, Microsoft SQL Server
Default value: NO

Syntax

UPDATE_MULT_ROWS= YES | NO

Syntax Description

YES
specifies that SAS/ACCESS processing continues if multiple rows are updated. This might produce unexpected results.

NO
specifies that SAS/ACCESS processing does not continue if multiple rows are updated.

Details

Some providers do not handle the following DBMS SQL statement well, and therefore update more than the current row with this statement:

```
UPDATE ... WHERE CURRENT OF CURSOR
```

UPDATE_MULT_ROWS= enables SAS/ACCESS to continue if multiple rows were updated.

UPDATE_SQL= LIBNAME Option

Determines the method that is used to update and delete rows in a data source

Valid in: SAS/ACCESS LIBNAME statement

DBMS support: ODBC, Microsoft SQL Server

Default value: YES (except for the Oracle drivers from Microsoft and Oracle)

Syntax

UPDATE_SQL=YES | NO

Syntax Description

YES
specifies that SAS/ACCESS uses Current-of-Cursor SQL to update or delete rows in a table.

NO
specifies that SAS/ACCESS uses the SQLSetPos() application programming interface (API) to update or delete rows in a table.

Details

This is the update/delete equivalent of the INSERT_SQL= option. The default for the Oracle drivers from Microsoft and Oracle is NO because these drivers do not support Current-Of-Cursor operations.

See Also

To apply this option to an individual data set, see the data set option "UPDATE_SQL= Data Set Option" on page 259.

UPDATEBUFF= LIBNAME Option

Specifies the number of rows that are processed in a single DBMS update or delete operation

Valid in: the SAS/ACCESS LIBNAME statement

DBMS support: Oracle

Default value: 1

Syntax

UPDATEBUFF=*positive-integer*

Syntax Description

positive-integer
is the number of rows in an operation. SAS allows the maximum that the DBMS allows.

Details

When updating with the VIEWTABLE window or the FSVIEW procedure, use UPDATEBUFF=1 to prevent the DBMS interface from trying to update multiple rows. By default, these features update only observation at a time (since by default they use record-level locking, they lock only the observation that is currently being edited).

See Also

To apply this option to an individual data set, see the data set option "UPDATEBUFF= Data Set Option" on page 260.

USE_ODBC_CL= LIBNAME Option

Indicates whether the Driver Manager uses the ODBC Cursor Library

Valid in: the SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. Please refer to your DBMS for details.

DBMS support: ODBC, Microsoft SQL Server

Default value: NO

Syntax

USE_ODBC_CL= YES | NO

Syntax Description

YES
specifies that the Driver Manager uses the ODBC Cursor Library. The ODBC Cursor Library supports block scrollable cursors and positioned update and delete statements.

NO
specifies that the Driver Manager uses the scrolling capabilities of the driver.

Details

For more information about the ODBC Cursor Library, see your vendor-specific documentation.

UTILCONN_TRANSIENT= LIBNAME Option

Enables utility connections to be maintained or dropped, as needed

Valid in: the SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. Please refer to your DBMS for details.

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DB2 UNIX/PC=NO, Informix=NO, ODBC=NO, MySQL=NO, OLEDB=NO, Oracle=NO, Microsoft SQL Server=NO, Sybase=NO, Teradata=NO; DB2 z/OS=YES

Syntax

UTILCONN_TRANSIENT= NO | YES

Syntax Description

NO
specifies that a utility connection is maintained for the lifetime of the libref.

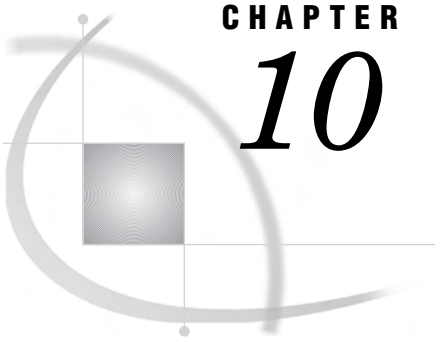
YES
specifies that a utility connection is automatically dropped as soon as it is no longer in use.

Details

For engines that can lock system resources as a result of operations such DELETE or RENAME, or as a result of queries on system tables or table indices, a utility connection is used. The utility connection prevents the COMMIT statements that are issued to unlock system resources from being submitted on the same connection that is being used for table processing. Keeping the COMMIT statements off of the table processing connection alleviates the problems they can cause such as invalidating cursors and committing pending updates on the tables being processed.

Since a utility connection exists for each LIBNAME statement, the number of connection to a DBMS can get large as multiple librefs are assigned across multiple SAS sessions. Setting UTILCONN_TRANSIENT= to YES keeps these connections from existing when they are not being used, thus reducing the number of current connections to the DBMS at any given point in time.

UTILCONN_TRANSIENT= has no effect on engines that do not support utility connections.



CHAPTER

10

Data Set Options for Relational Databases

Overview of Data Set Options for Relational Databases

You can specify SAS/ACCESS data set options on a SAS data set when you access DBMS data with the SAS/ACCESS LIBNAME statement . A data set option applies only to the data set on which it is specified, and it remains in effect for the duration of the DATA step or procedure. (See "LIBNAME Options for Relational Databases" on page 78 for options that can be assigned to a *group* of relational DBMS tables or views.)

For example, SAS/ACCESS data set options can be used as follows:

```
libname myoralib oracle;
proc print myoralib.mytable(data-set-option=value)
```

You can also use SAS/ACCESS data set options on a SAS data set when you access DBMS data using access descriptors, see "Using Descriptors with the ACCESS Procedure" on page 351. For example:

```
proc print mylib.myviewd(data-set-option=value)
```

Most data set options *cannot* be used on a PROC SQL DROP (table or view) statement.

You can use the CNTLLEV=, DROP=, FIRSTOBS=, IN=, KEEP=, OBS=, RENAME=, and WHERE= SAS data set options when you access DBMS data. The REPLACE= SAS data set option is not supported by SAS/ACCESS interfaces. For information about using SAS data set options, refer to the *SAS Language Reference: Dictionary*.

The information in this section explains all the applicable data set option. The information includes DBMS support and the corresponding LIBNAME options, and refers you to the documentation for your SAS/ACCESS interface when appropriate. The documentation for your SAS/ACCESS interface lists the data set options that are available for your DBMS and provides their default values.

Note: Specifying data set options in PROC SQL might reduce performance, because it prevents operations from being passed to the DBMS for processing. For more information, see "Overview of Optimizing Your SQL Usage" on page 37. △

AUTHID= Data Set Option

Enables you to qualify the specified table with an authorization ID, user ID, or group ID

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS

Default value: LIBNAME setting

Syntax

AUTHID=*authorization-ID*

Syntax Description

authorization-ID
is limited to 8 characters.

Details

If you specify a value for the AUTHID= option, the table name is qualified as *authid.tablename* before any SQL code is passed to the DBMS. If AUTHID= is not specified, the table name is not qualified before it is passed to the DBMS, and the DBMS uses your user ID as the qualifier. If you specify AUTHID= in a SAS/SHARE LIBNAME statement, the ID of the active server is the default ID.

SCHEMA= is an alias for this option.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "AUTHID= LIBNAME Option" on page 79.

AUTOCOMMIT= Data Set Option

Specifies whether to enable the DBMS autocommit capability

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: MySQL, Sybase

Default value: LIBNAME setting

Syntax

AUTOCOMMIT=YES | NO

Syntax Description

YES
specifies that all updates, inserts, and deletes are committed immediately after they are executed and no rollback is possible.

NO
specifies that SAS performs the commit after processing the number of row that are specified by using DBCOMMIT=, or the default number of rows if DBCOMMIT= is not specified.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "AUTOCOMMIT= LIBNAME Option" on page 80.

BL_BADFILE= Data Set Option

Identifies a file that contains records that were rejected during a bulk load

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: Oracle

Default value: creates a data file in the current directory or with the default file specifications

Syntax

BL_BADFILE= *path-and-filename*

Syntax Description

path-and-filename
is an SQL*Loader file to which rejected rows of data are written. On most platforms, the default file name takes the form BL_<*table*>_<*unique-ID*>.bad, where

table is the table name

unique-ID is a number that is used to prevent collisions in the event of two or more simultaneous bulk loads of a particular table. The SAS/ACCESS engine generates the number.

Details

If you do not specify this option and a BAD file does not exist, a file is created in the current directory (or with the default file specifications). If you do not specify this option and a BAD file already exists, the Oracle bulk loader reuses the file, replacing the contents with rejected rows from the new load.

Records can be rejected by either the SQL*Loader or by Oracle. For example, the SQL*Loader can reject a record that contains invalid input, and Oracle can reject a

record because it does not contain a unique key. If no records are rejected, the BAD file is not created.

On most operating systems, the BAD file is created in the same format as the DATA file, so the rejected records can be loaded after corrections have been made.

Operating Environment Information: On z/OS operating systems, the BAD file is created with default DCB attributes. For information about how to overcome this, refer to the section about SQL*Loader file attributes in the SQL*Loader chapter in the Oracle user's guide for z/OS. △

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_CODEPAGE= Data Set Option

Identifies the codepage that the DBMS engine uses to convert SAS character data to the current database codepage during a bulk load

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC

Default value: the current window's codepage ID.

Syntax

BL_CODEPAGE= *numeric-codepage-ID*

Syntax Description

numeric-codepage-ID
is a numeric value that represents a character set that is used to interpret multibyte character data and determine the character values.

Details

The value for this option must never be 0. If you do not wish any codepage conversions to take place, use the BL_OPTIONS= option to specify 'FORCEIN'. Codepage conversions only occur for DB2 character data types.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_CONTROL= Data Set Option

Identifies a file containing SQLLDR control statements that describe the data to be included in a bulk load

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: creates a control file in the current directory or with the default file specifications

Syntax

BL_CONTROL= *path-and-control-filename*

Syntax Description

path-and-control-filename
is an SQL*Loader file to which SQLLDR control statements are written. On most platforms, the default file name takes the form BL_<*table*>_<*unique-ID*>.ctl, where

table is the table name

unique-ID is a number used to prevent collisions in the event of two or more simultaneous bulk loads of a particular table. The SAS/ACCESS engine generates the number.

Details

If you do not specify this option and a control file does not exist, a file is created in the current directory (or with the default file specifications). If you do not specify this option and a control file already exists, the interface to Oracle reuses the file, replacing the contents with the new control statements.

The SAS/ACCESS interface for Oracle creates the control file by using information from the input data and SAS/ACCESS options. The file contains Data Definition Language (DDL) definitions that specify the location of the data and how the data corresponds to the database table. It is used to specify exactly how the loader should interpret the data that you are loading from the DATA file (.DAT file).

To specify this option, you must first specify YES for “BULKLOAD= Data Set Option” on page 189.

BL_COPY_LOCATION= Data Set Option

Specifies the directory to which DB2 will save a copy of the loaded data. This option is only valid when used in conjunction with BL_RECOVERABLE=YES

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC

Default value: none

Syntax

BL_COPY_LOCATION=*pathname*

BL_DATAFILE= Data Set Option

Identifies the file that contains the data that will be loaded or appended into a DBMS table during a bulk load

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle, DB2 UNIX/PC

Default value: creates a data file in the current directory or with the default file specifications

Syntax

BL_DATAFILE= *path-and-data-filename*

Syntax Description

path-and-data-filename
is a file that contains the rows of data to be loaded. On most platforms, the default file name takes the form BL_<*table*>_<*unique-ID*>.*ext*, where

table is the table name

unique-ID is a number used to prevent collisions in the event of two or more simultaneous bulk loads of a particular table. The SAS/ACCESS engine generates the number.

ext is the file extension (.DAT or .IXF) for the data file.

Details

The SAS/ACCESS engine creates this data file from the input SAS data set before calling the bulk loader. The data file contains SAS data that is ready to load into the DBMS.

By default, the data file is deleted after the load is completed. To override this behavior, specify BL_DELETE_DATAFILE=NO.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

Oracle Details: If you do not specify this option and a data file does not exist, the default action is to create a file in the current directory (or with the default file specifications). If you do not specify this option and a data file already exists, SAS/ACCESS reuses the file, replacing the contents with the new data. The exception to this is the interface to Oracle on the z/OS platform, where the data file is never reused. The interface to Oracle on z/OS will fail a bulk load rather than reuse a data file.

BL_DB2CURSOR= Data Set Option

Specifies a string that contains a valid DB2 SELECT statement that points to either local or remote objects (tables or views).

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: none

Syntax

BL_DB2CURSOR= 'SELECT * from *filename*'

Details

This option is used only if you specify it. You can use it to select load DB2 tables directly from other DB2 and non-DB2 objects. However, before you can select data from a remote location, your database administrator must first populate the communication database with the appropriate entries.

BL_DB2DEVT_PERM= Data Set Option

Specifies the unit address or generic device type that is used for the permanent data sets created by the LOAD utility, as well as SYSIN, SYSREC, and SYSPRINT when they are allocated by SAS

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: SYSDA

Syntax

BL_DB2DEVT_PERM= *unit-specification*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2DEVT_TEMP= Data Set Option

Specifies the unit address or generic device type that is used for the temporary data sets created by the LOAD utility (PNCH, COPY1, COPY2, RCPY1, RCPY2, WORK1, WORK2)

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: SYSDA

Syntax

BL_DB2DEVT_TEMP= *unit-specification*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2DISC= Data Set Option

Specifies the SYSDISC data set name for the LOAD utility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: a generated data set name

Syntax

BL_DB2DISC= *data-set-name*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2ERR= Data Set Option

Specifies the SYSERR data set name for the LOAD utility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: a generated data set name

Syntax

BL_DB2ERR= *data-set-name*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2IN= Data Set Option

Specifies the SYSIN data set name for the LOAD utility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: a generated data set name

Syntax

BL_DB2IN= *data-set-name*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2LDCT1= Data Set Option

Specifies a string in the LOAD utility control statement, between LOAD DATA and INTO TABLE

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: nothing

Syntax

BL_DB2LDCT1='*string*'

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2LDCT2= Data Set Option

Specifies a string in the LOAD utility control statement, between INTO TABLE *table-name* and (*field-specification*)

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: nothing

Syntax

BL_DB2LDCT2= '*string*'

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2LDEXT= Data Set Option

Specifies the mode of execution for the DB2 LOAD utility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: GENRUN

Syntax

BL_DB2LDEXT= GENRUN | GENONLY | USERUN

Syntax Description

GENRUN
generates the control file (SYSIN) and the data file (SYSREC) and invokes the utility with them.

GENONLY
generates the control file (SYSIN) and the data file (SYSREC) but does not invoke the utility. Use this method when you need to edit the control file or verify the generated control statement or data before you run the utility.

USERUN
uses existing control and data files, and runs the utility with them. The existing files can be from a previous run or from previously run batch utility jobs. Use this execution method when you are restarting an invocation of the utility.

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2MAP= Data Set Option

Specifies the SYSMAP data set name for the LOAD utility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: a generated data set name

Syntax

BL_DB2MAP= *data-set-name*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2PRINT= Data Set Option

Specifies the SYSPRINT data set name for the LOAD utility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: a generated data set name

Syntax

BL_DB2PRINT= *data-set-name*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2PRNLOG= Data Set Option

Determines whether the SYSPRINT output is written to the SAS log

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: YES

Syntax

BL_DB2PRNLOG= YES | NO

Syntax Description

YES
specifies that the SYSPRINT output is written to the SAS log.

NO
specifies that the SYSPRINT output is not written to the SAS log.

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2REC= Data Set Option

Specifies the SYSREC data set name for the LOAD utility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: a generated data set name

Syntax

BL_DB2REC= *data-set-name*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2RECSP= Data Set Option

Determines the number of cylinders to specify as the primary allocation for the SYSREC data set when it is created

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: 10

Syntax

BL_DB2RECSP= *primary-allocation*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2RSTRT= Data Set Option

Tells the LOAD utility whether the current load is a restart and, for a restart, indicates where to begin

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: NO

Syntax

BL_DB2RSTRT= NO | CURRENT | PHASE

Syntax Description

NO
specifies a new invocation of the LOAD utility, not a restart.

CURRENT
specifies to restart at the last commit point.

PHASE
specifies to restart at the beginning of the current phase.

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2SPC_PERM= Data Set Option

Determines the number of cylinders to specify as the primary allocation for the permanent data sets that are created by the LOAD utility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: 10

Syntax

BL_DB2SPC_PERM= *primary-allocation*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2SPC_TEMP= Data Set Option

Determines the number of cylinders to specify as the primary allocation for the temporary data sets that are created by the LOAD utility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS
Default value: 10

Syntax

BL_DB2SPC_TEMP= *primary-allocation*

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2TBLXST= Data Set Option

Indicates whether the LOAD utility runs against an existing table

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
DBMS Support: DB2 z/OS
Default value: NO

Syntax

BL_DB2TBLXST= YES | NO

Syntax Description

YES
specifies that the LOAD utility runs against an existing table. This is *not* a replacement operation. See details below.

NO
specifies that the LOAD utility does not run against an existing table.

Details

For details about this option, see the bulk load topic in the documentation for the interface to DB2 under z/OS.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DB2UTID= Data Set Option

Specifies a unique identifier for a given run of the DB2 LOAD utility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS Support: DB2 z/OS

Default value: user ID and second level DSN qualifier

Syntax

BL_DB2UTID= *utility-ID*

Details

For details about this option, see

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DELETE_DATAFILE= Data Set Option

Deletes the data file that is created for the DBMS bulk-load facility

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle, DB2 UNIX/PC

Default value: YES

Syntax

BL_DELETE_DATAFILE=YES | NO

Syntax Description

YES
deletes the data file that the SAS/ACCESS engine creates for the DBMS bulk-load facility.

NO
saves the data file from deletion.

Details

Setting BL_DELETE_DATAFILE= to YES deletes the temporary data file that is created after the load is completed. Only the data file is deleted.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DIRECT_PATH= Data Set Option

Sets the Oracle SQL*Loader DIRECT option

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: YES

Syntax

BL_DIRECT_PATH= YES | NO

Syntax Description

YES
sets the Oracle SQL*Loader option DIRECT to TRUE, enabling the SQL*Loader to use Direct Path Load to insert rows into a table.

NO
sets the Oracle SQL*Loader option DIRECT to FALSE, enabling the SQL*Loader to use Conventional Path Load to insert rows into a table.

Details

The Conventional Path Load reads in multiple data records and places them in a binary array. When the array is full, it is passed to Oracle for insertion, and Oracle uses the SQL interface with the array option.

The Direct Path Load creates data blocks that are already in the Oracle database block format. The blocks are then written directly into the database. This method is significantly faster, but there are restrictions. For more information about the SQL*Loader Direct and Conventional Path loads, see your Oracle utilities documentation for SQL*Loader.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_DISCARDFILE= Data Set Option

Identifies the file that contains the records that were filtered out of a bulk load because they did not match the criteria specified in the CONTROL file

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: creates a file in the current directory or with the default file specifications

Syntax

BL_DISCARDFILE= *path-and-discard-filename*

Syntax Description

path-and-discard-filename
is an SQL*Loader discard file containing rows that did not meet the specified criteria. On most platforms, the default file name takes the form BL_<*table*>_<*unique-ID*>.dsc, where

table is the table name

unique-ID is a number used to prevent collisions in the event of two or more simultaneous bulk loads of a particular table. The SAS/ACCESS engine generates the number.

Details

SQL*Loader creates the file of discarded rows only if there are discarded rows and if a discard file is requested. If you do not specify this option and a discard file does not exist, a discard file is created in the current directory (or with the default file specifications). If you do not specify this option and a discard file already exists, the Oracle bulk loader reuses the existing file, replacing the contents with discarded rows from the new load.

On most operating systems, the discard file has the same format as the data file, so the discarded records can be loaded after corrections are made.

Operating Environment Information: On z/OS operating systems, the discard file is created with default DCB attributes. For information about how to overcome this, refer to the section about SQL*Loader file attributes in the SQL*Loader chapter in the Oracle user's guide for z/OS. △

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

See Also

See "BL_BADFILE= Data Set Option" on page 157 to set the name and location of the file that contains *rejected* rows.

BL_INDEX_OPTIONS= Data Set Option

Enables you to specify SQL*Loader Index options with bulk loading

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
DBMS support: Oracle
Default value: none

Syntax

BL_INDEX_OPTIONS=*any valid SQL*Loader Index optionsegment-name*

Syntax Description

any valid SQL*Loader Index option

The value specified for this option must be a valid SQL*Loader index option, such as one of the following. Otherwise, an error will occur.

SINGLEROW	Use this option when loading either a direct path with APPEND on systems with limited memory or a small number of records into a large table. It inserts each index entry directly into the index, one record at a time. By default, DQL*Loader does not use this option to append records to a table.
SORTED INDEXES	This clause applies when you are loading a direct path. It tells the SQL*Loader that the incoming data has already been sorted on the specified indexes, allowing SQL*Loader to optimize performance. It allows the SQL*Loader to optimize index creation by eliminating the sort phase for this data when using the direct-path load method.

Details

You can now pass in SQL*Loader index options when bulk loading. For details about these options, refer to the Oracle utilities documentation.

SQLLDR_INDEX_OPTION= is an alias for this option.

Example

The following example shows how you can use this option.

```
proc sql;
connect to oracle  ( user=scott pw=tiger path=alien);
execute ( drop table blidxopts) by oracle;
execute ( create table blidxopts ( empno number, empname varchar2(20))) by
oracle;
execute ( drop index blidxopts_idx) by oracle;
execute ( create index blidxopts_idx on blidxopts ( empno ) ) by oracle;

quit;


libname x oracle user=scott pw=tiger path=alien;

data new;
empno=1; empname='one';
output;
empno=2; empname='two';
output;
run;


proc append base= x.blidxopts( bulkload=yes bl_index_options='sorted indexes
( blidxopts_idx)' ) data= new;
run;
```

BL_INDEXING_MODE= Data Set Option

Used to indicate which scheme the DB2 load utility should use with respect to index maintenance

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC

Default value: AUTOSELECT

Syntax

BL_INDEXING_MODE= AUTOSELECT | REBUILD | INCREMENTAL | DEFERRED

Syntax Description

AUTOSELECT
The load utility automatically decides between REBUILD or INCREMENTAL mode. This is the default.

REBUILD
All indexes are rebuilt.

INCREMENTAL
Indexes are extended with new data

DEFERRED
The load utility does not attempt index creation if this mode is specified. Indexes are marked as needing a refresh.

See Also

For more information about the usage of these values, see the DB2 *Data Movement Utilities Guide and Reference*.

BL_KEEPIDENTITY= Data Set Option

Determines whether the identity column that is created during a bulk load is populated with values generated by Microsoft SQL Server or with values provided by the user

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: OLE DB

Default value: LIBNAME setting

Syntax

BL_KEEPIDENTITY=YES | NO

Syntax Description

YES
specifies that the user must provide values for the identity column.

NO
specifies that the Microsoft SQL Server generates values for an identity column in the table.

Details

This option is only valid when you use the Microsoft SQL Server provider.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "BL_KEEPIDENTITY= LIBNAME Option" on page 81.

BL_KEEPNULLS= Data Set Option

Indicates how NULL values in Microsoft SQL Server columns that accept NULL are handled during a bulk load

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
DBMS support: OLE DB
Default value: LIBNAME setting

Syntax

BL_KEEPNULLS= YES | NO

Syntax Description

YES
preserves NULL values inserted by the OLE DB interface.

NO
replaces NULL values that are inserted by the OLE DB interface with a default value (as specified in the DEFAULT constraint).

Details

This options only affects values in Microsoft SQL Server columns that accept NULL and that have a DEFAULT constraint.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "BL_KEEPNULLS= LIBNAME Option" on page 81.

BL_LOAD_METHOD= Data Set Option

Specifies the method by which data is loaded into an Oracle table during bulk loading

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: INSERT when loading an empty table; APPEND when loading a table that contains data

Syntax

BL_LOAD_METHOD= INSERT | APPEND | REPLACE | TRUNCATE

Syntax Description

INSERT
requires the DBMS table to be empty before loading.

APPEND
appends rows to an existing DBMS table.

REPLACE
deletes all rows in the existing DBMS table and then loads new rows from the data file.

TRUNCATE
uses the *SQL truncate* command to achieve the best possible performance. The DBMS table's referential integrity constraints must first be disabled.

Details

The REPLACE and TRUNCATE values apply only when you are loading data into a table that already contains data. In this case, you can use REPLACE and TRUNCATE to override the default value of APPEND. Refer to your Oracle utilities documentation for information about using the TRUNCATE and REPLACE load methods.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_LOAD_REPLACE= Data Set Option

Specifies whether DB2 will append or replace rows during bulk loading

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC

Default value: NO

Syntax

BL_LOAD_REPLACE= NO | YES

Syntax Description

NO
the CLI LOAD interface appends new rows of data to the DB2 table.

YES
the CLI LOAD interface replaces the existing data in the table.

BL_LOG= Data Set Option

Identifies a log file that will contain information such as statistics and error information for a bulk load

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, Oracle, Teradata

Default value: DBMS-specific

Syntax

BL_LOG= *path-and-log-filename*

Syntax Description

path-and-log-filename
is a file to which information about the loading process is written.

Details

When the DBMS bulk-load facility is invoked, it creates a log file. The contents of the log file are DBMS-specific. The BL_ prefix distinguishes this log file from the one created by the SAS log. If BL_LOG= is specified with the same path and filename as an existing log, the new log replaces the existing log.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189. See the documentation for your SAS/ACCESS interface for additional details.

Oracle Details: When the SQL*Loader is invoked, it creates a log file. This file contains a detailed summary of the load, including a description of any errors. If SQL*Loader cannot create a log file, execution of the bulk load terminates. If a log file does not exist, the default action is to create a log file in the current directory or with the default file specifications. If a log file already exists, the Oracle bulk loader reuses the file, replacing the contents with information from the new load. On most platforms, the default file name takes the form BL_<*table*>_<*unique-ID*>.log, where

table is the table name

unique-ID is a number used to prevent collisions in the event of two or more simultaneous bulk loads of a particular table. The SAS/ACCESS engine generates the number.

DB2 UNIX/PC Details: If BL_LOG= is not specified, the log file is deleted automatically after a successful operation. See for more information.

Teradata Details: See the bulk load topic in the documentation for the Teradata interface for more information.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "BL_LOG= LIBNAME Option" on page 82.

BL_METHOD= Data Set Option

Specifies which bulk loading method to use for DB2

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC

Default value: None

Syntax

BL_METHOD= CLILOAD

Syntax Description

CLILOAD

activates the CLI LOAD interface to the LOAD utility. BULKLOAD=YES must also be specified in order to use the CLI LOAD interface.

BL_OPTIONS= Data Set Option

Passes options to the DBMS bulk-load facility, affecting how it loads and processes data

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, OLE DB, Oracle

Default value: DBMS-specific

Syntax

BL_OPTIONS='<*option…,option*>'

Syntax Description

option
specifies an option from the available options that are specific to each SAS/ACCESS interface. See details below.

Details

BL_OPTIONS= enables you to pass options to the DBMS bulk-load facility when it is invoked, thereby affecting how data is loaded and processed. You must separate multiple options with commas and enclose the entire string of options in single quotation marks.

DB2 under UNIX and PC Hosts Details: This option passes DB2 file type modifiers to DB2 LOAD or IMPORT commands to affect how data is loaded and processed. *Not all DB2 file type modifiers are appropriate for all situations.* You can specify one or more DB2 file type modifiers with .IXF files. For a list of file type modifiers, see the description of the LOAD and IMPORT utilities in the "DB2 Data Movement Utilities Guide and Reference."

OLE DB Details: By default, no options are specified. This option is only valid when you are using the SQL Server provider. This option takes the same values as the *-h* HINT option of the Microsoft BCP utility. For example, the ORDER= option sets the sort order of data in the data file, and can be used to improve performance if the file is sorted according to the clustered index on the table. Refer to the Microsoft SQL Server documentation for a complete list of supported bulk copy options.

Oracle Details: This option enables you to specify the SQL*Loader options ERRORS= and LOAD=. The ERRORS= option specifies the number of insert errors that will terminate the load. The default value of ERRORS=1000000 overrides the default value for the Oracle SQL*Loader ERRORS= option, which is 50. LOAD= specifies the maximum number of logical records to load. If the LOAD= option is not specified, all rows are loaded. Refer to your Oracle utilities documentation for a complete list of SQL*Loader options that you can specify in BL_OPTIONS=.

Example

In the following Oracle example, BL_OPTIONS= specifies the number of errors that are permitted during a load of 2,000 rows of data. Notice that the entire listing of options is enclosed in quotation marks.

```
bl_options='ERRORS=999,LOAD=2000'
```

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "BL_OPTIONS= LIBNAME Option" on page 83.

BL_PARFILE= Data Set Option

Creates a file that contains the SQL*Loader command line options

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle
Default value: none

Syntax

BL_PARFILE=<*parse-file*>

Syntax Description

parse-file
the name you give the file that will contain the SQL*Loader command line options. The name can also specify the path. If no path is specified, the file is created in the current directory.

Details

This option prompts the SQL*Loader to use the PARFILE= option. This SQL*Loader option enables you to specify SQL*Loader command line options in a file instead of as command line options. For example the SQL*Loader can be invoked by specifying user ID and control options as follows:

```
sqlldr userid=scott/tiger control=example.ctl
```

However, you can invoke it using the PARFILE = option as follows:

```
sqlldr parfile=example.par
```

Example.par will now contain the USERID= and CONTROL= options. One of the biggest advantages of using the BL_PARFILE= option is security, since the user ID and password are stored in a separate file.

The permissions on the file default to the operating system defaults. It is advisable to create this file in a protected directory so that the contents of the file are not accessible to unauthorized users.

You can display the contents of the parse file in the SAS log by using the option **SASTRACE=",,,d"**. However, the password will be blocked out and replaced with **xxxx**.

Note: The parse file is deleted at the end of SQL*Loader processing. △

Example

The following example demonstrates how SQL*Loader invocation is different when the BL_PARFILE= option is specified.

```
libname x oracle user=scott pw=tiger;
  /* SQL*Loader is invoked as follows without BL_PARFILE= */
   sqlldr userid=scott/tiger@oraclev9 control=bl_bltst_0.ctl log=bl_bltst_0.log
    bad=bl_bltst_0.bad discard=bl_bltst_0.dsc   */

 data x.bltst ( bulkload=yes);
 c1=1;
  run;
   /* Note in the DATA step below, which uses BL_PARFILE=, how SQL*Loader is invoked */
  sqlldr  parfile=test.par
   /* In this case all the options are written to the test.par file.  */
data x.bltst2 ( bulkload=yes bl_parfile='test.par');
```

```
 c1=1;
run;
```

BL_PRESERVE_BLANKS= Data Set Option

Determines how the SQL*Loader handles requests to insert blank spaces into CHAR/VARCHAR2 columns with the NOT NULL constraint

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: NO

Syntax

BL_PRESERVE_BLANKS=YES | NO

Syntax Description

YES
specifies that blank values are inserted as blank spaces.

CAUTION:
When this option is set to YES, *any trailing blank spaces are also inserted*. For this reason, use this option with caution. It is recommended that you only set this option to YES for CHAR columns. It is not recommended that you set this option to YES for VARCHAR2 columns, because trailing blank spaces are significant in VARCHAR2 columns. △

NO
specifies that blank values are inserted as NULL values.

Details

Operating Environment Information: This option is not supported on z/OS. △

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_RECOVERABLE= Data Set Option

Determines whether the LOAD process is recoverable

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, Oracle

Default value: NO for DB2 UNIX/PC, YES for Oracle

Syntax

BL_RECOVERABLE=YES | NO

Syntax Description

YES
specifies that the LOAD process is recoverable. For DB2, YES also specifies that the copy location for the data should be specified by BL_COPY_LOCATION=.

NO
specifies that the LOAD process is not recoverable. For Oracle, NO adds the UNRECOVERABLE keyword before the LOAD keyword in the control file.

Details for Oracle

To improve direct load performance, specify that this option is set to NO.

CAUTION:
Be aware that an unrecoverable load does not log loaded data into the redo log file and so Media recovery is disabled for the loaded table. For more information about the implications of using the UNRECOVERABLE parameter in Oracle, see your Oracle utilities manual. △

Example

This example for Oracle demonstrates the use of BL_RECOVERABLE= to specify that the load is unrecoverable.

```
data x.recover_no (bulkload=yes bl_recoverable=no); c1=1; run;
```

BL_REMOTE_FILE= Data Set Option

Specifies the base filename and location of DB2 LOAD temporary files

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC

Default value: none

Syntax

BL_REMOTE_FILE=*pathname-and-base-filename*

Syntax Description

pathname-and-base-filename
is the full pathname and base filename to which DB2 appends extensions (such as .log, .msg, and .dat files) to create temporary files during load operations. By default, the base filename takes the form BL_<*table*>_<*unique-ID*>, where

table is the table name

unique-ID is a number used to prevent collisions in the event of two or more simultaneous bulk loads of a particular table. The SAS/ACCESS engine generates the number.

Details

When you specify this option, the DB2 LOAD command is used (instead of the IMPORT command). See the bulk load topic in the SAS/ACCESS documentation for DB2 under UNIX and PC hosts for more information about these commands.

For *pathname*, specify a location on a DB2 server that is accessed exclusively by a single DB2 server instance, and for which the instance owner has read and write permissions. Make sure that each LOAD command is associated with a unique *pathname-and-base-filename* value.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

Note: Do NOT use BL_REMOTE_FILE= unless you have SAS Release 6.1 or later for both the DB2 client and server. Using the LOAD facility with a DB2 client or server prior to Release 6.1 might cause the tablespace to become unusable in the event of a load error. This might affect tables other than the table being loaded. △

BL_SERVER_DATAFILE= Data Set Option

Specifies the name and location of the data file as seen by the DB2 server instance

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC

Default value: same as BL_DATAFILE

Syntax

BL_SERVER_DATAFILE=*path-and-data-filename*

Syntax Description

pathname-and-data-filename
is the fully-qualified pathname and filename of the data file to be loaded, as seen by the DB2 server instance. By default, the base filename takes the form BL_<*table*>_<*unique-ID*>, where

table is the table name

unique-ID is a number used to prevent collisions in the event of two or more simultaneous bulk loads of a particular table. The SAS/ACCESS engine generates the number.

Details

If the path to the data file from the DB2 server instance is different than the path to the data file from the client, you must use BL_SERVER_DATAFILE= to specify the path from the DB2 server.

By enabling the DB2 server instance to directly access the data file specified by BL_DATAFILE=, this option facilitates use of the DB2 LOAD command. See the bulk load topic in the SAS/ACCESS documentation for DB2 under UNIX and PC hosts for more information about the LOAD command.

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189 and specify a value for "BL_REMOTE_FILE= Data Set Option" on page 183.

See Also

To specify the path from the client, see "BL_DATAFILE= Data Set Option" on page 160.

BL_SQLLDR_PATH= Data Set Option

Specifies the location of the SQLLDR executable file

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: SQLLDR

Syntax

BL_SQLLDR_PATH=*pathname*

Syntax Description

pathname
is the full pathname to the SQLLDR executable file so that the SAS/ACCESS interface for Oracle can invoke SQL*Loader.

Details

Normally there is no need to specify this option because the environment is set up to find the Oracle SQL*Loader automatically.

Operating Environment Information: This option is ignored on z/OS. △

To specify this option, you must first specify YES for "BULKLOAD= Data Set Option" on page 189.

BL_SUPPRESS_NULLIF= Data Set Option

Indicates whether to suppress the NULLIF clause for the specified columns when a table is created in order to increase performance

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: NO

Syntax

BL_SUPPRESS_NULLIF=< _ALL_=YES | NO > | (<*column-name-1*=YES | NO > <*column-name-n*=YES | NO >...)

Syntax Description

YES
column-name-1=YES indicates that the NULLIF clause should be suppressed for the specified column in the table.

NO
column-name-1=NO indicates that the NULLIF clause should not be suppressed for the specified column in the table.

ALL
specifies that the YES or NO applies to all columns in the table.

Details

If you specify more than one column name, the names must be separated with spaces.

The BL_SUPPRESS_NULLIF= option processes values from left to right, so if you specify a column name twice, or if you use the _ALL_ value, the last value overrides the first value that is specified for the column.

Example

The following example uses the BL_SUPPRESS_NULLIF= option in the DATA step to suppress the NULLIF clause for the columns C1 and C5 in the table.

```
data x.suppressnullif2_yes (bulkload=yes BL_SUPPRESS_NULLIF=(c1=yes c5=yes));
run;
```

The following example uses the BL_SUPPRESS_NULLIF= option in the DATA step to suppress the NULLIF clause for all of the columns in the table.

```
libname x oracle user=dbitest pw=tiger path=lupin_o9010;
```

```
%let num=1000000;   /* 1 million rows */

data x.testlmn ( bulkload=yes
                  BL_SUPPRESS_NULLIF=( _all_ =yes )
                  rename=(year=yearx) );
  set x.big1mil (obs= &num ) ;
run;
```

BL_WARNING_COUNT= Data Set Option

Specifies the maximum number of row warnings to allow before aborting the load operation

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC

Default value: 2147483646

Syntax

BL_WARNING_COUNT=*warning-count*

Details

Use this option to limit the maximum number of rows that generate warnings. Consult the log file for information about why the rows generated warnings.

To specify this option, you must first specify a value for "BL_REMOTE_FILE= Data Set Option" on page 183 and specify YES for "BULKLOAD= Data Set Option" on page 189.

BUFFERS= Data Set Option

Specifies the number of shared memory buffers to be used for transferring data from SAS to Teradata.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: 2

Syntax

BUFFERS=*number-of-shared-memory-buffers*

Syntax Description

number-of-shared-memory-buffers
a numeric value between 1 and 8 that specifies the number of buffers used for transferring data from SAS to Teradata.

Details

BUFFERS= specifies the number of data buffers to use for transferring data from SAS to Teradata. When using MULTILOAD=, data is transferred from SAS to Teradata using shared memory segments. The default shared memory buffer size is 64K. The default number of shared memory buffers used for the transfer is 2.

Use BUFFERS= to vary the number of buffers for data transfer from 1 to 8. Use the MBUFFSIZE=data set option to vary the size of the shared memory buffers from the size of each data row up to 1MB.

See Also

For more information about specifying the size of shared memory buffers, see MBUFFSIZE=.

BULK_BUFFER= Data Set Option

Specifies the number of bulk rows that the SAS/ACCESS engine can buffer for output

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Sybase

Default value: 100

Syntax

BULK_BUFFER=*numeric-value*

Syntax Description

numeric-value
is the maximum number of rows that are allowed. This value depends on the amount of memory that is available to your system.

Details

This option improves performance by specifying the number of rows that can be held in memory for efficient retrieval from the DBMS. A higher number signifies that more rows can be held in memory and accessed quickly during output operations.

BULKLOAD= Data Set Option

Loads rows of data as one unit

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, ODBC, Oracle, Sybase, Teradata

Default value: NO

Syntax

BULKLOAD=YES | NO

Syntax Description

YES
calls a DBMS-specific bulk-load facility in order to insert or append rows to a DBMS table.

NO
uses the dynamic SAS/ACCESS engine to insert or append data to a DBMS table.

Details

Using BULKLOAD=YES is the fastest way to insert rows into a DBMS table.

See the SAS/ACCESS documentation for your DBMS interface for details.

DB2 z/OS Details: BL_DB2LDUTIL= is an alias for this option.

Sybase Details: When BULKLOAD=NO, insertions are processed and rolled back as expected according to DBCOMMIT= and ERRLIMIT= values. If the ERRLIMIT= value is encountered, all uncommitted rows are rolled back. The commit intervals are determined by the DBCOMMIT= data set option.

When BULKLOAD=YES, the first error encountered causes the remaining rows (including the erroneous row) in the buffer to be rejected. No other errors within the same buffer will be detected, even if the ERRLIMIT= value is greater than one. In addition, all rows prior to the error are committed, even if DBCOMMIT= is set larger than the number of the erroneous row.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "BULKLOAD= LIBNAME Option" on page 83.

CAST= Data Set Option

Specifies whether data conversions should be performed on the Teradata DBMS server or by SAS

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Teradata
Default value: none

Syntax

CAST=YES | NO

Syntax Description

YES
forces data conversions (casting) to be done on the Teradata DBMS server and overrides any data overhead percentage limit.

NO
forces data conversions to be done by SAS and overrides any data overhead percentage limit.

Details

Internally, SAS numbers and dates are floating point values. Teradata has varying formats for numbers, including integers, floating point values, and decimal values. Number conversion must occur when you are reading Teradata numbers that are not floating point (Teradata FLOAT). SAS/ACCESS can use the Teradata CAST= function to cause Teradata to perform numeric conversions. The parallelism of Teradata makes it well suited to perform this work. This is especially true if you are running SAS on z/OS (MVS) where CPU activity can be costly.

CAST= can cause more data to be transferred from Teradata to SAS, as a result of the option forcing the Teradata type into a larger SAS type. For example, the CAST= transfer of a Teradata BYTEINT to SAS floating point adds seven overhead bytes to each row transferred.

The following Teradata types are candidates for casting:

- INTEGER
- BYTEINT
- SMALLINT
- DECIMAL
- DATE.

SAS/ACCESS limits data expansion for CAST= to 20 percent in order to trade rapid data conversion by Teradata for extra data transmission. If casting does not exceed a 20 percent data increase, all candidate columns are cast. If the increase exceeds this limit, then SAS attempts to cast Teradata DECIMAL types only. If casting only DECIMAL types still exceeds the increase limit, data conversions are done by SAS.

You can alter the casting rules by using either CAST= or "CAST_OVERHEAD_MAXPERCENT= LIBNAME Option" on page 86. With CAST_OVERHEAD_MAXPERCENT=, you can change the 20 percent overhead limit. With CAST=, you can override the percentage rules:

- CAST=YES forces Teradata to cast all candidate columns
- CAST=NO cancels all Teradata casting

CAST= only applies when you are reading Teradata tables into SAS. It does not apply when you are writing Teradata tables from SAS.

Also, CAST= only applies to SQL that SAS generates for you. If you supply your own SQL with the explicit SQL feature of PROC SQL, you must code your own casting clauses to force data conversions to occur in Teradata instead of SAS.

See Also

"CAST= LIBNAME Option" on page 84

CAST_OVERHEAD_MAXPERCENT= Data Set Option

Specifies the overhead limit for data conversions to be performed in Teradata instead of SAS

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Teradata

Default value: 20 percent

Syntax

CAST_OVERHEAD_MAXPERCENT=*<n>*

Syntax Description

<n>
Any positive numeric value. The engine default is 20.

Details

Teradata INTEGER, BYTEINT, SMALLINT, and DATE columns require conversion when read in to SAS. Conversions can be performed either by Teradata or by SAS. When performed in Teradata, using Teradata's CAST operator, the row size transmitted to SAS can increase. CAST_OVERHEAD_MAXPERCENT= limits the allowable increase, also called conversion overhead.

Examples

The following example demonstrates the use of CAST_OVERHEAD_MAXPERCENT= to increase the allowable overhead to 40 percent:

```
proc print data=mydblib.emp (cast_overhead_maxpercent=40);
where empno<1000;
run;
```

See Also

"CAST= LIBNAME Option" on page 84 for more information about conversions, conversion overhead, and casting.

COMMAND_TIMEOUT= Data Set Option

Specifies the number of seconds to wait before a command times out

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: OLE DB

Default value: LIBNAME setting

Syntax

COMMAND_TIMEOUT=*number-of-seconds*

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "COMMAND_TIMEOUT= LIBNAME Option" on page 87.

CURSOR_TYPE= Data Set Option

Specifies the cursor type for read only and updatable cursors

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, ODBC, OLE DB, Microsoft SQL Server

Default value: LIBNAME setting

Syntax

CURSOR_TYPE=DYNAMIC | KEYSET_DRIVEN | FORWARD_ONLY | STATIC

Syntax Description

DYNAMIC
specifies that the cursor reflects all of the changes that are made to the rows in a result set as you move the cursor. The data values and the membership of rows in the cursor can change dynamically on each fetch.

KEYSET_DRIVEN
specifies that the cursor determines which rows belong to the result set when the cursor is opened. However, changes that are made to these rows are reflected as you move the cursor.

FORWARD_ONLY
specifies that the cursor works like a DYNAMIC cursor except that it only supports fetching the rows sequentially. (This is not valid in OLE DB.)

STATIC
specifies that the cursor builds the complete result set when the cursor is opened. No changes made to the rows in the result set after the cursor is opened are reflected in the cursor. Static cursors are read-only.

Details

The driver is allowed to modify the default without an error. Not all database drivers support all cursor types. An error is returned if the specified cursor type is not supported.

OLE DB Details: By default, this option is not set and the provider uses a default. See your provider documentation for more information. See OLE DB programmer reference documentation for details about these properties. The OLE DB properties applied to an open row set are as follows:

CURSOR_TYPE=	OLE DB Properties Applied
DYNAMIC	DBPROP_OTHERINSERT=TRUE, DBPROP_OTHERUPDATEDELETE=TRUE
KEYSET_DRIVEN	DBPROP_OTHERINSERT=FALSE, DBPROP_OTHERUPDATEDELETE=TRUE
STATIC	DBPROP_OTHERINSERT=FALSE, DBPROP_OTHERUPDATEDELETE=FALSE

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "CURSOR_TYPE= LIBNAME Option" on page 94.

DBCOMMIT= Data Set Option

Causes an automatic COMMIT (a permanent writing of data to the DBMS) after a specified number of rows have been processed

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: the current LIBNAME setting

Syntax

DBCOMMIT=*n*

Syntax Description

n
is an integer greater than or equal to 0.

Details

DBCOMMIT= affects update, delete, and insert processing. The number of rows processed includes rows that are not processed successfully. When DBCOMMIT=0, a commit is issued only once (after the procedure or DATA step completes).

If the DBCOMMIT= option is explicitly set, SAS/ACCESS fails any update that has a WHERE clause.

Note: If you specify both DBCOMMIT= and ERRLIMIT= , and these options collide during processing, then the COMMIT is issued first and the ROLLBACK is issued second. Because the COMMIT (caused by the DBCOMMIT= option) is issued prior to the ROLLBACK (caused by the ERRLIMIT= option), the DBCOMMIT= option is said to override the ERRLIMIT= option in this situation. △

DB2 under UNIX and PC Hosts Details: When BULKLOAD=YES, the default is 10000.

Teradata Details: The Teradata interface alias for this option is CHECKPOINT. See the FastLoad capability description in the SAS/ACCESS documentation for Teradata for the default behavior of this option.

DBCOMMIT= and ERRLIMIT= are disabled for MultiLoad in order to prevent any conflict with ML_CHECKPOINT=.

Example

In the following example, a commit is issued after every 10 rows are processed:

```
data oracle.dept(dbcommit=10);
   set myoralib.staff;
run;
```

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option “DBCOMMIT= LIBNAME Option” on page 95.

DBCONDITION= Data Set Option

Specifies criteria for subsetting and ordering DBMS data

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: none

Syntax

DBCONDITION="*DBMS-SQL-query-clause*"

Syntax Description

DBMS-SQL-query-clause
is a DBMS-specific SQL query clause, such as WHERE, GROUP BY, HAVING, or ORDER BY.

Details

This option enables you to specify selection criteria in the form of DBMS-specific SQL query clauses, which the SAS/ACCESS engine passes directly to the DBMS for processing. When selection criteria are passed directly to the DBMS for processing, performance is often enhanced. The DBMS checks the criteria for syntax errors when it receives the SQL query.

The DBKEY= and DBINDEX= options are ignored when you use DBCONDITION=.

Example

In the following example, the function that is passed to the DBMS with the DBCONDITION= option causes the DBMS to return to SAS only the rows that satisfy the condition.

```
proc sql;
   create view smithnames as
      select lastname from myoralib.employees
             (dbcondition="where soundex(lastname) = soundex('SMYTHE')" )
              using libname myoralib oracle user=testuser pw=testpass path=dbmssrv;

select lastname from smithnames;
```

DBCREATE_TABLE_OPTS= Data Set Option

Specifies DBMS-specific syntax to be added to the CREATE TABLE statement

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: none

Syntax

DBCREATE_TABLE_OPTS=*'DBMS-SQL-clauses'*

Syntax Description

DBMS-SQL-clauses
are one or more DBMS-specific clauses that can be appended to the end of an SQL CREATE TABLE statement.

Details

This option enables you to add DBMS-specific clauses to the end of the SQL CREATE TABLE statement. The SAS/ACCESS engine passes the SQL CREATE TABLE statement and its clauses to the DBMS, which executes the statement and creates the DBMS table. This option applies only when you are creating a DBMS table by specifying a libref associated with DBMS data.

Example

In the following example, the DB2 table TEMP is created with the value of the DBCREATE_TABLE_OPTS= option appended to the CREATE TABLE statement.

```
libname mydblib db2 user=testuser
        pwd=testpass dsn=sample;

data mydblib.temp (DBCREATE_TABLE_OPTS='PARTITIONING
     KEY (X) USING HASHING');
x=1; output;
x=2; output;
run;
```

Given this data set option, the following DB2 SQL statement is passed by the SAS/ACCESS interface to DB2 in order to create the DB2 table:

```
CREATE TABLE TEMP (X DOUBLE) PARTITIONING
   KEY (X) USING HASHING
```

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "DBCREATE_TABLE_OPTS= LIBNAME Option" on page 98.

DBFORCE= Data Set Option

Specifies whether to force the truncation of data during insert processing

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: NO

Syntax

DBFORCE=YES | NO

Syntax Description

YES
specifies that the rows which contain data values that exceed the length of the DBMS column are inserted, and the data values are truncated to fit the DBMS column length.

NO
specifies that the rows which contain data values that exceed the DBMS column length are not inserted.

Details

This option determines how the SAS/ACCESS engine handles rows that contain data values that exceed the length of the DBMS column.

The SAS data set option FORCE= overrides this option when it is used with PROC APPEND or the PROC SQL UPDATE statement. The PROC SQL UPDATE statement does not provide a warning before truncating the data.

Example

In the following example, two librefs are associated with Oracle databases; the default databases and schemas are used and therefore are not specified. In the DATA step, MYDBLIB.DEPT is created from the Oracle data referenced by MYORALIB.STAFF. The LASTNAME variable is a character variable of length 20 in MYORALIB.STAFF. During the creation of MYDBLIB.DEPT, the LASTNAME variable is stored as a column of type character and length 10 by using DBFORCE=YES.

```
libname myoralib oracle user=tester1 password=tst1;
libname mydblib oracle user=lee password=dataman;

data mydblib.dept(dbtype=(lastname='char(10)')
     dbforce=yes);
   set myoralib.staff;
run;
```

See Also

"DBTYPE= Data Set Option" on page 214

DBGEN_NAME= Data Set Option

Specifies how SAS renames columns automatically when they contain characters that SAS does not allow

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS

Syntax

DBGEN_NAME=DBMS | SAS

Syntax Description

DBMS
specifies that disallowed characters are converted to underscores.

SAS
specifies that DBMS columns that contain disallowed characters are converted into valid SAS variable names, using the format _COL*n*, where *n* is the column number (starting with zero). If a name is converted to a name that already exists, a sequence number is appended to the end of the new name.

Details

SAS retains column names when reading data from DBMS tables, unless a column name contains characters that SAS does not allow, such as $ or @. SAS allows alphanumeric characters and the underscore (_).

This option is intended primarily for National Language Support, notably the conversion of Kanji to English characters because the English characters converted from Kanji are often those that are not allowed in SAS.

Note: The various SAS/ACCESS interfaces each handled name collisions differently in SAS Version 6. Some interfaces appended to the end of the name; other interfaces replaced the last character(s) in the name. Some interfaces used a single sequence number, other interfaces used unique counters. If you specify VALIDVARNAME= V6, name collisions are handled the same as they were in SAS Version 6. △

Examples

If you specify DBGEN_NAME=SAS, a DBMS column named DEPT$AMT is renamed to _COL*n* where *n* is the column number.

If you specify DBGEN_NAME=DBMS,a DBMS column named DEPT$AMT is renamed to DEPT_AMT.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option “DBGEN_NAME= LIBNAME Option” on page 99.

DBINDEX= Data Set Option

Detects and verifies that indexes exist on a DBMS table. If they do exist and are of the correct type, a join query that is passed to the DBMS might improve performance

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS specific

Syntax

DBINDEX=YES | NO | <'>*index-name*<'>

Syntax Description

YES
triggers the SAS/ACCESS engine to search for all indexes on a table and return them to SAS for evaluation. If a usable index is found, then the join WHERE clause is passed to the DBMS for processing. A usable index is expected to have at least the same attributes as the join column.

NO
no automated index search is performed.

index-name
verifies the index name that is specified for the index columns on the DBMS table. This requires the same type of call as when DBINDEX=YES is used.

Details

When processing a join that involves a large DBMS table and a relatively small SAS data set, you might be able to use DBINDEX= to improve performance.

CAUTION:
Improper use of this option can impair performance. See "Using the DBINDEX=, DBKEY=, and MULTI_DATASRC_OPT= Options" on page 43 for detailed information about using this option. △

Queries must be issued to the necessary DBMS control or system tables to extract index information about a specific table or validate the index that you specified.

The DBINDEX= option can be entered as a LIBNAME option, SAS data set option, or as an option with PROC SQL. The order that the engine will process this option is as follows:

1. DATA step or PROC SQL specification.
2. LIBNAME statement specification.

Note: If "DBKEY= Data Set Option" on page 200 is specified, it will take precedence over DBINDEX=. △

Example

The following SAS data set is used in these examples:

```
data s1;
   a=1; y='aaaaa'; output;
   a=2; y='bbbbb'; output;
   a=5; y='ccccc'; output;
run;
```

The following example demonstrates the use of DBINDEX= in the LIBNAME statement:

```
libname mydblib oracle user=myuser password=userpwd dbindex=yes;

proc sql;
select * from s1 aa, x.dbtab bb where aa.a=bb.a;
select * from s1 aa, mydblib.dbtab bb where aa.a=bb.a;
```

The DBINDEX= values for table dbtab are retrieved from the DBMS and compared with the join values. In this case, a match was found so the join is passed down to the DBMS using the index. If the index **a** was not found, the join would take place in SAS.

The following example demonstrates the use of DBINDEX= in the SAS DATA step:

```
data a;
s1;
set x.dbtab(dbindex=yes) key=a;
set mydblib.dbtab(dbindex=yes) key=a;
run;
```

The key is validated against the list from the DBMS. If **a** is an index, then a pass down occurs. Otherwise the join takes place in SAS.

The following example demonstrates the use of DBINDEX= in PROC SQL:

```
proc sql;
select * from s1 aa, x.dbtab(dbindex=yes) bb where aa.a=bb.a;
select * from s1 aa, mylib.dbtab(dbindex=yes) bb where aa.a=bb.a;
/*or*/
select * from s1 aa, x.dbtab(dbindex=a) bb where aa.a=bb.a;
select * from s1 aa, mylib.dbtab(dbindex=a) bb where aa.a=bb.a;
```

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "DBINDEX= LIBNAME Option" on page 100.

DBKEY= Data Set Option

Specifies a key column to optimize DBMS retrieval. Can improve performance when you are processing a join that involves a large DBMS table and a small SAS data set or DBMS table

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: none

Syntax

DBKEY=(<'>*column-1*<'><… <'>*column-n*<'>>)

Syntax Description

column
used by SAS to build an internal WHERE clause to search for matches in the DBMS table based on the key column. For example:

```
select * from sas.a, dbms.b(dbkey=x) where a.x=b.x;
```

In this example, DBKEY= specifies column **x**, which matches the key column designated in the WHERE clause. However, if the DBKEY= column does NOT match the key column in the WHERE clause, then DBKEY= is not used.

Examples

The following example uses DBKEY= with the MODIFY statement in a DATA step:

```
libname invty db2;
data invty.stock;
   set addinv;
   modify invty.stock(dbkey=partno) key=dbkey;
   INSTOCK=instock+nwstock;
   RECDATE=today();
   if _iorc_=0 then replace;
run;
```

To use more than one value for DBKEY=, you must include the second value as a join on the WHERE clause. In the following example, the PROC SQL brings the entire DBMS table into SAS and then proceeds with processing:

```
options sastrace=',,,d' sastraceloc=saslog nostsuffix;

proc sql;
create table work.barbkey as
select keyvalues.empid, employees.hiredate, employees.jobcode
     from mydblib.employees(dbkey=(empid jobcode))
     inner join work.keyvalues on employees.empid = keyvalues.empid;
  quit;
```

Details

When processing a join that involves a large DBMS table and a relatively small SAS data set, you might be able to use DBKEY= to improve performance.

When you specify DBKEY=, it is *strongly* recommended that an index exists for the key column in the underlying DBMS table. Performance can be severely degraded without an index.

CAUTION:
Improper use of this option can decrease performance. See "Using the DBINDEX=, DBKEY=, and MULTI_DATASRC_OPT= Options" on page 43 for detailed information about using this option. △

DBLABEL= Data Set Option

Specifies whether to use SAS variable labels or SAS variable names as the DBMS column names during output processing

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: NO

Syntax

DBLABEL=YES | NO

Syntax Description

YES
specifies that SAS variable *labels* are used as DBMS column names during output processing.

NO
specifies that SAS variable *names* are used as DBMS column names.

Details

This option is valid only for creating DBMS tables.

Example

In the following example, a SAS data set, NEW, is created with one variable C1. This variable is assigned a label of DEPTNUM. In the second DATA step, the MYDBLIB.MYDEPT table is created by using DEPTNUM as the DBMS column name. Setting DBLABEL=YES enables the label to be used as the column name.

```
data new;
   label c1='deptnum';
   c1=001;
run;

data mydblib.mydept(dblabel=yes);
   set new;
run;
```

```
proc print data=mydblib.mydept;
run;
```

DBLINK= Data Set Option

Specifies a link from your default database to another database on the server to which you are connected in the Sybase interface; and specifies a link from your local database to database objects on another server in the Oracle interface

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle, Sybase

Default value: LIBNAME setting

Syntax

DBLINK=*database-link*

Details

This option operates differently in each DBMS.

Oracle Details: A link is a database object that identifies an object that is stored in a remote database. A link contains stored path information and can also contain user name and password information for connecting to the remote database. If you specify a link, SAS uses the link to access remote objects. If you omit DBLINK=, SAS accesses objects in the local database.

Sybase Details: This option enables you to link to another database within the same server to which you are connected. If you omit DBLINK=, SAS can only access objects in your default database.

Example

In this example, SAS sends MYORADB.EMPLOYEES to Oracle as EMPLOYEES@SALES.HQ.ACME.COM.

```
proc print data=myoradb.employees(dblink='sales.hq.acme.com');
run;
```

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "DBLINK= LIBNAME Option" on page 103.

DBMASTER= Data Set Option

Designates which table is the larger table when you are processing a join that involves tables from two different types of databases

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata
Default value: none

Syntax

DBMASTER=*YES*

Syntax Description

YES
designates which of two tables references in a join operation is the larger table.

Example

In the following example, a table from an Oracle database and a table from a DB2 database are joined. DBMASTER= is set to YES to indicate that the Oracle table is the larger table. The DB2 table is the smaller table.

```
libname mydblib oracle user=testuser /*database 1 */
  pw=testpass path='myorapath'

libname mydblib2 db2 user=testuser /*database 2 */
  pw=testpass path='mydb2path';

proc sql;
   select * from mydblib.bigtab(dbmaster=yes), mydblib2.smalltab
bigtab.x=smalltab.x;
```

Details

This option can be used with the MULTI_DATASRC_OPT= option to specify which table reference in a join is the larger table. This can improve performance by eliminating the processing normally performed to determine this information. The specification of this option will be ignored, however, during the processing of any outer joins.

See Also

"MULTI_DATASRC_OPT= LIBNAME Option" on page 125

DBMAX_TEXT= Data Set Option

Determines the length of any very long DBMS character data type that is read into SAS or written from SAS when you are using a SAS/ACCESS engine

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase

Default value: 1024

Syntax

DBMAX_TEXT= *integer*

Syntax Description

integer
is a number between 1 and 32,767.

Details

This option applies to appending and updating rows in an existing table. It does not apply when creating a table.

DBMAX_TEXT= is usually used with a very long DBMS character data type, such as the Sybase TEXT data type or the Oracle LONG RAW data type.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "DBMAX_TEXT= LIBNAME Option" on page 104.

DBNULL= Data Set Option

Indicates whether NULL is a valid value for the specified columns when a table is created

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

DBNULL=<_ALL=YES | NO > | (<*column-name-1*=YES | NO >
<…<*column-name-n*=YES | NO >>)

Syntax Description

ALL
specifies that the YES or NO applies to all columns in the table. (This is valid in the interfaces to Informix, Oracle, Sybase, and Teradata only.)

YES
specifies that the NULL value is valid for the specified columns in the DBMS table.

NO
specifies that the NULL value is not valid for the specified columns in the DBMS table.

Details

This option is valid only for creating DBMS tables. If you specify more than one column name, the names must be separated with spaces.

The DBNULL= option processes values from left to right, so if you specify a column name twice, or if you use the _ALL_ value, the last value overrides the first value that is specified for the column.

Examples

In the following example, by using the DBNULL= option, the EMPID and JOBCODE columns in the new MYDBLIB.MYDEPT2 table are prevented from accepting NULL values. If the EMPLOYEES table contains NULL values in the EMPID or JOBCODE columns, the DATA step fails.

```
data mydblib.mydept2(dbnull=(empid=no jobcode=no));
   set mydblib.employees;
run;
```

In the following example, all columns in the new MYDBLIB.MYDEPT3 table except for the JOBCODE column are prevented from accepting NULL values. If the EMPLOYEES table contains NULL values in any column other than the JOBCODE column, the DATA step fails.

```
data mydblib.mydept3(dbnull=(_ALL_=no jobcode=YES));
   set mydblib.employees;
run;
```

See Also

"NULLCHAR= Data Set Option" on page 236
"NULLCHARVAL= Data Set Option" on page 237

DBNULLKEYS= Data Set Option

Controls the format of the WHERE clause with regard to NULL values when you use the DBKEY= data set option

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, ODBC, OLE DB, Oracle, Microsoft SQL Server
Default value: LIBNAME setting

Syntax

DBNULLKEYS= YES | NO

Details

If there might be NULL values in the transaction table or the master table for the columns that you specify in the DBKEY= option, then use DBNULLKEYS=YES. When you specify DBNULLKEYS=YES and specify a column that is not defined as NOT NULL in the DBKEY= data set option, SAS generates a WHERE clause that can find NULL values. For example, if you specify DBKEY=COLUMN and COLUMN is not defined as NOT NULL, SAS generates a WHERE clause with the following syntax:

```
WHERE ((COLUMN = ?) OR ((COLUMN IS NULL) AND (? IS NULL)))
```

This syntax enables SAS to prepare the statement once and use it for any value (NULL or NOT NULL) in the column. Note that this syntax has the potential to be much less efficient than the shorter form of the WHERE clause (presented below). When you specify DBNULLKEYS=NO or specify a column that is defined as NOT NULL in the DBKEY= option, SAS generates a simple WHERE clause.

If you know that there are no NULL values in the transaction table or the master table for the columns you specify in the DBKEY= option, you can use DBNULLKEYS=NO. If you specify DBNULLKEYS=NO and specify DBKEY=COLUMN, SAS generates a shorter form of the WHERE clause (regardless of whether or not the column specified in DBKEY= is defined as NOT NULL):

```
WHERE (COLUMN = ?)
```

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "DBNULLKEYS= LIBNAME Option" on page 104.

DBPROMPT= Data Set Option

Specifies whether SAS displays a window that prompts you to enter DBMS connection information

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
DBMS support: MySQL, Oracle, Sybase
Default value: NO

Syntax

DBPROMPT=YES | NO

Syntax Description

YES
displays the prompting window.

NO
does not display the prompting window.

Details

This data set option is supported only for view descriptors.

Oracle Details: The interface to Oracle allows you to enter 30 characters each for the USERNAME and PASSWORD and up to 70 characters for the PATH, depending on your platform and terminal type.

Examples

In the following example, connection information is specified in the ACCESS procedure. The DBPROMPT= data set option defaults to NO during the PRINT procedure because it is not specified.

```
proc access dbms=oracle;
   create alib.mydesc.access;
   user=testuser;
   password=testpass;
   table=dept;
   create vlib.myview.view;
   select all;
run;

proc print data=vlib.myview;
run;
```

In the following example, the DBPROMPT window opens during connection to the DBMS. Values that were previously specified during the creation of MYVIEW are pulled into the DBPROMPT window fields. You must edit or accept the connection information in the DBPROMPT window to proceed. The password value appears as a series of asterisks; you can edit it.

```
proc print data=vlib.myview(dbprompt=yes);
run;
```

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option “DBPROMPT= LIBNAME Option” on page 105.

DBSASLABEL= Data Set Option

Specifies how the engine returns column labels

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: COMPAT

Syntax

DBSASLABEL=COMPAT | NONE

Syntax Description

COMPAT
specifies that the labels returned should be compatible with what the application normally receives. In other words, engines exhibit their normal behavior.

NONE
specifies that the engine does not return a column label. The engine will return blanks for the column labels.

Details

By default, the SAS/ACCESS interface for your DBMS generates column labels from the column names, rather than from the real column labels.

This option enables the user to override this default behavior. It is useful in the PROC SQL context where column labels instead of column aliases are used as headers.

Examples

The following example demonstrates how DBSASLABEL= is used to return blank column labels so PROC SQL can use the column aliases as the column headers.

```
proc sql;
     select deptno as Department ID, loc as Location
     from mylib.dept(dbsaslabel=none);
```

Without the DBSASLABEL= option set to NONE, the aliases would be ignored and DEPTNO and LOC would be used as column headers in the result set.

DBSASTYPE= Data Set Option

Specifies data types to override the default SAS data types during input processing

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, MySQL, ODBC, OLE DB, Oracle, Informix, Microsoft SQL Server

Default value: DBMS-specific

Syntax

DBSASTYPE=*(column-name-1=<'>SAS-data-type<'><…column-name-n=<'>SAS-data-type<'>>)*

Syntax Description

column-name
specifies a DBMS column name.

SAS-data-type
specifies a SAS data type. SAS data types include the following: CHAR(*n*), NUMERIC, DATETIME, DATE, TIME. See the documentation for your SAS/ACCESS interface for details.

Details

By default, the SAS/ACCESS interface for your DBMS converts each DBMS data type to a SAS data type during input processing. When you need a different data type, you can use this option to override the default and assign a SAS data type to each specified DBMS column. Some conversions might not be supported; if a conversion is not supported, SAS prints an error to the log.

Examples

In the following example, DBSASTYPE= specifies a data type to use for the column MYCOLUMN when SAS is printing ODBC data. If the data in this DBMS column is stored in a format that SAS does not support, such as SQL_DOUBLE(20), this enables SAS to print the values.

```
proc print data=mylib.mytable
   (dbsastype=(mycolumn='CHAR(20)'));
run;
```

In the following example, the data stored in the DBMS FIBERSIZE column has a data type that provides more precision than what SAS could accurately support, such as DECIMAL(20). If you used only PROC PRINT on the DBMS table, the data might be rounded or displayed as a missing value. Instead, you could use DBSASTYPE= to convert the column to a character field of the length 21. Because the DBMS performs the conversion before the data is brought into SAS, there is no loss of precision.

```
proc print data=mylib.specprod
   (dbsastype=(fibersize='CHAR(21)'));
run;
```

The following example, uses DBSASTYPE= to append one table to another when the data types are not comparable. If the SAS data set has a variable EMPID defined as CHAR(20) and the DBMS table has an EMPID column defined as DECIMAL (20), you can use DBSASTYPE= to make them match:

```
proc append base=dblib.hrdata (dbsastype=(empid='CHAR(20)'))
            data=saslib.personnel;
run;
```

DBSASTYPE= specifies to SAS that the EMPID is defined as a character field of length 20. When a row is inserted from the SAS data set into a DBMS table, the DBMS performs a conversion of the character field to the DBMS data type DECIMAL(20).

DBSLICE= Data Set Option

Specifies user-supplied WHERE clauses to partition a DBMS query for threaded reads

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, Microsoft SQL Server, ODBC, Oracle, Sybase, Teradata

Default value: none

Syntax

DBSLICE=("*WHERE-clause-1*" " *WHERE-clause-2*" < ... "*WHERE-clause-n*">)

DBSLICE=(<*server*=>"*WHERE-clause-1*" <*server*=>" *WHERE-clause-2*" < … <*server*=>"*WHERE-clause-n*">)

Syntax Description

Two syntax diagrams are shown here to highlight the simpler version. In many cases, the first, simpler syntax is sufficient. The optional *server*= form is valid only for ODBC and DB2 under UNIX and PC hosts.

WHERE-clause
The WHERE clauses in the syntax signifies DBMS-valid WHERE clauses that partition the data. The clauses should not cause any omissions or duplications of rows in the results set. For example, if EMPNUM can be null, the following DBSLICE= specification omits rows, creating an *incorrect* result set:

```
DBSLICE=("EMPNUM<1000" "EMPNUM>=1000")
```

A correct form is:

```
DBSLICE=("EMPNUM<1000" "EMPNUM>=1000" "EMPNUM IS NULL")
```

In the following example, DBSLICE= creates an *incorrect* set by duplicating SALES with value zero:

```
DBSLICE=(''SALES<=0 or SALES=NULL'' ''SALES>=0'')
```

server
identifies a particular server node in a DB2 partitioned database or in a Microsoft SQL Server partitioned view. Used for the best possible read performance, this enables your SAS thread to directly connect to the node containing the data partition corresponding to your WHERE clause.

Details

If your table reference is eligible for threaded reads (that is, if it is a read-only LIBNAME table reference), DBSLICE= forces a threaded read to occur, partitioning the table with the WHERE clauses you supply. Use DBSLICE= when SAS is unable to generate threaded reads automatically, or if you can provide better partitioning.

DBSLICE= is appropriate for experienced programmers familiar with the layout of their DBMS tables. A well-tuned DBSLICE= specification will usually outperform SAS automatic partitioning. For example, a well-tuned DBSLICE= specification might better distribute data across threads by taking advantage of a column that SAS/ACCESS cannot use when it automatically generates partitioning WHERE clauses.

DBSLICE= delivers optimal performance for DB2 under UNIX and for Microsoft SQL Server. Conversely, DBSLICE= can degrade performance compared to automatic partitioning. For example, Teradata invokes the FastExport Utility for automatic partitioning. If this is overridden with DBSLICE=, WHERE clauses are generated instead. Even with well planned WHERE clauses, performance is degraded because FastExport is innately faster.

CAUTION:

When using DBSLICE=, you are responsible for data integrity. If your WHERE clauses omit rows from the result set or retrieves the same row on more than one thread, your input DBMS result set is incorrect and your SAS program generates incorrect results. △

Examples

In the following example, DBSLICE= partitions on the column GENDER which can only have the values **m**, **M**, **f**, and **F**. This DBSLICE= clause will not work for all DBMSs due to the use of UPPER and single quotation marks (some DBMSs require double quotation marks around character literals). Two threads are created.

```
proc reg SIMPLE
data=lib.customers(DBSLICE="UPPER(GENDER)='M'" "UPPER(GENDER)='F'"));
var age weight;
where years_active>1;
run;
```

The following example partitions on the non-NULL column CHILDREN, the number of children in a family. Three threads are created.

```
data local;
set lib.families(DBSLICE=("CHILDREN<2" "CHILDREN>2" "CHILDREN=2"));
where religion="P";
run;
```

DBSLICEPARM= Data Set Option

Controls the scope of DBMS threaded reads and the number of DBMS connections

Valid in: DATA and PROC Steps (when accessing DBMS data using SAS/ACCESS software) (also available as a SAS configuration file option, SAS invocation option, global SAS option, and LIBNAME option)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, Microsoft SQL Server, ODBC, Oracle, Sybase, Teradata

Default value: THREADED_APPS,2 (DB2 z/OS, Oracle, and Teradata)
THREADED_APPS,2 *or* 3 (DB2 UNIX/PC, Informix, Microsoft SQL Server, ODBC, and Sybase)

Syntax

DBSLICEPARM=(NONE | THREADED_APPS |ALL <, *max-threads*>)

Syntax Description

Two syntax diagrams are shown here in order to highlight the simpler version. In most cases, the simpler version suffices.

NONE
disables DBMS threaded reads. SAS reads tables on a single DBMS connection, as it did with SAS Version 8 and earlier.

THREADED_APPS
makes fully threaded SAS procedures (threaded applications) eligible for threaded reads.

ALL
makes all read-only librefs eligible for threaded reads. This includes SAS threaded applications, as well as the SAS DATA step and numerous SAS procedures.

max-threads
specifies with a positive integer value the maximum number of connections per table read. A partition or portion of the data is read on each connection. The combined rows across all partitions are the same irrespective of the number of connections. That is, changes to the number of connections do not change the result set. Increasing the number of connections instead redistributes the same result set across more connections.

There are diminishing returns when increasing the number of connections. With each additional connection, more burden is placed on the DBMS, and a smaller percentage of time is saved in SAS. Therefore, you should consult your DBMS-specific documentation for threaded reads before using this parameter.

Details

DBSLICEPARM= can be used in numerous locations, and the usual rules of option precedence apply. A table option has the highest precedence, then a LIBNAME option, and so on. A SAS configuration file option has the lowest precedence because DBSLICEPARM= in any of the other locations overrides that configuration setting.

DBSLICEPARM=ALL and DBSLICEPARM=THREADED_APPS make SAS programs eligible for threaded reads. To see if threaded reads are actually generated, turn on SAS tracing and run a program, as shown in the following example:

```
options sastrace=',,,d' sastraceloc=saslog nostsuffix;
proc print data=lib.dbtable(dbsliceparm=(ALL));
   where dbcol>1000;
run;
```

If you want to directly control the threading behavior, use the DBSLICE= data set option.

For DB2 UNIX/PC, Informix, Microsoft SQL Server, ODBC, and Sybase, the default thread number is dependent on whether an application passes in the number of threads (CPUCOUNT=) and whether the data type of the column selected for the data partitioning purpose is binary.

Examples

The following code demonstrates how to use DBSLICEPARM= in a PC SAS configuration file entry to turn off threaded reads for all SAS users:

```
--dbsliceparm NONE
```

The following code demonstrates how to use DBSLICEPARM= as an z/OS invocation option to turn on threaded reads for read-only references to DBMS tables throughout a SAS job:

```
sas o(dbsliceparm=ALL)
```

The following code demonstrates how to use DBSLICEPARM= as a SAS global option, most likely as one of the first statements in your SAS code, to increase maximum threads to three for SAS threaded apps:

```
option dbsliceparm=(threaded_apps,3);
```

The following code demonstrates how to use DBSLICEPARM= as a LIBNAME option to turn on threaded reads for read-only table references that use this particular libref:

```
libname dblib oracle user=scott password=tiger dbsliceparm=ALL;
```

The following code demonstrates how to use DBSLICEPARM= as a table level option to turn on threaded reads for this particular table, requesting up to four connections:

```
proc reg SIMPLE;
   data=dblib.customers (dbsliceparm=(all,4));
   var age weight;
     where years_active>1;
run;
```

See Also

"DBSLICEPARM= LIBNAME Option" on page 108

DBTYPE= Data Set Option

Specifies a data type to use instead of the default DBMS data type when SAS creates a DBMS table

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata
Default value: DBMS-specific

Syntax

DBTYPE=(*column-name-1*=<'>*DBMS-type*<'>

<*…column-name-n=<’>DBMS-type<’>>*)

Syntax Description

column-name
specifies a DBMS column name.

DBMS-type
specifies a DBMS data type. See the documentation for your SAS/ACCESS interface for the default data types for your DBMS.

Details

By default, the SAS/ACCESS interface for your DBMS converts each SAS data type to a predetermined DBMS data type when outputting data to your DBMS. When you need a different data type, use DBTYPE= to override the default data type chosen by the SAS/ACCESS engine.

Teradata Details: In Teradata, you can use DBTYPE= to specify data attributes for a column. See your Teradata CREATE TABLE documentation for information about the data type attributes that you can specify. If you specify DBNULL=NO for a column, do not also use DBTYPE= to specify NOT NULL for that column. If you do, ’NOT NULL’ is inserted twice in the column definition. This causes Teradata to generate an error message.

Examples

In the following example, DBTYPE= specifies the data types that are used when you create columns in the DBMS table.

```
data mydblib.newdept(dbtype=(deptno='number(10,2)' city='char(25)'));
   set mydblib.dept;
run;
```

The following example creates a new Teradata table, NEWDEPT, specifying the Teradata data types for the DEPTNO and CITY columns.

```
data mydblib.newdept(dbtype=(deptno='byteint' city='char(25)'));
set dept;
run;
```

The following example creates a new Teradata table, NEWEMPLOYEES, and specifies a data type and attributes for the EMPNO column. The example encloses the Teradata type and attribute information in double quotation marks. Single quotation marks conflict with those that are required by the Teradata FORMAT attribute. If you use single quotation marks, SAS returns syntax error messages.

```
data mydblib.newemployees(dbtype= (empno="SMALLINT FORMAT '9(5)'
     CHECK (empno >= 100 AND empno <= 2000)"));
set mydblib.employees;
run;
```

See Also

“DBFORCE= Data Set Option” on page 196

ERRLIMIT= Data Set Option

Specifies the number of errors that are allowed before SAS stops processing and issues a rollback

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: 1

Syntax

ERRLIMIT=*integer*

Syntax Description

integer
is a positive integer that represents the number of errors after which SAS stops processing and issues a rollback.

Details

SAS calls the DBMS to issue a rollback after a specified number of errors occurs during the processing of inserts, deletes, updates, and appends. If ERRLIMIT= is set to 0, SAS processes all rows, regardless of the number of errors that occur. The SAS log displays the total number of rows processed and the number of failed rows, if applicable.

The DBCOMMIT= option overrides the ERRLIMIT= option. If you specify a value for DBCOMMIT= other than zero, then rollbacks affected by the ERRLIMIT= option might not include records that are processed unsuccessfully because they were already committed by DBCOMMIT=.

Note: This option cannot be used from a SAS client session in a SAS/SHARE environment. △

Teradata Details: DBCOMMIT= and ERRLIMIT= are disabled for MultiLoad in order to prevent any conflict with ML_CHECKPOINT=.

Example

In the following example, SAS stops processing and issues a rollback to the DBMS at the occurrence of the tenth error. The MYDBLIB libref was assigned in a prior LIBNAME statement.

```
data mydblib.employee3 (errlimit=10);
   set mydblib.employees;
   where salary > 40000;
run;
```

IGNORE_ READ_ONLY_COLUMNS= Data Set Option

Specifies whether to ignore or include columns whose data types are read-only when generating an SQL statement for inserts or updates

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, ODBC, OLE DB, Microsoft SQL Server

Default value: NO

Syntax

IGNORE_READ_ONLY_COLUMNS=YES | NO

Syntax Description

YES
specifies that the SAS/ACCESS engine ignores columns whose data types are read-only when you are generating insert and update SQL statements

NO
specifies that the SAS/ACCESS engine does not ignore columns whose data types are read-only when you are generating insert and update SQL statements

Details

Several databases include data types that can be read-only, such as Microsoft SQL Server's timestamp data type. Several databases also have properties that allow certain data types to be read-only, such as Microsoft SQL Server's identity property.

When the IGNORE_READ_ONLY_COLUMNS option is set to NO (the default), and a DBMS table contains a column that is read-only, an error is returned indicating that the data could not be modified for that column.

Examples

For the following example, a database that contains the table Products is created with two columns: ID and PRODUCT_NAME. The ID column is defined by a read-only data type and PRODUCT_NAME is a character column.

```
CREATE TABLE products (id int IDENTITY PRIMARY KEY, product_name varchar(40))
```

If you have a SAS data set that contains the name of your products, you can insert the data from the SAS data set into the Products table:

```
data work.products;
   id=1;
   product_name='screwdriver';
   output;
   id=2;
   product_name='hammer';
   output;
```

```
   id=3;
   product_name='saw';
   output;
   id=4;
   product_name='shovel';
   output;
run;
```

With IGNORE_READ_ONLY_COLUMNS=NO (the default), an error is returned by the database because in this example, the ID column can not be updated. However, if you set the option to YES and execute a PROC APPEND, the append succeeds, and the SQL statement that is generated does not contain the ID column. :

```
libname x odbc uid=dbitest pwd=dbigrp1 dsn=lupinss
              ignore_read_only_columns=yes;
options sastrace=',,,d' sastraceloc=saslog nostsuffix;
proc append base=x.PRODUCTS data=work.products;
run;
```

See Also

To apply this option to an individual data set, see the LIBNAME option "IGNORE_READ_ONLY_COLUMNS= LIBNAME Option" on page 116.

IN= Data Set Option

Enables you to specify the database or tablespace in which you want to create a new table

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC

Default value: LIBNAME setting

Syntax

IN=*'database-name.tablespace-name'*|'DATABASE *database-name*'

Syntax Description

database-name.tablespace-name
specifies the names of the database and tablespace, which are separated by a period.

DATABASE *database-name*
specifies only the database name. In this case, you specify the word DATABASE, then a space and the database name. Enclose the entire specification in single quotation marks.

Details

The IN= option is relevant only when you are creating a new table. If you omit this option, the default is to create the table in the default database or tablespace.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "IN= LIBNAME Option" on page 117.

INSERT_SQL= Data Set Option

Determines the method that is used to insert rows into a data source

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: ODBC, OLE DB, Microsoft SQL Server

Default value: LIBNAME setting

Syntax

INSERT_SQL=YES | NO

Syntax Description

YES
specifies that the SAS/ACCESS engine uses the data source's SQL insert method to insert new rows into a table.

NO
specifies that the SAS/ACCESS engine uses an alternate (DBMS-specific) method to add new rows to a table.

Details

Flat file databases (such as dBase, FoxPro, and text files) generally have improved insert performance when INSERT_SQL=NO, but other databases might have inferior insert performance (or might fail) with this setting, so you should experiment to determine the optimal setting for your situation.

ODBC Details: The ODBC default is YES, except for Microsoft Access which has a default of NO. When INSERT_SQL=NO, the SQLSetPos (SQL_ADD) function inserts rows in groups that are the size of the INSERTBUFF= option value. The SQLSetPos (SQL_ADD) function does not work unless it is supported by your ODBC driver.

OLE DB Details: By default, the OLE DB interface attempts to use the most efficient row insertion method for each data source. You can use the INSERT_SQL option to override the default in the event that it is not optimal for your situation. The OLE DB alternate method (used when this option is set to NO) uses the OLE DB IRowsetChange interface.

SQL Server Details: The SQL Server default is YES. When INSERT_SQL=NO, the SQLSetPos (SQL_ADD) function inserts rows in groups that are the size of the

INSERTBUFF= option value. The SQLSetPos (SQL_ADD) function does not work unless it is supported by your ODBC driver.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "INSERT_SQL= LIBNAME Option" on page 118.

INSERTBUFF= Data Set Option

Specifies the number of rows in a single DBMS insert

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
DBMS support: DB2 UNIX/PC, ODBC, OLE DB, Oracle, Microsoft SQL Server, MySQL
Default value: LIBNAME setting

Syntax

INSERTBUFF=*positive-integer*

Syntax Description

positive-integer
specifies the number of rows to insert.

Details

SAS allows the maximum number of rows that is allowed by the DBMS. The optimal value for this option varies with factors such as network type and available memory. You might need to experiment with different values to determine the best value for your site.

When you assign a value that is greater than INSERTBUFF=1, the SAS application notes that indicate the success or failure of the insert operation might be incorrect. This is because these notes represent information for only a single insert, even when multiple inserts are performed.

If the DBCOMMIT= option is specified with a value that is less than the value of INSERTBUFF=, then DBCOMMIT= overrides INSERTBUFF=.

Note: When you insert rows with the VIEWTABLE window or the FSEDIT or FSVIEW procedure, use INSERTBUFF=1 to prevent the DBMS interface from trying to insert multiple rows. These features do not support inserting more than one row at a time. △

Note: Additional driver-specific restrictions might apply. △

DB2 under UNIX and PC Hosts Details: You must specify INSERT_SQL=YES in order to use this option. If one row in the insert buffer fails, all rows in the insert buffer fail.

Microsoft SQL Server Details: You must specify INSERT_SQL=YES in order to use this option.

MySQL Details: The default is 0. Any value greater than 0 turns on the INSERTBUFF= option. The engine then calculates how many rows it can insert at one time, based on the row size. If one row in the insert buffer fails, all rows in the insert buffer might fail, depending on your storage type.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "INSERTBUFF= LIBNAME Option" on page 119.

KEYSET_SIZE= Data Set Option

Specifies the number of rows in the cursor that are keyset driven

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: ODBC, Microsoft SQL Server

Default value: LIBNAME setting

Syntax

KEYSET_SIZE=*number-of-rows*

Syntax Description

number-of-rows
is a positive integer from 0 through the number of rows in the cursor.

Details

This option is valid only when CURSOR_TYPE=KEYSET_DRIVEN.

If KEYSET_SIZE=0, then the entire cursor is keyset driven. If a value greater than 0 is specified for KEYSET_SIZE=, then the value chosen indicates the number of rows, within the cursor, that function as a keyset-driven cursor. When you scroll beyond the bounds that are specified by KEYSET_SIZE=, then the cursor becomes dynamic and new rows might be included in the cursor; this becomes the new keyset and the cursor functions as a keyset-driven cursor again. Whenever the value specified is between 1 and the number of rows in the cursor, the cursor is considered to be a mixed cursor, because part of the cursor functions as a keyset-driven cursor and part of the cursor functions as a dynamic cursor.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "KEYSET_SIZE= LIBNAME Option" on page 120.

LOCATION= Data Set Option

Enables you to further specify exactly where a table resides

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS

Default value: LIBNAME setting

Syntax

LOCATION=*location-name*

Details

The location name maps to the location in the SYSIBM.SYSLOCATIONS catalog in the communication database.

In the DBMS engine, the location is converted to the first level of a three-level table name: *location-name*.AUTHID.TABLE. The connection to the remote DBMS subsystem is done implicitly by the DBMS when it receives a three-level name in an SQL statement.

If you specify LOCATION=, you must also specify the AUTHID= option.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "LOCATION= LIBNAME Option" on page 121.

LOCKTABLE= Data Set Option

Places exclusive or shared locks on tables

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Informix

Default value: LIBNAME setting

Syntax

LOCKTABLE=EXCLUSIVE | SHARE

Syntax Description

EXCLUSIVE

locks a table exclusively, preventing other users from accessing any table that you open in the libref.

SHARE
locks a table in shared mode, allowing other users or processes to read data from the tables, but preventing users from updating data.

Details

You can lock tables only if you are the owner or have been granted the necessary privilege. If you omit LOCKTABLE=, no locking occurs.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "LOCKTABLE= LIBNAME Option" on page 122.

MBUFFSIZE= Data Set Option

Specifies the size of the shared memory buffers to be used for transferring data from SAS to Teradata.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: 64K

Syntax

MBUFFSIZE=*size-of-shared-memory-buffers*

Syntax Description

size-of-shared-memory-buffers
a numeric value (between the size of a row being loaded and 1MB) that specifies the buffer size.

Details

MBUFFSIZE= specifies the size of data buffers used for transferring data from SAS to Teradata. Two data set options are available for tuning the number and size of data buffers used for transferring data from SAS to Teradata.

When using MULTILOAD=, data is transferred from SAS to Teradata using shared memory segments. The default shared memory buffer size is 64K. The default number of shared memory buffers used for the transfer is 2.

Use the MBUFFSIZE= data set option to vary the size of the shared memory buffers from the size of each data row up to 1MB.

Use BUFFERS= to vary the number of buffers for data transfer from 1 to 8.

See Also

See BUFFERS= for information about changing the number of shared memory buffers.

ML_CHECKPOINT= Data Set Option

Specifies the interval between checkpoint operations, in minutes.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: 0

Syntax

ML_CHECKPOINT=*checkpoint-rate*

Syntax Description

checkpoint-rate
a numeric value that specifies the interval between checkpoint operations, in minutes.

Details

ML_CHECKPOINT=0 is the default; no checkpoints are taken if the default is used. If ML_CHECKPOINT= is between 1 and 59 inclusive, checkpoints are taken at the specified intervals, in minutes. If the value of ML_CHECKPOINT= is greater than or equal to 60, then a checkpoint operation occurs after a multiple of the specified rows are loaded.

ML_CHECKPOINT= functions very much like CHECKPOINT in the native Teradata MultiLoad utility, but it differs from the DBCOMMIT= data set option. Note that DBCOMMIT= is disabled for MultiLoad to prevent any conflict.

For more information about using checkpoints for MultiLoad, refer to Teradata's MultiLoad utility documentation.

See Also

For more information about using checkpoints and restarting MultiLoad jobs, see MULTILOAD=.

ML_ERROR1= Data Set Option

Specifies the name of a temporary table that MultiLoad uses to track errors that were generated during the acquisition phase of a bulk-load operation.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: none

Syntax

ML_ERROR1=*temporary-table-name*

Syntax Description

temporary-table-name
specifies the name of a temporary table that MultiLoad uses to track errors that were generated during the acquisition phase of a bulk-load operation.

Details

The Teradata MultiLoad utility uses four different temporary tables when performing the bulk-load operation. MultiLoad uses a log table to track restart information, two error tables to track errors, and a work table to hold data before the insert operation is made.

By default, the SAS/ACCESS MultiLoad facility generates names for these temporary tables:

Restart table	SAS_ML_RS_*randnum*
Acquisition error table	SAS_ML_ET_*randnum*
Application error table	SAS_ML_UT_*randnum*
Work table	SAS_ML_WT_*randnum*

In the table names, *randnum* represents a random number.

The data set option ML_ERROR1 is used to specify the name of a table to be used for storing the errors that were generated during the acquisition phase of the MultiLoad bulk-load operation. Upon restarting the job, ML_ERROR1 is used to specify the name of the table that is used for storing errors generated during a previously aborted MultiLoad job.

For more information about the temporary table names used by MultiLoad, refer to Teradata's documentation on the MultiLoad utility.

Note: ML_ERROR1 is not to be used with data set option ML_LOG=, which provides a common prefix for all of the temporary tables that are used by the Teradata MultiLoad utility. △

See Also

To specify a common prefix for all of the temporary tables that are used by the Teradata MultiLoad utility, see ML_LOG=.

ML_ERROR2= Data Set Option

Specifies the name of a temporary table that MultiLoad uses to track errors that were generated during the application phase of a bulk-load operation.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: none

Syntax

ML_ERROR2=*temporary-table-name*

Syntax Description

temporary-table-name
specifies the name of a temporary table that MultiLoad uses to track errors that were generated during the application phase of a bulk-load operation.

Details

The Teradata MultiLoad utility uses four different temporary tables when performing the bulk-load operation. MultiLoad uses a log table to track restart information, two error tables to track errors, and a work table to hold data before the insert operation is made.

By default, the SAS/ACCESS MultiLoad facility generates names for these temporary tables:

Restart table	SAS_ML_RS_*randnum*
Acquisition error table	SAS_ML_ET_*randnum*
Application error table	SAS_ML_UT_*randnum*
Work table	SAS_ML_WT_*randnum*

In these table names, *randnum* represents a random number.

The data set option ML_ERROR2 is used to specify the name of a table to be used for storing the errors that were generated during the application phase of the MultiLoad bulk-load operation. Upon restarting the job, ML_ERROR2 is used to specify the name of the table that is used for storing errors generated during a previously aborted MultiLoad job.

For more information about the temporary table names used by MultiLoad, refer to Teradata's documentation on the MultiLoad utility.

Note: ML_ERROR2 is not to be used with data set option ML_LOG=, which provides a common prefix for all of the temporary tables that are used by the Teradata MultiLoad utility. △

See Also

To specify a common prefix for all of the temporary tables that are used by the Teradata MultiLoad utility, see ML_LOG=

ML_LOG= Data Set Option

Specifies a prefix for the names of the temporary tables that MultiLoad uses during a bulk-load operation.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: none

Syntax

ML_LOG=*prefix-for-MultiLoad-temporary-tables*

Syntax Description

prefix-for-MultiLoad-temporary-tables
specifies the prefix to use when naming Teradata tables that the Teradata MultiLoad utility uses during a bulk-load operation.

Details

The Teradata MultiLoad utility uses four different temporary tables when performing the bulk-load operation. MultiLoad uses a log table to track restart information, two error tables to track errors, and a work table to hold data before the insert operation is made. By default, the SAS/ACCESS MultiLoad facility generates names for these temporary tables:

Restart table	SAS_ML_RS_*randnum*
Acquisition error table	SAS_ML_ET_*randnum*
Application error table	SAS_ML_UT_*randnum*
Work table	SAS_ML_WT_*randnum*

In these table names, *randnum* represents a random number.

To specify a different name for the tables listed above, use the data set options ML_RESTART=, ML_ERROR1=, ML_ERROR2=, and ML_WORK=, respectively.

The data set option ML_LOG= can be used to specify a prefix for the temporary table names used for by MultiLoad. For example, if **ML_LOG=MY_ERRORS** is used, the following table names are generated:

Restart table	MY_ERRORS_RS
Acquisition error table	MY_ERRORS_ET
Application error table	MY_ERRORS_UT
Work table	MY_ERRORS_WT

ML_LOG= can also be used in a restart step to specify the same prefix that was used by a previous MultiLoad job that used ML_LOG= and failed.

For more information about the temporary table names used by MultiLoad, refer to Teradata's documentation on the MultiLoad utility.

Note: ML_LOG= is not to be used with data set options ML_RESTART= , ML_ERROR1=, ML_ERROR2=, and ML_WORK=, which provide specific names to the temporary files. △

ML_RESTART= Data Set Option

Specifies the name of a temporary table that is used by MultiLoad to track checkpoint information.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: none

Syntax

ML_RESTART=*temporary-table-name*

Syntax Description

temporary-table-name
specifies the name of the temporary table that is used by the Teradata MultiLoad utility to track checkpoint information.

Details

The Teradata MultiLoad utility uses four different temporary tables when performing the bulk-load operation. MultiLoad uses a log table to track restart information, two error tables to track errors, and a work table to hold data before the insert operation is made.

By default, the SAS/ACCESS MultiLoad facility generates names for these temporary tables:

Restart table	SAS_ML_RS_*randnum*
Acquisition error table	SAS_ML_ET_*randnum*
Application error table	SAS_ML_UT_*randnum*
Work table	SAS_ML_WT_*randnum*

In these table names, *randnum* represents a random number.

Use ML_RESTART= to specify the name of a table to store checkpoint information. Upon restart, ML_RESTART= is used to specify the name of the log table that was used for storing checkpoint information in the earlier failed run.

For more information about the temporary table names used by the Teradata MultiLoad utility, refer to Teradata's documentation on the MultiLoad utility.

Note: ML_RESTART= is not to be used with the data set option ML_LOG=, which provides a common prefix for all of the temporary tables that are used by the Teradata MultiLoad utility. △

See Also

To specify a common prefix for all of the temporary tables that are used by the Teradata MultiLoad utility, see ML_LOG=.

ML_WORK= Data Set Option

Specifies the name of a temporary table that MultiLoad uses to store intermediate data.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: none

Syntax

ML_WORK=*temporary-table-name*

Syntax Description

temporary-table-name
specifies the name of a temporary table that MultiLoad uses to store intermediate data that is received by the MultiLoad utility during a bulk-load operation.

Details

The Teradata MultiLoad utility uses four different temporary tables when performing the bulk-load operation. MultiLoad uses a log table to track restart information, two

error tables to track errors, and a work table to hold data before the insert operation is made.

By default, the SAS/ACCESS MultiLoad facility generates names for these temporary tables:

Restart table	SAS_ML_RS_*randnum*
Acquisition error table	SAS_ML_ET_*randnum*
Application error table	SAS_ML_UT_*randnum*
Work table	SAS_ML_WT_*randnum*

In these table names,*randnum* represents a random number.

The data set option ML_WORK= is used to specify the name of the table to be used for storing the intermediate data that is received by the MultiLoad utility during a bulk-load operation. Upon restarting the job, ML_WORK= is used to specify the name of the table that is used for storing intermediate data during a previously aborted MultiLoad job.

For more information about the temporary table names used by MultiLoad, refer to Teradata's documentation on the MultiLoad utility.

Note: ML_WORK= is not to be used with data set option ML_LOG=, which provides a common prefix for all of the temporary tables that are used by the Teradata MultiLoad utility. △

See Also

To specify a common prefix for all of the temporary tables that are used by the Teradata MultiLoad utility, see ML_LOG=.

MULTILOAD= Data Set Option

Specifies whether Teradata insert and append operations should make use of the Teradata MultiLoad utility.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: NO

Syntax

MULTILOAD=YES | NO

Syntax Description

YES
uses the Teradata MultiLoad utility, if available, to load Teradata tables.

NO
sends inserts to Teradata tables one row at a time.

Details

Bulk Loading

The SAS/ACCESS MultiLoad facility provides a bulk-loading method of loading both empty and existing Teradata tables. Unlike FastLoad, MultiLoad can append data to existing tables.

Data Buffers

Two data set options are available for tuning the number and the size of data buffers that are used for transferring data from SAS to Teradata. Data is transferred from SAS to Teradata using shared memory. The default shared memory buffer size is 64K. The default number of shared memory buffers used for the transfer is 2. The BUFFERS= data set option can be used to vary the number of buffers for data transfer from 1 to 8. The MBUFFSIZE= data set option can be used to vary the size of the shared memory buffers from the size of each data row up to 1MB.

Temporary Tables

The Teradata MultiLoad utility uses four different temporary tables when performing the bulk-load operation. MultiLoad uses a log table to track restart information, two error tables to track errors, and a work table to hold data before the insert operation is made.

By default, the SAS/ACCESS MultiLoad facility generates names for these temporary tables:

Restart table	SAS_ML_RS_*randnum*
Acquisition error table	SAS_ML_ET_*randnum*
Application error table	SAS_ML_UT_*randnum*
Work table	SAS_ML_WT_*randnum*

In these table names, *randnum* represents a random number.

To specify a different name for the tables listed above, use the data set options ML_RESTART=, ML_ERROR1=, ML_ERROR2=, and ML_WORK=, respectively.

The data set option ML_LOG= can be used to specify a prefix for the temporary table names used by MultiLoad. For example, if **`ML_LOG=MY_ERRORS`** is used, the following table names are generated:

Restart table	MY_ERRORS_RS
Acquisition error table	MY_ERRORS_ET
Application error table	MY_ERRORS_UT
Work table	MY_ERRORS_WT

The following order is used for naming the temporary tables used by MultiLoad:

1. If ML_LOG= is specified, the specified prefix is used when naming temporary tables for MultiLoad.
2. If ML_LOG= is not specified, then the values specified for ML_ERROR1, ML_ERROR2, ML_WORK, ML_RESTART are used.
3. If none of the table naming options are specified, temporary table names are generated by default.

Note: ML_LOG cannot be used with any of these options: ML_ERROR1, ML_ERROR2, ML_WORK, and ML_RESTART. △

Restarting MultiLoad

The MultiLoad bulk-load operation (or MultiLoad job) works in two phases: the first phase is called an acquisition phase, during which data is transferred from SAS to Teradata work tables. The second phase is called the application phase, during which data is applied to the target table.

If the MultiLoad job fails during the acquisition phase, you can restart the job from the last successful checkpoint. The exact observation from which the MultiLoad job must be restarted is displayed in the SAS log. If the MultiLoad job fails in the application phase (that is, when data is loaded onto the target tables from the worktable) you must restart the MultiLoad job outside of SAS. The MultiLoad restart script is displayed in The SAS log. You can run the generated MultiLoad script outside of SAS to complete the load.

The ML_CHECKPOINT= data set option allows the user to specify the checkpoint rate. Specify a value for ML_CHECKPOINT= if restart capability is desired. If checkpoint tracking is not used and the MultiLoad fails in the acquisition phase, the load needs to be restarted from the beginning. ML_CHECKPOINT=0 is the default; no checkpoints are taken if the default is used.

If ML_CHECKPOINT is between 1 and 59 inclusive, checkpoints are taken at the specified interval in minutes. If ML_CHECKPOINT is greater than or equal to 60, then a checkpoint operation occurs after a multiple of the specified rows are loaded.

Note: ML_CHECKPOINT= functions very much like the Teradata MultiLoad utility's checkpoint, differs from the DBCOMMIT= data set option. △

The following restrictions apply when restarting a failed MultiLoad job:

- The failed MultiLoad job must have specified a checkpoint rate other than zero using the ML_CHECKPOINT= data set option. Otherwise, restarting begins from the first record of the source data.

  Note, however, that checkpoints are relevant only to the acquisition phase of MultiLoad. Even if ML_CHECKPOINT=0 is specified, a checkpoint takes place at the end of the acquisition phase. If the job fails after that (in the application

phase) you must restart the job outside of SAS using the MultiLoad script written to the SAS log.

For example, the following MultiLoad job takes a checkpoint every 1000 records.

```
libname trlib teradata user=testuser pw=XXXXXX server=dbc;

/* Create data to MultiLoad */
data work.testdata;
     do x=1 to 50000;
        output;
     end;
end;

data trlib.mlfloat(MultiLoad=yes ML_CHECKPOINT=1000);
set work.testdata;
run;
```

- You must restart the failed MultiLoad job as an append process since the target table will already exist. It is also necessary to identify the work tables, restart table, and the error tables used in the original job.

  For example, suppose that the DATA step shown above failed with the following error message in the SAS log:

```
ERROR: MultiLoad failed with DBS error 2644 after a checkpoint was
taken for 13000 records.
 Correct error and restart as an append process with dataset options
     ML_RESTART=SAS_ML_RS_1436199780, ML_ERROR1=SAS_ML_ET_1436199780,
     ML_ERROR2=SAS_ML_UT_1436199780, and ML_WORK=SAS_ML_WT_1436199780.
If the first run used FIRSTOBS=n, then use the value (7278+n-1) for FIRSTOBS
in the restart.
     Otherwise use FIRSTOBS=7278.
 Note that sometimes, the FIRSTOBS value used on the restart may be an earlier
position than the last checkpoint because restart is block-oriented and not
record-oriented.
```

  After fixing the error, the job must be restarted as an append process and must specify the same work, error, and restart tables used in the earlier run. A FIRSTOBS= value is used on the source table to specify the record from which to restart.

```
/* Restart a MultiLoad job that failed in the acquisition phase
   after correcting the error */
proc append data=work.testdata(FIRSTOBS=7278)
  base=trmlib.mlfloat(MultiLoad=YES ML_RESTART=SAS_ML_RS_1436199780
    ML_ERROR1=SAS_ML_ET_1436199780 ML_ERROR2=SAS_ML_UT_1436199780
    ML_WORK=SAS_ML_WT_1436199780 ML_CHECKPOINT=1000);
run;
```

- If ML_LOG= is used in the run that failed, then you can specify the same value for ML_LOG= on the restart so that four data set options do not have to be specified to identify the temporary tables used by MultiLoad.

  For example, suppose that the original run used ML_LOG= as follows:

```
data trlib.mlfloat(MultiLoad=yes ML_CHECKPOINT=1000 ML_LOG=MY_ERRORS);
  set work.testdata;
run;
```

Suppose that the DATA step shown above fails with the following error. The restart capability only needs data set option ML_LOG= to identify all the necessary tables.

```
ERROR: MultiLoad failed with DBS error 2644 after a checkpoint
was taken for 13000 records.  Correct error and restart as an append process
with dataset options
     ML_RESTART=SAS_ML_RS_1436199780, ML_ERROR1=SAS_ML_ET_1436199780,
     ML_ERROR2=SAS_ML_UT_1436199780, and ML_WORK=SAS_ML_WT_1436199780.
If the first run used FIRSTOBS=n, then use the value (7278+n-1) for FIRSTOBS
in the restart.
     Otherwise use FIRSTOBS=7278.
 Note that sometimes, the FIRSTOBS value used on the restart may be an earlier
position than the last checkpoint because restart is block-oriented and not
record-oriented.
```

```
proc append data=work.testdata(FIRSTOBS=7278)
   base=trlib.mlfloat(MultiLoad=YES ML_LOG=MY_ERRORS ML_CHECKPOINT=1000);
run;
```

- If the MultiLoad process fails in the application phase, SAS has already transferred all data to be loaded to Teradata. Restarting a MultiLoad job must be performed outside of SAS using the script written in the SAS log. Refer to Teradata's MultiLoad documentation for instructions on how to run MultiLoad scripts. An example of a script written in the SAS log is shown below:

```
=-=-= MultiLoad restart script starts here =-=-=
.LOGTABLE MY_ERRORS_RS;
.LOGON boom/mloaduser,********;
.begin import mload tables "mlfloat" CHECKPOINT 0 WORKTABLES
   MY_ERRORS_WT ERRORTABLES
   MY_ERRORS_ET MY_ERRORS_UT
/*TIFY HIGH EXIT SASMLNE.DLL TEXT '2180*/;
.layout saslayout indicators;
.FIELD "x" *  FLOAT;
.DML Label SASDML;
insert into "mlfloat".*;
.IMPORT INFILE DUMMY
/*SMOD SASMLAM.DLL '2180 2180 2180 */
FORMAT UNFORMAT LAYOUT SASLAYOUT
APPLY SASDML;
.END MLOAD;
.LOGOFF;
=-=-=  MultiLoad restart script ends here  =-=-=
ERROR: MultiLoad failed with DBS error 2644 in the application phase.
  Run the MultiLoad restartscript listed above outside of SAS
  to restart the job.
```

- If the original run used a value for FIRSTOBS= for the source data, use the formula provided in the SAS log error message to calculate the value for FIRSTOBS= upon restart. This is illustrated in the following examples:

```
/* Create data to MultiLoad */
data work.testdata;
     do x=1 to 50000;
        output;
```

```
     end;
run;

libname trlib teradata user=testuser pw=testpass server=boom;

/* Load 40,000 rows to the Teradata table */
data trlib.mlfloat(MultiLoad=yes ML_CHECKPOINT=1000 ML_LOG=MY_ERRORS);
set work.testdata(FIRSTOBS=10001);
run;
```

Suppose that the DATA step shown above failed with this error message:

```
ERROR: MultiLoad failed with DBS error 2644 after a checkpoint
was taken for 13000 records.
Correct the error and restart the load as an append process with
data set option ML_LOG=MY_ERRORS.
If the first run used FIRSTOBS=n, then use the value (7278+n-1)
for FIRSTOBS in the restart.
   Otherwise use FIRSTOBS=7278.
Note that sometimes, the FIRSTOBS value specified on the restart
  may be an earlier position than the last checkpoint
  because MultiLoad restart is block-oriented and not
  record-oriented.
```

The FIRSTOBS for the restart step can be calculated using the formula provided, that is, FIRSTOBS=7278+100001-1=17278. Use FIRSTOBS=17278 on the source data.

```
proc append data=work.testdata(FIRSTOBS=17278)
   base=trlib.mlfloat(MultiLoad=YES ML_LOG=MY_ERRORS ML_CHECKPOINT=1000);
run;
```

Note:

- DBCOMMIT= is disabled for MultiLoad in order to prevent any conflict with ML_CHECKPOINT=.
- ERRLIMIT= is not available for MultiLoad because the number of errors are known only at the end of each load phase.
- For restart to work correctly, the data source must return data in the same order. If the order of data read varies from one run to another and the load job fails in the application phase, delete temporary tables and restart the load as a new process. If the job fails in the application phase, restart the job outside of SAS as usual since the data needed to complete the load has already been transferred.
- The restart capability in MultiLoad is block-oriented, and not record-oriented. This means that while a checkpoint has been taken at, for example, 5000 records, it might be necessary to restart from an earlier record; for example, record 4000. This is because the block of data containing record 5001 might have started at record 4000. The exact record where restart should take place is displayed in the SAS log.

△

Examples

The following example uses MultiLoad to load SAS data to an alternate database. Note that it specifies **`database=mloaduser`** in the LIBNAME statement.

```
libname trlib teradata user=testuser pw=testpass server=dbc database=mloaduser;
/*MultiLoad 20000 observations into alternate database mloaduser */

data trlib.trmload14(DBCREATE_TABLE_OPTS="PRIMARY INDEX(IDNUM)" MultiLoad=yes
   ML_LOG=TRMLOAD14  ML_CHECKPOINT=5000);
 set permdata.BIG1MIL(drop=year obs=20000);
run;
```

This example extracts data from one table using FastExport and loads data into another table using MultiLoad.

```
libname trlib teradata user=testuser pw=testpass server=dbc;

/*  Create data to load  */
data trlib.trodd(DBCREATE_TABLE_OPTS="PRIMARY INDEX(IDNUM)" MultiLoad=yes);
  set permdata.BIG1MIL(drop=year obs=10000);
where mod(IDNUM,2)=1;
run;

/*  FastExport from one table and MultiLoad into another  */
proc append data=trlib.treven(dbsliceparm=all)  base=trlib.trall(MultiLOAD=YES);
run;
```

See Also

- See SLEEP= for information about specifying how long to wait before retrying a logon operation.
- See TENACITY= for information about specifying how many hours to continue to retry a logon operation.

NULLCHAR= Data Set Option

Indicates how missing SAS character values are handled during insert, update, DBINDEX=, and DBKEY= processing

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: SAS

Syntax

NULLCHAR= SAS | YES | NO

Syntax Description

SAS

indicates that missing character values in SAS data sets are treated as NULL values if the DBMS allows NULLs. Otherwise, they are treated as the NULLCHARVAL= value.

YES
indicates that missing character values in SAS data sets are treated as NULL values if the DBMS allows NULLs. Otherwise, an error is returned.

NO
indicates that missing character values in SAS data sets are treated as the NULLCHARVAL= value (regardless of whether the DBMS allows NULLs for the column).

Details

This option affects insert and update processing and also applies when you use the DBINDEX= and DBKEY= options.

This option works in conjunction with the NULLCHARVAL= data set option, which determines what is inserted when NULL values are not allowed.

All missing SAS numeric values (represented in SAS as '.') are treated by the DBMS as NULLs.

Oracle Details: See the topic on bulk loading in the documentation for the interface to Oracle for interactions between NULLCHAR and BULKLOAD=.

See Also

"NULLCHARVAL= Data Set Option" on page 237
"DBNULL= Data Set Option" on page 205

NULLCHARVAL= Data Set Option

Defines the character string that replaces missing SAS character values during insert, update, DBINDEX=, and DBKEY= processing

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: a blank character

Syntax

NULLCHARVAL=*'character-string'*

Details

This option affects insert and update processing and also applies when you use the DBINDEX= and DBKEY= options.

This option works with the NULLCHAR= option, which determines whether or not a missing SAS character value is treated as a NULL value.

If NULLCHARVAL= is longer than the maximum column width, one of the following occurs:

- The string is truncated if DBFORCE=YES.
- The operation fails if DBFORCE=NO.

See Also

"NULLCHAR= Data Set Option" on page 236
"DBFORCE= Data Set Option" on page 196
"DBNULL= Data Set Option" on page 205

OR_PARTITION= Data Set Option

Allows reading, updating, and deleting from a particular partition in a partitioned table, also inserting and bulk loading into a particular partition in a partitioned table

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: none

Syntax

OR_PARTITION =*name of a partition in a partitioned Oracle table*

Syntax Description

name of a partition in a partitioned Oracle table

The name of the partition must be valid or an error will occur.

Details

Use this option in cases where you are working with only one particular partition at a time in a partitioned table. Specifying this option will boost performance because you are limiting your access to only one partition of a table instead of the entire table.

This option is appropriate when reading, updating, and deleting from a partitioned table, also when inserting into a partitioned table or bulk loading to a table. You can use it to boost performance.

Example

The following example shows one way you can use this option.

```
libname x oracle user=scott pw=tiger path=oraclev9;
```

```
proc delete data=x.orparttest; run;
data x.ORparttest ( dbtype=(NUM='int')
  DBCREATE_TABLE_OPTS='partition by range (NUM)
              (partition p1 values less than (11),
               partition p2 values less than (21),
               partition p3 values less than (31),
               partition p4 values less than (41),
               partition p5 values less than (51),
               partition p6 values less than (61),
               partition p7 values less than (71),
               partition p8 values less than (81)
              )' );
 do i=1 to 80;
    NUM=i;
 output;
 end;
 run;


options sastrace=",,t,d" sastraceloc=saslog nostsuffix;

/* input */
proc print data=x.orparttest ( or_partition=p4 );
run;


/* update */
proc sql;

/* update should fail with 14402, 00000, "updating partition key column would
cause a partition change"
// *Cause:  An UPDATE statement attempted to change the value of a partition
//          key column causing migration of the row to another partition
// *Action: Do not attempt to update a partition key column or make sure that
//          the new partition key is within the range containing the old
//          partition key.
*/
update x.orparttest ( or_partition=p4 ) set num=100;

update x.orparttest ( or_partition=p4 ) set num=35;


select * from x.orparttest ( or_partition=p4 );
select * from x.orparttest ( or_partition=p8 );


/* delete */
delete from x.orparttest ( or_partition=p4 );

select * from x.orparttest;
quit;


/* load to an existing table */
```

```
data new; do i=31 to 39; num=i; output;end;
run;
data new2; do i=1 to 9; num=i; output;end;
run;


proc append base= x.orparttest ( or_partition=p4 )   data= new;
run;

/* insert should fail 14401, 00000, "inserted partition key is outside
specified partition"
// *Cause:  the concatenated partition key of an inserted record is outside
//          the ranges of the two concatenated partition bound lists that
//          delimit the partition named in the INSERT statement
// *Action: do not insert the key or insert it in another partition
*/
proc append base= x.orparttest ( or_partition=p4 )   data= new2;
run;


/* load to an existing table */
proc append base= x.orparttest ( or_partition=p4 bulkload=yes
bl_load_method=truncate )   data= new;
run;

/* insert should fail 14401   */
proc append base= x.orparttest ( or_partition=p4 bulkload=yes
bl_load_method=truncate )   data= new2;
run;
```

Below are a series of sample scenarios that illustrate how you can use this option. The first shows how to create the ORPARTTEST table, on which all remaining example depend.

```
libname x oracle user=scott pw=tiger path=oraclev9;
proc delete data=x.orparttest; run;
data x.ORparttest ( dbtype=(NUM='int')
  DBCREATE_TABLE_OPTS='partition by range (NUM)
              (partition p1 values less than (11),
               partition p2 values less than (21),
               partition p3 values less than (31),
               partition p4 values less than (41),
               partition p5 values less than (51),
               partition p6 values less than (61),
               partition p7 values less than (71),
               partition p8 values less than (81)
              )' );
 do i=1 to 80;
    NUM=i; output;
 end;
 run;
```

Only the P4 partition is read in the following example.

```
proc print data=x.orparttest ( or_partition-p4 );
run;
```

Rows that belong to only the single P4 partition are updated in the following example.

```
proc sql;
update x.orparttest ( or_partition=p4 ) set num=35;
quit;
```

The above example also illustrates how a particular partition can be updated. However, updates and even inserts to the partition key column are done in such a way that it must be migrated to a different partition in the table. Therefore, the following example fails because the value 100 does not belong to the P4 partition.

```
proc sql;
update x.orparttest ( or_partition=p4 ) set num=100;
quit;
```

All rows in the P4 partition are deleted in the following example.

```
proc sql;
delete from x.orparttest ( or_partition=p4 );
quit;
```

In this next example, rows are added to the P4 partition in the table.

```
data new;
   do i=31 to 39; num=i; output;end;
run;
proc append base= x.orparttest ( or_partition=p4 );
   data= new;
run;
```

The following example also adds rows to the P4 partition but uses the SQL*Loader instead.

```
proc append base= x.orparttest ( or_partition=p4 bulkload=yes );
   data= new;
run;
```

OR_UPD_NOWHERE= Data Set Option

Specifies whether SAS uses an extra WHERE clause when updating rows with no locking

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: LIBNAME setting

Syntax

OR_UPD_NOWHERE =YES | NO

Syntax Description

YES
SAS does not use an additional WHERE clause to determine whether each row has changed since it was read. Instead, SAS uses the SERIALIZABLE isolation level (available with Oracle 7.3 and later) for update locking. If a row changes after the serializable transaction starts, the update on that row fails.

NO
SAS uses an additional WHERE clause to determine whether each row has changed since it was read. If a row has changed since being read, the update fails.

Details

Use this option when you are updating rows without locking (UPDATE_LOCK_TYPE=NOLOCK).

By default (OR_UPD_NOWHERE=YES), updates are performed in serializable transactions. This enables you to avoid extra WHERE clause processing and potential WHERE clause floating point precision problems.

Specify OR_UPD_NOWHERE=NO for compatibility when you are updating a SAS Version 6 view descriptor.

ORACLE_73_OR_ABOVE= is an alias for this option.

Note: Due to the published Oracle bug 440366, sometimes an update on a row fails even if the row has not changed. Oracle offers the following solution: When creating a table, increase the number of INITRANS to at least 3 for the table. △

Example

In the following example, you create a small Oracle table called TEST and then update the TEST table once using the default setting (OR_UPD_NOWHERE=YES) and once specifying OR_UPD_NOWHERE=NO.

```
libname oralib oracle user=testuser pw=testpass update_lock_type=no;

data oralib.test;
c1=1;
c2=2;
c3=3;
run;

options sastrace=",,,d" sastraceloc=saslog;

proc sql;
   update oralib.test set c2=22;
   update oralib.test(or_upd_nowhere=no) set c2=222;
quit;
```

This code uses the SASTRACELOC= and SASTRACE= options to send the output to the SAS log.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option “OR_UPD_NOWHERE= LIBNAME Option” on page 126.

ORHINTS= Data Set Option

Specifies Oracle hints to pass to Oracle from a SAS statement or SQL procedure

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: none

Syntax

ORHINTS =*'Oracle-hint'*

Syntax Description

Oracle-hint
specifies an Oracle hint for SAS/ACCESS to pass to the DBMS as part of an SQL query.

Details

If you specify an Oracle hint, SAS passes the hint to Oracle. If you omit ORHINTS=, SAS does not send any hints to Oracle.

Examples

The following example runs a SAS procedure on DBMS data and SAS converts the procedure to one or more SQL queries. ORHINTS= enables you to specify an Oracle hint for SAS to pass as part of the SQL query.

```
libname mydblib oracle user=testuser password=testpass path='myorapath';

proc print data=mydblib.payroll(orhints='/*+ ALL_ROWS */');
run;
```

In the following example, SAS sends the Oracle hint **'/*+ ALL_ROWS */'** to Oracle as part of the following statement:

```
SELECT /*+ ALL_ROWS */ * FROM PAYROLL
```

PRESERVE_COL_NAMES= Data Set Option

Preserves spaces, special characters, and case sensitivity in DBMS column names when you create DBMS tables.

Valid in: DATA and PROC steps (when creating DBMS tables using SAS/ACCESS software).

DBMS support: DB2 z/OS, DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Teradata

Default value: LIBNAME setting

Syntax

PRESERVE_COL_NAMES=NO | YES

Syntax Description

NO
specifies that column names that are used in DBMS table creation are derived from SAS variable names by using the SAS variable name normalization rules (see the VALIDVARNAME = system option for more information). However, the database applies its DBMS-specific normalization rules to the SAS variable names when it creates the DBMS column names.

The use of name literals to create column names that use database keywords or special symbols other than the underscore character might be illegal when DBMS normalization rules are applied. To include nonstandard SAS symbols or database keywords, specify PRESERVE_COL_NAMES=YES.

YES
specifies that column names that are used in table creation are passed to the DBMS with special characters and the exact, case-sensitive spelling of the name preserved.

Details

This option applies only when you use SAS/ACCESS to create a new DBMS table. When you create a table, you assign the column names by using one of the following methods:

- To control the case of the DBMS column names, specify variables with the desired case and set PRESERVE_COL_NAMES=YES. If you use special symbols or blanks, you must set VALIDVARNAME=ANY and use name literals. For more information, see the section about names in *SAS/ACCESS for Relational Databases: Reference* and the section about system options in *SAS Language Reference: Dictionary*.
- To enable the DBMS to normalize the column names according to its naming conventions, specify variables with any case and set PRESERVE_COLUMN_NAMES=NO.

Note: When you use SAS/ACCESS to read from, insert rows into, or modify data in an existing DBMS table, SAS identifies the database column names by their spelling. Therefore, when the database column exists, the case of the variable does not matter. △

For more information, see the topic about naming in the documentation for your SAS/ACCESS interface.

Specify the alias PRESERVE_NAMES= if you plan to specify both the PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options in your LIBNAME statement. Using this alias saves you some time when coding.

To use column names in your SAS program that are not valid SAS names, you must use one of the following techniques:

- Use the DQUOTE= option in PROC SQL and then reference your columns using double quotation marks. For example:

```
proc sql dquote=ansi;
   select "Total$Cost" from mydblib.mytable;
```

- Specify the global system option VALIDVARNAME=ANY and use name literals in the SAS language. For example:

```
proc print data=mydblib.mytable;
   format 'Total$Cost'n 22.2;
```

Note that if you are *creating* a table in PROC SQL, you must also include the PRESERVE_COL_NAMES=YES option. For example:

```
libname mydblib oracle user=testuser password=testpass;
   proc sql dquote=ansi;
     create table mydblib.mytable (preserve_col_names=yes) ("my$column" int);
```

PRESERVE_COL_NAMES= does not apply to the Pass-Through Facility.

See Also

To assign this option to a *group* of relational DBMS tables or views, see "PRESERVE_COL_NAMES= LIBNAME Option" on page 128.

QUALIFIER= Data Set Option

Specifies the qualifier to use when you are reading database objects, such as DBMS tables and views

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: MySQL, ODBC, OLE DB, Microsoft SQL Server

Default value: LIBNAME setting

Syntax

QUALIFIER=<*qualifier-name*>

Details

If this option is omitted, the default qualifier name, if any, is used for the data source. QUALIFIER= can be used for any data source, such as a DBMS object, that allows three-part identifier names: *qualifier.schema.object*.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "QUALIFIER= LIBNAME Option" on page 131.

QUERY_TIMEOUT= Data Set Option

Specifies the number of seconds of inactivity to wait before canceling a query

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, ODBC, Microsoft SQL Server

Default value: LIBNAME setting

Syntax

QUERY_TIMEOUT=*number-of-seconds*

Details

QUERY_TIMEOUT= 0 indicates that there is no time limit for a query. This option is useful when you are testing a query, you suspect that a query might contain an endless loop, or the data is locked by another user.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "QUERY_TIMEOUT= LIBNAME Option" on page 133.

READ_ISOLATION_LEVEL= Data Set Option

Specifies which level of read isolation locking to use when you are reading data

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

READ_ISOLATION_LEVEL=*DBMS-specific-value*

Syntax Description

dbms-specific-value
See the documentation for your SAS/ACCESS interface for the values for your DBMS.

Details

In the interfaces to ODBC and DB2 under UNIX and PC hosts, this option is ignored if READ_LOCK_TYPE= is not set to ROW.

The degree of isolation defines the following:

- the degree to which rows that are read and updated by the current application are available to other concurrently executing applications
- the degree to which update activity of other concurrently executing application processes can affect the current application.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "READ_ISOLATION_LEVEL= LIBNAME Option" on page 135.

READ_LOCK_TYPE= Data Set Option

Specifies how data in a DBMS table is locked during a read transaction

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

READ_LOCK_TYPE=ROW | PAGE | TABLE | NOLOCK | VIEW

Syntax Description

Not all values are valid for every interface. See details below.

ROW
locks a row if any of its columns are accessed. (This value is valid in the DB2 under UNIX and PC hosts, Microsoft SQL Server, ODBC, and Oracle interfaces.)

PAGE
locks a page of data, which is a DBMS-specific number of bytes. (This value is valid in the Sybase interface.)

TABLE
locks the entire DBMS table. If you specify READ_LOCK_TYPE=TABLE, you must also specify the LIBNAME option CONNECTION=UNIQUE, or you will receive an error message. Setting CONNECTION=UNIQUE ensures that your table lock is not lost, for example, due to another table closing and committing rows in the same connection. (This value is valid in the DB2 under z/OS, DB2 under UNIX and PC hosts, ODBC, Oracle, Microsoft SQL Server, and Teradata interfaces.)

NOLOCK
does not lock the DBMS table, pages, or any rows during a read transaction. (This value is valid in the Oracle and Sybase interfaces and in the ODBC and Microsoft SQL Server interfaces when you use the Microsoft SQL Server driver.)

VIEW
locks the entire DBMS view. (This value is valid in the Teradata interface.)

Details

If you omit READ_LOCK_TYPE=, you get either the default action for the DBMS that you are using, or a lock for the DBMS that was set with the LIBNAME statement.

See the documentation for your SAS/ACCESS interface for additional details.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "READ_LOCK_TYPE= LIBNAME Option" on page 136.

READ_MODE_WAIT= Data Set Option

Specifies during SAS/ACCESS read operations whether Teradata waits to acquire a lock or fails your request when the DBMS resource is locked by a different user

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Teradata

Default value: LIBNAME setting

Syntax

READ_MODE_WAIT= YES | NO

Syntax Description

YES
specifies that Teradata waits to acquire the lock, and SAS/ACCESS waits indefinitely until it can acquire the lock.

NO
specifies that Teradata fails the lock request if the specified DBMS resource is locked.

Details

If you specify READ_MODE_WAIT=NO, and a different user holds a *restrictive* lock, then the executing SAS step fails. SAS/ACCESS continues to process the job by executing the next step. If you specify READ_MODE_WAIT=YES, SAS/ACCESS waits indefinitely until it can acquire the lock.

A *restrictive* lock means that another user is holding a lock that prevents you from obtaining your desired lock. Until the other user releases the restrictive lock, you cannot obtain your lock. For example, another user's table-level WRITE lock prevents you from obtaining a READ lock on the table.

For more information, see the SAS/ACCESS documentation for Teradata.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "READ_MODE_WAIT= LIBNAME Option" on page 137.

READBUFF= Data Set Option

Specifies the number of rows of DBMS data to read into the buffer

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
DBMS support: DB2 UNIX/PC, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase
Default value: LIBNAME setting

Syntax

READBUFF=*integer*

Syntax Description

integer
is the maximum value that is allowed by the DBMS.

Details

This option improves performance by specifying a number of rows that can be held in memory for input into SAS. Buffering data reads can decrease network activities and increase performance. However, because SAS stores the rows in memory, higher values for READBUFF= use more memory. In addition, if too many rows are selected at once, then the rows that are returned to the SAS application might be out of date.

When READBUFF=1, only one row is retrieved at a time. The higher the value for READBUFF=, the more rows the SAS/ACCESS engine retrieves in one fetch operation.

DB2 UNIX/PC Details: By default, the SQLFetch API call is used and no internal SAS buffering is performed. Setting READBUFF=1 or greater causes the SQLExtendedFetch API call to be used. ROWSET_SIZE= is an alias for this option.

ODBC and Microsoft SQL Server Details: By default, the SQLFetch API call is used and no internal SAS buffering is performed. Setting READBUFF=1 or greater causes the SQLExtendedFetch API call to be used. ROWSET_SIZE= is an alias for this option.

OLE DB Details: ROWSET_SIZE= is an alias for this option.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "READBUFF= LIBNAME Option" on page 138.

SASDATEFMT= Data Set Option

Changes the SAS date format of a DBMS column

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, Informix, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: DBMS-specific

Syntax

SASDATEFMT=(*DBMS-date-col-1='SAS-date-format'*
<… *DBMS-date-col-n='SAS-date-format'*>)

Syntax Description

DBMS-date-col
specifies the name of a date column in a DBMS table.

SAS-date-format
specifies a SAS date format that has an equivalent (like-named) informat. For example, DATETIME21.2 is both a SAS format and a SAS informat, so it is a valid value for the *SAS-date-format* argument.

Details

If the date format of a SAS column does not match the date format of the corresponding DBMS column, you must convert the SAS date values to the appropriate DBMS date values. The SASDATEFMT= option enables you to convert date values from the default SAS date format to another SAS date format that you specify.

Use the SASDATEFMT= option to prevent date type mismatches in the following circumstances:

- during input operations to convert DBMS date values to the correct SAS DATE, TIME, or DATETIME values
- during output operations to convert SAS DATE, TIME, or DATETIME values to the correct DBMS date values.

The column names specified in this option must be DATE, DATETIME, or TIME columns; columns of any other type are ignored.

The format specified must be a valid date format; output with any other format is unpredictable.

If the SAS date format and the DBMS date format match, this option is not needed.

The default SAS date format is DBMS-specific and is determined by the data type of the DBMS column. See the documentation for your SAS/ACCESS interface.

Note: For non-English date types, SAS automatically converts the data to the SAS type of NUMBER. The SASDATEFMT= option does not currently handle these date types, but you can use a PROC SQL view to convert the DBMS data to a SAS date format as you retrieve the data, or use a format statement in other contexts. △

Oracle details: It is recommended that "DBSASTYPE= Data Set Option" on page 209 be used instead of SASDATEFMT=.

Examples

In the following example, the APPEND procedure adds SAS data from the SASLIB.DELAY data set to the Oracle table that is accessed by MYDBLIB.INTERNAT.

Using SASDATEFMT=, the default SAS format for the Oracle column DATES is changed to the DATE9. format. Data output from SASLIB.DELAY into the DATES column in MYDBLIB.INTERNAT now converts from the DATE9. format to the Oracle format assigned to that type.

```
libname mydblib oracle user=testuser password=testpass;
libname saslib 'your-SAS-library';

proc append base=mydblib.internat(sasdatefmt=(dates='date9.'))force
   data=saslib.delay;
run;
```

In the following example, SASDATEFMT= converts DATE1, a SAS DATETIME value, to a Teradata date column named DATE1.

```
libname x teradata user=testuser password=testpass;

proc sql noerrorstop;
   create table x.dateinfo ( date1 date );
   insert into x.dateinfo
   ( sasdatefmt=( date1='datetime21.') )
     values ( '31dec2000:01:02:30'dt );
```

In the following example, SASDATEFMT= converts DATE1, a Teradata date column, to a SAS DATETIME type named DATE1.

```
libname x teradata user=testuser password=testpass;

data sas_local;
format date1 datetime21.;
set x.dateinfo( sasdatefmt=( date1='datetime21.') );
run;
```

SCHEMA= Data Set Option

Enables you to read a data source, such as a DBMS table and view, in the specified schema

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 UNIX/PC, Informix, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: LIBNAME setting

Syntax

SCHEMA=*schema-name*

Syntax Description

schema-name
is the name assigned to a logical classification of objects in a relational database.

Details

For this option to work, you must have appropriate privileges to the schema that is specified.

The values for SCHEMA= are usually case-sensitive, so be careful when you specify this option.

Oracle Details: If PRESERVE_TAB_NAMES=NO, SAS converts the SCHEMA= value to uppercase because all values in the Oracle data dictionary are converted to uppercase unless quoted.

Teradata Details: If you omit this option, a libref points to your default Teradata database, which often has the same name as your user name. You can use this option to point to a different database. This option enables you to view or modify a different user's DBMS tables or views if you have the required Teradata privileges. (For example, to read another user's tables, you must have the Teradata privilege SELECT for that user's tables.) The Teradata interface alias for SCHEMA= is DATABASE=. For more information about changing the default database, see the DATABASE statement in your Teradata documentation.

Examples

In the following example, SCHEMA= causes MYDB.TEMP_EMPS to be interpreted by DB2 as SCOTT.TEMP_EMPS.

```
proc print data=mydb.temp_emps
     schema=SCOTT;
run;
```

In the following example, SAS sends any reference to Employees as Scott.Employees.

```
libname mydblib oracle user=testuser password=testpass path="myorapath";

proc print data=employees (schema=scott);
run;
```

In the following example, user TESTUSER prints the contents of the Employees table, which is located in the Donna database.

```
libname mydblib teradata user=testuser pw=testpass;

proc print data=mydblib.employees(schema=donna);
run;
```

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "SCHEMA= LIBNAME Option" on page 140.

SEGMENT_NAME= Data Set Option

Enables you to control the segment in which you create a table

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Sybase

Default value: none

Syntax

SEGMENT_NAME=*segment-name*

Syntax Description

segment-name
specifies the name of the segment in which to create a table.

SLEEP= Data Set Option

Specifies the number of minutes that MultiLoad waits before it retries logging in to Teradata.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: 6

Syntax

SLEEP=*number-of-minutes*

Syntax Description

number-of-minutes
the number of minutes that MultiLoad waits before it retries logging on to Teradata.

Details

Use the SLEEP= data set option to indicate to MultiLoad how long to wait before it retries logging on to Teradata when the maximum number of utilities are already running. (The maximum number of Teradata utilities that can run concurrently varies from 5 to 15, depending upon the database server setting.) The default value for SLEEP= is 6 minutes. The value specified for SLEEP must be greater than 0.

SLEEP= is used in conjunction with TENACITY=, which specifies the time in hours that MultiLoad must continue to retry the the logon operation. SLEEP= and TENACITY= function very much like the SLEEP and TENACITY run-time options of the native Teradata MultiLoad utility.

See Also

See TENACITY= for information about specifying how long to continue to retry a logon operation.

TENACITY= Data Set Option

Specifies how many hours MultiLoad continues to retry logging on to Teradata if the maximum number of Teradata utilities are already running.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

DBMS support: Teradata

Default value: 4

Syntax

TENACITY=*number-of-hours*

Syntax Description

number-of-hours
the number of hours to continue to retry logging on to Teradata.

Details

Use the TENACITY= data set option to indicate to MultiLoad how long to continue retrying a logon operation when the maximum number of utilities are already running. (The maximum number of Teradata utilities that can run concurrently varies from 5 to 15, depending upon the database server setting.) The default value for TENACITY= is 4 hours. The value specified for TENACITY must be greater than 0.

TENACITY= is used in conjunction with SLEEP=, which specifies the number of minutes that MultiLoad waits before it retries logging on to Teradata. SLEEP= and TENACITY= function very much like the SLEEP and TENACITY run-time options of the native Teradata MultiLoad utility.

The following message is written to the SAS log if the time period specified by TENACITY= is exceeded.

```
ERROR:  MultiLoad failed unexpectedly with returncode 12
```

Note: Check the MultiLoad log for more information about the cause of the MultiLoad failure. SAS does not receive any informational messages from Teradata either when the currently run MultiLoad process waits because the maximum number of utilities are already running, or if MultiLoad is terminated because the time limit specified for TENACITY= has been exceeded.

The native Teradata MultiLoad utility sends messages associated with SLEEP and TENACITY only to the MultiLoad log; therefore, nothing is written to the SAS log. △

See Also

See SLEEP= for information about specifying how long to wait before retrying a logon operation.

TRAP151= Data Set Option

Enables columns that cannot be updated to be removed from a FOR UPDATE OF clause so updating of columns can proceed as normal

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS

Default value: NO

Syntax

TRAP151=YES | NO

Syntax Description

YES
removes the non-updatable column that is designated in the error-151 and reprepares the statement for processing. This process is repeated until all columns that cannot be updated are removed, and all remaining columns can be updated.

NO
disables TRAP151=. TRAP151= is disabled by default. It is not necessary to specify NO.

Examples

In the following example, DB2DBUG is turned on so that you can see what occurs when TRAP151=YES:

Output 10.1 SAS Log for TRAP151=YES

```
proc fsedit data=x.v4(trap151=yes);
run;
SELECT * FROM V4 FOR FETCH ONLY
SELECT * FROM V4 FOR FETCH ONLY
SELECT "A","X","Y","B","Z","C" FROM V4 FOR UPDATE OF "A","X","Y","B","Z","C"
DB2 SQL Error, sqlca->sqlcode=-151
WARNING: SQLCODE -151: repreparing SELECT as:
  SELECT "A","X","Y","B","Z","C" FROM V4 FOR UPDATE OF "A","Y","B","Z","C"
DB2 SQL Error, sqlca->sqlcode=-151
WARNING: SQLCODE -151: repreparing SELECT as:
  SELECT "A","X","Y","B","Z","C" FROM V4 FOR UPDATE OF "A","B","Z","C"
DB2 SQL Error, sqlca->sqlcode=-151
WARNING: SQLCODE -151: repreparing SELECT as:
  SELECT "A","X","Y","B","Z","C" FROM V4 FOR UPDATE OF "A","B","C"
COMMIT WORK
NOTE:  The PROCEDURE FSEDIT used 0.13 CPU seconds and 14367K.
```

The following example features the same code with TRAP151 turned off:

Output 10.2 SAS Log for TRAP151=NO

```
proc fsedit data=x.v4(trap151=no);
run;
SELECT * FROM V4 FOR FETCH ONLY
SELECT * FROM V4 FOR FETCH ONLY
SELECT "A","X","Y","B","Z","C" FROM V4 FOR UPDATE OF "A","X","Y","B","Z","C"
DB2 SQL Error, sqlca->sqlcode=-151
ERROR: DB2 prepare error;  DSNT4081 SQLCODE= ---151, ERROR;
   THE UPDATE STATEMENT IS INVALID BECAUSE THE CATALOG DESCRIPTION OF COLUMN C
   INDICATES THAT IT CANNOT BE UPDATED.
COMMIT WORK
NOTE:  The SAS System stopped processing this step because of errors.
NOTE:  The PROCEDURE FSEDIT used 0.08 CPU seconds and 14367K.
```

UPDATE_ISOLATION_LEVEL= Data Set Option

Defines the degree of isolation of the current application process from other concurrently running application processes

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, MySQL, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: LIBNAME setting

Syntax

UPDATE_ISOLATION_LEVEL=*DBMS-specific-value*

Syntax Description

dbms-specific-value
See the documentation for your SAS/ACCESS interface for the values for your DBMS.

Details

The degree of isolation identifies the following:

- the degree to which rows that are read and updated by the current application are available to other concurrently executing applications
- the degree to which update activity of other concurrently executing application processes can affect the current application.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "UPDATE_ISOLATION_LEVEL= LIBNAME Option" on page 148.

UPDATE_LOCK_TYPE= Data Set Option

Specifies how data in a DBMS table is locked during an update transaction

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: DB2 z/OS, DB2 UNIX/PC, ODBC, OLE DB, Oracle, Microsoft SQL Server, Sybase, Teradata

Default value: LIBNAME setting

Syntax

UPDATE_LOCK_TYPE=ROW | PAGE | TABLE | NOLOCK | VIEW

Syntax Description

Not all values are valid for every interface. See details below.

ROW
locks a row if any of its columns are going to be updated. (This is valid in the DB2 under UNIX and PC hosts, Microsoft SQL Server, ODBC, OLE DB, and Oracle interfaces.)

PAGE
locks a page of data, which is a DBMS specific number of bytes. (This is valid in the Sybase interface.)

TABLE
locks the entire DBMS table. (This is valid in the DB2 under z/OS, DB2 under UNIX and PC hosts, ODBC, Oracle, Microsoft SQL Server, and Teradata interfaces.)

NOLOCK
does not lock the DBMS table, page, or any rows when reading them for update. (This is valid in the Microsoft SQL Server, ODBC, Oracle and Sybase interfaces.)

VIEW
locks the entire DBMS view. (This is valid in the Teradata interface.)

Details

If you omit UPDATE_LOCK_TYPE=, you get either the default action for the DBMS that you are using, or a lock for the DBMS that was set with the LIBNAME statement. You can set a lock for one DBMS table by using the data set option or for a group of DBMS tables by using the LIBNAME option.

For more information, see the documentation for your SAS/ACCESS interface.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "UPDATE_LOCK_TYPE= LIBNAME Option" on page 148.

UPDATE_MODE_WAIT= Data Set Option

Specifies during SAS/ACCESS update operations whether the DBMS waits to acquire a lock or fails your request when the DBMS resource is locked by a different user

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
DBMS support: Teradata
Default value: LIBNAME setting

Syntax

UPDATE_MODE_WAIT= YES | NO

Syntax Description

YES
specifies that Teradata waits to acquire the lock, so SAS/ACCESS waits indefinitely until it can acquire the lock.

NO
specifies that Teradata fails the lock request if the specified DBMS resource is locked.

Details

If you specify UPDATE_MODE_WAIT=NO and if a different user holds a *restrictive* lock, then your SAS step fails and SAS/ACCESS continues the job by processing the

next step. If you specify UPDATE_MODE_WAIT=YES, SAS/ACCESS waits indefinitely until it can acquire the lock.

A *restrictive* lock means that a different user is holding a lock that prevents you from obtaining your desired lock. Until the other user releases the restrictive lock, you cannot obtain your lock. For example, another user's table-level WRITE lock prevents you from obtaining a READ lock on the table.

Use SAS/ACCESS locking options only when Teradata's standard locking is undesirable.

For more information, see the documentation for the interface to Teradata.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "UPDATE_MODE_WAIT= LIBNAME Option" on page 149.

UPDATE_SQL= Data Set Option

Determines the method that is used to update and delete rows in a data source

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: ODBC, Microsoft SQL Server

Default value: LIBNAME setting

Syntax

UPDATE_SQL=YES | NO

Syntax Description

YES
specifies that SAS/ACCESS uses Current-of-Cursor SQL to update or delete rows in a table.

NO
specifies that SAS/ACCESS uses the SQLSetPos() API to update or delete rows in a table.

Details

This is the update and delete equivalent of the INSERT_SQL= option.

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "UPDATE_SQL= LIBNAME Option" on page 151.

UPDATEBUFF= Data Set Option

Specifies the number of rows that are processed in a single DBMS update or delete operation

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

DBMS support: Oracle

Default value: LIBNAME setting

Syntax

UPDATEBUFF=*positive-integer*

Syntax Description

positive-integer
is the maximum value that is allowed by the DBMS.

Details

When updating with the VIEWTABLE window or PROC FSVIEW, use UPDATEBUFF=1 to prevent the DBMS interface from trying to update multiple rows. By default, these features update only one observation at a time (since by default they use record-level locking, they lock only the observation that is currently being edited).

See Also

To assign this option to a *group* of relational DBMS tables or views, see the LIBNAME option "UPDATEBUFF= LIBNAME Option" on page 151.

CHAPTER

11

Macro Variables and System Options for Relational Databases

Introduction to Macro Variables and System Options for Relational Databases

This section describes the system options and macro variables that are available to use with SAS/ACCESS software. It describes only those components of the macro facility that depend on the SAS/ACCESS engines. Most features of the SAS macro facility are portable. For more information, refer to the *SAS Macro Language: Reference* and the SAS Help for the macro facility.

Macro Variables for Relational Databases

The automatic macro variables SYSDBMSG, SYSDBRC, SQLXMSG, and SQLXRC are portable, but their values are determined by the SAS/ACCESS engine and your DBMS. Initially, the macro variables SYSDBMSG and SQLXMSG are blank, whereas SYSDBRC and SQLXRC are set to **0**.

SAS/ACCESS generates several return codes and error messages while it processes your programs. This information is available to you through the following two SAS macro variables:

SYSDBMSG
: contains DBMS-specific error messages that are generated when you use SAS/ACCESS software to access your DBMS data.

SYSDBRC
: contains DBMS-specific error codes that are generated when you use SAS/ACCESS software to access your DBMS data. Error codes that are returned are text, not numbers.

You can use these two variables anywhere while you are accessing DBMS data. Because only one set of macro variables is provided, it is possible that, if tables from two different DBMSs are accessed, it might not be clear from which DBMS the error message originated. To address this problem, the name of the DBMS is inserted at the beginning of the SYSDBMSG macro variable message or value. The contents of the SYSDBMSG and SYSDBRC macro variables can be printed in the SAS log by using the %PUT macro. They are reset after each SAS/ACCESS LIBNAME statement, DATA

step, or procedure is executed. In the following statement, %SUPERQ masks special characters such as &, %, and any unbalanced parentheses or quotation marks that might exist in the text stored in the SYSDBMSG macro.

```
%put %superq(SYSDBMSG)
```

These special characters can cause unpredictable results if you use the following statement:

```
%put &SYSDBMSG
```

It is more advantageous to use %SUPERQ.

If you try to connect to Oracle and use the incorrect password, you receive the messages shown in the following output.

Output 11.1 SAS Log for an Oracle Error

```
2? libname mydblib oracle user=pierre pass=paris path="orav7";

ERROR: Oracle error trying to establish connection.  Oracle error is
       ORA-01017: invalid username/password; logon denied
ERROR: Error in the LIBNAME or FILENAME statement.
 3? %put %superq(sysdbmsg);

Oracle: ORA-01017: invalid username/passsword; logon denied
 4? %put &sysdbrc;

-1017
 5?
```

You can also use SYMGET to retrieve error messages:

```
msg=symget("SYSDBMSG");
```

For example:

```
data_null_;
msg=symget("SYSDBMSG");
put msg;
run;
```

The Pass-Through Facility generates return codes and error messages that are available to you through the following two SAS macro variables:

SQLXMSG
: contains DBMS-specific error messages.

SQLXRC
: contains DBMS-specific error codes.

SQLXMSG and SQLXRC can be used only with the Pass-Through Facility. See "Return Codes" on page 278.

The contents of the SQLXMSG and SQLXRC macro variables can be printed in the SAS log by using the %PUT macro. SQLXMSG is reset to a blank string and SQLXRC is reset to **0** when any Pass-Through Facility statement is executed.

System Options for Relational Databases

DBSRVTP= and DBSLICEPARM= are SAS system options for databases. SASTRACE=, SASTRACELOC=, and VALIDVARNAME= are SAS system options that have applications specific to SAS/ACCESS.

Note: The SAS system option REPLACE= is not supported by the SAS/ACCESS interfaces. △

DBSRVTP= System Option

Specifies whether SAS/ACCESS engines put a hold (or block) on the originating client while making performance-critical calls to the database. This option applies when SAS is invoked as a server responding to multiple clients

Valid in: SAS invocation

Default value: NONE

Syntax

DBSRVTP= 'ALL' | 'NONE' |'(*engine-name(s)*')

Syntax Description

ALL
indicates that SAS does not use any blocking operations for all underlying SAS/ ACCESS products that support this option.

NONE
indicates that SAS uses standard blocking operations for all SAS/ACCESS products.

engine-name(s)
indicates that SAS does not use any blocking operations for the specified SAS/ ACCESS interface(s). You can specify one or more of the following engine names. If you specify more than one name, separate them with blank spaces and enclose the list in parentheses.

db2 (only supported under UNIX and PC hosts)

informix

odbc (indicates that SAS uses nonblocking operations for the SAS/ACCESS interfaces to ODBC and Microsoft SQL Server).

oledb

Oracle

sybase

teradata (not supported on OS/390)

Details

This option can be used to help throughput of the SAS server by supporting multiple simultaneous execution streams, if the server utilizes certain SAS/ACCESS interfaces.

Better throughput is accomplished when the underlying SAS/ACCESS engine does not hold or block the originating client, such that any one client using a SAS/ACCESS product does not keep the SAS server from responding to other clients' requests. SAS/SHARE software and SAS Integration Technologies are two ways of invoking SAS as a server.

This option is a system invocation option, which means the value is set when SAS is invoked. Because the DBSRVTP= option uses multiple native threads, enabling this option uses the underlying DBMS's threading support. Some databases handle threading better than others, so you might want to invoke DBSRVTP= for some DBMSs and not others. Refer to your documentation for your DBMS for more information.

The option accepts a string where values are the engine name of a SAS/ACCESS product, ALL, or NONE. If multiple values are specified, enclose the values in quotation marks and parentheses, and separate the values with a space.

This option is applicable on all Windows platforms, AIX, SLX, and MVS (Oracle only). On some of these hosts, SAS can be invoked with the -SETJMP system option. Setting -SETJMP disables the DBSRVTP= option.

Examples

The following examples invoke SAS from the UNIX command line:

```
sas -dbsrvtp all


sas -dbsrvtp '(oracle db2)'


sas -dbsrvtp teradata


sas -dbsrvtp '(SYBASE informix odbc oledb)'


sas -dbsrvtp none
```

SASTRACE= System Option

Generates trace information from a DBMS engine

Valid in: OPTIONS statement, configuration file, SAS invocation
Default value: NONE

Syntax

SASTRACE= ',,,d' | ',,d,' | 'd,' | ',,,s' | ',,,sa' | ',,t,'

Syntax Description

',,,d'

specifies that all SQL statements sent to the DBMS are sent to the log. These statements include the following:

SELECT

CREATE

DROP

INSERT

UPDATE

DELETE

SYSTEM CATALOG

COMMIT

ROLLBACK

For those engines that do not generate SQL statements, the API calls, including all parameters, are sent to the log.

',,d,'
specifies that all routine calls are sent to the log. When this option is selected, all function enters and exits, as well as pertinent parameters and return codes, are traced. The information, however, will vary from engine to engine.

This option is most useful if you are having a problem and need to send a SAS log to technical support for troubleshooting.

'd,'
specifies that all DBMS calls, such as API and Client calls, connection information, column bindings, column error information, and row processing are sent to the log. However, this information will vary from engine to engine.

This option is most useful if you are having a problem and need to send a SAS log to technical support for troubleshooting.

',,,s'
specifies that a summary of timing information for calls made to the DBMS is sent to the log.

',,,sa'
specifies that timing information for each call made to the DMBS, along with a summary, is sent to the log.

',,t,'
specifies that all threading information is sent to the log. This information includes:

- the number of threads spawned
- the number of observations each thread contains
- the exit code of the thread, should it fail.

SAS/ACCESS Specific Details

The SASTRACE= options have behavior that is specific to SAS/ACCESS software. SASTRACE= is a very powerful tool to use when you want to see the commands that are sent to your DBMS by the SAS/ACCESS engine. SASTRACE= output is DBMS-specific. However, most SAS/ACCESS engines will show you statements like SELECT or COMMIT as the DBMS processes them for the SAS application. The following details will help you manage SASTRACE= output in your DBMS:

- When using SASTRACE= on PC platforms, you must also specify SASTRACELOC=.

□ In order to turn SAS tracing off, you can specify the following option:

```
options sastrace=off;
```

□ Log output is much easier to read if you specify NOSTSUFFIX.

Note: NOSTSUFFIX is not supported on MVS. △

The following code is entered without specifying the option, and the resulting log is longer and harder to decipher.

```
options sastrace=',,,d' sastraceloc=saslog;
proc print data=mydblib.snow_birthdays;
run;
```

The resulting log is as follows:

```
0 1349792597 sastb_next 2930 PRINT
ORACLE_5: Prepared: 1 1349792597 sastb_next 2930 PRINT
SELECT * FROM scott.SNOW_BIRTHDAYS 2 1349792597 sastb_next 2930 PRINT
3 1349792597 sastb_next 2930 PRINT
16 proce print data=mydblib.snow_birthdays; run;

4 1349792597 sastb_next 2930 PRINT
ORACLE_6: Executed: 5 1349792597 sastb_next 2930 PRINT
Prepared statement ORACLE_5 6 1349792597 sastb_next 2930 PRINT
7 1349792597 sastb_next 2930 PRINT
```

However, by using NOSTSUFFIX, the log file becomes much easier to read:

```
options sastrace=',,,d' sastraceloc=saslog nostsuffix;
proc print data=mydblib.snow_birthdays;
run;
```

The resulting log is as follows:

```
ORACLE_1: Prepared:
SELECT * FROM scott.SNOW_BIRTHDAYS

12 proc print data=mydblib.snow_birthdays; run;

ORACLE_2: Executed:
Prepared statement ORACLE_1
```

Examples

The examples in this section are based on the following data set, and are shown using NOSTSUFFIX and SASTRACELOC=SASLOG.

```
data work.winter_birthdays;
   input empid birthdat date9. lastname $18.;
   format birthdat date9.;
datalines;
678999 28DEC1966 PAVEO             JULIANA        3451
456788 12JAN1977 SHIPTON           TIFFANY        3468
890123 20FEB1973 THORSTAD          EDVARD         3329
;
run;
```

Example 1: SQL Trace ',,,d'

```
options sastrace=',,,d' sastraceloc=saslog nostsuffix;
libname mydblib oracle user=scott password=tiger schema=bday_data;
```

```
data mydblib.snow_birthdays;
   set work.winter_birthdays;
run;

libname mydblib clear;
```

The output is written to the SAS log, as specified by the SASTRACELOC=SASLOG option.

Output 11.2 SAS Log Output from the SASTRACE= ',,,d' System Option

```
30   data work.winter_birthdays;
31      input empid birthdat date9. lastname $18.;
32      format birthdat date9.;
33   datalines;

NOTE: The data set WORK.WINTER_BIRTHDAYS has 3 observations and 3 variables.
NOTE: DATA statement used (Total process time):
     real time           0.03 seconds
     cpu time            0.04 seconds
37   ;
38   run;
39   options sastrace=',,,d' sastraceloc=saslog nostsuffix;

40   libname mydblib oracle user=scott password=XXXXX schema=bday_data;
NOTE: Libref MYDBLIB was successfully assigned as follows:
     Engine:        ORACLE
     Physical Name:

41   proc delete data=mydblib.snow_birthdays; run;

ORACLE_1: Prepared:
SELECT * FROM SNOW_BIRTHDAYS

ORACLE_2: Executed:
DROP TABLE  SNOW_BIRTHDAYS

NOTE: Deleting MYDBLIB.SNOW_BIRTHDAYS (memtype=DATA).
NOTE: PROCEDURE DELETE used (Total process time):
     real time           0.26 seconds
     cpu time            0.12 seconds

42   data mydblib.snow_birthdays;
43      set work.winter_birthdays;
44   run;

ORACLE_3: Prepared:
SELECT * FROM SNOW_BIRTHDAYS

NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
```

```
ORACLE_4: Executed:
CREATE TABLE SNOW_BIRTHDAYS(empid NUMBER ,birthdat DATE,lastname VARCHAR2 (18))

ORACLE_5: Prepared:
INSERT  INTO SNOW_BIRTHDAYS (empid,birthdat,lastname) VALUES
(:empid,TO_DATE(:birthdat,'DDMONYYYY','NLS_DATE_LANGUAGE=American'),:lastname)

NOTE: There were 3 observations read from the data set WORK.WINTER_BIRTHDAYS.

ORACLE_6: Executed:
Prepared statement ORACLE_5

ORACLE:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-*
NOTE: The data set MYDBLIB.SNOW_BIRTHDAYS has 3 observations and 3 variables.
ORACLE:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-*
NOTE: DATA statement used (Total process time):
     real time           0.47 seconds
     cpu time            0.13 seconds

ORACLE_7: Prepared:
SELECT * FROM SNOW_BIRTHDAYS

45   proc print data=mydblib.snow_birthdays; run;

ORACLE_8: Executed:
Prepared statement ORACLE_7

NOTE: There were 3 observations read from the data set MYDBLIB.SNOW_BIRTHDAYS.
NOTE: PROCEDURE PRINT used (Total process time):
     real time           0.04 seconds
     cpu time            0.04 seconds

46
47   libname mydblib clear;
NOTE: Libref MYDBLIB has been deassigned.
```

Example 2: Log Trace ',,d'

```
options sastrace=',,d,' sastraceloc=saslog nostsuffix;
libname mydblib oracle user=scott password=tiger schema=bday_data;

data mydblib.snow_birthdays;
   set work.winter_birthdays;
run;

libname mydblib clear;
```

The output is written to the SAS log, as specified by the SASTRACELOC=SASLOG option.

Output 11.3 SAS Log Output from the SASTRACE= ',,d,' System Option

```
84   options sastrace=',,d,' sastraceloc=saslog nostsuffix;

ACCESS ENGINE: Entering DBICON
ACCESS ENGINE: Number of connections is 1
ORACLE: orcon()
ACCESS ENGINE: Successful physical conn id 1
ACCESS ENGINE: Exiting DBICON, Physical Connect id = 1, with rc=0X00000000
85   libname mydblib oracle user=dbitest password=XXXXX schema=bday_data;
ACCESS ENGINE:  CONNECTION= SHAREDREAD
NOTE: Libref MYDBLIB was successfully assigned as follows:
      Engine:        ORACLE
      Physical Name: lupin

86   data mydblib.snow_birthdays;
87      set work.winter_birthdays;
88   run;

ACCESS ENGINE:  Entering yoeopen
ACCESS ENGINE: Entering dbiopen
ORACLE: oropen()
ACCESS ENGINE: Successful dbiopen, open id 0, connect id 1
ACCESS ENGINE: Exit dbiopen with rc=0X00000000
ORACLE: orqall()
ORACLE: orprep()
ACCESS ENGINE: Entering dbiclose
ORACLE: orclose()
ACCESS ENGINE: DBICLOSE open_id 0, connect_id 1
ACCESS ENGINE: Exiting dbiclos with rc=0X00000000
ACCESS ENGINE:  Access Mode is XO_OUTPUT
ACCESS ENGINE:  Access Mode is XO_SEQ
ACCESS ENGINE:  Shr flag is XHSHRMEM
ACCESS ENGINE: Entering DBICON
ACCESS ENGINE:  CONNECTION= SHAREDREAD
ACCESS ENGINE: Number of connections is 2
ORACLE: orcon()
ACCESS ENGINE: Successful physical conn id 2
ACCESS ENGINE: Exiting DBICON, Physical Connect id = 2, with rc=0X00000000
ACCESS ENGINE: Entering dbiopen
ORACLE: oropen()
ACCESS ENGINE: Successful dbiopen, open id 0, connect id 2
ACCESS ENGINE: Exit dbiopen with rc=0X00000000
ACCESS ENGINE: Exit yoeopen with SUCCESS.
ACCESS ENGINE:  Begin yoeinfo
ACCESS ENGINE: Exit yoeinfo with SUCCESS.
ORACLE: orovar()
NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
ORACLE: oroload()
ACCESS ENGINE: Entering dbrload with SQL Statement set to
          CREATE TABLE SNOW_BIRTHDAYS(empid NUMBER ,birthdat DATE,lastname VARCHAR2 (18))

ORACLE: orexec()
ORACLE: orexec() END
ORACLE: orins()
ORACLE: orubuf()
ORACLE: orubuf()
ORACLE: SAS date : 28DEC1966
ORACLE: orins()
ORACLE: SAS date : 12JAN1977
ORACLE: orins()
ORACLE: SAS date : 20FEB1973
NOTE: There were 3 observations read from the data set WORK.WINTER_BIRTHDAYS.
ORACLE: orforc()
ORACLE: orflush()
NOTE: The data set MYDBLIB.SNOW_BIRTHDAYS has 3 observations and 3 variables.
ACCESS ENGINE:  Enter yoeclos
ACCESS ENGINE: Entering dbiclose
ORACLE: orclose()
ORACLE: orforc()
ORACLE: orflush()
```

```
ACCESS ENGINE: DBICLOSE open_id 0, connect_id 2
ACCESS ENGINE: Exiting dbiclos with rc=0X00000000
ACCESS ENGINE: Entering DBIDCON
ORACLE:  ordcon
ACCESS ENGINE: Physical disconnect on id = 2
ACCESS ENGINE: Exiting DBIDCON with rc=0X00000000, rc2=0X00000000
ACCESS ENGINE: Exit yoeclos with rc=0x00000000
NOTE: DATA statement used (Total process time):
      real time           0.21 seconds
      cpu time            0.06 seconds


ACCESS ENGINE: Entering DBIDCON
ORACLE:  ordcon
ACCESS ENGINE: Physical disconnect on id = 1
ACCESS ENGINE: Exiting DBIDCON with rc=0X00000000, rc2=0X00000000
89   libname mydblib clear;
NOTE: Libref MYDBLIB has been deassigned.
```

Example 3: DBMS Trace 'd,'

```
options sastrace='d,' sastraceloc=saslog nostsuffix;
libname mydblib oracle user=scott password=tiger schema=bday_data;

data mydblib.snow_birthdays;
   set work.winter_birthdays;
run;

libname mydblib clear;
```

The output is written to the SAS log, as specified by the SASTRACELOC=SASLOG option.

Output 11.4 SAS Log Output from the SASTRACE= 'd,' System Option

```
ORACLE: PHYSICAL connect successful.
ORACLE: USER=scott
ORACLE: PATH=lupin
ORACLE: SCHEMA=bday_data
110  libname mydblib oracle user=dbitest password=XXXXX path=lupin schema=bday_data;

NOTE: Libref MYDBLIB was successfully assigned as follows:
      Engine:        ORACLE
      Physical Name: lupin

111  data mydblib.snow_birthdays;
112     set work.winter_birthdays;
113  run;

ORACLE: PHYSICAL connect successful.
ORACLE: USER=scott
ORACLE: PATH=lupin
ORACLE: SCHEMA=bday_data
NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
ORACLE:  INSERTBUFF option value set to 10.
NOTE: There were 3 observations read from the data set WORK.WINTER_BIRTHDAYS.
ORACLE:  Rows processed: 3
ORACLE:  Rows failed   : 0
NOTE: The data set MYDBLIB.SNOW_BIRTHDAYS has 3 observations and 3 variables.
ORACLE: Successfully disconnected.
ORACLE: USER=scott
ORACLE: PATH=lupin
NOTE: DATA statement used (Total process time):
      real time           0.21 seconds
      cpu time            0.04 seconds


ORACLE: Successfully disconnected.
ORACLE: USER=scott
ORACLE: PATH=lupin
114  libname mydblib clear;
NOTE: Libref MYDBLIB has been deassigned.
```

Example 4: Time Trace ',,,s'

```
options sastrace=',,,s' sastraceloc=saslog nostsuffix;
libname mydblib oracle user=scott password=tiger schema=bday_data;

data mydblib.snow_birthdays;
   set work.winter_birthdays;
run;

libname mydblib clear;
```

The output is written to the SAS log, as specified by the SASTRACELOC=SASLOG option.

Output 11.5 SAS Log Output from the SASTRACE= ',,,s' System Option

```
118  options sastrace=',,,s' sastraceloc=saslog nostsuffix;

119  libname mydblib oracle user=dbitest password=XXXXX schema=bday_data;

NOTE: Libref MYDBLIB was successfully assigned as follows:
      Engine:        ORACLE
      Physical Name: lupin

120  data mydblib.snow_birthdays;
121     set work.winter_birthdays;
122  run;

NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
NOTE: There were 3 observations read from the data set WORK.WINTER_BIRTHDAYS.
NOTE: The data set MYDBLIB.SNOW_BIRTHDAYS has 3 observations and 3 variables.

Summary Statistics for ORACLE are:
Total SQL execution seconds were:                   0.127079
Total SQL prepare seconds were:                     0.004404
Total SQL row insert seconds were:                  0.004735
Total seconds used by the ORACLE ACCESS engine were     0.141860

NOTE: DATA statement used (Total process time):
      real time           0.21 seconds
      cpu time            0.04 seconds


123  libname mydblib clear;
NOTE: Libref MYDBLIB has been deassigned.
```

Example 5: Time All Trace ',,,sa'

```
options sastrace=',,,sa' sastraceloc=saslog nostsuffix;
libname mydblib oracle user=scott password=tiger schema=bday_data;

data mydblib.snow_birthdays;
   set work.winter_birthdays;
run;

libname mydblib clear;
```

The output is written to the SAS log, as specified by the SASTRACELOC=SASLOG option.

Output 11.6 SAS Log Output from the SASTRACE= ',,,sa' System Option

```
146  options sastrace=',,,sa' sastraceloc=saslog nostsuffix;
147
148  libname mydblib oracle user=dbitest password=XXXXX path=lupin schema=dbitest insertbuff=1;

NOTE: Libref MYDBLIB was successfully assigned as follows:
      Engine:        ORACLE
      Physical Name: lupin

149  data mydblib.snow_birthdays;
150     set work.winter_birthdays;
151  run;

NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
ORACLE:  The insert time in seconds is    0.004120
ORACLE:  The insert time in seconds is    0.001056
ORACLE:  The insert time in seconds is    0.000988
NOTE: There were 3 observations read from the data set WORK.WINTER_BIRTHDAYS.
NOTE: The data set MYDBLIB.SNOW_BIRTHDAYS has 3 observations and 3 variables.

Summary Statistics for ORACLE are:
Total SQL execution seconds were:                   0.130448
Total SQL prepare seconds were:                     0.004525
Total SQL row insert seconds were:                  0.006158
Total seconds used by the ORACLE ACCESS engine were     0.147355

NOTE: DATA statement used (Total process time):
      real time           0.20 seconds
      cpu time            0.00 seconds


152
153  libname mydblib clear;
NOTE: Libref MYDBLIB has been deassigned.
```

Example 6: Threaded Trace ',,t,'

```
options sastrace=',,t,' sastraceloc=saslog nostsuffix;
libname mydblib oracle user=scott password=tiger schema=bday_data;

data mydblib.snow_birthdays(DBTYPE=(empid'number(10');
   set work.winter_birthdays;
run;

proc print data=mydblib.snow_birthdays(dbsliceparm=(all,3));
run;
```

The output is written to the SAS log, as specified by the SASTRACELOC=SASLOG option.

Output 11.7 SAS Log Output from the SASTRACE= ',,t,' System Option

```
165  options sastrace=',,t,' sastraceloc=saslog nostsuffix;
166  data mydblib.snow_birthdays(DBTYPE=(empid='number(10)'));
167     set work.winter_birthdays;
168  run;

NOTE: SAS variable labels, formats, and lengths are not written to DBMS tables.
NOTE: There were 3 observations read from the data set WORK.WINTER_BIRTHDAYS.
NOTE: The data set MYDBLIB.SNOW_BIRTHDAYS has 3 observations and 3 variables.
NOTE: DATA statement used (Total process time):
       real time          0.21 seconds
       cpu time           0.06 seconds

169  proc print data=mydblib.snow_birthdays(dbsliceparm=(all,3));
170  run;

ORACLE:  DBSLICEPARM option set and 3 threads were requested
ORACLE:  No application input on number of threads.
ORACLE:  Thread 1 contains 1 obs.
ORACLE:  Thread 2 contains 0 obs.
ORACLE:  Thread 3 contains 2 obs.
ORACLE:  Threaded read enabled. Number of threads created: 3
NOTE: There were 3 observations read from the data set MYDBLIB.SNOW_BIRTHDAYS.
NOTE: PROCEDURE PRINT used (Total process time):
       real time           1.12 seconds
       cpu time            0.17 seconds
```

For more information about tracing threaded reads, refer to "Generating Trace Information for Threaded Reads" on page 50.

Note: The SASTRACE= options can also be used in conjunction with each other. For example, SASTRACE=',,d,d'. △

SASTRACELOC= System Option

Prints SASTRACE information to a specified location

Valid in: OPTIONS statement, configuration file, SAS invocation

Default value: stdout

Syntax

SASTRACELOC=stdout | SASLOG | FILE *'path-and-filename'*

Details

SASTRACELOC= enables you to specify where to put the trace messages that are generated by SASTRACE=. By default, the output goes to the default output location for your operating environment. You can send the output to a SAS log by specifying SASTRACELOC=SASLOG.

Note: This option and its values might differ for each host. △

Example

The following example on a PC platform writes the trace information to the TRACE.LOG file in the work directory on the C drive.

```
options sastrace=',,,d' sastraceloc=file 'c:\work\trace.log';
```

VALIDVARNAME= System Option

Controls the type of SAS variable names that can be used or created during a SAS session

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Default value: V7

Syntax

VALIDVARNAME= V7 | V6 | UPCASE | ANY

SAS/ACCESS Specific Details

VALIDVARNAME= enables you to control which rules apply for SAS variable names. For more information about the VALIDVARNAME= system option, see the *SAS Language Reference: Dictionary*. The settings are as follows:

VALIDVARNAME=V7
indicates that a DBMS column name is changed to a valid SAS name by using the following rules:

- Up to 32 mixed-case alphanumeric characters are allowed.
- Names must begin with an alphabetic character or an underscore.
- Invalid characters are changed to underscores.
- Any column name that is not unique when it is normalized is made unique by appending a counter (0,1,2,...) to the name.

This is the default value for SAS Version 7 and later.

VALIDVARNAME=V6
indicates that only those variable names that are considered valid in Version 6 are considered valid SAS variable names. When V6 is specified in Pass-Through Facility code, the DBMS engine truncates column names to 8 characters as it did in Version 6. If required, numbers are appended to the end of the truncated name to make it unique.

VALIDVARNAME=UPCASE
indicates that a DBMS column name is changed to a valid SAS name as described in VALIDVARNAME=V7 except that variable names are in uppercase.

VALIDVARNAME=ANY
allows any characters in DBMS column names to appear as valid characters in SAS variable names. Symbols, such as the equal sign (=) and the asterisk (*), must be contained in a *'variable-name'*n construct. You must use ANY whenever you want to read DBMS column names that do not follow the SAS naming conventions.

Example

The following example shows how the Pass-Through Facility works with VALIDVARNAME=V6.

```
options validvarname=v6;
proc sql;
   connect to oracle (user=testuser pass=testpass);
   create view myview as
     select amount_b, amount_s
       from connection to oracle
         (select "Amount Budgeted$", "Amount Spent$"
            from mytable);
quit;

proc contents data=myview;
run;
```

The output from this example would show that "Amount Budgeted$" becomes AMOUNT_B and "Amount Spent$" becomes AMOUNT_S.

See Also

"SAS Names and Support for DBMS Names" in *SAS/ACCESS for Relational Databases: Reference*

CHAPTER

12

The Pass-Through Facility for Relational Databases

Overview of the SQL Procedure's Interactions with SAS/ACCESS

The SQL procedure implements structured query language (SQL) for SAS software. See the *Base SAS Procedures Guide* for information about PROC SQL. SAS/ACCESS software for relational databases interacts with PROC SQL in the following ways:

- You can assign a libref to a DBMS using the SAS/ACCESS LIBNAME statement and then reference the new libref in a PROC SQL statement to query, update, or delete DBMS data. See the Chapter 9, "The LIBNAME Statement for Relational Databases," on page 73 for more information about this feature.
- You can embed LIBNAME information in a PROC SQL view and then automatically connect to the DBMS every time the PROC SQL view is processed. See "SQL Views with Embedded LIBNAME Statements" on page 76 for more information.
- You can send DBMS-specific SQL statements directly to a DBMS using an extension to PROC SQL called the Pass-Through Facility. See "Syntax for the Pass-Through Facility for Relational Databases" on page 278 for the syntax for this feature.

Overview of the Pass-Through Facility

The Pass-Through Facility uses SAS/ACCESS to connect to a DBMS and to send statements directly to the DBMS for execution. This facility is an alternative to the SAS/ACCESS LIBNAME statement. It enables you to use the SQL syntax of your DBMS, and it supports any non-ANSI standard SQL that is supported by your DBMS.

Note: Not all SAS/ACCESS interfaces support this feature. See Chapter 8, "SAS/ACCESS Features by Host," on page 65 to determine whether this feature is available in your environment. △

The Pass-Through Facility enables you to complete the following tasks:

- establish and terminate connections with a DBMS using the facilities CONNECT and DISCONNECT statements
- send dynamic, non-query, DBMS-specific SQL statements to a DBMS using the facility's EXECUTE statement

- retrieve data directly from a DBMS using the facilities CONNECTION TO component in the FROM clause of a PROC SQL SELECT statement.

You can use Pass-Through Facility statements in a PROC SQL query or you can store them in an SQL view. When you create an SQL view, any arguments that you specify in the CONNECT statement are stored with the view. Therefore, when the view is used in a SAS program, SAS can establish the appropriate connection to the DBMS.

Syntax for the Pass-Through Facility for Relational Databases

This section presents the syntax for the Pass-Through Facility statements and the CONNECTION TO component. For DBMS-specific details, see the documentation for your SAS/ACCESS interface.

PROC SQL <*option(s)*>;

CONNECT TO *dbms-name* <AS *alias*> <(<*database-connection-arguments*> <*connect-statement-arguments*>)>;

DISCONNECT FROM *dbms-name* | *alias*;

EXECUTE (*dbms-specific-SQL-statement*) BY *dbms-name* | *alias*;

SELECT *column-list* FROM **CONNECTION TO** *dbms-name* | *alias (dbms-query)*

Return Codes

As you use the PROC SQL statements that are available in the Pass-Through Facility, any error return codes and error messages are written to the SAS log. These codes and messages are available to you through the following two SAS macro variables:

SQLXRC
contains the DBMS return code that identifies the DBMS error.

SQLXMSG
contains descriptive information about the DBMS error that is generated by the DBMS.

The contents of the SQLXRC and SQLXMSG macro variables are printed in the SAS log using the %PUT macro. They are reset after each Pass-Through Facility statement has been executed.

See “Macro Variables for Relational Databases” on page 261 for more information about these return codes.

CONNECT Statement

Establishes a connection with the DBMS

Valid in: PROC SQL steps (when accessing DBMS data using SAS/ACCESS software)

Syntax

CONNECT TO *dbms-name* <AS *alias*> <(<*database-connection-arguments*> <*connect-statement-arguments*>)>;

The CONNECT statement establishes a connection with the DBMS. You establish a connection to send DBMS-specific SQL statements to the DBMS or to retrieve DBMS data. The connection remains in effect until you issue a DISCONNECT statement or terminate the SQL procedure.

To connect to a DBMS using the Pass-Through Facility, complete the following steps:

1. Initiate a PROC SQL step.
2. Use the Pass-Through Facility's CONNECT statement, identify the DBMS (such as Oracle or DB2), and (optionally) assign an alias.
3. Specify any attributes for the connection (such as multiple, shared, unique).
4. Specify any arguments that are needed to connect to the database.

The CONNECT statement is optional for some DBMSs. However, if it is not specified, the default values for all of the database connection arguments are used.

Any return code or message that is generated by the DBMS is available in the macro variables SQLXRC and SQLXMSG after the statement executes. See "Macro Variables for Relational Databases" on page 261 for more information about these macro variables.

Arguments

You use the following arguments with the CONNECT statement:

dbms-name
identifies the database management system to which you want to connect. You must specify the DBMS name for your SAS/ACCESS interface. You may also specify an optional alias.

alias
specifies for the connection an optional alias that has 1 to 32 characters. If you specify an alias, the keyword AS must appear before the alias. If an alias is not specified, the DBMS name is used as the name of the Pass-Through connection.

database-connection-arguments
specifies the DBMS-specific arguments that are needed by PROC SQL to connect to the DBMS. These arguments are optional for most databases, but if they are included, they must be enclosed in parentheses. See the documentation for your SAS/ACCESS interface for information about these arguments.

connect-statement-arguments
specifies arguments that indicate whether you can make multiple connections, shared or unique connections, and so on, to the database. These arguments enable the Pass-Through Facility to use some of the LIBNAME statement's connection management features. These arguments are optional, but if they are included, they must be enclosed in parentheses.

CONNECTION= SHARED | GLOBAL
indicates whether multiple CONNECT statements for a DBMS can use the same connection.

The CONNECTION= option enables you to control the number of connections, and therefore transactions, that your SAS/ACCESS engine executes and supports for each Pass-Through CONNECT statement.

When CONNECTION=GLOBAL, multiple CONNECT statements that use identical values for CONNECTION=, CONNECTION_GROUP=, DBCONINIT=, DBCONTERM=, and any database connection arguments can share the same connection to the DBMS.

When CONNECTION=SHARED, the CONNECT statement makes one connection to the DBMS. Only Pass-Through statements that use this alias share the connection. SHARED is the default value for CONNECTION=.

In the following example, the two CONNECT statements share the same connection to the DBMS because CONNECTION=GLOBAL. Only the first CONNECT statement actually makes the connection to the DBMS, while the last DISCONNECT statement is the only statement that disconnects from the DBMS.

```
proc sql;

/*...SQL Pass-Through statements referring to mydbone...*/

connect to oracle as mydbone
   (user=testuser pw=testpass
     path='myorapath'
     connection=global);

/*...SQL Pass-Through statements referring to mydbtwo...*/

connect to oracle as mydbtwo
   (user=testuser pw=testpass
     path='myorapath'
     connection=global);

disconnect from mydbone;
disconnect from mydbtwo;
quit;
```

CONNECTION_GROUP=*connection-group-name*
specifies a connection that can be shared among several CONNECT statements in the Pass-Through Facility.

Default value: none

By specifying the name of a connection group, you can share one DBMS connection among several different CONNECT statements. The connection to the DBMS can be shared only if each CONNECT statement specifies the same CONNECTION_GROUP= value and specifies identical DBMS connection arguments.

When CONNECTION_GROUP= is specified, it implies that the value of the CONNECTION= option will be GLOBAL.

DBCONINIT=<'>*DBMS-user-command*<'>
specifies a user-defined initialization command to be executed immediately after the connection to the DBMS.

You can specify any DBMS command that can be passed by the SAS/ACCESS engine to the DBMS and that does not return a result set or output parameters. The command executes immediately after the DBMS connection is established successfully. If the command fails, a disconnect occurs, and the CONNECT statement fails. You must specify the command as a single, quoted string, unless it is an environment variable.

DBCONTERM='*DBMS-user-command*'
specifies a user-defined termination command to be executed before the disconnect from the DBMS that occurs with the DISCONNECT statement.

Default value: none

The termination command that you select can be a script, stored procedure, or any DBMS SQL language statement that might provide

additional control over the interaction between the SAS/ACCESS engine and the DBMS. You can specify any valid DBMS command that can be passed by the SAS/ACCESS engine to the DBMS and that does not return a result set or output parameters. The command executes immediately before SAS terminates each connection to the DBMS. If the command fails, SAS provides a warning message but the disconnect still occurs. You must specify the command as a quoted string.

DBGEN_NAME= DBMS | SAS
specifies whether to automatically rename DBMS columns containing characters that SAS does not allow, such as $, to valid SAS variable names. See "DBGEN_NAME= LIBNAME Option" on page 99 for further information.

DBMAX_TEXT=*integer*
determines the length of any very long DBMS character data type that is read into SAS or written from SAS when using a SAS/ACCESS engine. This option applies to reading, appending, and updating rows in an existing table. It does not apply when you are creating a table.

Examples of a long DBMS data type are the SYBASE TEXT data type or the Oracle LONG RAW data type.

DBPROMPT=YES | NO
specifies whether SAS displays a window that prompts the user to enter DBMS connection information prior to connecting to the DBMS.

Default value: NO

If you specify DBPROMPT=YES, SAS displays a window that interactively prompts you for the DBMS connection arguments when the CONNECT statement is executed. Therefore, it is not necessary to provide connection arguments with the CONNECT statement. If you do specify connection arguments with the CONNECT statement and you specify DBPROMPT=YES, the connection argument values are displayed in the window. These values can be overridden interactively.

If you specify DBPROMPT=NO, SAS does not display the prompting window.

The DBPROMPT= option interacts with the DEFER= option to determine when the prompt window appears. If DEFER=NO, the DBPROMPT window opens when the CONNECT statement is executed. If DEFER=YES, the DBPROMPT window opens the first time a Pass-Through statement is executed. The DEFER= option normally defaults to NO, but defaults to YES if DBPROMPT=YES. You can override this default by explicitly setting DEFER=NO.

DEFER=NO | YES
determines when the connection to the DBMS occurs.

Default value: NO

If DEFER=YES, the connection to the DBMS occurs when the first Pass-Through statement is executed. If DEFER=NO, the connection to the DBMS occurs when the CONNECT statement occurs.

VALIDVARNAME=V6
indicates that only those variable names considered valid SAS variable names in Version 6 of SAS software are considered valid. Specify this connection argument if you want Pass-Through to operate in Version 6 compatibility mode.

By default DBMS column names are changed to valid SAS names by using the following rules:

- Up to 32 mixed-case alphanumeric characters are allowed.

- Names must begin with an alphabetic character or an underscore.
- Characters that are not permitted are changed to underscores.
- Any column name that is not unique when it is normalized is made unique by appending a counter (0,1,2,...) to the name.

When VALIDVARNAME=V6 is specified, the SAS/ACCESS engine for the DBMS truncates column names to eight characters, as it did in Version 6. If required, numbers are appended to the ends of the truncated names to make them unique. Setting this option overrides the value of the SAS system option VALIDVARNAME= during (and only during) the Pass-Through connection.

The following example shows how the Pass-Through Facility uses VALIDVARNAME=V6 as a connection argument. Using this option causes the output to show the DBMS column "Amount Budgeted$" as AMOUNT_B and "Amount Spent$" as AMOUNT_S.

```
proc sql;
connect to oracle (user=gloria password=teacher
                   validvarname=v6)
create view budget2000 as
 select amount_b, amount_s
 from connection to oracle
     (select "Amount Budgeted$", "Amount Spent$"
      from annual_budget);
quit;
proc contents data=budget2000;
run;
```

For this example, if you *omit* VALIDVARNAME=V6 as a connection argument, you must add it in an OPTIONS= statement in order for PROC CONTENTS to work:

```
options validvarname=v6;
proc contents data=budget2000;
run;
```

Thus, using it as a connection argument saves you coding later.

Note: In addition to the arguments listed here, several other LIBNAME options are available for use with the CONNECT statement. See the section about the Pass-Through Facility in the documentation for your SAS/ACCESS interface to determine which LIBNAME options are available in the Pass-Through Facility for your DBMS. When used with the Pass-Through Facility CONNECT statement, these options have the same effect as they do in a LIBNAME statement. △

CONNECT Statement Example

The following example connects to a Sybase server and assigns the alias SYBCON1 to it. Sybase is a case-sensitive database. Therefore, the database objects are in uppercase, as they were created.

```
proc sql;
connect to sybase as sybcon1
   (server=SERVER1 database=PERSONNEL
    user=testuser password=testpass
    connection=global);
%put &sqlxmsg &sqlxrc;
```

Note: You might be able to omit the CONNECT statement and implicitly connect to a database using default settings. See the documentation for your SAS/ACCESS interface for more information. △

DISCONNECT Statement

Terminates the connection to the DBMS

Valid in: PROC SQL steps (when accessing DBMS data using SAS/ACCESS software)

Syntax

DISCONNECT FROM *dbms-name* | *alias*

The DISCONNECT statement ends the connection with the DBMS. If the DISCONNECT statement is omitted, an implicit DISCONNECT is performed when PROC SQL terminates. The SQL procedure continues to execute until you submit a QUIT statement, another SAS procedure, or a DATA step.

Any return code or message that is generated by the DBMS is available in the macro variables SQLXRC and SQLXMSG after the statement executes. See "Macro Variables for Relational Databases" on page 261 for more information about these macro variables.

Arguments

Use one of the following arguments with the DISCONNECT statement:

dbms-name
specifies the database management system from which you want to disconnect. You must specify the DBMS name for your SAS/ACCESS interface, or use an alias in the DISCONNECT statement.

Note: If you used the CONNECT statement to connect to the DBMS, the DISCONNECT statement's DBMS name or alias must match the name or alias that you specified in the CONNECT statement. △

alias
specifies an alias that was defined in the CONNECT statement.

Example

To exit the Pass-Through Facility, use the facilities DISCONNECT statement and then QUIT the PROC SQL statement. The following example disconnects the user from a DB2 database with the alias DBCON1 and terminates the SQL procedure:

```
proc sql;
connect to db2 as dbcon1 (ssid=db2a);
...more SAS statements...
disconnect from dbcon1;
quit;
```

EXECUTE Statement

Sends DBMS-specific, non-query SQL statements to the DBMS

Valid in: PROC SQL steps (when accessing DBMS data using SAS/ACCESS software)

Syntax

EXECUTE (*dbms-specific-sql-statement*) BY *dbms-name* | *alias*;

The EXECUTE statement sends dynamic non-query, DBMS-specific SQL statements to the DBMS and processes those statements.

In some SAS/ACCESS interfaces, you can issue an EXECUTE statement directly without first explicitly connecting to a DBMS (see "CONNECT Statement" on page 278). If you omit the CONNECT statement, an implicit connection is performed (by using default values for all database connection arguments) when the first EXECUTE statement is passed to the DBMS. See the documentation for your SAS/ACCESS interface for details.

The EXECUTE statement cannot be stored as part of an Pass-Through Facility query in a PROC SQL view.

Arguments

(*dbms-specific-sql-statement*)
a dynamic non-query, DBMS-specific SQL statement. This argument is required and must be enclosed in parentheses. However, the SQL statement cannot contain a semicolon because a semicolon represents the end of a statement in SAS. The SQL statement might be case-sensitive, depending on your DBMS, and it is passed to the DBMS exactly as you type it.

On some DBMSs, this argument can be a DBMS stored procedure. However, stored procedures with output parameters are not supported in the Pass-Through Facility. Furthermore, if the stored procedure contains more than one query, only the first query is processed.

Any return code or message that is generated by the DBMS is available in the macro variables SQLXRC and SQLXMSG after the statement executes. See "Macro Variables for Relational Databases" on page 261 for more information about these macro variables.

dbms-name
identifies the database management system to which you direct the DBMS-specific SQL statement. The keyword BY must appear before the *dbms-name* argument. You must specify either the DBMS name for your SAS/ACCESS interface or an alias.

alias
specifies an alias that was defined in the CONNECT statement. (You cannot use an alias if the CONNECT statement was omitted.)

Useful Statements to Include in EXECUTE Statements

You can pass the following statements to the DBMS by using the Pass-Through Facility's EXECUTE statement.

CREATE
creates a DBMS table, view, index, or other DBMS object, depending on how the statement is specified.

DELETE
deletes rows from a DBMS table.

DROP
deletes a DBMS table, view, or other DBMS object, depending on how the statement is specified.

GRANT
gives users the authority to access or modify objects such as tables or views.

INSERT
adds rows to a DBMS table.

REVOKE
revokes the access or modification privileges that were given to users by the GRANT statement.

UPDATE
modifies the data in one column of a row in a DBMS table.

For more information and restrictions on these and other SQL statements, see the SQL documentation for your DBMS.

CONNECTION TO Component

Retrieves and uses DBMS data in a PROC SQL query or view

Valid in: PROC SQL step SELECT statements (when accessing DBMS data using SAS/ACCESS software)

Syntax

CONNECTION TO *dbms-name* | *alias (dbms-query)*

The CONNECTION TO component specifies the DBMS connection that you want to use or that you want to create (if you have omitted the CONNECT statement). CONNECTION TO then enables you to retrieve DBMS data directly through a PROC SQL query.

You use the CONNECTION TO component in the FROM clause of a PROC SQL SELECT statement:

```
PROC SQL;
   SELECT column-list
      FROM CONNECTION TO dbms-name (dbms-query)
           other optional PROC SQL clauses
QUIT;
```

CONNECTION TO can be used in any FROM clause, including those in nested queries (that is, subqueries).

You can store a Pass-Through Facility query in an SQL view and then use that view in SAS programs. When you create an SQL view, any options that you specify in the corresponding CONNECT statement are stored too. Thus, when the SQL view is used in a SAS program, SAS can establish the appropriate connection to the DBMS.

On many relational databases, you can issue a CONNECTION TO component in a PROC SQL SELECT statement directly without first connecting to a DBMS (see "CONNECT Statement" on page 278). If you omit the CONNECT statement, an implicit connection is performed when the first PROC SQL SELECT statement that contains a CONNECTION TO component is passed to the DBMS. Default values are used for all DBMS connection arguments. See the documentation for your SAS/ACCESS interface for details.

Because relational databases and SAS have different naming conventions, some DBMS column names might be changed when you retrieve DBMS data through the CONNECTION TO component. See Chapter 2, "SAS Names and Support for DBMS Names," on page 7 for more information.

Arguments

dbms-name
identifies the database management system to which you direct the DBMS-specific SQL statement. See the documentation for your SAS/ACCESS interface for the name for your DBMS.

alias
specifies an alias, if one was defined in the CONNECT statement.

(dbms-query)
specifies the query that you are sending to the DBMS. The query can use any DBMS-specific SQL statement or syntax that is valid for the DBMS. However, the query cannot contain a semicolon because a semicolon represents the end of a statement in SAS.

You must specify a query argument in the CONNECTION TO component, and the query must be enclosed in parentheses. The query is passed to the DBMS exactly as you type it. Therefore, if your DBMS is case-sensitive, you must use the correct case for DBMS object names.

On some DBMSs, the *dbms-query* argument can be a DBMS stored procedure. However, stored procedures with output parameters are not supported in the Pass-Through Facility. Furthermore, if the stored procedure contains more than one query, only the first query is processed.

Example

After you connect (explicitly using the CONNECT statement or implicitly using default settings) to a DBMS, you can send a DBMS-specific SQL query to the DBMS using the facilities CONNECTION TO component. You issue a SELECT statement to indicate which columns you want to retrieve, identify your DBMS (such as Oracle or DB2), and issue your query using the SQL syntax of your DBMS.

The following example sends an Oracle SQL query, shown by highlighting, to the Oracle database for processing. The results from the Oracle SQL query serve as a virtual table for the PROC SQL FROM clause. In this example, MYCON is a connection alias.

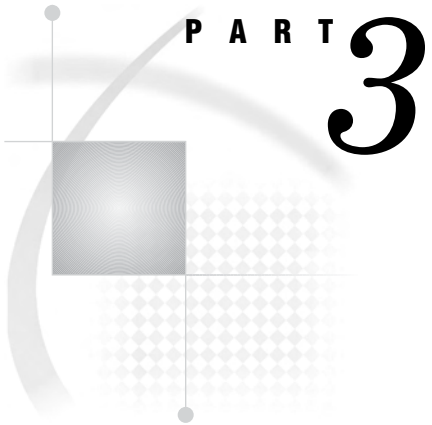
```
proc sql;
connect to oracle as mycon (user=testuser
```

```
   password=testpass path='myorapath');
%put &sqlxmsg;

select *
   from connection to mycon
    (select empid, lastname, firstname,
            hiredate, salary
     from employees where
          hiredate>='31-DEC-88');
%put &sqlxmsg;

disconnect from mycon;
quit;
```

The SAS %PUT macro displays the &SQLXMSG macro variable for error codes and information from the DBMS. See "Macro Variables for Relational Databases" on page 261 for more information.

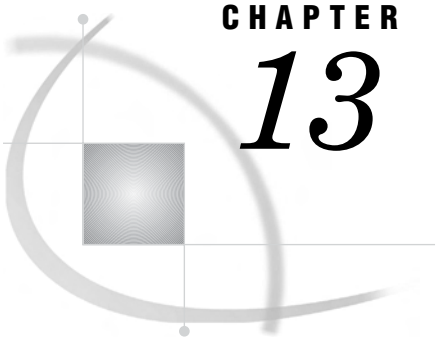
The following example gives the query a name and stores it as the SQL view samples.HIRES88:

```
libname samples 'SAS-data-library';

proc sql;
connect to oracle as mycon (user=testuser
   password=testpass path='myorapath');
%put &sqlxmsg;

create view samples.hires88 as
   select *
      from connection to mycon
         (select empid, lastname, firstname,
           hiredate, salary
          from employees where
               hiredate>='31-DEC-88');
%put &sqlxmsg;

disconnect from mycon;

quit;
```



PART 3

Sample Code



CHAPTER

13

Accessing DBMS Data with the LIBNAME Statement

About the LIBNAME Statement Sample Code

This section contains examples that use the LIBNAME statement to associate librefs with DBMS objects, such as tables and views. The LIBNAME statement is the recommended method for accessing DBMS data from within SAS.

These examples work with all of the SAS/ACCESS relational interfaces. To run these examples, complete the following steps:

1. Modify and submit the ACCAUTO.SAS file, which will create the appropriate LIBNAME statements for each database.
2. Submit the ACCDATA.sas program to create the DBMS tables and SAS data sets that the sample code uses.
3. Submit the ACCRUN.sas program to run the samples.

These programs are available in the SAS Sample Library. If you need assistance locating the Sample Library, contact your SAS support consultant. See "Descriptions of the Sample Data" on page 319 for information about the tables that are used in the sample code.

Note: Before you rerun an example that *updates* DBMS data, resubmit the ACCDATA.sas program to re-create the DBMS tables. △

Creating SAS Data Sets from DBMS Data

After you associate a SAS/ACCESS libref with your DBMS data, you can use the libref just as you would use any SAS libref. The following examples illustrate basic uses of the DATA step with librefs that reference DBMS data.

Using the PRINT Procedure with DBMS Data

In the following example, the interface to DB2 creates the libref MyDbLib and associates the libref with tables and views that reside on DB2. The DATA= option specifies a libref that references DB2 data. The PRINT procedure prints a New Jersey staff phone list from the DB2 table Staff. Information for staff from states other than New Jersey is not printed. The DB2 table Staff is not modified.

```
libname mydblib db2 ssid=db2;

proc print data=mydblib.staff
  (keep=lname fname state hphone);
  where state = 'NJ';
  title 'New Jersey Phone List';
run;
```

Output 13.1 Using the PRINT Procedure with DBMS Data

New Jersey Phone List 1

Obs	LNAME	FNAME	STATE	HPHONE
1	ALVAREZ	CARLOS	NJ	201/732-8787
2	BAREFOOT	JOSEPH	NJ	201/812-5665
3	DACKO	JASON	NJ	201/732-2323
4	FUJIHARA	KYOKO	NJ	201/812-0902
5	HENDERSON	WILLIAM	NJ	201/812-4789
6	JOHNSON	JACKSON	NJ	201/732-3678
7	LAWRENCE	KATHY	NJ	201/812-3337
8	MURPHEY	JOHN	NJ	201/812-4414
9	NEWKIRK	SANDRA	NJ	201/812-3331
10	NEWKIRK	WILLIAM	NJ	201/732-6611
11	PETERS	RANDALL	NJ	201/812-2478
12	RHODES	JEREMY	NJ	201/812-1837
13	ROUSE	JEREMY	NJ	201/732-9834
14	VICK	THERESA	NJ	201/812-2424
15	YANCEY	ROBIN	NJ	201/812-1874

Combining DBMS Data and SAS Data

The following example shows how to read DBMS data into SAS and create additional variables to perform calculations or subsetting operations on the data. The example creates the SAS data set Work.HighWage from the DB2 table Payroll and adds a new

variable, Category. The Category variable is based on the value of the salary column in the DB2 table Payroll. The Payroll table is not modified.

```
libname mydblib db2 ssid=db2;

data highwage;
  set mydblib.payroll(drop=sex birth hired);
  if salary>60000 then
    CATEGORY="High";
  else if salary<30000 then
    CATEGORY="Low";
  else
    CATEGORY="Avg";
run;

options obs=20;

proc print data=highwage;
  title "Salary Analysis";
  format salary dollar10.2;
run;
```

Output 13.2 Combining DBMS Data and SAS Data

Salary Analysis 1

OBS	IDNUM	JOBCODE	SALARY	CATEGORY
1	1919	TA2	$34,376.00	Avg
2	1653	ME2	$35,108.00	Avg
3	1400	ME1	$29,769.00	Low
4	1350	FA3	$32,886.00	Avg
5	1401	TA3	$38,822.00	Avg
6	1499	ME3	$43,025.00	Avg
7	1101	SCP	$18,723.00	Low
8	1333	PT2	$88,606.00	High
9	1402	TA2	$32,615.00	Avg
10	1479	TA3	$38,785.00	Avg
11	1403	ME1	$28,072.00	Low
12	1739	PT1	$66,517.00	High
13	1658	SCP	$17,943.00	Low
14	1428	PT1	$68,767.00	High
15	1782	ME2	$35,345.00	Avg
16	1244	ME2	$36,925.00	Avg
17	1383	BCK	$25,823.00	Low
18	1574	FA2	$28,572.00	Low
19	1789	SCP	$18,326.00	Low
20	1404	PT2	$91,376.00	High

Reading Data from Multiple DBMS Tables

You can use the DATA step to read data from multiple data sets. This example merges data from the two Oracle tables Staff and SuperV in the SAS data set Work.Combined.

```
libname mydblib oracle user=testuser password=testpass path='@alias';
```

```
data combined;
  merge mydblib.staff mydblib.superv(in=super
    rename=(supid=idnum));
  by idnum;
  if super;
run;

proc print data=combined;
  title "Supervisor Information";
run;
```

Output 13.3 Reading Data from Multiple DBMS Tables

Supervisor Information 1

Obs	IDNUM	LNAME	FNAME	CITY	STATE	HPHONE	JOBCAT
1	1106	MARSHBURN	JASPER	STAMFORD	CT	203/781-1457	PT
2	1118	DENNIS	ROGER	NEW YORK	NY	718/383-1122	PT
3	1126	KIMANI	ANNE	NEW YORK	NY	212/586-1229	TA
4	1352	RIVERS	SIMON	NEW YORK	NY	718/383-3345	NA
5	1385	RAYNOR	MILTON	BRIDGEPORT	CT	203/675-2846	ME
6	1401	ALVAREZ	CARLOS	PATERSON	NJ	201/732-8787	TA
7	1405	DACKO	JASON	PATERSON	NJ	201/732-2323	SC
8	1417	NEWKIRK	WILLIAM	PATERSON	NJ	201/732-6611	NA
9	1420	ROUSE	JEREMY	PATERSON	NJ	201/732-9834	ME
10	1431	YOUNG	DEBORAH	STAMFORD	CT	203/781-2987	FA
11	1433	YANCEY	ROBIN	PRINCETON	NJ	201/812-1874	FA
12	1442	NEWKIRK	SANDRA	PRINCETON	NJ	201/812-3331	PT
13	1564	WALTERS	ANNE	NEW YORK	NY	212/587-3257	SC
14	1639	CARTER-COHEN	KAREN	STAMFORD	CT	203/781-8839	TA
15	1677	KRAMER	JACKSON	BRIDGEPORT	CT	203/675-7432	BC
16	1834	LEBLANC	RUSSELL	NEW YORK	NY	718/384-0040	BC
17	1882	TUCKER	ALAN	NEW YORK	NY	718/384-0216	ME
18	1935	FERNANDEZ	KATRINA	BRIDGEPORT	CT	203/675-2962	NA
19	1983	DEAN	SHARON	NEW YORK	NY	718/384-1647	FA

Using the DATA Step's UPDATE Statement with DBMS Data

You can also use the DATA step's UPDATE statement to create a SAS data set with DBMS data. This example creates the SAS data set Work.Payroll with data from the Oracle tables Payroll and Payroll2. The Oracle tables are not modified.

The columns in the two Oracle tables must match; however, Payroll2 can have additional columns. Any additional columns in Payroll2 are added to the Payroll data set. The UPDATE statement requires unique values for IdNum to correctly merge the data from Payroll2.

```
libname mydblib oracle user=testuser password=testpass;

data payroll;
  update mydblib.payroll
         mydblib.payroll2;
  by idnum;

proc print data=payroll;
  format birth datetime9. hired datetime9.;
```

```
  title 'Updated Payroll Data';
run;
```

Output 13.4 Creating a SAS Data Set with DBMS Data by Using the UPDATE Statement

Updated Payroll Data 1

Obs	IDNUM	SEX	JOBCODE	SALARY	BIRTH	HIRED
1	1009	M	TA1	28880	02MAR1959	26MAR1992
2	1017	M	TA3	40858	28DEC1957	16OCT1981
3	1036	F	TA3	42465	19MAY1965	23OCT1984
4	1037	F	TA1	28558	10APR1964	13SEP1992
5	1038	F	TA1	26533	09NOV1969	23NOV1991
6	1050	M	ME2	35167	14JUL1963	24AUG1986
7	1065	M	ME3	38090	26JAN1944	07JAN1987
8	1076	M	PT1	69742	14OCT1955	03OCT1991
9	1094	M	FA1	22268	02APR1970	17APR1991
10	1100	M	BCK	25004	01DEC1960	07MAY1988
11	1101	M	SCP	18723	06JUN1962	01OCT1990
12	1102	M	TA2	34542	01OCT1959	15APR1991
13	1103	F	FA1	23738	16FEB1968	23JUL1992
14	1104	M	SCP	17946	25APR1963	10JUN1991
15	1105	M	ME2	34805	01MAR1962	13AUG1990
16	1106	M	PT3	94039	06NOV1957	16AUG1984
17	1107	M	PT2	89977	09JUN1954	10FEB1979
18	1111	M	NA1	40586	14JUL1973	31OCT1992
19	1112	M	TA1	26905	29NOV1964	07DEC1992
20	1113	F	FA1	22367	15JAN1968	17OCT1991

Using the SQL Procedure with DBMS Data

Rather than performing operations on your data in SAS, you can perform operations on data directly in your DBMS by using the LIBNAME statement and the SQL procedure. The following examples use the SQL procedure to query, update, and create DBMS tables.

Querying a DBMS Table

This example uses the SQL procedure to query the Oracle table Payroll. The PROC SQL query retrieves all job codes and provides a total salary amount for each job code.

```
libname mydblib oracle user=testuser password=testpass;

title 'Total Salary by Jobcode';

proc sql;
  select jobcode label='Jobcode',
         sum(salary) as total
         label='Total for Group'
         format=dollar11.2
  from mydblib.payroll
  group by jobcode;
quit;
```

Output 13.5 Querying a DBMS Table

```
                Total Salary by Jobcode

                                Total for
                 Jobcode            Group

                     BCK       $232,148.00
                     FA1       $253,433.00
                     FA2       $447,790.00
                     FA3       $230,537.00
                     ME1       $228,002.00
                     ME2       $498,076.00
                     ME3       $296,875.00
                     NA1       $210,161.00
                     NA2       $157,149.00
                     PT1       $543,264.00
                     PT2       $879,252.00
                     PT3        $21,009.00
                     SCP       $128,162.00
                     TA1       $249,492.00
                     TA2       $671,499.00
                     TA3       $476,155.00
```

The next example uses the SQL procedure to query flight information from the Oracle table Delay. The WHERE clause specifies that only flights to London and Frankfurt are retrieved.

```
libname mydblib oracle user=testuser password=testpass;

 title 'Flights to London and Frankfurt';

 proc sql;
   select dates format=datetime9.,
     dest from mydblib.delay
   where (dest eq "FRA") or
     (dest eq "LON")
   order by dest;
quit;
```

Note: By default, both the WHERE clause and the ORDER BY clause are processed by the DBMS for optimized performance. See "Overview of Optimizing Your SQL Usage" on page 37 for more information. △

Output 13.6 Querying a DBMS Table with a WHERE clause

```
              Flights to London and Frankfurt

                         DATES  DEST

                       01MAR1998  FRA
                       04MAR1998  FRA
                       07MAR1998  FRA
                       03MAR1998  FRA
                       05MAR1998  FRA
                       02MAR1998  FRA
                       04MAR1998  LON
                       07MAR1998  LON
                       02MAR1998  LON
                       06MAR1998  LON
                       05MAR1998  LON
                       03MAR1998  LON
                       01MAR1998  LON
```

The next example uses the SQL procedure to query the DB2 table InterNat for information about international flights with over 200 passengers. Note that the output is sorted by using a PROC SQL query and that the TITLE, LABEL, and FORMAT keywords are not ANSI standard SQL; they are SAS extensions that you can use in PROC SQL.

```
libname mydblib db2 ssid=db2;

proc sql;
   title  'International Flights by Flight Number';
   title2 'with Over 200 Passengers';
   select flight   label="Flight Number",
          dates    label="Departure Date"
                   format datetime9.,
          dest     label="Destination",
          boarded  label="Number Boarded"
     from mydblib.internat
    where boarded > 200
    order by flight;
quit;
```

Output 13.7 Querying a DBMS Table with SAS Extensions

```
            International Flights by Flight Number
                   with Over 200 Passengers

          Flight  Departure                 Number
          Number       Date  Destination   Boarded
          ----------------------------------------
          219     04MAR1998  LON               232
          219     07MAR1998  LON               241
          622     07MAR1998  FRA               210
          622     01MAR1998  FRA               207
```

Querying Multiple DBMS Tables

You can also retrieve data from multiple DBMS tables in a single query by using the SQL procedure. This example joins the Oracle tables Staff and Payroll to query salary information for employees who earn more than $40,000.

```
libname mydblib oracle user=testuser password=testpass;

title 'Employees with salary greater than $40,000';

options obs=20;

proc sql;
  select a.lname, a.fname, b.salary
    format=dollar10.2
  from mydblib.staff a, mydblib.payroll b
  where (a.idnum eq b.idnum) and
    (b.salary gt 40000);
quit;
```

Note: By default, SAS/ACCESS passes the entire join to the DBMS for processing in order to optimize performance. See "Passing Joins to the DBMS" on page 38 for more information. △

Output 13.8 Querying Multiple Oracle Tables

```
Employees with salary greater than $40,000
LNAME            FNAME                SALARY
-------------------------------------------
WELCH            DARIUS            $40,858.00
VENTER           RANDALL           $66,558.00
THOMPSON         WAYNE             $89,977.00
RHODES           JEREMY            $40,586.00
DENNIS           ROGER             $111379.00
KIMANI           ANNE              $40,899.00
O'NEAL           BRYAN             $40,079.00
RIVERS           SIMON             $53,798.00
COHEN            LEE               $91,376.00
GREGORSKI        DANIEL            $68,096.00
NEWKIRK          WILLIAM           $52,279.00
ROUSE            JEREMY            $43,071.00
```

The next example uses the SQL procedure to join and query the DB2 tables March, Delay, and Flight. The query retrieves information about delayed international flights during the month of March.

```
libname mydblib db2 ssid=db2;

title "Delayed International Flights in March";

proc sql;
  select distinct march.flight, march.dates format datetime9.,
      delay format=2.0
    from mydblib.march, mydblib.delay,
      mydblib.internat
```

```
  where march.flight=delay.flight and
      march.dates=delay.dates and
      march.flight=internat.flight and
      delay>0
  order by delay descending;
quit;
```

Note: By default, SAS/ACCESS passes the entire join to the DBMS for processing in order to optimize performance. See "Passing Joins to the DBMS" on page 38 for more information. △

Output 13.9 Querying Multiple DB2 Tables

```
          Delayed International Flights in March

                FLIGHT      DATES  DELAY
                ------------------------
                622     04MAR1998     30
                219     06MAR1998     27
                622     07MAR1998     21
                219     01MAR1998     18
                219     02MAR1998     18
                219     07MAR1998     15
                132     01MAR1998     14
                132     06MAR1998      7
                132     03MAR1998      6
                271     01MAR1998      5
                132     02MAR1998      5
                271     04MAR1998      5
                271     05MAR1998      5
                271     02MAR1998      4
                219     03MAR1998      4
                271     07MAR1998      4
                219     04MAR1998      3
                132     05MAR1998      3
                219     05MAR1998      3
                271     03MAR1998      2
```

The next example uses the SQL procedure to retrieve the combined results of two queries to the Oracle tables Payroll and Payroll2. An OUTER UNION in PROC SQL concatenates the data.

```
libname mydblib oracle user=testuser password=testpass;

title "Payrolls 1 & 2";

proc sql;
  select idnum, sex, jobcode, salary,
         birth format datetime9., hired format datetime9.
     from mydblib.payroll
  outer union corr
     select *
        from mydblib.payroll2
        order by idnum, jobcode, salary;
quit;
```

Output 13.10 Querying Multiple DBMS Tables

```
                        Payrolls 1 & 2                1

IDNUM   SEX  JOBCODE     SALARY       BIRTH       HIRED
-------------------------------------------------------
1009    M    TA1          28880   02MAR1959   26MAR1992
1017    M    TA3          40858   28DEC1957   16OCT1981
1036    F    TA3          39392   19MAY1965   23OCT1984
1036    F    TA3          42465   19MAY1965   23OCT1984
1037    F    TA1          28558   10APR1964   13SEP1992
1038    F    TA1          26533   09NOV1969   23NOV1991
1050    M    ME2          35167   14JUL1963   24AUG1986
1065    M    ME2          35090   26JAN1944   07JAN1987
1065    M    ME3          38090   26JAN1944   07JAN1987
1076    M    PT1          66558   14OCT1955   03OCT1991
1076    M    PT1          69742   14OCT1955   03OCT1991
1094    M    FA1          22268   02APR1970   17APR1991
1100    M    BCK          25004   01DEC1960   07MAY1988
1101    M    SCP          18723   06JUN1962   01OCT1990
1102    M    TA2          34542   01OCT1959   15APR1991
1103    F    FA1          23738   16FEB1968   23JUL1992
1104    M    SCP          17946   25APR1963   10JUN1991
1105    M    ME2          34805   01MAR1962   13AUG1990
```

Updating DBMS Data

In addition to querying data, you can also update data directly in your DBMS. You can update rows, columns, and tables by using the SQL procedure. The following example adds a new row to the DB2 table SuperV.

```
libname mydblib db2 ssid=db2;

proc sql;
insert into mydblib.superv
  values('1588','NY','FA');
quit;

proc print data=mydblib.superv;
  title "New Row in AIRLINE.SUPERV";
run;
```

Note: Depending on how your DBMS processes insert, the new row might not be added as the last physical row of the table. △

Output 13.11 Adding to DBMS Data

New Row in AIRLINE.SUPERV 1

OBS	SUPID	STATE	JOBCAT
1	1677	CT	BC
2	1834	NY	BC
3	1431	CT	FA
4	1433	NJ	FA
5	1983	NY	FA
6	1385	CT	ME
7	1420	NJ	ME
8	1882	NY	ME
9	1935	CT	NA
10	1417	NJ	NA
11	1352	NY	NA
12	1106	CT	PT
13	1442	NJ	PT
14	1118	NY	PT
15	1405	NJ	SC
16	1564	NY	SC
17	1639	CT	TA
18	1401	NJ	TA
19	1126	NY	TA
20	1588	NY	FA

The next example deletes all employees who work in Connecticut from the DB2 table Staff.

```
libname mydblib db2 ssid=db2;

proc sql;
  delete from mydblib.staff
    where state='CT';
quit;

options obs=20;

proc print data=mydblib.staff;
   title "AIRLINE.STAFF After Deleting Connecticut Employees";
run;
```

Note: If you omit a WHERE clause when you delete rows from a table, all rows in the table are deleted. △

Output 13.12 Deleting DBMS Data

AIRLINE.STAFF After Deleting Connecticut Employees 1

OBS	IDNUM	LNAME	FNAME	CITY	STATE	HPHONE
1	1400	ALHERTANI	ABDULLAH	NEW YORK	NY	212/586-0808
2	1350	ALVAREZ	MERCEDES	NEW YORK	NY	718/383-1549
3	1401	ALVAREZ	CARLOS	PATERSON	NJ	201/732-8787
4	1499	BAREFOOT	JOSEPH	PRINCETON	NJ	201/812-5665
5	1101	BAUCOM	WALTER	NEW YORK	NY	212/586-8060
6	1402	BLALOCK	RALPH	NEW YORK	NY	718/384-2849
7	1479	BALLETTI	MARIE	NEW YORK	NY	718/384-8816
8	1739	BRANCACCIO	JOSEPH	NEW YORK	NY	212/587-1247
9	1658	BREUHAUS	JEREMY	NEW YORK	NY	212/587-3622
10	1244	BUCCI	ANTHONY	NEW YORK	NY	718/383-3334
11	1383	BURNETTE	THOMAS	NEW YORK	NY	718/384-3569
12	1574	CAHILL	MARSHALL	NEW YORK	NY	718/383-2338
13	1789	CARAWAY	DAVIS	NEW YORK	NY	212/587-9000
14	1404	COHEN	LEE	NEW YORK	NY	718/384-2946
15	1065	COPAS	FREDERICO	NEW YORK	NY	718/384-5618
16	1876	CHIN	JACK	NEW YORK	NY	212/588-5634
17	1129	COUNIHAN	BRENDA	NEW YORK	NY	718/383-2313
18	1988	COOPER	ANTHONY	NEW YORK	NY	212/587-1228
19	1405	DACKO	JASON	PATERSON	NJ	201/732-2323
20	1983	DEAN	SHARON	NEW YORK	NY	718/384-1647

Creating a DBMS Table

You can create new tables in your DBMS by using the SQL procedure. This example uses the SQL procedure to create the Oracle table GTForty by using data from the Oracle Staff and Payroll tables.

```
libname mydblib oracle user=testuser password=testpass;

proc sql;
  create table mydblib.gtforty as
  select lname as lastname,
         fname as firstname,
         salary as Salary
           format=dollar10.2
  from mydblib.staff a,
       mydblib.payroll b
  where (a.idnum eq b.idnum) and
        (salary gt 40000);
quit;

options obs=20;

proc print data=mydblib.gtforty noobs;
  title 'Employees with salaries over $40,000';
  format salary dollar10.2;
run;
```

Output 13.13 Creating a DBMS Table

```
     Employees with salaries over $40,000

LASTNAME            FIRSTNAME               SALARY

WELCH               DARIUS              $40,858.00
VENTER              RANDALL             $66,558.00
MARSHBURN           JASPER              $89,632.00
THOMPSON            WAYNE               $89,977.00
RHODES              JEREMY              $40,586.00
KIMANI              ANNE                $40,899.00
CASTON              FRANKLIN            $41,690.00
STEPHENSON          ADAM                $42,178.00
BANADYGA            JUSTIN              $88,606.00
O'NEAL              BRYAN               $40,079.00
RIVERS              SIMON               $53,798.00
MORGAN              ALFRED              $42,264.00
```

Using Other SAS Procedures with DBMS Data

The following examples illustrate basic uses of other SAS procedures with librefs that refer to DBMS data.

Using the MEANS Procedure

This example uses the PRINT and MEANS procedures on a SAS data set created from the Oracle table March. The MEANS procedure provides information about the largest number of passengers on each flight.

```
libname mydblib oracle user=testuser password=testpass;

title 'Number of Passengers per Flight by Date';

proc print data=mydblib.march noobs;
  var dates boarded;
  by flight dest;
  sumby flight;
  sum boarded;
  format dates datetime9.;
run;

title 'Maximum Number of Passengers per Flight';

proc means data=mydblib.march fw=5 maxdec=1 max;
  var boarded;
  class flight;
run;
```

Output 13.14 Using the PRINT and MEANS Procedures

```
                    Number of Passengers per Flight by Date

------------------------------ FLIGHT=132 DEST=YYZ -------------------------------

                                     DATE    BOARDED

                                  01MAR1998      115
                                  02MAR1998      106
                                  03MAR1998       75
                                  04MAR1998      117
                                  05MAR1998      157
                                  06MAR1998      150
                                  07MAR1998      164
                                  ---------    -------
                                     FLIGHT      884


------------------------------ FLIGHT=219 DEST=LON -------------------------------

                                     DATE    BOARDED

                                  01MAR1998      198
                                  02MAR1998      147
                                  03MAR1998      197
                                  04MAR1998      232
                                  05MAR1998      160
                                  06MAR1998      163
                                  07MAR1998      241
                                  ---------    -------
                                     FLIGHT     1338
```

```
                  Maximum Number of Passengers per Flight

                            The MEANS Procedure

                       Analysis Variable : BOARDED

                         FLIGHT   N Obs        Max
                         132          7      164.0

                         219          7      241.0
```

Using the DATASETS Procedure

This example uses the DATASETS procedure to view a list of DBMS tables, in this case, in an Oracle database.

Note: The MODIFY and ALTER statements in PROC DATASETS are not available for use with librefs that refer to DBMS data. △

```
libname mydblib oracle user=testuser password=testpass;

title 'Table Listing';

proc datasets lib=mydblib;
  contents data=_all_ nods;
run;
```

Output 13.15 Using the DATASETS Procedure

```
    Table Listing
 The DATASETS Procedure

-----Directory-----

Libref:         MYDBLIB
Engine:         Oracle
Physical Name:
Schema/User:    testuser


#  Name        Memtype
----------------------
1  BIRTHDAY    DATA
2  CUST        DATA
3  CUSTOMERS   DATA
4  DELAY       DATA
5  EMP         DATA
6  EMPLOYEES   DATA
7  FABORDER    DATA
8  INTERNAT    DATA
9  INVOICES    DATA
10 INVS        DATA
```

Using the CONTENTS Procedure

This example shows output from the CONTENTS procedure when it is run on a DBMS table. Note that PROC CONTENTS shows all of the SAS metadata that is derived from the DBMS table by the SAS/ACCESS interface.

```
libname mydblib oracle user=testuser password=testpass;

title 'Contents of the DELAY Table';

proc contents data=mydblib.delay;
run;
```

Output 13.16 Using the CONTENTS Procedure

```
              Contents of the DELAY Table

                The CONTENTS Procedure

Data Set Name: MYDBLIB.DELAY          Observations:         .
Member Type:   DATA                   Variables:            7
Engine:        Oracle                 Indexes:              0
Created:       .                      Observation Length:   0
Last Modified: .                      Deleted Observations: 0
Protection:                           Compressed:           NO
Data Set Type:                        Sorted:               NO
Label:


          -----Alphabetic List of Variables and Attributes-----

#    Variable    Type    Len    Pos    Format         Informat       Label
-----------------------------------------------------------------------------
2    DATES       Num       8      8    DATETIME20.    DATETIME20.    DATES
7    DELAY       Num       8     64                                  DELAY
5    DELAYCAT    Char     15     32    $15.           $15.           DELAYCAT
4    DEST        Char      3     24    $3.            $3.            DEST
6    DESTYPE     Char     15     48    $15.           $15.           DESTYPE
1    FLIGHT      Char      3      0    $3.            $3.            FLIGHT
3    ORIG        Char      3     16    $3.            $3.            ORIG
```

Using the RANK Procedure

This example uses the RANK procedure to rank flights in the DB2 table Delay by number of minutes delayed.

```
libname mydblib db2 ssid=db2;

options obs=20;

proc rank data=mydblib.delay descending
    ties=low out=ranked;
  var delay;
  ranks RANKING;
run;

proc print data=ranked;
  title 'Ranking of Delayed Flights';
  format delay 2.0
         dates datetime9.;
run;
```

Output 13.17 Using the RANK Procedure

Ranking of Delayed Flights 1

OBS	FLIGHT	DATES	ORIG	DEST	DELAYCAT	DESTYPE	DELAY	RANKING
1	114	01MAR1998	LGA	LAX	1-10 Minutes	Domestic	8	9
2	202	01MAR1998	LGA	ORD	No Delay	Domestic	-5	42
3	219	01MAR1998	LGA	LON	11+ Minutes	International	18	4
4	622	01MAR1998	LGA	FRA	No Delay	International	-5	42
5	132	01MAR1998	LGA	YYZ	11+ Minutes	International	14	8
6	271	01MAR1998	LGA	PAR	1-10 Minutes	International	5	13
7	302	01MAR1998	LGA	WAS	No Delay	Domestic	-2	36
8	114	02MAR1998	LGA	LAX	No Delay	Domestic	0	28
9	202	02MAR1998	LGA	ORD	1-10 Minutes	Domestic	5	13
10	219	02MAR1998	LGA	LON	11+ Minutes	International	18	4
11	622	02MAR1998	LGA	FRA	No Delay	International	0	28
12	132	02MAR1998	LGA	YYZ	1-10 Minutes	International	5	13
13	271	02MAR1998	LGA	PAR	1-10 Minutes	International	4	19
14	302	02MAR1998	LGA	WAS	No Delay	Domestic	0	28
15	114	03MAR1998	LGA	LAX	No Delay	Domestic	-1	32
16	202	03MAR1998	LGA	ORD	No Delay	Domestic	-1	32
17	219	03MAR1998	LGA	LON	1-10 Minutes	International	4	19
18	622	03MAR1998	LGA	FRA	No Delay	International	-2	36
19	132	03MAR1998	LGA	YYZ	1-10 Minutes	International	6	12
20	271	03MAR1998	LGA	PAR	1-10 Minutes	International	2	25

Using the TABULATE Procedure

This example uses the TABULATE procedure on the Oracle table Payroll to display a chart of the number of employees for each job code.

```
libname mydblib oracle user=testuser password=testpass;

title "Number of Employees by Jobcode";

proc tabulate data=mydblib.payroll format=3.0;
   class jobcode;
   table jobcode*n;
   keylabel n="#";
run;
```

Output 13.18 Using the TABULATE Procedure

Number of Employees by Jobcode 1

JOBCODE															
BCK	FA1	FA2	FA3	ME1	ME2	ME3	NA1	NA2	PT1	PT2	PT3	SCP	TA1	TA2	TA3
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#
9	11	16	7	8	14	7	5	3	8	10	2	7	9	20	12

Using the APPEND Procedure

In this example, the DB2 table Payroll2 is appended to the DB2 table Payroll with the APPEND procedure. The Payroll table is updated on DB2.

Note: When you append data to a DBMS table, you are actually inserting rows into a table. The rows can be inserted into the DBMS table in any order. △

```
libname mydblib db2 ssid=db2;

proc append base=mydblib.payroll
            data=mydblib.payroll2;
run;

proc print data=mydblib.payroll;
  title 'PAYROLL After Appending PAYROLL2';
  format birth datetime9. hired datetime9.;
run;
```

Note: In cases where a DBMS table that you are using is in the same database space as a table that you are creating or updating, you must use the LIBNAME option CONNECTION=SHARED to prevent a deadlock. △

Output 13.19 Using the APPEND Procedure

PAYROLL After Appending PAYROLL2 1

OBS	IDNUM	SEX	JOBCODE	SALARY	BIRTH	HIRED
1	1919	M	TA2	34376	12SEP1960	04JUN1987
2	1653	F	ME2	35108	15OCT1964	09AUG1990
3	1400	M	ME1	29769	05NOV1967	16OCT1990
4	1350	F	FA3	32886	31AUG1965	29JUL1990
5	1401	M	TA3	38822	13DEC1950	17NOV1985
6	1499	M	ME3	43025	26APR1954	07JUN1980
7	1101	M	SCP	18723	06JUN1962	01OCT1990
8	1333	M	PT2	88606	30MAR1961	10FEB1981
9	1402	M	TA2	32615	17JAN1963	02DEC1990
10	1479	F	TA3	38785	22DEC1968	05OCT1989
11	1403	M	ME1	28072	28JAN1969	21DEC1991
12	1739	M	PT1	66517	25DEC1964	27JAN1991
13	1658	M	SCP	17943	08APR1967	29FEB1992
14	1428	F	PT1	68767	04APR1960	16NOV1991
15	1782	M	ME2	35345	04DEC1970	22FEB1992
16	1244	M	ME2	36925	31AUG1963	17JAN1988
17	1383	M	BCK	25823	25JAN1968	20OCT1992
18	1574	M	FA2	28572	27APR1960	20DEC1992
19	1789	M	SCP	18326	25JAN1957	11APR1978
20	1404	M	PT2	91376	24FEB1953	01JAN1980

Calculating Statistics from DBMS Data

This example uses the FREQ procedure to calculate statistics on the DB2 table Invoices.

```
libname mydblib db2 ssid=db2;

proc freq data=mydblib.invoices(keep=invnum country);
  tables country;
  title 'Invoice Frequency by Country';
run;
```

The following output shows the one-way frequency table that this example generates.

Output 13.20 Using the FREQ Procedure

```
                          Invoice Frequency by Country                    1
                            The FREQ Procedure

                                 COUNTRY

                                                      Cumulative    Cumulative
COUNTRY                 Frequency     Percent     Frequency      Percent
-----------------------------------------------------------------------------
Argentina                       2       11.76             2        11.76
Australia                       1        5.88             3        17.65
Brazil                          4       23.53             7        41.18
USA                            10       58.82            17       100.00
```

Selecting and Combining DBMS Data

This example uses a WHERE statement in a DATA step to create a list that includes only unpaid bills over $300,000.

```
libname mydblib oracle user=testuser password=testpass;

proc sql;
  create view allinv as
      select paidon, billedon, invnum, amtinus, billedto
      from mydblib.invoices
quit;

data notpaid (keep=invnum billedto amtinus billedon);
   set allinv;
      where paidon is missing and amtinus>=300000.00;
run;

proc print data=notpaid label;
  format amtinus dollar20.2 billedon datetime9.;
  label  amtinus=amountinus billedon=billedon
         invnum=invoicenum billedto=billedto;
  title 'High Bills--Not Paid';
run;
```

Output 13.21 Using the WHERE Statement

```
                          High Bills--Not Paid                        1

       Obs     billedon    invoicenum              amountinus    billedto

         1    05OCT1998       11271            $11,063,836.00    18543489
         2    10OCT1998       11286            $11,063,836.00    43459747
         3    02NOV1998       12051             $2,256,870.00    39045213
         4    17NOV1998       12102            $11,063,836.00    18543489
         5    27DEC1998       12471             $2,256,870.00    39045213
         6    24DEC1998       12476             $2,256,870.00    38763919
```

Joining DBMS and SAS Data

This example shows how to combine SAS and DBMS data using the SAS/ACCESS LIBNAME statement. The example creates a SQL view, Work.Emp_Csr, from the DB2 table Employees and joins the view with a SAS data set, TempEmps, to select only interns who are family members of existing employees.

```
libname mydblib db2 ssid=db2;

title 'Interns Who Are Family Members of Employees';

proc sql;
 create view emp_csr as
 select * from mydblib.employees
  where dept in ('CSR010', 'CSR011', 'CSR004');

  select tempemps.lastname, tempemps.firstnam,
         tempemps.empid, tempemps.familyid,
         tempemps.gender, tempemps.dept,
         tempemps.hiredate
    from emp_csr, samples.tempemps
    where emp_csr.empid=tempemps.familyid;

quit;
```

Output 13.22 Combining an SQL View with a SAS Data Set

```
                Interns Who Are Family Members of Employees                 1

   lastname          firstnam   empid    familyid    gender   dept     hiredate
   ---------------------------------------------------------------------------
   SMITH             ROBERT     765112   234967      M        CSR010   04MAY1998
   NISHIMATSU-LYNCH  RICHARD    765111   677890      M        CSR011   04MAY1998
```

CHAPTER

14

Accessing DBMS Data with the Pass-Through Facility

About the Pass-Through Facility Sample Code

The examples in this section demonstrate how to use the Pass-Through Facility to access and update DBMS data. The Pass-Through Facility enables you to read and write data between SAS and a DBMS, however, it is recommended that you use the LIBNAME statement to access your DBMS data more easily and directly.

To run these examples, complete the following steps:

1. Modify and submit the ACCAUTO.SAS file, which will create the appropriate LIBNAME statements for each database.
2. Submit the ACCDATA.sas program to create the DBMS tables and SAS data sets that the sample code uses.
3. Submit the ACCRUN.sas program to run the samples.

These programs are available in the SAS Sample Library. If you need assistance locating the Sample Library, contact your SAS support consultant. See “Descriptions of the Sample Data” on page 319 for information about the tables that are used in the sample code.

Note: Before you rerun an example that *updates* DBMS data, resubmit the ACCDATA.sas program to re-create the DBMS tables. △

Retrieving DBMS Data with a Pass-Through Query

This section describes how to retrieve DBMS data by using the statements and components of the Pass-Through Facility. The following example, creates a brief listing of the companies who have received invoices, the amount of the invoices, and the dates on which the invoices were sent. This example accesses Oracle data.

First, the code specifies a PROC SQL CONNECT statement to connect to a particular Oracle database that resides on a remote server. It refers to the database with the alias MyDb. Then it lists the columns to select from the Oracle tables in the PROC SQL SELECT clause.

Note: If desired, you can use a column list that follows the table alias, such as **`as t1(invnum,billedon,amtinus,name)`** to rename the columns. This is not necessary,

however. If you choose to rename the columns by using a column list, you must specify them in the same order in which they appear in the SELECT statement in the Pass-Through query, so that the columns map one-to-one. When you use the new names in the first SELECT statement, you can specify the names in any order. Add the NOLABEL option to the query to display the renamed columns. △

The PROC SQL SELECT statement uses a CONNECTION TO component in the FROM clause to retrieve data from the Oracle table. The Pass-Through query (in italics) is enclosed in parentheses and uses Oracle column names. This query joins data from the Invoices and Customers tables by using the BilledTo column, which references the primary key column Customers.Customer. In this Pass-Through query, Oracle can take advantage of its keyed columns to join the data in the most efficient way. Oracle then returns the processed data to SAS.

Note: The order in which processing occurs is not the same as the order of the statements in the example. The first SELECT statement (the PROC SQL query) displays and formats the data that is processed and returned to SAS by the second SELECT statement (the Pass-Through query). △

```
options linesize=120;

proc sql;
connect to oracle as mydb (user=testuser password=testpass);
%put &sqlxmsg;

title 'Brief Data for All Invoices';
   select invnum, name, billedon format=datetime9.,
          amtinus format=dollar20.2
   from connection to mydb
     (select invnum, billedon, amtinus, name
       from invoices, customers
       where invoices.billedto=customers.customer
       order by billedon, invnum);
%put &sqlxmsg;

disconnect from mydb;
quit;
```

The SAS %PUT statement writes the contents of the &SQLXMSG macro variable to the SAS log so that you can check it for error codes and descriptive information from the Pass-Through Facility. The DISCONNECT statement terminates the Oracle connection and the QUIT statement ends the SQL procedure.

The following output shows the results of the Pass-Through query.

Output 14.1 Data Retrieved by a Pass-Through Query

Brief Data for All Invoices

INVOICENUM	NAME	BILLEDON	AMTINUS
11270	LABORATORIO DE PESQUISAS VETERINARIAS DESIDERIO FINAMOR	05OCT1998	$2,256,870.00
11271	LONE STAR STATE RESEARCH SUPPLIERS	05OCT1998	$11,063,836.00
11273	TWENTY-FIRST CENTURY MATERIALS	06OCT1998	$252,148.50
11276	SANTA CLARA VALLEY TECHNOLOGY SPECIALISTS	06OCT1998	$1,934,460.00
11278	UNIVERSITY BIOMEDICAL MATERIALS	06OCT1998	$1,400,825.00
11280	LABORATORIO DE PESQUISAS VETERINARIAS DESIDERIO FINAMOR	07OCT1998	$2,256,870.00
11282	TWENTY-FIRST CENTURY MATERIALS	07OCT1998	$252,148.50
11285	INSTITUTO DE BIOLOGIA Y MEDICINA NUCLEAR	10OCT1998	$2,256,870.00
11286	RESEARCH OUTFITTERS	10OCT1998	$11,063,836.00
11287	GREAT LAKES LABORATORY EQUIPMENT MANUFACTURERS	11OCT1998	$252,148.50
12051	LABORATORIO DE PESQUISAS VETERINARIAS DESIDERIO FINAMOR	02NOV1998	$2,256,870.00
12102	LONE STAR STATE RESEARCH SUPPLIERS	17NOV1998	$11,063,836.00
12263	TWENTY-FIRST CENTURY MATERIALS	05DEC1998	$252,148.50
12468	UNIVERSITY BIOMEDICAL MATERIALS	24DEC1998	$1,400,825.00
12476	INSTITUTO DE BIOLOGIA Y MEDICINA NUCLEAR	24DEC1998	$2,256,870.00
12478	GREAT LAKES LABORATORY EQUIPMENT MANUFACTURERS	24DEC1998	$252,148.50
12471	LABORATORIO DE PESQUISAS VETERINARIAS DESIDERIO FINAMOR	27DEC1998	$2,256,870.00

The following example changes the Pass-Through query into an SQL view by adding a CREATE VIEW statement to the query, removing the ORDER BY clause from the CONNECTION TO component, and adding the ORDER BY clause to a separate SELECT statement that prints only the new SQL view. *

```
libname samples 'your-SAS-data-library';

proc sql;
connect to oracle as mydb (user=testuser password=testpass);
%put &sqlxmsg;

   create view samples.brief as
     select invnum, name, billedon format=datetime9.,
            amtinus format=dollar20.2
         from connection to mydb
            (select invnum, billedon, amtinus, name
               from invoices, customers
               where invoices.billedto=customers.customer);
%put &sqlxmsg;

disconnect from mydb;

options ls=120 label;

title 'Brief Data for All Invoices';
select * from samples.brief
   order by billedon, invnum;

quit;
```

The output from the Samples.Brief view is the same as shown in Output 14.1.

* If you have data that is usually sorted, it is more efficient to keep the ORDER BY clause in the Pass-Through query and let the DBMS sort the data.

When an SQL view is created from a Pass-Through query, the query's DBMS connection information is stored with the view. Therefore, when you reference the SQL view in a SAS program, you automatically connect to the correct database, and you retrieve the most current data in the DBMS tables.

Combining an SQL View with a SAS Data Set

The following example joins SAS data with Oracle data that is retrieved by using a Pass-Through query in a PROC SQL SELECT statement.

Information about student interns is stored in the SAS data file, Samples.TempEmps. The Oracle data is joined with this SAS data file to determine whether any of the student interns have a family member who works in the CSR departments.

To join the data from Samples.TempEmps with the data from the Pass-Through query, you assign a table alias (Query1) to the query. Doing so enables you to qualify the query's column names in the WHERE clause.

```
options ls=120;

title 'Interns Who Are Family Members of Employees';

proc sql;
connect to oracle as mydb;
%put &sqlxmsg;

select tempemps.lastname, tempemps.firstnam, tempemps.empid,
       tempemps.familyid, tempemps.gender, tempemps.dept,
       tempemps.hiredate
   from connection to mydb
      (select * from employees) as query1, samples.tempemps
   where query1.empid=tempemps.familyid;
%put &sqlxmsg;

disconnect from mydb;
quit;
```

Output 14.2 Combining a PROC SQL View with a SAS Data Set

```
                Interns Who Are Family Members of Employees                1

lastname          firstnam   empid    familyid    gender   dept     hiredate
-------------------------------------------------------------------------------
SMITH             ROBERT     765112   234967      M        CSR010   04MAY1998
NISHIMATSU-LYNCH  RICHARD    765111   677890      M        CSR011   04MAY1998
```

Note: When SAS data is joined to DBMS data through a Pass-Through query, PROC SQL cannot optimize the query. In this case it is much more efficient to use a SAS/ACCESS LIBNAME statement. Another way to increase efficiency is to extract the DBMS data and place it in a new SAS data file, assign SAS indexes to the appropriate variables, and join the two SAS data files. △

Using a Pass-Through Query in a Subquery

The following example shows how to use a subquery that contains a Pass-Through query. A subquery is a nested query and is usually part of a WHERE or HAVING clause. Summary functions cannot appear in a WHERE clause, so using a subquery is often a good technique. A subquery is contained in parentheses and returns one or more values to the outer query for further processing.

This example creates an SQL view, Samples.AllEmp, based on Sybase data. Sybase objects, such as table names and columns, are case sensitive. Database identification statements and column names are converted to uppercase unless they are enclosed in quotation marks.

The outer PROC SQL query retrieves data from the SQL view; the subquery uses a Pass-Through query to retrieve data. This query returns the names of employees who earn less than the average salary for each department. The macro variable, Dept, substitutes the department name in the query.

```
libname mydblib sybase server=server1 database=personnel
   user=testuser password=testpass;
libname samples 'your-SAS-data-library';

/* create SQL view */
proc sql;

   create view samples.allemp as
      select * from mydblib.employees;

quit;

/* use the Pass-Through Facility to retrieve data */
proc sql stimer;

title "Employees Who Earn Below the &dept Average Salary";

connect to sybase(server=server1 database=personnel
   user=testuser password=testpass);
%put &sqlxmsg;

%let dept='ACC%';

select empid, lastname
   from samples.allemp
   where dept like &dept and salary <
         (select avg(salary) from connection to sybase
                (select SALARY from EMPLOYEES
                    where DEPT like &dept));
%put &sqlxmsg;
disconnect from sybase;
quit;
```

When a PROC SQL query contains subqueries or inline views, the innermost query is evaluated first. In this example, data is retrieved from the Employees table and returned to the subquery for further processing. Notice that the Pass-Through query is enclosed in parentheses (in italics) and another set of parentheses encloses the entire subquery.

When a comparison operator such as < or > is used in a WHERE clause, the subquery must return a single value. In this example, the AVG summary function returns the average salary of employees in the department, $57,840.86. This value is inserted in the query, as if the query were written:

```
where dept like &dept and salary < 57840.86;
```

Employees who earn less than the department's average salary are listed in the following output.

Output 14.3 Output from a Pass-Through Query in a Subquery

```
Employees Who Earn Below the 'ACC%' Average Salary

                 EMPID  LASTNAME
                -----------------
                123456  VARGAS
                135673  HEMESLY
                423286  MIFUNE
                457232  LOVELL
```

It might appear to be more direct to omit the Pass-Through query and to instead access Samples.AllEmp a second time in the subquery, as if the query were written as follows:

```
%let dept='ACC%';

proc sql stimer;
select empid, lastname
   from samples.allemp
   where dept like &dept and salary <
         (select avg(salary) from samples.allemp
             where dept like &dept);
quit;
```

However, as the SAS log below indicates, the PROC SQL query with the Pass-Through subquery performs better. (The STIMER option on the PROC SQL statement provides statistics on the SAS process.)

Output 14.4 SAS Log Comparing the Two PROC SQL Queries

```
213
214  %let dept='ACC%';
215
216  select empid, lastname, firstnam
217     from samples.allemp
218     where dept like &dept and salary <
219           (select avg(salary)
220               from connection to sybase
221                  (select SALARY from EMPLOYEES
222                      where DEPT like &dept));
NOTE: The SQL Statement used 0:00:00.2 real 0:00:00.20 cpu.
223  %put &sqlxmsg;

224  disconnect from sybase;
NOTE: The SQL Statement used 0:00:00.0 real 0:00:00.0 cpu.
225  quit;
NOTE: The PROCEDURE SQL used 0:00:00.0 real 0:00:00.0 cpu.

226
227  %let dept='ACC%';
228
229  proc sql stimer;
NOTE: The SQL Statement used 0:00:00.0 real 0:00:00.0 cpu.
230  select empid, lastname, firstnam
231     from samples.allemp
232     where dept like &dept and salary <
233           (select avg(salary)
234               from samples.allemp
235               where dept like &dept);
NOTE: The SQL Statement used 0:00:06.0 real 0:00:00.20 cpu.
```

CHAPTER

15

Sample Data for SAS/ACCESS for Relational Databases

Introduction to the Sample Data

This section provides information about the DBMS tables that are used in the LIBNAME statement and Pass-Through Facility sample code chapters. The sample code uses tables that contain fictitious airline and textile industry data to show how the SAS/ACCESS interfaces work with data that is stored in relational DBMS tables.

Descriptions of the Sample Data

The following PROC CONTENTS output excerpts describe the DBMS tables and SAS data sets that are used in the sample code.

Output 15.1 Description of the March DBMS Data

```
-----Alphabetic List of Variables and Attributes-----
  #    Variable    Type    Len    Pos    Format     Informat

  7    boarded     Num       8     24
  8    capacity    Num       8     32
  2    dates       Num       8      0    DATE9.     DATE7.
  3    depart      Num       8      8    TIME5.     TIME5.
  5    dest        Char      3     46
  1    flight      Char      3     40
  6    miles       Num       8     16
  4    orig        Char      3     43
```

Output 15.2 Description of the Delay DBMS Data

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Format    Informat

 2    dates       Num       8      0    DATE9.    DATE7.
 7    delay       Num       8      8
 5    delaycat    Char     15     25
 4    dest        Char      3     22
 6    destype     Char     15     40
 1    flight      Char      3     16
 3    orig        Char      3     19
```

Output 15.3 Description of the InterNat DBMS Data

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Format    Informat

 4    boarded     Num       8      8
 2    dates       Num       8      0    DATE9.    DATE7.
 3    dest        Char      3     19
 1    flight      Char      3     16
```

Output 15.4 Description of the Schedule DBMS Data

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Format    Informat

 2    dates       Num       8      0    DATE9.    DATE7.
 3    dest        Char      3     11
 1    flight      Char      3      8
 4    idnum       Char      4     14
```

Output 15.5 Description of the Payroll DBMS Data

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Format    Informat

 5    birth       Num       8      8    DATE9.    DATE7.
 6    hired       Num       8     16    DATE9.    DATE7.
 1    idnum       Char      4     24
 3    jobcode     Char      3     29
 4    salary      Num       8      0
 2    sex         Char      1     28
```

Output 15.6 Description of the Payroll2 DBMS Data

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Format    Informat

 5    birth       Num       8      8    DATE9.    DATE7.
 6    hired       Num       8     16    DATE9.    DATE7.
 1    idnum       Char      4     24
 3    jobcode     Char      3     29
 4    salary      Num       8      0
 2    sex         Char      1     28
```

Output 15.7 Description of the Staff DBMS Data

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos

 4    city        Char     15     34
 3    fname       Char     15     19
 6    hphone      Char     12     51
 1    idnum       Char      4      0
 2    lname       Char     15      4
 5    state       Char      2     49
```

Output 15.8 Description of the Superv DBMSData

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Label

 3    jobcat      Char      2      6    Job Category
 2    state       Char      2      4
 1    supid       Char      4      0    Supervisor Id
```

Output 15.9 Description of the Invoices DBMSData

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Format

 3    AMTBILL     Num       8      8
 5    AMTINUS     Num       8     16
 6    BILLEDBY    Num       8     24
 7    BILLEDON    Num       8     32    DATE9.
 2    BILLEDTO    Char      8     48
 4    COUNTRY     Char     20     56
 1    INVNUM      Num       8      0
 8    PAIDON      Num       8     40    DATE9.
```

Output 15.10 Description of the Employees DBMS Data

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Format

 7    BIRTHDTE    Num       8     32    DATE9.
 4    DEPT        Char      6     40
 1    EMPID       Num       8      0
 9    FRSTNAME    Char     15     65
 6    GENDER      Char      1     46
 2    HIREDATE    Num       8      8    DATE9.
 5    JOBCODE     Num       8     24
 8    LASTNAME    Char     18     47
10    MIDNAME     Char     15     80
11    PHONE       Char      4     95
 3    SALARY      Num       8     16
```

Output 15.11 Description of the Customers DBMS Data

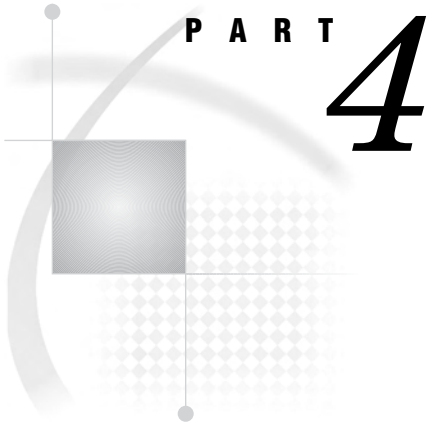
```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Format

 8    ADDRESS     Char     40    145
 9    CITY        Char     25    185
 7    CONTACT     Char     30    115
 4    COUNTRY     Char     20     23
 1    CUSTOMER    Char      8      8
10    FIRSTORD    Num       8      0    DATE9.
 6    NAME        Char     60     55
 5    PHONE       Char     12     43
 2    STATE       Char      2     16
 3    ZIPCODE     Char      5     18
```

Output 15.12 Description of the Faborder DBMS Data

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Format

 6    DATEORD     Num       8     32    DATE9.
 4    FABCHARG    Num       8     24
 3    LENGTH      Num       8     16
 1    ORDERNUM    Num       8      0
 9    PROCSBY     Num       8     56
 7    SHIPPED     Num       8     40    DATE9.
 5    SHIPTO      Char      8     64
10    SPECFLAG    Char      1     72
 2    STOCKNUM    Num       8      8
 8    TAKENBY     Num       8     48
```

Output 15.13 Description of the TempEmps SAS Data Set

```
-----Alphabetic List of Variables and Attributes-----
 #    Variable    Type    Len    Pos    Format    Informat

 3    dept        Char      6     24
 1    empid       Num       8      0
 8    familyid    Num       8     16
 6    firstnam    Char     15     49
 4    gender      Char      1     30
 2    hiredate    Num       8      8    DATE9.    DATE.
 5    lastname    Char     18     31
 7    middlena    Char     15     64
```



PART 4

Converting SAS/ACCESS Descriptors to SQL Views

CHAPTER 16

The CV2VIEW Procedure

Overview of the CV2VIEW Procedure

The CV2VIEW procedure converts SAS/ACCESS view descriptors and access descriptors into SQL views. We recommend that you consider converting your descriptors for the following reasons:

- Descriptors are no longer the recommended method for accessing relational database data. Converting to SQL views enables you to use the LIBNAME statement, which is the preferred method. The LIBNAME statement provides greater control over DBMS operations such as locking, spooling, and data type conversions. The LIBNAME statement can also handle long field names, whereas descriptors cannot.
- SQL views are platform-independent. SAS/ACCESS descriptors are not.

It is important to note that the CV2VIEW procedure in SAS 9.1 will work for converting both 64-bit SAS/ACCESS view descriptors (created in either 64-bit SAS Version 8 or 64-bit SAS 9.1) and 32-bit SAS/ACCESS access descriptors (created in 32-bit SAS Version 6 and Version 8).

If the descriptor that you want to convert is READ, WRITE, or ALTER protected, then those values are applied to the output SQL view. For security reasons, these values do not appear if you save the generated SQL to a file. The PASSWORD portion of the LIBNAME statement is also not visible. This prevents the generated SQL statements from being manually submitted without modification.

Procedure Syntax

The syntax for the CV2VIEW procedure is as follows:

PROC CV2VIEW DBMS= *dbms-name* | ALL;

FROM_VIEW= *libref.input-descriptor*;

FROM_LIBREF= *input-library*;

TO_VIEW= *libref.output-view*;

TO_LIBREF= *output-library*;

TYPE= SQL | VIEW | ACCESS;

SAVEAS= *external-filename*;

SUBMIT;

REPLACE ALL | VIEW | FILE;

PROC CV2VIEW Statement

PROC CV2VIEW DBMS= *dbms-name* | ALL;

Arguments

dbms-name
specifies the name of a supported database from which you want to obtain descriptors. Valid values for *dbms-name* are **DB2**, **Oracle**, and **SYBASE**.

ALL
specifies that you want the descriptors from all supported databases.

FROM_VIEW= Statement

Specifies the name of the view descriptor or access descriptor that you want to convert

Requirement: You must specify either the FROM_VIEW= statement or the FROM_LIBREF= statement.

Requirement: The FROM_VIEW= and TO_VIEW= statements are always used together.

Restriction: If you specify DBMS=ALL, then you cannot use the FROM_VIEW= statement.

FROM_VIEW=*libref.input-descriptor*;

Arguments

libref
specifies the libref that contains the view descriptor or access descriptor that you want to convert.

input-descriptor
specifies the view descriptor or access descriptor that you want to convert.

FROM_LIBREF= Statement

Specifies the library that contains the view descriptors or access descriptors that you want to convert

Requirement: You must specify either the FROM_VIEW= statement or the FROM_LIBREF= statement.

Requirement: The FROM_LIBREF= and TO_LIBREF= statements are always used together.

FROM_LIBREF= *input-library*;

Argument

input-library
specifies a previously assigned library that contains the view descriptors or access descriptors that you want to convert. All descriptors that are in the specified library and that access data in the specified DBMS are converted into SQL views. If you specify DBMS=ALL, then all descriptors that are in the specified library and that access any supported DBMS are converted.

REPLACE= Statement

Specifies whether existing views and files are replaced

REPLACE= ALL | FILE | VIEW ;

Arguments

ALL
replaces the TO_VIEW= file if it already exists and replaces the SAVEAS= file if it already exists.

FILE
replaces the SAVEAS= file if it already exists. If the file already exists, and if REPLACE=FILE or REPLACE=ALL are not specified, the generated PROC SQL code is appended to the file.

VIEW
replaces the TO_VIEW= file if it already exists.

SAVEAS= Statement

Saves the generated PROC SQL statements to a file

Interaction: If you specify the SAVEAS= statement, the generated SQL is not automatically submitted, so you must use the SUBMIT statement.

SAVEAS=*external-filename*;

Argument

external-filename
enables you to save the PROC SQL statements that are generated by PROC CV2VIEW to an external file. You can modify this file and submit it on another platform.

Details

PROC CV2VIEW inserts comments in the generated SQL to replace any statements that contain passwords. For example, if a view descriptor is READ, WRITE, or ALTER protected, then the output view has the same level of security. However, the file that contains the SQL statements does not show the password values. This is also the case for the password in the LIBNAME statement.

SUBMIT Statement

Causes PROC CV2VIEW to submit the generated PROC SQL statements when you specify the SAVEAS= statement

Tip: If you do not use the SAVEAS= statement, PROC CV2VIEW automatically submits the generated SQL, so you do not need to specify the SUBMIT statement.

SUBMIT;

TO_VIEW= Statement

Specifies the name of the new (converted) SQL view

Requirement: You must specify either the TO_VIEW= statement or the TO_LIBREF= statement.

Requirement: The FROM_VIEW= and TO_VIEW= statements are always used together.

Restriction: If you specify DBMS=ALL, then you cannot use the TO_VIEW= statement.

Interaction: Use the REPLACE= statement to control whether the output file is overwritten or appended if it already exists.

TO_VIEW=*libref.output-view*;

Arguments

libref
specifies the libref where you will store the new SQL view.

output-view
specifies the name for the new SQL view that you want to create.

TO_LIBREF= Statement

Specifies the library that contains the new (converted) SQL views

Requirement: You must specify either the TO_VIEW= statement or the TO_LIBREF= statement.

Requirement: The FROM_LIBREF= and TO_LIBREF= statements are always used together.

Interaction: Use the REPLACE= statement if a file with the name of one of your output views already exists. If a file with the name of one of your output views already exists and you do not specify the REPLACE statement, PROC CV2VIEW does not convert that view.

TO_LIBREF= *output-library*;

Argument

output-library
specifies the name of a previously assigned library where you want to store the new SQL views.

Details

The names of the input view descriptors or access descriptors are used as the output view names. In order to individually name your output views, use the FROM_VIEW= statement and the TO_VIEW= statement.

TYPE= Statement

Specifies what type of conversion should occur

TYPE= SQL | VIEW | ACCESS;

Arguments

SQL
specifies that PROC CV2VIEW converts descriptors to SQL views. This is the default behavior.

VIEW
specifies that PROC CV2VIEW converts descriptors to native view descriptor format. This is most useful in the 32-bit to 64-bit case. This will not convert view descriptors across different operating systems.

ACCESS
specifies that PROC CV2VIEW converts access descriptors to native access descriptor format. This is most useful in the 32-bit to 64-bit case. This will not convert access descriptors across different operating systems.

Details

When TYPE=VIEW or TYPE=ACCESS, SAVEAS=, SUBMIT, and REPLACE= or REPLACE_FILE= are not valid options.

CV2VIEW Procedure Examples

Example 1: Converting an Individual View Descriptor

In this example, PROC CV2VIEW converts the MYVIEW view descriptor to the SQL view NEWVIEW. The MYVIEW view descriptor is ALTER, READ, and WRITE protected. The PROC SQL statements that are generated by PROC CV2VIEW are submitted and saved to an external file named SQL.SAS.

```
libname input '/username/descriptors/';
libname output '/username/sqlviews/';

proc cv2view dbms=oracle;
from_view = input.myview (alter=apwd);
to_view = output.newview;
saveas = '/username/vsql/sql.sas';
submit;
replace file;
run;
```

PROC CV2VIEW generates the following PROC SQL statements.

```
/* SOURCE DESCRIPTOR: MYVIEW */
PROC SQL DQUOTE=ANSI;
  CREATE VIEW OUTPUT.NEWVIEW
  (
/* READ=  */
/* WRITE= */
/* ALTER= */
  LABEL=EMPLINFO
  )
  AS SELECT
     "EMPLOYEE  " AS EMPLOYEE INFORMAT= 5.0 FORMAT= 5.0
                  LABEL= 'EMPLOYEE  ' ,
     "LASTNAME  " AS LASTNAME INFORMAT= $10. FORMAT= $10.
                  LABEL= 'LASTNAME  ' ,
     "SEX       " AS SEX INFORMAT= $6. FORMAT= $6.
                  LABEL= 'SEX       ' ,
     "STATUS    " AS STATUS INFORMAT= $9. FORMAT= $9.
                  LABEL= 'STATUS    ' ,
     "DEPARTMENT" AS DEPARTME INFORMAT= 7.0 FORMAT= 7.0
                  LABEL= 'DEPARTMENT' ,
     "CITYSTATE " AS CITYSTAT INFORMAT= $15. FORMAT= $15.
                  LABEL= 'CITYSTATE '
  FROM _CVLIB_."EMPLINFO"
  USING LIBNAME _CVLIB_
  Oracle
/* PW=  */
  USER=ordevxx PATH=OracleV8 PRESERVE_TAB_NAMES=YES;
  QUIT;
```

The REPLACE FILE statement causes an existing file named SQL.SAS to be overwritten. Without this statement the text would be appended to SQL.SAS if the user has the appropriate privileges.

The LABEL value of **EMPLINFO** is the name of the underlying database table that is referenced by the view descriptor.

If the underlying DBMS is Oracle or DB2, the CV2VIEW procedure adds the PRESERVE_TAB_NAMES= option to the embedded LIBNAME statement to enable you to access those tables that have mixed-case or embedded-blank table names.

Note: This SQL syntax fails if you try to submit it because the PW field of the LIBNAME statement is replaced with a comment in order to protect the password. The ALTER, READ, and WRITE protection is commented out for the same reason. You can add the passwords to the code and then submit the SQL to re-create the view. △

Example 2: Converting a Library of View Descriptors for a Single DBMS

In this example, PROC CV2VIEW converts all of the Oracle view descriptors in the input library into SQL views. If an error occurs during the conversion of a view descriptor, the procedure moves to the next view. The PROC SQL statements that are generated by PROC CV2VIEW are both submitted and saved to an external file named SQL.SAS.

```
libname input '/username/descriptors/';
libname output '/username/sqlviews/';
```

```
proc cv2view dbms=oracle;
from_libref = input;
to_libref = output;
saveas = '/username/vsql/manyview.sas';
submit;
run;
```

PROC CV2VIEW generates the following PROC SQL statements for one of the views.

```
/* SOURCE DESCRIPTOR: PPCV2R */
PROC SQL DQUOTE=ANSI;
  CREATE VIEW OUTPUT.PPCV2R
  (
  LABEL=EMPLOYEES
  )
  AS SELECT
     "EMPID     " AS EMPID INFORMAT= BEST22. FORMAT= BEST22.
                  LABEL= 'EMPID     ' ,
     "HIREDATE  " AS HIREDATE INFORMAT= DATETIME16. FORMAT= DATETIME16.
                  LABEL= 'HIREDATE  ' ,
     "JOBCODE   " AS JOBCODE INFORMAT= BEST22. FORMAT= BEST22.
                  LABEL= 'JOBCODE   ' ,
     "SEX       " AS SEX INFORMAT= $1. FORMAT= $1.
                  LABEL= 'SEX       '
  FROM _CVLIB_."EMPLOYEES" (
      SASDATEFMT = ( "HIREDATE"= DATETIME16. ) )
  USING LIBNAME _CVLIB_
  Oracle
  /* PW= */
  USER=ordevxx PATH=OracleV8 PRESERVE_TAB_NAMES=YES;
  QUIT;
```

The SAVEAS= statement causes all of the generated SQL for all of the Oracle view descriptors to be stored in the MANYVIEW.SAS file.

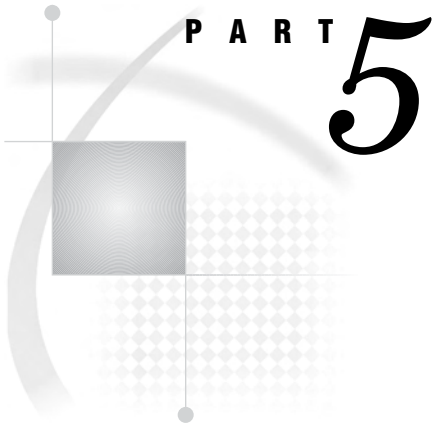
If the underlying DBMS is Oracle or DB2, the CV2VIEW procedure adds the PRESERVE_TAB_NAMES= option to the embedded LIBNAME statement to enable you to access those tables that have mixed-case or embedded-blank table names.

Example 3: Converting a Library of View Descriptors for All Supported DBMSs

In this example, PROC CV2VIEW converts all of the view descriptors that are in the input library and that access data in any supported DBMS. If an error occurs during the conversion of a view descriptor, then the procedure moves to the next view. The PROC SQL statements that are generated by PROC CV2VIEW are automatically submitted but are not saved to an external file (because the SAVEAS= statement is not used).

```
libname input '/username/descriptors/';
libname output '/username/sqlviews/';
```

```
proc cv2view dbms=all;
from_libref = input;
to_libref = output;
run;
```



PART 5

Appendixes

APPENDIX

1

The ACCESS Procedure for Relational Databases

Overview of the ACCESS Procedure for Relational Databases

Accessing DBMS Data

Although the ACCESS procedure is still supported for the database systems and environments on which it was available in Version 6, it is no longer the recommended method for accessing relational DBMS data. It is recommended that you access your DBMS data more directly, using the LIBNAME statement or Pass-Through Facility.

Note: Not all SAS/ACCESS interfaces support this feature. See Chapter 8, "SAS/ACCESS Features by Host," on page 65 to determine whether this feature is available in your environment. △

This section provides general reference information for the ACCESS procedure; see the SAS/ACCESS documentation for your DBMS for DBMS-specific details.

The ACCESS procedure, along with the DBLOAD procedure and an interface view engine, creates an interface between SAS and data in other vendors' databases. The ACCESS procedure enables you to create and update descriptors.

About ACCESS Procedure Statements

The ACCESS procedure has several types of statements:

- *Database connection statements* are used to connect to your DBMS. For details, see the SAS/ACCESS documentation for your DBMS.
- *Creating and updating statements* are CREATE and UPDATE.
- *Table and editing statements* include ASSIGN, DROP, FORMAT, LIST, QUIT, RENAME, RESET, SELECT, SUBSET, TABLE, and UNIQUE.

The following table summarizes the PROC ACCESS options and statements that are required to accomplish common tasks.

Table A1.1 Statement Sequence for Accomplishing Tasks with the ACCESS Procedure

To do this...	Use these statements and options
Create an access descriptor	**PROC ACCESS** *statement-options*; **CREATE** *libref.member-name*.ACCESS; *database-connection-statements*; *editing-statements*; **RUN**;
Create an access descriptor and a view descriptor	**PROC ACCESS** *statement-options*; **CREATE** *libref.member-name*.ACCESS; *database-connection-statements*; *editing-statements*; **CREATE** *libref.member-name*.VIEW; **SELECT** *column-list*; *editing-statements*; **RUN**;
Create a view descriptor from an existing access descriptor	**PROC ACCESS** *statement-options*, including ACCDESC=*libref.access-descriptor*; **CREATE** *libref.member-name*.VIEW; **SELECT** *column-list*; *editing-statements*; **RUN**;
Update an access descriptor	**PROC ACCESS** *statement-options*; **UPDATE** *libref.member-name*.ACCESS; *database-connection-statements*; *editing-statements*; **RUN**;

To do this...	Use these statements and options
Update an access descriptor and a view descriptor	**PROC ACCESS** *statement-options*; **UPDATE** *libref.member-name*.ACCESS; *database-connection-statements*; *editing-statements*; **UPDATE** *libref.member-name*.VIEW; *editing-statements*; **RUN**;
Update an access descriptor and create a view descriptor	**PROC ACCESS** *statement-options*; **UPDATE** *libref.member-name*.ACCESS; *database-connection-statements*; *editing-statements*; **CREATE** *libref.member-name*.VIEW; **SELECT** *column-list*; *editing-statements*; **RUN**;
Update a view descriptor from an existing access descriptor	**PROC ACCESS** *statement-options*, including ACCDESC=*libref.access-descriptor*; **UPDATE** *libref.member-name*.VIEW; *editing-statements*; **RUN**;
Create a SAS data set from a view descriptor	**PROC ACCESS** *statement-options*, including DBMS=*dbms-name*; VIEWDESC=*libref.member*; OUT=*libref.member*; **RUN**;

Procedure Syntax

The general syntax for the ACCESS procedure is presented here; see the SAS/ACCESS documentation for your DBMS for DBMS-specific details.

PROC ACCESS<*options*>;

database-connection-statements;

CREATE *libref.member-name.*ACCESS | VIEW <*password-option*>;

UPDATE *libref.member-name.*ACCESS | VIEW <*password-option*>;

TABLE= <'>*table-name*<'>;

ASSIGN <=>YES | NO | Y | N;

DROP <'>*column-identifier-1*<'> <…<'>*column-identifier-n*<'>>;

FORMAT <'>*column-identifier-1*<'> <=> *SAS-format-name-1*
<…<'>*column-identifier-n*<'> <=> *SAS-format-name-n*>;

LIST <ALL | VIEW |<'>*column-identifier*<'>>;

QUIT;

RENAME <'>*column-identifier-1*<'> <=> *SAS-variable-name-1*
<…<'>*column-identifier-n*<'> <=> *SAS-variable-name-n*>;

RESET ALL |<'>*column-identifier-1*<'> <…<'>*column-identifier-n*<'>>;

SELECT ALL |<'>*column-identifier-1*<'> <…<'>*column-identifier-n*<'>>;

SUBSET *selection-criteria*;

UNIQUE <=> YES | NO | Y | N;

RUN;

PROC ACCESS Statement

PROC ACCESS <*options*>;

Options

ACCDESC=*libref.access-descriptor*
specifies an access descriptor. ACCDESC= is used with the DBMS= option to create or update a view descriptor that is based on the specified access descriptor. You can use a SAS data set option on the ACCDESC= option to specify any passwords that have been assigned to the access descriptor.

Note: The ODBC interface does not support this option. △

DBMS=*database-management-system*
specifies which database management system you want to use. This DBMS-specific option is required. See the SAS/ACCESS documentation for your DBMS.

OUT=*libref.member-name*
specifies the SAS data file to which DBMS data is output.

VIEWDESC=*libref.view-descriptor*
specifies a view descriptor through which you extract the DBMS data.

Database Connection Statements

Provide DBMS-specific connection information

database-connection-statements;

Database connection statements are used to connect to your DBMS. For the statements to use with your DBMS, see the documentation for your SAS/ACCESS interface.

ASSIGN Statement

Indicates whether SAS variable names and formats are generated

Applies to: access descriptor

Interacts with: FORMAT, RENAME, RESET, UNIQUE

Default: NO

ASSIGN <=>YES | NO | Y | N;

YES
generates unique SAS variable names from the first eight characters of the DBMS column names. If you specify **YES**, you cannot specify the RENAME, FORMAT, RESET or UNIQUE statements when you create view descriptors that are based on the access descriptor.

NO
enables you to modify SAS variable names and formats when you create an access descriptor and when you create view descriptors that are based on this access descriptor.

Details

The ASSIGN statement indicates whether SAS variable names are automatically generated and whether you can change SAS variable names and formats in the view descriptors that are created from the access descriptor. Each time the SAS/ACCESS interface encounters a CREATE statement to create an access descriptor, the ASSIGN statement is reset to the default **NO** value.

During an access descriptor's creation, you use the RENAME statement to change SAS variable names and the FORMAT statement to change SAS formats.

When you specify **YES**, names are generated according to the following rules:

- you can change the SAS variable names only in the access descriptor
- the SAS variable names that are saved in an access descriptor are *always* used when view descriptors are created from the access descriptor; you cannot change them in the view descriptors
- the ACCESS procedure allows names only up to eight characters.

CREATE Statement

Creates a SAS/ACCESS descriptor file

Applies to: access descriptor or view descriptor

CREATE *libref.member-name.*ACCESS | VIEW <*password-option*>;

libref.member-name
identifies the libref of the SAS data library where you will store the descriptor and identifies the descriptor's name.

ACCESS
specifies an access descriptor.

VIEW
specifies a view descriptor.

password-option
specifies a password.

Details

The CREATE statement is required. The CREATE statement names the access descriptor or view descriptor that you are creating. Use a three-level name where the first level identifies the libref of the SAS data library where you will store the descriptor, the second level is the descriptor's name, and the third level specifies the type of SAS file (specify **ACCESS** for an access descriptor or **VIEW** for a view descriptor).

See Table A1.1 on page 338 for the appropriate sequence of statements for creating access and view descriptors.

Example

The following example creates an access descriptor AdLib.Employ on the Oracle table Employees, and a view descriptor Vlib.Emp1204 based on AdLib.Employ, in the same PROC ACCESS step.

```
proc access dbms=oracle;

   /* create access descriptor  */

   create adlib.employ.access;
   database='qa:[dubois]textile';
   table=employees;
   assign=no;
   list all;

   /* create view descriptor  */

   create vlib.emp1204.view;
   select empid lastname hiredate salary dept
   gender birthdate;
   format empid 6.
          salary dollar12.2
          jobcode 5.
          hiredate datetime9.
          birthdate datetime9.;
   subset where jobcode=1204;
run;
```

DROP Statement

Drops a column so that it cannot be selected in a view descriptor

Applies to: access and view descriptors
Interacts with: RESET, SELECT

DROP <'>*column-identifier-1*<'> <…<'>*column-identifier-n*<'>>;

column-identifier
specifies the column name or the positional equivalent from the LIST statement, which is the number that represents the column's place in the access descriptor. For example, to drop the third and fifth columns, submit the following statement:

```
drop 3 5;
```

Details

The DROP statement drops the specified column(s) from a descriptor. You can drop a column when creating or updating an access descriptor; you can also drop a column when updating a view descriptor. If you drop a column when creating an access descriptor, you cannot select that column when creating a view descriptor that is based on the access descriptor. The underlying DBMS table is unaffected by this statement.

To display a column that was previously dropped, specify that column name in the RESET statement. However, doing so also resets all the column's attributes (such as SAS variable name, format, and so on) to their default values.

FORMAT Statement

Changes a SAS format for a DBMS column

Applies to: access descriptor or view descriptor
Interacts with: ASSIGN, DROP, RESET

FORMAT <'>*column-identifier-1*<'> <=>*SAS-format-name-1*
<…<'>*column-identifier-n*<'> <=> *SAS-format-name-n*>;

column-identifier
specifies the column name or the positional equivalent from the LIST statement, which is the number that represents the column's place in the access descriptor. If the column name contains lowercase characters, special characters, or national characters, enclose the name in quotation marks.

SAS-format-name
specifies the SAS format to be used.

Details

The FORMAT statement changes SAS variable formats from their default formats. The default SAS variable format is based on the data type of the DBMS column. See the SAS/ACCESS documentation for your DBMS for information about default formats that SAS assigns to your DBMS data types.

You can use the FORMAT statement with a view descriptor only if the ASSIGN statement that was used when creating the access descriptor was specified with the **NO** value. When you use the FORMAT statement with access descriptors, the FORMAT statement also reselects columns that were previously dropped with the DROP statement.

For example, to associate the DATE9. format with the BIRTHDATE column and with the second column in the access descriptor, submit the following statement:

```
format 2=date9. birthdate=date9.;
```

The equal sign (=) is optional. For example, you can use the FORMAT statement to specify new SAS variable formats for four DBMS table columns:

```
format productid  4.
       weight     e16.9
       fibersize  e20.13
       width      e16.9;
```

LIST Statement

Lists columns in the descriptor and gives information about them

Applies to: access descriptor or view descriptor

Default: ALL

LIST <ALL | VIEW |<'>*column-identifier*<'>>;

ALL
lists all the DBMS columns in the table, the positional equivalents, the SAS variable names, and the SAS variable formats that are available for a descriptor.

VIEW
lists all the DBMS columns that are selected for a view descriptor, their positional equivalents, their SAS names and formats, and any subsetting clauses.

column-identifier
lists information about a specified DBMS column, including its name, positional equivalent, SAS variable name and format, and whether it has been selected. If the column name contains lowercase characters, special characters, or national characters, enclose the name in quotation marks.

The *column-identifier* argument can be either the column name or the positional equivalent, which is the number that represents the column's place in the descriptor. For example, to list information about the fifth column in the descriptor, submit the following statement:

```
list 5;
```

Details

The LIST statement lists columns in the descriptor, along with information about the columns. The LIST statement can be used only when *creating* an access descriptor or a view descriptor. The LIST information is written to your SAS log.

Note: To review the contents of an existing view descriptor, use the CONTENTS procedure. △

When you use LIST for an access descriptor, ***NON-DISPLAY*** appears next to the column description for any column that has been dropped; ***UNSUPPORTED*** appears next to any column whose data type is not supported by your DBMS interface view engine. When you use LIST for a view descriptor, ***SELECTED*** appears next to the column description for columns that you have selected for the view.

Specify LIST last in your PROC ACCESS code in order to see the entire descriptor. If you are creating or updating multiple descriptors, specify LIST before each CREATE or UPDATE statement in order to list information about all of the descriptors that you are creating or updating.

QUIT Statement

Terminates the procedure

Applies to: access descriptor or view descriptor

QUIT;

Details

The QUIT statement terminates the ACCESS procedure without any further descriptor creation. Changes made since the last CREATE, UPDATE, or RUN statement are not saved; changes are saved only when a new CREATE, UPDATE, or RUN statement is submitted.

RENAME Statement

Modifies the SAS variable name

Applies to: access descriptor or view descriptor

Interacts with: ASSIGN, RESET

RENAME <'>*column-identifier-1*<'> <=> *SAS-variable-name-1*
<…<'>*column-identifier-n*<'> <=> *SAS-variable-name-n*>;

column-identifier
specifies the DBMS column name or the positional equivalent from the LIST statement, which is the number that represents the column's place in the descriptor. If the column name contains lowercase characters, special characters, or national characters, enclose the name in quotation marks. The equal sign (=) is optional.

SAS-variable-name
specifies a SAS variable name.

Details

The RENAME statement sets or modifies the SAS variable name that is associated with a DBMS column.

Two factors affect the use of the RENAME statement: whether you specify the ASSIGN statement when you are creating an access descriptor, and the kind of descriptor you are creating.

- If you omit the ASSIGN statement or specify it with a **NO** value, the renamed SAS variable names that you specify in the access descriptor are retained throughout an ACCESS procedure execution. For example, if you rename the CUSTOMER column to CUSTNUM when you create an access descriptor, that column continues to be named CUSTNUM when you select it in a view descriptor unless a RESET statement or another RENAME statement is specified.

  When creating a view descriptor that is based on this access descriptor, you can specify the RESET statement or another RENAME statement to rename the variable again, but the new name applies only in that view. When you create other view descriptors, the SAS variable names are derived from the access descriptor.
- If you specify the **YES** value in the ASSIGN statement, you can use the RENAME statement to change SAS variable names only while creating an access descriptor. As described earlier in the ASSIGN statement, SAS variable names and formats that are saved in an access descriptor are always used when creating view descriptors that are based on the access descriptor.

For example, to rename the SAS variable names that are associated with the seventh column and the nine-character FIRSTNAME column in a descriptor, submit the following statement:

```
rename
7 birthdy  'firstname'=fname;
```

Note: When you are creating a view descriptor, the RENAME statement automatically selects the renamed column for the view. That is, if you rename the SAS variable associated with a DBMS column, you do not have to issue a SELECT statement for that column. △

RESET Statement

Resets DBMS columns to their default settings

Applies to: access descriptor or view descriptor

Interacts with: ASSIGN, DROP, FORMAT, RENAME, SELECT

RESET ALL |<'>*column-identifier-1*<'> <…<'>*column-identifier-n*<'>>;

ALL
resets all columns in an access descriptor to their default names and formats and reselects any dropped columns. ALL deselects all columns in a view descriptor so that no columns are selected for the view.

column-identifier
can be either the DBMS column name or the positional equivalent from the LIST statement, which is the number that represents the column's place in the access descriptor. If the column name contains lowercase characters, special characters, or national characters, enclose the name in quotation marks. For example, to reset the SAS variable name and format associated with the third column, submit the following statement:

```
reset
3;
```

For access descriptors, the specified column is reset to its default name and format settings. For view descriptors, the specified column is no longer selected for the view.

Details

The RESET statement resets column attributes to their default values. This statement has different effects on access and view descriptors.

For *access descriptors*, the RESET statement resets the specified column names to the default names that are generated by the ACCESS procedure. The RESET statement also changes the current SAS variable format to the default SAS format. Any previously-dropped columns that are specified in the RESET statement become available.

Note: When creating an access descriptor, if you omit the ASSIGN statement or set it to **NO**, the default SAS variable names are blanks. If you set ASSIGN=YES, the default names are the first eight characters of each DBMS column name. △

For *view descriptors*, the RESET statement clears any columns that were included in the SELECT statement (that is, it deselects the columns). When you create a view descriptor that is based on an access descriptor created without an ASSIGN statement or with ASSIGN=NO, resetting and then reselecting (within the same procedure execution) a SAS variable changes the SAS variable names and formats to their default values. When you create a view descriptor that is based on an access descriptor created with ASSIGN=YES, the RESET statement does not have this effect.

SELECT Statement

Selects DBMS columns for the view descriptor

Applies to: view descriptor

Interacts with: RESET

SELECT ALL |<'>*column-identifier-1*<'> <…<'>*column-identifier-n* <'>>;

ALL
includes in the view descriptor all the columns that were defined in the access descriptor and that were not dropped.

column-identifier
can be either the DBMS column name or the positional equivalent from the LIST statement, which is the number that represents the column's place in the access descriptor on which the view is based. For example, to select the first three columns, submit the following statement:

```
select 1 2 3;
```

If the column name contains lowercase characters, special characters, or national characters, enclose the name in quotation marks.

Details

The SELECT statement is required. The SELECT statement specifies which DBMS columns in an access descriptor to include in a view descriptor.

SELECT statements are cumulative within a view creation. That is, if you submit the following two SELECT statements, columns 1, 5, and 6 are selected:

```
select 1;
select 5 6;
```

To clear your current selections when creating a view descriptor, use the **RESET ALL** statement.

SUBSET Statement

Adds or modifies selection criteria for a view descriptor

Applies to: view descriptor

SUBSET *selection-criteria*;

selection-criteria
one or more DBMS-specific SQL expressions that are accepted by your DBMS, such as WHERE, ORDER BY, HAVING, and GROUP BY. Use DBMS column names, not SAS variable names, in your selection criteria.

Details

You can use the SUBSET statement to specify selection criteria when you create a view descriptor. This statement is optional; if you omit it, the view retrieves all the data (that is, all the rows) in the DBMS table.

For example, for a view descriptor that retrieves rows from a DBMS table, you could submit the following SUBSET statement:

```
subset where firstorder is not null;
```

If you have multiple selection criteria, enter them all in one SUBSET statement, as in the following example:

```
subset where firstorder is not null
  and country = 'USA'
  order by country;
```

Unlike other ACCESS procedure statements, the SUBSET statement is ccase-sensitive. The SQL statement is sent to the DBMS exactly as you type it. Therefore, you must use the correct case for any DBMS object names. See the SAS/ACCESS documentation for your DBMS for details.

SAS does not check the SUBSET statement for errors. The statement is verified only when the view descriptor is used in a SAS program.

If you specify more than one SUBSET statement per view descriptor, the last SUBSET overwrites the earlier SUBSETs. To delete the selection criteria, submit a SUBSET statement without any arguments.

TABLE= Statement

Identifies the DBMS table on which the access descriptor is based

Applies to: access descriptor

TABLE= <'>*table-name*<'>;

table-name
a valid DBMS table name. If it contains lowercase characters, special characters, or national characters, you must enclose it in quotation marks. See the SAS/ACCESS documentation for your DBMS for details on the TABLE= statement.

Details

This statement is required with the CREATE statement and optional with the UPDATE statement.

UNIQUE Statement

Generates SAS variable names based on DBMS column names

Applies to: view descriptor

Interacts with: ASSIGN

UNIQUE <=> YES | NO | Y | N;

YES
causes the SAS/ACCESS interface to append numbers to any duplicate SAS variable names, thus making each variable name unique.

NO
causes the SAS/ACCESS interface to continue to allow duplicate SAS variable names to exist. You must resolve these duplicate names before saving (and thereby creating) the view descriptor.

Details

The UNIQUE statement specifies whether the SAS/ACCESS interface should generate unique SAS variable names for DBMS columns for which SAS variable names have not been entered.

The UNIQUE statement is affected by whether you specified the ASSIGN statement when you created the access descriptor on which the view is based:

- If you specified the ASSIGN=YES statement, you cannot specify UNIQUE when creating a view descriptor. **YES** causes SAS to generate unique names, so UNIQUE is not necessary.
- If you omitted the ASSIGN statement or specified ASSIGN=NO, you must resolve any duplicate SAS variable names in the view descriptor. You can use UNIQUE to generate unique names automatically, or you can use the RENAME statement to resolve duplicate names yourself. See "RENAME Statement" on page 345 for information.

If duplicate SAS variable names exist in the access descriptor on which you are creating a view descriptor, you can specify UNIQUE to resolve the duplication.

Note: It is recommended that you use the UNIQUE statement and specify UNIQUE=YES. If you omit the UNIQUE statement or specify UNIQUE=NO and SAS encounters duplicate SAS variable names in a view descriptor, your job fails. △

The equal sign (=) is optional in the UNIQUE statement.

UPDATE Statement

Updates a SAS/ACCESS descriptor file

Applies to: access descriptor or view descriptor

UPDATE *libref.member-name*.ACCESS | VIEW <*password-option*>;

libref.member-name
identifies the libref of the SAS data library where you will store the descriptor and identifies the descriptor's name.

ACCESS
specifies an access descriptor.

VIEW
specifies a view descriptor.

password-option
specifies a password.

Details

The UPDATE statement identifies an existing access descriptor or view descriptor that you want to update. UPDATE is normally used to update database connection information, such as user IDs and passwords. If your descriptor requires many changes, it might be easier to use the CREATE statement to overwrite the old descriptor with a new one.

Note: Altering a DBMS table might invalidate descriptor files that are based on the DBMS table, or it might cause these files to be out of date. If you re-create a table, add a new column to a table, or delete an existing column from a table, use the UPDATE statement to modify your descriptors to use the new information. △

Rules that apply to the CREATE statement also apply to the UPDATE statement. For example, the SUBSET statement is valid only for updating view descriptors.

Note: The following statements are not supported when using the UPDATE statement: ASSIGN, RESET, SELECT, and UNIQUE. △

See Table A1.1 on page 338 for the appropriate sequence of statements for updating descriptors.

Using Descriptors with the ACCESS Procedure

What Are Descriptors?

Descriptors work with the ACCESS procedure by providing information about DBMS objects to SAS, enabling you to access and update DBMS data from within a SAS session or program.

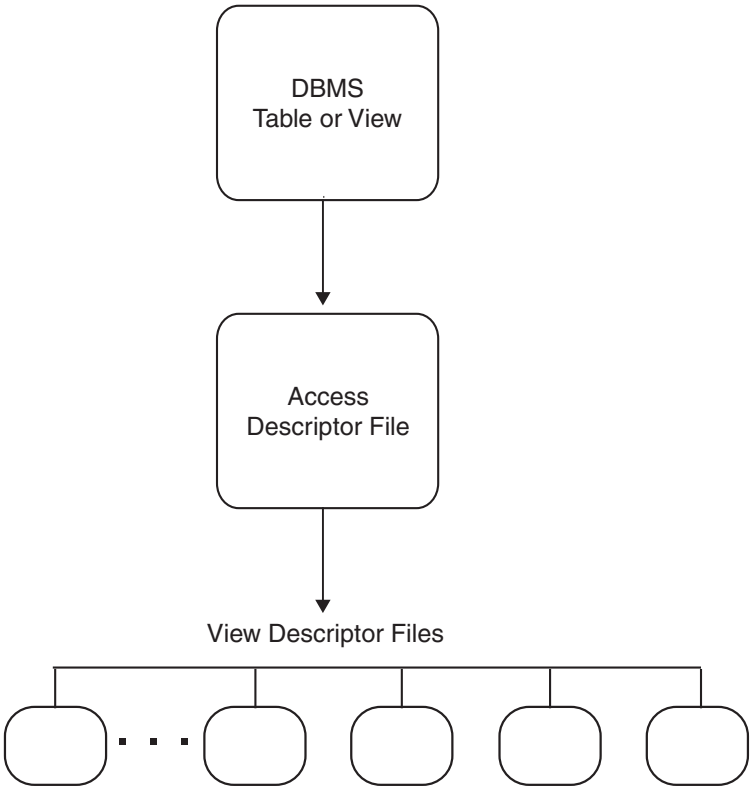
There are two types of descriptors, *access descriptors* and *view descriptors*. Access descriptors provide SAS with information about the structure and attributes of a DBMS table or view. An access descriptor, in turn, is used to create one or more view descriptors, or SAS data views, of the DBMS data.

Access Descriptors

Typically, each DBMS table or view has a single access descriptor that provides connection information, data type information, and names for databases, tables, and columns.

You use an access descriptor to create one or more view descriptors. When creating a view descriptor, you select the columns and specify criteria for the rows you want to retrieve. The figure below illustrates the descriptor creation process. Note that an access descriptor, which contains the metadata of the DBMS table, must be created before view descriptors can be created.

Figure 17.1 Creating an Access Descriptor and View Descriptors for a DBMS Table



View Descriptors

You use a view descriptor in a SAS program much as you would any SAS data set. For example, you can specify a view descriptor in the DATA= statement of a SAS procedure or in the SET statement of a DATA step.

You can also use a view descriptor to copy DBMS data into a SAS data file; this is called *extracting* the data. When you need to use DBMS data in a number of procedures or DATA steps, extracting the data into a SAS data file might use fewer resources than repeatedly accessing the data directly.

Note: The SAS/ACCESS interface view engine usually tries to pass WHERE conditions to the DBMS for processing, because in most cases it is more efficient for a DBMS to process WHERE conditions than for SAS to do the processing. △

Accessing Data Sets and Descriptors

SAS enables you to control access to SAS data sets and access descriptors by associating one or more SAS passwords with them. When you create an access descriptor, the connection information that you provide is stored in the access descriptor and in any view descriptors based on that access descriptor. The password is stored in an encrypted form. When these descriptors are accessed, the connection information that was stored is also used to access the DBMS table or view. To ensure data security, you might want to change the protection on the descriptors to prevent others from seeing the connection information stored in the descriptors.

When you create or update view descriptors, you can use a SAS data set option after the ACCDESC= option to specify the access descriptor's password (if one exists). In this case, you are *not* assigning a password to the view descriptor that is being created or updated. Rather, using the password grants you permission to use the access descriptor to create or update the view descriptor. For example:

```
proc access dbms=sybase accdesc=adlib.customer
            (alter=rouge);
  create vlib.customer.view;
  select all;
run;
```

By specifying the ALTER level of password, you can read the AdLib.Customer access descriptor and create the Vlib.Customer view descriptor.

Examples of Using the ACCESS Procedure

Example 1: Updating an Access Descriptor

The following example updates an access descriptor AdLib.Employ on the Oracle table Employees. The original access descriptor includes all of the columns in the table. The updated access descriptor omits the Salary and BirthDate columns.

```
proc access dbms=oracle ad=adlaib.employ;

   /* update access descriptor  */

   update adlib.employ.access;
   drop salary birthdate;
   list all;
run;
```

Using the LIST statement enables you to write all of the variables to the SAS log so that you can see the complete access descriptor before you update it.

Example 2: Creating a View Descriptor

The following example re-creates a view descriptor, VLIB.EMP1204, that is based on an access descriptor, ADLIB.EMPLOY, that was previously updated.

```
proc access dbms=oracle;

   /* re-create view descriptor  */

   create vlib.emp1204.view;
   select empid hiredate dept jobcode gender
      lastname firstname middlename phone;
```

```
   format empid 6.
          jobcode 5.
          hiredate datetime9.;
   subset where jobcode=1204;
run;
```

Because SELECT and RESET are not supported when UPDATE is used, the view descriptor Vlib.Emp1204 must be re-created to omit the Salary and BirthDate columns.

APPENDIX

2

The DBLOAD Procedure for Relational Databases

Overview of the DBLOAD Procedure for Relational Databases

Sending Data from SAS to a DBMS

Although it is still supported for the database systems and environments on which it was available in Version 6, the DBLOAD procedure is no longer the recommended method for sending data from SAS to a DBMS. It is recommended that you access your DBMS data more directly, using the LIBNAME statement or the Pass-Through Facility.

Note: Not all SAS/ACCESS interfaces support this feature. See Chapter 8, "SAS/ACCESS Features by Host," on page 65 to determine whether this feature is available in your environment. △

Properties of the DBLOAD Procedure

This section provides general reference information for the DBLOAD procedure; see the SAS/ACCESS documentation for your DBMS for DBMS-specific details.

The DBLOAD procedure, along with the ACCESS procedure and an interface view engine, creates an interface between SAS and data in other vendors' databases.

The DBLOAD procedure enables you to create and load a DBMS table, append rows to an existing table, and submit non-query DBMS-specific SQL statements to the DBMS for processing. The procedure constructs DBMS-specific SQL statements to create and load, or append, to a DBMS table by using one of the following:

- a SAS data file
- an SQL view or DATA step view
- a view descriptor that was created with the SAS/ACCESS interface to your DBMS or with another SAS/ACCESS interface product
- another DBMS table referenced by a SAS libref that was created with the SAS/ACCESS LIBNAME statement.

The DBLOAD procedure associates each SAS variable with a DBMS column and assigns a default name and data type to each column. It also specifies whether each column accepts NULL values. You can use the default information or change it as necessary. When you are finished customizing the columns, the procedure creates the DBMS table and loads or appends the input data.

About DBLOAD Procedure Statements

There are several types of DBLOAD statements:

- *Database connection statements* are used to connect to your DBMS. See the SAS/ACCESS documentation for your DBMS for details.
- *Creating and loading statements* are LOAD and RUN.
- *Table and editing statements* are used to specify how a table is populated.

The following table summarizes the PROC DBLOAD options and statements required to accomplish common tasks.

Table A2.1 Statement Sequence for Accomplishing Common Tasks with the DBLOAD Procedure

To do this...	Use these options and statements
Create and load a DBMS table	**PROC DBLOAD** *statement-options*; *database-connection-options*; **TABLE=** <'>*table-name*<'>; **LOAD**; **RUN**;
Submit a dynamic, non-query DBMS-SQL statement to DBMS (without creating a table)	**PROC DBLOAD** *statement-options*; *database-connection-options*; **SQL** *DBMS-specific-SQL-statements*; **RUN**;

Note: LOAD must appear before RUN to create and load a table or append data to a table. △

Procedure Syntax

The general syntax for the DBLOAD procedure is presented here; see the SAS/ACCESS documentation for your DBMS for DBMS-specific details.

PROC DBLOAD <*options*>;

database connection statements;

TABLE= <'>*table-name*<'>;

ACCDESC= <*libref.*>*access-descriptor*;

COMMIT= *commit-frequency*;

DELETE *variable-identifier-1*
<*…variable-identifier-n*>;

ERRLIMIT= *error-limit*;

LABEL;

LIMIT= *load-limit*;

LIST <ALL | COLUMN | *variable-identifier*>;

NULLS *variable-identifier-1* = Y | N
<*…variable-identifier-n* = Y | N>;

QUIT;

RENAME *variable-identifier-1* = <'>*column-name-1*<'>
<*…variable-identifier-n* = <'>*column-name-n*<'>>;

RESET ALL | *variable-identifier-1* <*…variable-identifier-n*>;

SQL *DBMS-specific-SQL-statement*;

TYPE *variable-identifier-1 = 'column-type-1'* <*…variable-identifier-n = 'column-type-n'*>;

WHERE *SAS-where-expression*;

LOAD;

RUN;

PROC DBLOAD Statement

PROC DBLOAD <*options*>;

Options

DBMS=*database-management-system*
specifies which database management system you want to access. This DBMS-specific option is required. See the SAS/ACCESS documentation for your DBMS.

DATA=<*libref.*>*SAS-data-set*
specifies the input data set. The input data can be retrieved from a SAS data file, an SQL view, a DATA step view, a SAS/ACCESS view descriptor, or another DBMS table that is referenced by a SAS/ACCESS libref. If the SAS data set is permanent, you must use its two-level name, *libref.SAS-data-set*. If you omit the DATA= option, the default is the last SAS data set that was created.

APPEND
appends data to an existing DBMS table that you identify by using the TABLE= statement. When you specify APPEND, the input data specified with the DATA= option is inserted into the existing DBMS table. Your input data can be in the form of a SAS data set, SQL view, or SAS/ACCESS view (view descriptor).

CAUTION:
When you use APPEND, you must ensure that your input data corresponds exactly to the columns in the DBMS table. If your input data does not include values for all columns in the DBMS table, you might corrupt your DBMS table by inserting data into the wrong columns. Use the COMMIT, ERRLIMIT, and LIMIT statements to help safeguard against data corruption. Use the DELETE and RENAME statements to drop and rename SAS input variables that do not have corresponding DBMS columns. △

All PROC DBLOAD statements and options can be used with APPEND, except for the NULLS and TYPE statements, which have no effect when used with APPEND. The LOAD statement is required.

The following example appends new employee data from the SAS data set NEWEMP to the DBMS table EMPLOYEES. The COMMIT statement causes a DBMS commit to be issued after every 100 rows are inserted. The ERRLIMIT statement causes processing to stop after five errors occur.

```
proc dbload dbms=oracle data=newemp append;
   user=testuser;
   password=testpass;
   path='myorapath';
   table=employees;
   commit=100;
   errlimit=5;
   load;
```

```
run;
```

Note: By omitting the APPEND option from the DBLOAD statement, you can use the PROC DBLOAD SQL statements to create a DBMS table and append to it in the same PROC DBLOAD step. △

Database Connection Statements

Provide DBMS connection information

database-connection-statements

These statements are used to connect to your DBMS and vary depending on which SAS/ACCESS interface you are using. See the documentation for your SAS/ACCESS interface for details. Examples include USER=, PASSWORD=, and DATABASE=.

ACCDESC= Statement

Creates an access descriptor based on the new DBMS table

ACCDESC=<*libref.*>*access-descriptor*;

Details

The ACCDESC= statement creates an access descriptor based on the DBMS table that you are creating and loading. If you specify ACCDESC=, the access descriptor is automatically created after the new table is created and loaded. You must specify an access descriptor if it does not already exist.

COMMIT= Statement

Issues a commit or saves rows after a specified number of inserts

Default: 1000

COMMIT=*commit-frequency*;

Details

The COMMIT= statement issues a commit (that is, generates a DBMS-specific SQL COMMIT statement) after the specified number of rows has been inserted.

Using this statement might improve performance by releasing DBMS resources each time the specified number of rows has been inserted.

If you omit the COMMIT= statement, a commit is issued (or a group of rows is saved) after each 1,000 rows are inserted and after the last row is inserted.

The *commit-frequency* argument must be a non-negative integer.

DELETE Statement

Does not load specified variables into the new table

DELETE *variable-identifier-1 <…variable-identifier-n>*;

Details

The DELETE statement drops the specified SAS variables before the DBMS table is created. The *variable-identifier* argument can be either the SAS variable name or the positional equivalent from the LIST statement. The positional equivalent is the number that represents the variable's place in the data set. For example, if you want to drop the third variable, submit the following statement:

```
delete 3;
```

When you drop a variable, the positional equivalents of the variables do not change. For example, if you drop the second variable, the third variable is still referenced by the number 3, not 2. If you drop more than one variable, separate the identifiers with spaces, not commas.

ERRLIMIT= Statement

Stops the loading of data after a specified number of errors

Default: 100; see the SAS/ACCESS documentation for your DBMS for possible exceptions

ERRLIMIT=*error-limit*;

Details

The ERRLIMIT= statement stops the loading of data after the specified number of DBMS SQL errors has occurred. Errors include observations that fail to be inserted and commits that fail to execute. The ERRLIMIT= statement defaults to 10 when used with APPEND.

The *error-limit* argument must be a non-negative integer. To allow an unlimited number of DBMS SQL errors to occur, specify ERRLIMIT=0. If the SQL CREATE TABLE statement that is generated by the procedure fails, the procedure terminates.

LABEL Statement

Causes DBMS column names to default to SAS labels

Interacts with: RESET

Default: DBMS column names default to SAS variable names

LABEL;

Details

The LABEL statement causes the DBMS column names to default to the SAS variable labels when the new table is created. If a SAS variable has no label, the variable name is used. If the label is too long to be a valid DBMS column name, the label is truncated.

For the LABEL statement to take effect, the RESET statement must be used *after* the LABEL statement.

LIMIT= Statement

Limits the number of observations that are loaded

Default: 5000

LIMIT=*load-limit*;

Details

The LIMIT= statement places a limit on the number of observations that can be loaded into the new DBMS table. The *load-limit* argument must be a non-negative integer. To load all the observations from your input data set, specify LIMIT=0.

LIST Statement

Lists information about the variables to be loaded

Default: ALL

LIST <ALL | FIELD | *variable-identifier*>;

Details

The LIST statement lists information about some or all of the SAS variables to be loaded into the new DBMS table. By default, the list is sent to the SAS log.

The LIST statement can take the following arguments:

ALL
lists information about all the variables in the input SAS data set, whether or not those variables are selected for the load.

FIELD
lists information about only the input SAS variables that are selected for the load.

variable-identifier
lists information about only the specified variable. The *variable-identifier* argument can be either the SAS variable name or the positional equivalent. The positional equivalent is the number that represents the variable's position in the data set. For example, if you want to list information for the column associated with the third SAS variable, submit the following statement:

```
list 3;
```

You can specify LIST as many times as you want while creating a DBMS table; specify LIST before the LOAD statement to see the entire table.

LOAD Statement

Creates and loads the new DBMS table

Valid: in the DBLOAD procedure (required statement for loading or appending data)

LOAD;

Details

The LOAD statement informs the DBLOAD procedure to execute the action that you request, including loading or appending data. This statement is required to create and load a new DBMS table or to append data to an existing table.

When you create and load a DBMS table, you must place statements or groups of statements in a certain order after the PROC DBLOAD statement and its options, as listed inTable A2.1 on page 357 .

Example

The following example creates the SummerTemps table in Oracle based on the DLib.TempEmps data file.

```
proc dbload dbms=oracle data=dlib.tempemps;
    user=testuser; password=testpass;
    path='testpath';
    table=summertemps;
    rename firstnam=firstname
```

```
        middlena=middlename;
    type hiredate 'date'
         empid 'number(6,0)'
         familyid 'number(6,0)';
    nulls 1=n;
    list;
    load;
run;
```

NULLS Statement

Specifies whether DBMS columns accept NULL values

Default: Y

NULLS *variable-identifier-1* = Y | N <…*variable-identifier-n* = Y | N>;

Details

Note: Some DBMSs have three valid values for this statement, Y, N, and D. See the SAS/ACCESS documentation for your DBMS for further details. △

The NULLS statement specifies whether the DBMS columns that are associated with the listed input SAS variables allow NULL values. Specify Y to accept NULL values. Specify N to reject NULL values and to require data in that column.

If you specify N for a numeric column, none of the observations that contain missing values in the corresponding SAS variable are loaded into the table, and a message is written to the SAS log. The current error count is increased by one for each observation that is not loaded. See "ERRLIMIT= Statement" on page 360 for more information.

If a character column contains blanks (the SAS missing value) and you have specified N for the DBMS column, then blanks are inserted. If you specify Y, NULL values are inserted.

The *variable-identifier* argument can be either the SAS variable name or the positional equivalent from the LIST statement. The positional equivalent is the number that represents the variable's place in the data set. For example, if you want the column that is associated with the third SAS variable to accept NULL values, submit the following statement:

```
nulls 3=y;
```

If you omit the NULLS statement, the DBMS default action occurs. You can list as many variables as you want in one NULLS statement. If you have previously defined a column as NULLS=N, you can use the NULLS statement to redefine it to accept NULL values.

QUIT Statement

Terminates the procedure

Valid: in the DBLOAD procedure (control statement)

QUIT;

Details

The QUIT statement terminates the DBLOAD procedure without further processing.

RENAME Statement

Renames DBMS columns

Interacts with: DELETE, LABEL, RESET

RENAME *variable-identifier-1 = <'>column-name-1<'> <…variable-identifier-n = <'>column-name-n<'>>*;

Details

The RENAME statement changes the names of the DBMS columns that are associated with the listed SAS variables. If you omit the RENAME statement, all the DBMS column names default to the corresponding SAS variable names (unless the LABEL statement is specified).

The *variable-identifier* argument can be either the SAS variable name or the positional equivalent from the LIST statement. The positional equivalent is the number that represents the variable's place in the data set. For example, if you want to rename the column associated with the third SAS variable, submit the following statement:

```
rename 3=employeename;
```

Note: The *column-name* argument must be a valid DBMS column name. If the column name includes lowercase characters, special characters, or national characters, you must enclose the column name in single or double quotation marks. If no quotation marks are used, the DBMS column name is created in uppercase. To preserve case, use the following syntax: **`rename 3='"employeename"'`** △

The RENAME statement enables you to include variables that you have previously deleted. For example, suppose you submit the following statements:

```
delete 3;
rename 3=empname;
```

The DELETE statement drops the third variable. The RENAME statement includes the third variable and assigns the name EMPNAME and the default column type to it.

You can list as many variables as you want in one RENAME statement. The RENAME statement overrides the LABEL statement for columns that are renamed. COLUMN is an alias for the RENAME statement.

RESET Statement

Resets column names and data types to their default values

Interacts with: DELETE, LABEL, RENAME, TYPE

RESET ALL | *variable-identifier-1* <*…variable-identifier-n*>;

Details

The RESET statement resets the columns that are associated with the listed SAS variables to the default values for DBMS column name, column data type, and ability to accept NULL values. If you specify ALL, all columns are reset to their default values, and any dropped columns are restored with their default values. The default values are as follows:

column name
: defaults to the SAS variable name, or to the SAS variable label (if you have used the LABEL statement).

column type
: is generated from the SAS variable format.

nulls
: uses the DBMS default value.

The *variable-identifier* argument can be either the SAS variable name or the positional equivalent from the LIST statement. The positional equivalent is the number that represents the variable's place in the data set. For example, if you want to reset the column associated with the third SAS variable, submit the following statement:

```
reset 3;
```

The RESET statement must be used after the LABEL statement for the LABEL statement to take effect.

SQL Statement

Submits a DBMS-specific SQL statement to the DBMS

SQL *DBMS-specific-SQL-statement*;

Details

The SQL statement submits a dynamic, non-query, DBMS-specific SQL statement to the DBMS. You can use the DBLOAD statement to submit these DBMS-specific SQL statements whether or not you create and load a DBMS table.

You must enter the keyword SQL before each DBMS-specific SQL statement that you submit. The *SQL-statement* argument can be any valid dynamic DBMS-specific SQL statement except the SELECT statement. However, you can enter a SELECT statement as a substatement within another statement, such as in a CREATE VIEW statement. You must use DBMS-specific SQL object names and syntax in the DBLOAD SQL statement.

You cannot create a DBMS table and reference it in your DBMS-specific SQL statements within the same PROC DBLOAD step. The new table is not created until the RUN statement is processed.

To submit dynamic, non-query DBMS-specific SQL statements to the DBMS *without* creating a DBMS table, you use the DBMS= option, any database connection statements, and the SQL statement.

Example

The following PROC DBLOAD example grants UPDATE privileges to user MARURI on the DB2 SasDemo.Orders table.

```
proc dbload dbms=db2;
   in sample;
   sql grant update on sasdemo.orders to maruri;
run;
```

TABLE= Statement

Names the DBMS table to be created and loaded

TABLE= <'>*DBMS-specific-syntax*<'>;

Details

When you create and load or append to a DBMS table, the TABLE= statement is required. It must follow other database connection statements such as DATABASE= or USER=. The TABLE= statement specifies the name of the DBMS table to be created and loaded into a DBMS database. The table name must be a valid table name for the DBMS. (See the SAS/ACCESS documentation for your DBMS for the syntax.) If your table name contains lowercase characters, special characters, or national characters, it must be enclosed in quotation marks.

In addition, you must specify a table name that does not already exist. If a table by that name exists, an error message is written to the SAS log, and the table specified in this statement is not loaded.

When you are submitting dynamic DBMS-specific SQL statements to the DBMS without creating and loading a table, do not use this statement.

TYPE Statement

Changes default DBMS data types in the new table

TYPE *variable-identifier-1 = 'column-type-1'* <*…variable-identifier-n = 'column-type-n'*>;

Details

The TYPE statement changes the default DBMS column data types that are associated with the corresponding SAS variables.

The *variable-identifier* argument can be either the SAS variable name or the positional equivalent from the LIST statement. The positional equivalent is the number that represents the variable's place in the data set. For example, if you want to change the data type of the DBMS column associated with the third SAS variable, submit the following statement:

```
type 3='char(17)';
```

The argument *column-type* must be a valid data type for the DBMS and must be enclosed in quotation marks.

If you omit the TYPE statement, the column data types are generated with default DBMS data types that are based on the SAS variable formats. You can change as many data types as you want in one TYPE statement. See the documentation for your SAS/ACCESS interface for a complete list of the default conversion data types for the DBLOAD procedure.

WHERE Statement

Loads a subset of data into the new table

WHERE *SAS-where-expression*;

Details

The WHERE statement causes a subset of observations to be loaded into the new DBMS table. The *SAS-where-expression* must be a valid SAS WHERE statement that uses SAS variable names (not DBMS column names) as defined in the input data set. The following example loads only the observations in which the SAS variable COUNTRY has the value **BRAZIL**:

```
where country='Brazil';
```

For more information about the syntax of the SAS WHERE statement, see *SAS Language Reference: Dictionary*.

Example of Using the DBLOAD Procedure

Example 1: Appending a Data Set to a DBMS Table

The following example appends new employee data from the SAS data set NewEmp to the DBMS table Employees. The COMMIT statement causes a DBMS commit to be issued after every 100 rows are inserted. The ERRLIMIT statement causes processing to stop after 10 errors occur.

```
proc dbload dbms=oracle data=newemp append;
   user=testuser;
   password=testpass;
   path='myorapath';
   table=employees;
   commit=100;
   errlimit=10;
   load;
run;
```

Note: By omitting the APPEND option from the DBLOAD statement, you can use the PROC DBLOAD SQL statements to create a DBMS table and append to it in the same PROC DBLOAD step. △

APPENDIX

3

Recommended Reading

Recommended Reading

Here is the recommended reading list for this title:

- *SAS/ACCESS Supplement for DB2 under z/OS (SAS/ACCESS for Relational Databases)*
- *SAS/ACCESS Supplement for DB2 under UNIX and PC Hosts*
- *SAS/ACCESS Supplement for Informix (SAS/ACCESS for Relational Databases)*
- *SAS/ACCESS Supplement for Microsoft SQL Server (SAS/ACCESS for Relational Databases)*
- *SAS/ACCESS Supplement for ODBC (SAS/ACCESS for Relational Databases)*
- *SAS/ACCESS Supplement for OLE DB (SAS/ACCESS for Relational Databases)*
- *SAS/ACCESS Supplement for Oracle (SAS/ACCESS for Relational Databases)*
- *SAS/ACCESS Supplement for SYBASE (SAS/ACCESS for Relational Databases)*
- *SAS/ACCESS Supplement for Teradata*
- *SAS Language Reference: Concepts*
- *SAS Language Reference: Dictionary*
- *Base SAS Procedures Guide*
- SAS Companion that is specific to your operating environment

For a complete list of SAS publications, see the current *SAS Publishing Catalog*. To order the most current publications or to receive a free copy of the catalog, contact a SAS representative at

SAS Publishing Sales
SAS Campus Drive
Cary, NC 27513
Telephone: (800) 727-3228*
Fax: (919) 677-8166
E-mail: sasbook@sas.com
Web address: **support.sas.com/pubs**
* For other SAS Institute business, call (919) 677-8000.

Customers outside the United States should contact their local SAS office.

Glossary

This glossary defines SAS software terms that are used in this document as well as terms that relate specifically to SAS/ACCESS software.

access descriptor
a SAS/ACCESS file that describes data that is managed by a data management system. After creating an access descriptor, you can use it as the basis for creating one or more view descriptors. See also view and view descriptor.

browsing data
the process of viewing the contents of a file. Depending on how the file is accessed, you can view SAS data either one observation (row) at a time or as a group in a tabular format. You cannot update data that you are browsing.

bulk load
to load large amounts of data into a database object, using methods that are specific to a particular DBMS. Bulk loading enables you to rapidly and efficiently add multiple rows of data to a table as a single unit.

client
(1) a computer or application that requests services, data, or other resources from a server. (2) in the X Window System, an application program that interacts with the X server and can perform tasks such as terminal emulation or window management. For example, SAS is a client because it requests windows to be created, results to be displayed, and so on.

column
in relational databases, a vertical component of a table. Each column has a unique name, contains data of a specific type, and has certain attributes. A column is analogous to a variable in SAS terminology.

column function
an operation that is performed for each value in the column that is named as an argument of the function. For example, AVG(SALARY) is a column function.

commit
the process that ends a transaction and makes permanent any changes to the database that the user made during the transaction. When the commit process occurs, locks on the database are released so that other applications can access the changed data. The SQL COMMIT statement initiates the commit process.

DATA step view
a type of SAS data set that consists of a stored DATA step program. Like other SAS data views, a DATA step view contains a definition of data that is stored elsewhere; the view does not contain the physical data. The view's input data can come from one or more sources, including external files and other SAS data sets. Because a DATA step view only reads (opens for input) other files, you cannot update the view's underlying data.

data type
a unit of character or numeric information in a SAS data set. A data value represents one variable in an observation.

data value
in SAS, a unit of character or numeric information in a SAS data set. A data value represents one variable in an observation.

database
an organized collection of related data. A database usually contains named files, named objects, or other named entities such as tables, views, and indexes

database management system (DBMS)
an organized collection of related data. A database usually contains named files, named objects, or other named entities such as tables, views, and indexes

editing data
the process of viewing the contents of a file with the intent and the ability to change those contents. Depending on how the file is accessed, you can view the data either one observation at a time or in a tabular format.

engine
a component of SAS software that reads from or writes to a file. Each engine enables SAS to access files that are in a particular format. There are several types of engines.

file
a collection of related records that are treated as a unit. SAS files are processed and controlled by SAS and are stored in SAS data libraries.

format
a collection of related records that are treated as a unit. SAS files are processed and controlled by SAS and are stored in SAS data libraries. In SAS/ACCESS software, the default formats vary according to the interface product.

index
(1) in SAS software, a component of a SAS data set that enables SAS to access observations in the SAS data set quickly and efficiently. The purpose of SAS indexes is to optimize WHERE-clause processing and to facilitate BY-group processing. (2) in other software vendors' databases, a named object that directs the DBMS to the storage location of a particular data value for a particular column. Some DBMSs have additional specifications. These indexes are also used to optimize the processing of WHERE clauses and joins. Depending on the SAS interface to a database product and how selection criteria are specified, SAS may or may not be able to use the indexes of the DBMS to speed data retrieval.

Depending on how selection criteria are specified, SAS might use DBMS indices to speed data retrieval.

informat
a pattern or set of instructions that SAS uses to determine how data values in an input file should be interpreted. SAS provides a set of standard informats and also enables you to define your own informats.

interface view engine
a SAS engine that is used by SAS/ACCESS software to retrieve data from files that have been formatted by another vendor's software. Each SAS/ACCESS interface has its own interface view engine, which reads the interface product data and returns the data in a form that SAS can understand (that is, in a SAS data set). SAS automatically uses an interface view engine; the engine name is stored in SAS/ACCESS descriptor files so that you do not need to specify the engine name in a LIBNAME statement.

libref
a name that is temporarily associated with a SAS data library. The complete name of a SAS file consists of two words, separated by a period. The libref, which is the first word, indicates the library. The second word is the name of the specific SAS file. For example, in VLIB.NEWBDAY, the libref VLIB tells SAS which library contains the file NEWBDAY. You assign a libref with a LIBNAME statement or with an operating system command.

member
a SAS file in a SAS data library.

member name
a name that is given to a SAS file in a SAS data library.

member type
a SAS name that identifies the type of information that is stored in a SAS file. Member types include ACCESS, DATA, CATALOG, PROGRAM, and VIEW.

missing value
in SAS, a term that describes the contents of a variable that contains no data for a particular row or observation. By default, SAS prints or displays a missing numeric value as a single period, and it prints or displays a missing character value as a blank space.

observation
a row in a SAS data set. All of the data values in an observation are associated with a single entity such as a customer or a state. Each observation contains one data value for each variable. In a database product table, an observation is analogous to a row. Unlike rows in a database product table or file, observations in a SAS data file have an inherent order.

Pass-Through Facility
a group of SQL procedure statements that send and receive data directly between a relational database management system and SAS. The Pass-Through Facility includes the CONNECT, DISCONNECT, and EXECUTE statements, and the CONNECTION TO component. SAS/ACCESS software is required in order to use the Pass-Through Facility.

PROC SQL view
a SAS data set (of type VIEW) that is created by the SQL procedure. A PROC SQL view contains no data. Instead, it stores information that enables it to read data values from other files, which can include SAS data files, SAS/ACCESS views, DATA step views, or other PROC SQL views. A PROC SQL view's output can be either a subset or a superset of one or more files.

query
a set of instructions that requests particular information from one or more data sources.

referential integrity
a set of rules that a DBMS uses to ensure that whenever a data value in one table is changed, the appropriate change is also made to any related values in other tables or in the same table. Referential integrity is also used to ensure that related data is not deleted or changed accidentally.

relational database management system
a database management system that organizes and accesses data according to relationships between data items. Oracle and DB2 are examples of relational database management systems.

rollback
in most databases, the process that restores the database to its state when changes were last committed, voiding any recent changes. The SQL ROLLBACK statement initiates the rollback processes. See also commit.

row
in relational database management systems, the horizontal component of a table. A row is analogous to a SAS observation.

SAS data file
a type of SAS data set that contains data values as well as descriptor information that is associated with the data. The descriptor information includes information such as the data types and lengths of the variables, as well as the name of the engine that was used to create the data. A PROC SQL table is a SAS data file. SAS data files are of member type DATA.

SAS data library
a collection of one or more SAS files that are recognized by SAS and that are referenced and stored as a unit. Each file is a member of the library.

SAS data set
a file whose contents are in one of the native SAS file formats. There are two types of SAS data sets: SAS data files and SAS data views. SAS data files contain data values in addition to descriptor information that is associated with the data. SAS data views contain only the descriptor information plus other information that is required for retrieving data values from other SAS data sets or from files whose contents are in other software vendors' file formats.

SAS data view
a file whose contents are in one of the native SAS file formats. There are two types of SAS data sets: SAS data files and SAS data views. SAS data files contain data values in addition to descriptor information that is associated with the data. SAS data views contain only the descriptor information plus other information that is required for retrieving data values from other SAS data sets or from files whose contents are in other software vendors' file formats.

SAS/ACCESS views
See view descriptor and SAS data view.

server
in a network, a computer that is reserved for servicing other computers in the network. Servers can provide several different types of services, such as file services and communication services. Servers can also enable users to access shared resources such as disks, data, and modems.

Structured Query Language (SQL)
the standardized, high-level query language that is used in relational database management systems to create and manipulate database management system objects. SAS implements SQL through the SQL procedure.

table
a two-dimensional representation of data, in which the data values are arranged in rows and columns.

trigger
a type of user-defined stored procedure that is executed whenever a user issues a data-modification command such as INSERT, DELETE, or UPDATE for a specified table or column. Triggers can be used to implement referential integrity or to maintain business constraints.

variable
a column in a SAS data set. A variable is a set of data values that describe a given characteristic across all observations.

view
a definition of a virtual data set. The definition is named and stored for later use. A view contains no data; it merely describes or defines data that is stored elsewhere. SAS data views can be created by the ACCESS and SQL procedures.

view descriptor
a file created by SAS/ACCESS software that defines part or all of the database management system (DBMS) data or PC file data that is described by an access descriptor. The access descriptor describes the data in a single DBMS table, DBMS view, or PC file.

wildcard
a file created by SAS/ACCESS software that defines part or all of the database management system (DBMS) data or PC file data that is described by an access descriptor. The access descriptor describes the data in a single DBMS table, DBMS view, or PC file.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Your Turn

If you have comments or suggestions about *SAS/ACCESS 9.1.3 for Relational Databases: Reference, Third Edition*, please send them to us on a photocopy of this page, or send us electronic mail.

For comments about this book, please return the photocopy to

SAS Publishing
SAS Campus Drive
Cary, NC 27513
E-mail: **yourturn@sas.com**

For suggestions about the software, please return the photocopy to

SAS Institute Inc.
Technical Support Division
SAS Campus Drive
Cary, NC 27513
E-mail: **suggest@sas.com**