SAS/ACCESS® 9.3 for Relational Databases

Reference, Second Edition

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2012. *SAS/ACCESS® 9.3 for Relational Databases: Reference, Second Edition*. Cary, NC: SAS Institute Inc.

SAS/ACCESS® 9.3 for Relational Databases: Reference, Second Edition

ISBN 978-1-60764-890-1

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

1st printing, August 2012
1st electronic book, August 2012
2nd electronic book, September 2014

Contents

What's New in SAS/ACCESS 9.3 for Relational Databases

Overview

SAS/ACCESS 9.3 for Relational Databases has these new features and enhancements.

- new SAS/ACCESS Interface to Hadoop in March 2012
- updated operating environments for various SAS/ACCESS interfaces
- new nicknames
- new function and options for the Teradata interface
- documentation enhancements

Operating Environments Updates

Operating environments for specific SAS/ACCESS interfaces have changed. See SAS/ACCESS Features by Host for the operating environments that your interface supports.

New Nicknames

These interfaces have new nicknames.

- SAS/ACCESS Interface to Aster *n*Cluster: **`aster`**
- SAS/ACCESS Interface to Greenplum: **`greenplm`**
- SAS/ACCESS Interface to Sybase IQ: **`sybaseiq`**

Second Maintenance Release for SAS 9.3: General Changes

In the second maintenance release for SAS 9.3, the default value for the SQLGENERATION= LIBNAME and system options includes Aster *n*Cluster and Greenplum.

SAS/ACCESS Interface to Aster *n*Cluster

In the second maintenance release for SAS 9.3, the default PORT= number for the LIBNAME statement has changed.

SAS/ACCESS Interface to Hadoop

In the March 2012 release for SAS 9.3, SAS/ACCESS Interface to Hadoop is a new database engine. It provides direct, transparent access to Apache Hadoop through LIBNAME statements and the SQL pass-through facility. You can use various LIBNAME statement options and data set options that the LIBNAME engine supports to control the data that is returned to SAS.

For details about SAS interfaces to Hadoop, see these topics or documents.

- SAS/ACCESS Interface to Hadoop
- SAS/ACCESS Interface to Hadoop: Supported Features
- PROC HADOOP in *Base SAS Procedures Guide*
- FILENAME statement (file access method) in *SAS Statements: Reference*

In the second maintenance release for SAS 9.3, these items are new or enhanced.

- You can specify a Hadoop configuration file in a CONNECTION statement or as a connection option in the LIBNAME statement.
- DBCREATE_TABLE_EXTERNAL= LIBNAME and data set options are new.
- The DBCREATE_TABLE_LOCATION= data set option is new.
- DBCREATE_TABLE_OPTS= LIBNAME and data set options are new.
- The LENGTH and TRIMN functions are now automatically passed down by default.

SAS/ACCESS Interface to Oracle

In the second maintenance release for SAS 9.3, these items are new.

- DB_OBJECTS= LIBNAME option
- OR_BINARY_DOUBLE= LIBNAME option

SAS/ACCESS Interface to Teradata

These items are new.

- Temporal qualifiers let you specify time-dimension criteria for retrieving data from Teradata. For details, see the section about temporal data types, including how to use them in a LIBNAME statement or a DATA step.
- Using the DBCONSTRAINT= data set option, you can specify with table-level definitions where to create a table.

In the second maintenance release for SAS 9.3, these items are new or enhanced.

- The enhanced LOGDB= LIBNAME option now also works with the TPT API.
- The SLEEP= and TENACITY= data set options are enhanced, and corresponding SLEEP= and TENACITY= LIBNAME options are new.
- The new TR_ENABLE_INTERRUPT= LIBNAME option lets you cancel an executing Teradata query.

Documentation Enhancements

This document includes these changes.

- SAS In-Database information is now in the *SAS In-Database Products: Administrator’s Guide* and the *SAS In-Database Products: User's Guide*. See also the In-Database Processing with SAS/ACCESS topic.
- For the MySQL interface, you can use the RESULTS= LIBNAME option to specify where you want to store query results.

Accessibility Features of SAS/ACCESS Interfaces

SAS/ACCESS relational database interfaces include accessibility and compatibility features that improve usability of the product for users with disabilities. These features are related to accessibility standards for electronic information technology adopted by the U.S. Government under Section 508 of the U.S. Rehabilitation Act of 1973, as amended.

Recommended Reading

Here is the recommended reading list for this title.

- *Base SAS Procedures Guide*
- *Base SAS Utilities: Reference*
- *SAS/ACCESS Interface to PC Files: Reference*
- *SAS Companion* for your operating environment
- *SAS Component Objects: Reference*
- *SAS Data Set Options: Reference*
- *SAS Formats and Informats: Reference*
- *SAS Functions and CALL Routines: Reference*
- *SAS In-Database Products: Administrator's Guide*
- *SAS In-Database Products: User's Guide*
- *SAS Intelligence Platform: Security Administration Guide*
- *SAS Language Reference: Concepts*
- *SAS Macro Language: Reference*
- *SAS/SHARE User's Guide*
- *SAS System Options: Reference*
- *SAS Statements: Reference*

For a complete list of SAS publications, go to support.sas.com/bookstore. If you have questions about which titles you need, please contact a SAS Publishing Sales Representative:

SAS Publishing Sales
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-3228
Fax: 1-919-677-8166
E-mail: sasbook@sas.com
Web address: support.sas.com/bookstore

Part 1

Concepts

Chapter 1

Overview of SAS/ACCESS Interface to Relational Databases

About This Document

This document provides conceptual, reference, and usage information for SAS/ACCESS software for data sources for relational database management systems (DBMSs), data warehouse appliances, and Hadoop Distributed File System (HDFS). The information in this document applies to all data sources that SAS/ACCESS software supports.

The availability and behavior of SAS/ACCESS features vary from one interface to another. You should therefore consult the information in both the general and DBMS-specific sections of this document when working with your particular SAS/ACCESS interface. You should also refer to the SAS system requirements and configuration guide documents that are available at **http://support.sas.com**.

This document is intended for applications programmers and end users with these skills.

- They are familiar with the basics of their DBMS or data warehouse appliance and its Structured Query Language (SQL).
- If they are using Hadoop, they are familiar with Hadoop and the Hadoop Distributed File System (HDFS). They have knowledge of fundamental Hadoop library and processing using HDFS. They are also familiar with the basics of Hadoop, Hive, and HiveQL. For details, see the Apache Hadoop and Hive Web sites at **http://hadoop.apache.org** and **https://cwiki.apache.org/confluence/display/Hive**.

- They know how to use their operating environment.
- They can use basic SAS commands and statements.

Database administrators might also want to read this document to understand how to implement and administer specific interfaces.

Methods for Accessing Relational Database Data

SAS/ACCESS Interface to Relational Databases is a family of interfaces—each licensed separately—with which you can interact with data in other vendor databases from within SAS. SAS/ACCESS provides these methods for accessing relational DBMS data.

- To assign SAS librefs to DBMS objects such as schemas and databases, you can use the LIBNAME statement on page 93 . After you associate a database with a libref, you can use a SAS two-level name to specify any table or view in the database. You can then work with the table or view as you would with a SAS data set.
- To interact with a data source using its native SQL syntax without leaving your SAS session, you can use the SQL pass-through facility on page 403. SQL statements are passed directly to the data source for processing.
- For indirect access to DBMS data, you can use the ACCESS on page 881 and DBLOAD on page 901 procedures. Although SAS still supports these procedures for database systems and environments on which they were available for SAS 6, they are no longer the recommended method for accessing DBMS data.

For information about when to use each method, see “Selecting a SAS/ACCESS Method ” on page 4.

Not all SAS/ACCESS interfaces support all of these features. To determine which features are available in your environment, see “Introduction” on page 77.

Selecting a SAS/ACCESS Method

Methods for Accessing DBMS Tables and Views

You can often complete a task in SAS/ACCESS in several ways. For example, you can access DBMS tables and views by using the LIBNAME statement on page 93 or the SQL pass-through facility on page 403. Before processing complex or data-intensive operations, you might want to test several methods first to determine the most efficient one for your particular task.

SAS/ACCESS LIBNAME Statement Advantages

You should use the SAS/ACCESS LIBNAME statement for the fastest and most direct method of accessing your DBMS data except when you need to use SQL that is not ANSI-standard. ANSI-standard SQL is required when you use the SAS/ACCESS library engine in the SQL procedure. However, the SQL pass-through facility accepts all SQL extensions that your DBMS provides.

Here are the advantages of using the SAS/ACCESS LIBNAME statement.

- 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, lets you specify how data is processed, and lets you easily view your DBMS tables in SAS.
- You do not need to know the SQL language of your DBMS to access and manipulate data on your DBMS. You can use such SAS procedures as PROC SQL or DATA step programming on any libref that references DBMS data. You can read, insert, update, delete, and append data. You can also create and drop DBMS tables by using SAS syntax.
- The LIBNAME statement gives you 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 processing of joins and WHERE clauses by passing them directly to the DBMS, which takes advantage of the indexing and other processing capabilities of your DBMS. For more information, see "Overview: Optimizing Your SQL Usage" on page 45.
- The engine can pass some functions directly to the DBMS for processing.

SQL Pass-Through Facility Advantages

Here are the advantages of using the SQL pass-through facility.

- You can use SQL pass-through facility statements so the DBMS can 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.
- SQL pass-through facility statements let the DBMS optimize queries when queries have summary functions (such as AVG and COUNT), GROUP BY clauses, or columns that expressions create (such as the COMPUTED function). The DBMS optimizer can use indexes on DBMS columns to process queries more rapidly.
- On some DBMSs, you can use SQL pass-through facility statements with SAS/AF applications to handle transaction processing of DBMS data. Using a SAS/AF application gives you complete control of COMMIT and ROLLBACK transactions. SQL pass-through facility statements give you better access to DBMS return codes.
- The SQL pass-through facility accepts all extensions to ANSI SQL that your DBMS provides.

SAS/ACCESS Features for Common Tasks

Here is 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*
	SQL pass-through facility
	View descriptors**

Task	SAS/ACCESS Features
Create DBMS objects, such as tables	LIBNAME statement*
	DBLOAD procedure
	SQL pass-through facility EXECUTE statement
Update, delete, or insert rows into DBMS tables	LIBNAME statement*
	View descriptors**
	SQL pass-through facility EXECUTE statement
Append data to DBMS tables	DBLOAD procedure with APPEND option
	LIBNAME statement and APPEND procedure*
	SQL pass-through facility EXECUTE statement
	SQL pass-through facility INSERT statement
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 DROP TABLE statement*
	LIBNAME statement and DATASETS procedure DELETE statement*
	DBLOAD procedure with SQL DROP TABLE statement
	SQL pass-through facility 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 lets you 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. Here are the types of SAS data views.

DATA step views
: 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. To convert existing view descriptors into SQL views, use the CV2View procedure on page 869.

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).

You can use SQL views and SAS/ACCESS views to update their underlying data if the view is based on only one DBMS table or if it is based on a DBMS view that is based on only one DBMS table and if the view has no calculated fields. You cannot use DATA step views to update the underlying data; you can use them only to 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. This table lists the recommended methods for creating SAS views.

Table 1.2 *Creating SAS Views*

Feature for Accessing DBMS Data	SAS View Technology to Use
SAS/ACCESS LIBNAME statement	SQL view or DATA step view of the DBMS table
SQL pass-through facility	SQL view with CONNECTION TO component

Choosing Your Degree of Numeric Precision

Factors That Can Cause Calculation Differences

Different factors affect numeric precision. This issue is common for many people, including SAS users. Though computers and software can help, you are limited in how precisely you can calculate, compare, and represent data. Therefore, only those people who generate and use data can determine the exact degree of precision that suits their enterprise needs.

As you decide the degree of precision that you want, you need to consider that these system factors can cause calculation differences:

- hardware limitations
- differences among operating systems
- different software or different versions of the same software
- different database management systems (DBMSs)

These factors can also cause differences:

- the use of finite number sets to represent infinite real numbers

- how numbers are stored, because storage sizes can vary

You also need to consider how conversions are performed—on, between, or across any of these system or calculation factors.

Examples of Problems That Result in Numeric Imprecision

Overview

Depending on the degree of precision that you want, calculating the value of r can result in a tiny residual in a floating-point unit. When you compare the value of **r** to 0.0, you might find that r≠0.0. The numbers are very close but not equal. This type of discrepancy in results can stem from problems in representing, rounding, displaying, and selectively extracting data.

Representing Data

Some numbers can be represented exactly, but others cannot. As shown in this example, the number 10.25, which terminates in binary, can be represented exactly.

```
data x;
   x=10.25;
   put x hex16.;
run;
```

The output from this DATA step is an exact number: 4024800000000000. However, the number 10.1 cannot be represented exactly, as this example shows.

```
data x;
   x=10.1;
   put x hex16.;
run;
```

The output from this DATA step is an inexact number: 4024333333333333.

Rounding Data

As this example shows, rounding errors can result from platform-specific differences. No solution exists for such situations.

```
data x;
   x=10.1;
   put x hex16.;
   y=100000;
   newx=(x+y)-y;
   put newx hex16.;
 run;
```

In Windows and Linux environments, the output from this DATA step is 402433333333*3333* (8/10-byte hardware double). In the Solaris x64 environment, the output is 402433333333*4000* (8/8-byte hardware double).

Displaying Data

For certain numbers such as x.5, the precision of displayed data depends on whether you round up or down. Low-precision formatting (rounding down) can produce different results on different platforms. In this example, the same high-precision (rounding up) result occurs for X=8.3, X=8.5, or X=hex16. However, a different result occurs for X=8.1 because this number does not yield the same level of precision.

```
data;
   x=input('C047DFFFFFFFFFFF', hex16.);
   put x= 8.1 x= 8.3 x= 8.5 x= hex16.;
run;
```

Here is the output under Windows or Linux (high-precision formatting).

```
x=-47.8
x=-47.750 x=-47.7500
x=C047DFFFFFFFFFFF
```

Here is the output under Solaris x64 (low-precision formatting).

```
x=-47.7
x=-47.750 x=-47.7500
x=C047DFFFFFFFFFFF
```

To fix the problem that this example illustrates, you must select a number that yields the next precision level—in this case, 8.2.

Selectively Extracting Data

Results can also vary when you access data that is stored on one system by using a client on a different system. This example illustrates running a DATA step from a Windows client to access SAS data in the z/OS environment.

```
data z(keep=x);
   x=5.2;
   output;
   y=1000;
   x=(x+y)-y;   /*almost 5.2 */
   output;
run;

proc print data=z;
run;
```

Here is the output this DATA step produces.

```
Obs   x
1    5.2
2    5.2
```

The next example illustrates the output that you receive when you execute the DATA step interactively under Windows or under z/OS.

```
data z1;
   set z(where=(x=5.2));
run;
```

Here is the corresponding z/OS output.

```
NOTE: There were 1 observations read from the data set WORK.Z.
WHERE x=5.2;
NOTE: The data set WORK.Z1 has 1 observations and 1 variables.
The DATA statement used 0.00 CPU seconds and 14476K.
```

In the above example, the expected count was not returned correctly under z/OS because the imperfection of the data and finite precision are not taken into account. You cannot use equality to obtain a correct count because it does not include the “almost 5.2” cases in that count. To obtain the correct results under z/OS, you must run this DATA step:

```
data z1;
   set z(where=(compfuzz(x,5.2,1e-10)=0));
run;
```

Here is the z/OS output from this DATA step.

```
NOTE: There were 2 observations read from the data set WORK.Z.
WHERE COMPFUZZ(x, 5.2, 1E-10)=0;
NOTE: The data set WORK.Z1 has 2 observations and 1 variables.
```

Your Options When Choosing the Degree of Precision That You Need

After you determine the degree of precision that your enterprise needs, you can refine your software. You can use macros, sensitivity analyses, or fuzzy comparisons such as extractions or filters to extract data from databases or from different versions of SAS.

If you are running SAS 9.2, use the COMPFUZZ (fuzzy comparison) function. Otherwise, use this macro.

```
/*****************************************************************************/
/* This macro defines an EQFUZZ operator.  The subsequent DATA step shows    */
/* how to use this operator to test for equality within a certain tolerance. */
/*****************************************************************************/
%macro eqfuzz(var1, var2, fuzz=1e-12);
abs((&var1 - &var2) / &var1) < &fuzz
%mend;

data _null_;
   x=0;
   y=1;
   do i=1 to 10;
      x+0.1;
   end;
   if x=y then put 'x exactly equal to y';
   else if %eqfuzz(x,y) then put 'x close to y';
   else put 'x nowhere close to y';
run;
```

When you read numbers in from an external DBMS that supports precision beyond 15 digits, you can lose that precision. You cannot do anything about this for existing databases. However, when you design new databases, you can set constraints to limit precision to about 15 digits or you can select a numeric DBMS data type to match the numeric SAS data type. For example, select the BINARY_DOUBLE type in Oracle (precise up to 15 digits) instead of the NUMBER type (precise up to 38 digits).

When you read numbers in from an external DBMS for noncomputational purposes, use the DBSASTYPE= data set option, as shown in this example.

```
libname ora oracle user=scott password=tiger path=path;
data sasdata;
   set ora.catalina2( dbsastype= ( c1='char(20)') ) ;
run;
```

This option retrieves numbers as character strings and preserves precision beyond 15 digits. For details, see the DBSASTYPE= data set option on page 298 .

References

See these resources for more detail about numeric precision, including variables that can affect precision.

- The Aggregate. 2008. "Numerical Precision, Accuracy, and Range," Aggregate.org: *Unbridled Computing*. Lexington, KY: University of Kentucky. Available at **`http://aggregate.org/NPAR`**. Accessed on July 30, 2012.
- IEEE. 2008. "IEEE 754: Standard for Binary Floating-Point Arithmetic." Available at **`http://grouper.ieee.org/groups/754/index.html`**. This standard defines 32-bit and 64-bit floating-point representations and computational results. Accessed on July 30, 2012.
- SAS Institute Inc. 2007. TS-230. *Dealing with Numeric Representation Error in SAS Applications*. Cary, NC: SAS Institute Inc. Available at **`http://support.sas.com/techsup/technote/ts230.html`**. Accessed on July 30, 2012.
- SAS Institute Inc. 2007. TS-654. *Numeric Precision 101*. Cary, NC: SAS Institute Inc. Available at **`http://support.sas.com/techsup/technote/ts654.pdf`**. This document is an overview of numeric precision and how it is represented in SAS applications. Accessed on July 30, 2012.

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, show special consideration when you use names of such DBMS objects 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 these 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

These SAS language elements have a maximum length of eight 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: Concepts*.

Case Sensitivity

When SAS encounters mixed-case or case-sensitive names in SAS code, SAS 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. 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.

Here are two examples of name literals.

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

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, you must set the system option to VALIDVARNAME=ANY. For more information, see VALIDVARNAME= on page 398.

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 (_). (Nonstandard names include those with blank spaces or such special characters as @, #, % that are not allowed in SAS names.) For example, the DBMS column name Amount Budgeted$ becomes the SAS variable name Amount_Budgeted_.

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 receive 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 eight characters. Long DBMS column names are truncated to 8-character SAS variable names within the SAS access descriptor. You can use the RENAME on page 890 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 receive an error message advising you that long variable names are not supported in the DBLOAD procedure. You can use the RENAME on page 911 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 on page 912 statement. The DBLOAD SQL statement is passed to the DBMS exactly as you enter 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 this example.

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

You can rename tables using this method for all SAS/ACCESS engines. However, if you change a table name, any view that depends on that table no longer works unless the view references the new table name.

Renaming SAS/ACCESS Columns

You can use the RENAME on page 911 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 on page 890 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 these options.

PRESERVE_COL_NAMES=YES
: 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. For more information, see the PRESERVE_COL_NAMES= LIBNAME on page 167 and data set on page 336options.

PRESERVE_TAB_NAMES=YES
: a SAS/ACCESS LIBNAME option. When set to YES, this option preserves blank spaces, special characters, and mixed case in DBMS table names. Specify the PRESERVE_NAMES=YES | NO alias 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. For more information, see the PRESERVE_TAB_NAMES= LIBNAME option on page 168.

DQUOTE=ANSI
: 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
: a global system option that can override the SAS naming conventions. For more information, see the VALIDVARNAME= system option on page 398.

Examples on page 19 that use these options are available. The availability of these options and their default settings are DBMS-specific, so see 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 tables in this section illustrate how SAS/ACCESS processes DBMS names when it retrieves data from a DBMS. This information applies generally to all interfaces. However, in some cases you need not specify these options because the option default values are DBMS-specific. For details, see the DBMS-specific reference section for your SAS/ACCESS interface. Examples on page 19 are available that illustrate the different types of naming actions and defaults.

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 tables in this section illustrate how SAS/ACCESS handles variable names when it creates such DBMS objects as tables and views. This information applies generally to all interfaces. However, in some cases you need not specify these options because the option default values are DBMS-specific. For details, see the documentation for your DBMS. Examples on page 19 are available that illustrate the different types of naming actions and defaults.

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

SAS Variable Name as Input	Desired DBMS Column Name	Options
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
Case-sensitive SAS data set name, such as Payroll	Case-sensitive DBMS table name, such as Payroll	PRESERVE_TAB_NAMES=YES
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. You can use both PRESERVE_COL_NAMES=YES and the PROC SQL DQUOTE=ANSI options to preserve the case and nonstandard characters in the column names. You need not quote the column aliases to preserve mixed case. You need only 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 pay1 Oracle table name in this example is created in lowercase, Oracle stores the name in uppercase as PAY1. To store the table name as "pay1", 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. SEX, JOBCODE, and SALARY columns in the PAYROLL Oracle table were created in uppercase. They therefore retain this case in the new table unless you rename them. Here is partial output from the example.

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

This example uses PROC PRINT to print the DBMS table PAYROLL. The DBMS table was created in uppercase. Because PRESERVE_TAB_NAMES=YES, you must specify the table name in uppercase. (If you set the PRESERVE_TAB_NAMES=NO, you can specify the DBMS table name in lowercase.) 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;
```

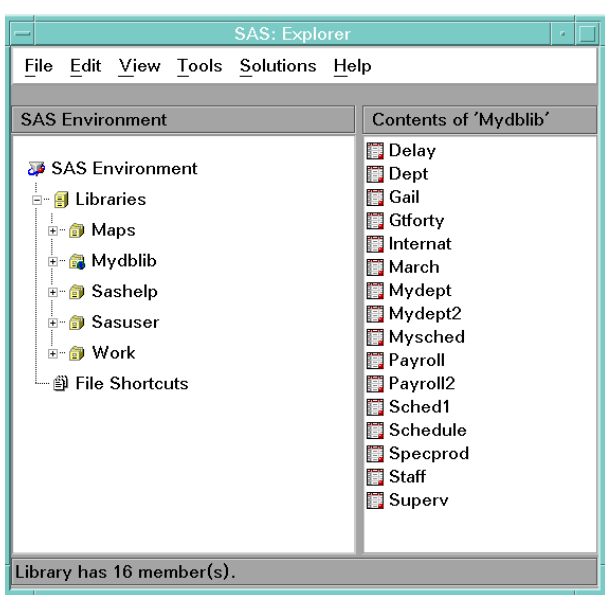
Output 2.3 *DBMS Table with a Case-Sensitive Name*

```
                        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
```

This next example submits a SAS/ACCESS LIBNAME statement. It then opens the SAS Explorer window, which lists the Oracle tables and views that the MYDBLIB libref references. The 16 members are listed and all member names are in the case (initial capitalization) that the Explorer window sets. Table names are capitalized due to the PRESERVE_TAB_NAMES=NO default.

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

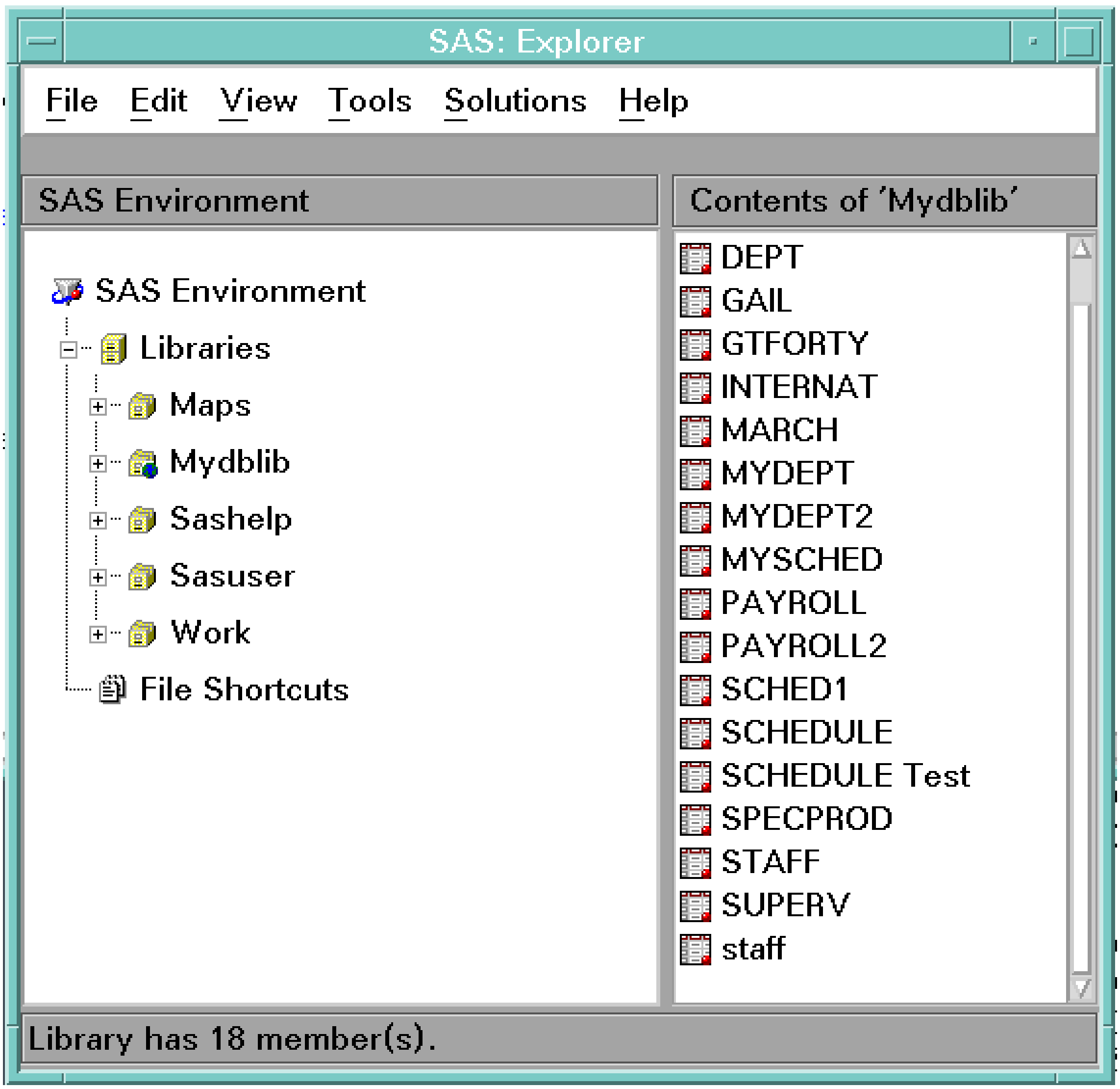
Display 2.1 *SAS Explorer Window Listing DBMS Objects*



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 that the MYDBLIB libref references.

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

Display 2.2 *SAS Explorer Window Listing Case-Sensitive DBMS Objects*



The 18 members are listed, including one in lowercase and one with 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

This example creates a DBMS table with a blank space in its name. Double quotation marks are used to specify the table name, International Delays. You can also set both of the preserve names LIBNAME options 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";
```

You can 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 contain a space in their names, you must enclose the names in double quotation marks. 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

This example creates a table using name literals. To use name literals, you must specify the SAS option VALIDVARNAME=ANY. 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

This example uses PROC SQL to create a DBMS table that is based on data from other DBMS tables. To preserve the case sensitivity of the aliased column names, use PRESERVE_COL_NAMES=YES. Partial output follows the example.

```
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 Greater Than $40,000';
run;
```

Output 2.7 *Updating DBMS Data*

```
     Employees with Salaries Greater Than $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

This 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 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

DBMS-Specific Naming Conventions

For more information about naming conventions for your interface, see these DBMS-specific topics.

- Aster nCluster on page 428
- DB2 under UNIX and PC Hosts on page 458
- DB2 under z/OS on page 502
- Greenplum on page 530
- Hadoop on page 543
- HP Neoview on page 570
- Informix on page 587

- Microsoft SQL Server on page 605
- MySQL on page 617
- Netezza on page 636
- ODBC on page 668
- OLE DB on page 692
- Oracle on page 719
- Sybase on page 748
- Sybase IQ on page 770
- Teradata on page 818

Chapter 3
Data Integrity and Security

Introduction to Data Integrity and Security

This section briefly describes DBMS security issues on page 29 and then presents measures that you can take on the SAS side on page 30 of the interface to help protect DBMS data from accidental update or deletion. This section also provides information about how SAS handles null values on page 35 that help you achieve consistent results.

DBMS Security

Privileges

The database administrator controls who has privileges to access or update DBMS objects. This person 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. SQL pass-through facility on page 403 to EXECUTE an SQL statement, or by issuing a GRANT statement from the DBLOAD procedure on page 901

You should give users only the privileges on the DBMS that they must have. Privileges are granted on whole tables or views. You must explicitly grant user privileges on the DBMS tables or views that underlie a view so that users can use that view.

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, from the SAS side of the interface, you can take steps like these.

- Specify the SAS/ACCESSDBPROMPT= on page 136 LIBNAME option to avoid saving connection information in your code.
- Create SQL views and protecting them from unauthorized access by applying passwords.

These and other approaches are discussed in detail in subsequent 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

File Type	READ=	WRITE=	ALTER=
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 these methods to assign, change, or delete a SAS password:

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

Here is the syntax for using PROC DATASETS to assign a password to an access descriptor, a view descriptor, or a SAS data file.

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 these 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 that VLIB.JOBC204 accesses—for example, by using a PROC SQL UPDATE. Assign a WRITE level of protection to prevent data updates.

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, you can achieve this 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. (See DBPROMPT= on page 136.) The following statement shows this: It causes a dialog box to prompt the user to enter connection information, such as a user name 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 in 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 user name changes, the SAS program does not need to be modified. Another advantage is that the same SAS program can be used by any valid user name 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 modifying your code. By using the prompt window, the new server name can be specified dynamically.

The DBPROMPT= option is not available in 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 can extract the data (or a subset of the data) and not distribute information about either the access descriptor or view descriptor.

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. For more information, see the *Base SAS Procedures Guide*.

Defining Views and Schemas

If you want to provide access to some but not all fields in a DBMS table, 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.

This example uses 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 this 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= on page 113 and DEFER= on page 141 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 example below, 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;
```

This example uses 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;
```

To assign more than one SAS libref to your DBMS server when you do not plan to update the DBMS tables, SAS/ACCESS lets you optimize how the engine makes connections. Your SAS librefs can share a single read-only connection to the DBMS if you use the CONNECTION=GLOBALREAD option. This example shows 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 in the LIBNAME statement, the SAS/ACCESS engine connects to the DBMS the first time a DBMS object is referenced in a SAS program:

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

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 on page 718 that SAS/ACCESS Interface to Oracle supports.

To lock the data pages of a table while SAS is reading the data to prevent other processes from updating the table, use the READLOCK_TYPE= on page 176 option, as shown in this example:

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

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

Here the SAS/ACCESS Oracle engine obtains a TABLE SHARE lock on the table so that other processes cannot update the data while your SAS program reads it.

In this next 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;
```

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

To specify DBMS commands or stored procedures to run immediately after a DBMS connection or before a DBMS disconnect, use the DBCONINIT= on page 125 and DBCONTERM= on page 126 options. Here is an 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. Therefore, 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= on page 130 option. Similarly, you can use the DBLIBTERM= on page 131 option to specify a command to run before the disconnection of only the first library connection. Here is an 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 receive 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. Conversely, SAS/ACCESS translates DBMS null values to SAS missing values when reading DBMS data into SAS.

However, there is 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 these 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. This interpretation 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 41 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 these strategies.

- Use the DIRECT_SQL= on page 145 LIBNAME option to control whether SAS or the DBMS handles processing.
- Use the SQL pass-through facility on page 403 to ensure that the DBMS handles processing.
- Add the *is not null* expression to WHERE clauses and ON clauses to ensure that you obtain the same result regardless of whether SAS or the DBMS does the processing.

Use the NULLCHAR= on page 329 data set option 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 this example.

```
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;
```

You can 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= on page 383 system option to improve the performance of the clients. This option tells the SAS server whether to put a hold (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 need not wait for the originating client to complete its call to the database.

Limiting Retrieval

Row and Column Selection

Limiting the number of rows that the DBMS returns to SAS is an extremely important performance consideration. The less data that the SAS job requests, the faster the job runs.

Wherever possible, specify selection criteria that limits the number of rows that the DBMS returns to SAS. Use the SAS WHERE clause to retrieve a subset of the DBMS data.

If you are interested in only the first few rows of a table, consider adding the OBS= option. SAS passes this option to the DBMS to limit the number of rows to transmit across the network, which can significantly improve performance against larger tables. To do this if you are using SAS Enterprise Guide, select **View** ⇨ **Explorer**, select the table that you want from the list of tables, and select the member that you want to see the contents of the table.

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 print data (keep=salary dept);
   where dept='ACC024';
quit;
```

The DBMS generates SQL that is similar to this:

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

Without the KEEP option, the DBMS processes SQL that is similar to this code:

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

This results in all 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, this code results in all columns from the EMPLOYEES table being retrieved into SAS. When creating the output data set, the KEEP statement is applied.

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

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

Here is how you can 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, access it directly from the database every time. Do not follow the extraction suggestions 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 work with a large DBMS table and plan to use the same DBMS data in several procedures or DATA steps during the same SAS session.

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 SELECT statement. Sort data only when it is needed for your program.

Here are 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= on page 281 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 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, the host sort rules, and 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 is available for these DBMSs.

Table 4.1 *DBMS-Specific Temporary Table Support*

DBMS	Temporary Table Support	DBMSTEMP= LIBNAME Support
Aster *n*Cluster	yes	yes
DB2 under UNIX and PC Hosts (p. 446)	yes	yes
DB2 under z/OS (p. 475)	yes	yes
Greenplum	yes	yes
HP Neoview (p. 565)	yes	yes

DBMS	Temporary Table Support	DBMSTEMP= LIBNAME Support
Informix (p. 584)	yes	no
Microsoft SQL Server	yes	yes
MySQL	no	no
Netezza (p. 631)	yes	yes
ODBC (p. 663)	yes	yes
OLE DB (p. 685)	yes	yes
Oracle (p. 709)	yes	yes
Sybase (p. 740)	yes	no
Sybase IQ	yes	yes
Teradata (p. 793)	yes	yes

The value of DBMS temporary table support in SAS is increased performance potential. By pushing processing to the DBMS in certain situations, you can achieve an overall performance gain. These processes provide a general outline of how to use DBMS temporary tables.

General Temporary Table Use

Follow these steps to use temporary tables on the DBMS.

1. Establish a global connection to the DBMS that persists 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.

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

Pushing Heterogeneous Joins

Follow these steps to push heterogeneous joins to the DBMS.

1. Establish a global connection to the DBMS that persists 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

Follow these steps to push updates (process transactions) to the DBMS.

1. Establish a global connection to the DBMS that persists 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.

Although these processing scenarios are purposely generic, they apply to each DBMS that supports temporary tables. For details, see the “DBMSTEMP= LIBNAME Option” on page 133.

Chapter 5

Optimizing Your SQL Usage

Overview: 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 access 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 these situations.

- When you use the SQL pass-through facility on page 403, 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 on page 93, 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 SQL 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 ” for information about passing WHERE clauses to the DBMS.

Note: Certain conditions 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.Also, when you join tables across multiple tables, implicit pass-through uses the first connection. Consequently, LIBNAME options from subsequent connections are ignored. You can use the SASTRACE= system option on page 385 to determine whether SAS processes an operation or whether pass-through passes it to the DBMS for processing.

To prevent operations from being passed to the DBMS, use the DIRECT_SQL= LIBNAME option on page 145.

Passing Functions to the DBMS Using PROC SQL

When you use the SAS/ACCESS LIBNAME statement, it automatically tries to pass the SAS SQL aggregate functions (MIN, MAX, AVG, MEAN, FREQ, N, SUM, and COUNT) to the DBMS because these are SQL ANSI-defined aggregate functions.

Here is a sample query of the Oracle EMP table being 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 this query.

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

SAS/ACCESS can also translate other SAS functions into DBMS-specific functions so that pass-through can pass them to the DBMS.

In this example, the SAS UPCASE function is translated into the Oracle UPPER function.

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

Here is the translated query that Oracle processes.

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

Functions that pass-through passes are different for each DBMS. Select your DBMS to see a list of functions that your SAS/ACCESS interface translates.

- Aster nCluster on page 425
- DB2 under UNIX and PC Hosts on page 452
- DB2 under z/OS on page 492
- Greenplum on page 525
- Hadoop on page 539
- HP Neoview on page 566
- Informix on page 585
- Microsoft SQL Server on page 603

- MySQL on page 616
- Netezza on page 632
- ODBC on page 665
- OLE DB on page 687
- Oracle on page 715
- Sybase on page 745
- Sybase IQ on page 767
- Teradata on page 796

Passing Joins to the DBMS

When you perform a join across SAS/ACCESS librefs in a single DBMS, PROC SQL can often pass the join to the DBMS for processing. Before implementing a join, PROC SQL checks to see whether the DBMS can process the join. A comparison is made using the SAS/ACCESS LIBNAME statement on page 93 for the librefs. Certain criteria must be met for the join to proceed. Select your DBMS to see the criteria that it requires before PROC SQL can pass the join.

- Aster nCluster on page 426
- DB2 under UNIX and PC Hosts on page 453
- DB2 under z/OS on page 493
- Greenplum on page 526
- HP Neoview on page 567
- Informix on page 586
- MySQL on page 617
- Netezza on page 633
- ODBC on page 665
- OLE DB on page 688
- Oracle on page 716
- Sybase on page 746
- Sybase IQ on page 767
- Teradata on page 797

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

These 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 such DBMSs as Informix, Oracle, and Sybase that support nonstandard outer join syntax, outer joins between two or more tables with these 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. A literal operand cannot be passed to the DBMS. Because these DBMSs do not support this, all ON clauses are transformed into WHERE clauses before trying 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, outer joins can neither consist of more than two tables nor contain a WHERE clause.
  - Sybase evaluates multijoins with WHERE clauses differently than SAS. Therefore, instead of passing multiple joins or joins with additional WHERE clauses to the DBMS, use the SAS/ACCESS DIRECT_SQL= LIBNAME option on page 145 to let PROC SQL process the join internally.

Note: If PROC SQL cannot successfully pass down a complete query to the DBMS, it might try again to pass down a subquery. You can analyze the SQL that is passed to the DBMS by turning on SAS tracing options. SAS trace information displays the exact queries that are being passed to the DBMS for processing.

In this example, TABLE1 and TABLE2 are large DBMS tables. Each has a column named DeptNo, and the value for with equal values. You want to retrieve the rows from an inner join of these tables. PROC SQL detects the join between two tables in the DBLIB library (which references an Oracle database), and SAS/ACCESS passes the join 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 and generates this Oracle code.

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

In this 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 and generates this 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 is passed to the DBMS for execution as long as it contains no WHERE clause. The DBMS deletes all rows but does not delete the table itself.

This example shows how a DELETE statement is passed to Oracle to empty the EMP table.

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

Oracle then executes this code.

```
delete from emp
```

When Passing Joins to the DBMS Will Fail

By default, SAS/ACCESS tries to pass certain types of SQL statements directly to the DBMS for processing. Most notable are SQL join statements that would otherwise be processed as individual queries to each data source that belonged to the join. In that instance, PROC SQL would then perform the join internally. 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 on 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. Here are the primary reasons why join statements might fail to be passed.

The DBMS does not accept the generated SQL syntax.
: 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 does not attempt to pass the statement to the DBMS for direct processing.

Using data set options in the query
: Specifying any data set option on a table that is referenced in the SQL query prohibits the statement from successfully passing to the DBMS for direct processing.

Using certain LIBNAME options
Specifying LIBNAME options that request such member-level controls as table locks (“READ_LOCK_TYPE= LIBNAME Option” on page 176 or “UPDATE_LOCK_TYPE= LIBNAME Option” on page 197 LIBNAME options) prohibits the statement from successfully passing to the DBMS for direct processing.

Using SAS functions on the SELECT clause
Specifying SAS functions on the SELECT clause can prevent joins from being passed.

Using the DIRECT SQL= LIBNAME option setting.
The DIRECT SQL= LIBNAME option on page 145 default setting is YES. PROC SQL attempts to pass SQL joins directly to the DBMS for processing. Other settings for DIRECT_SQL= influence the nature of the SQL statements that PROC SQL tries to pass down to the DBMS or if it tries 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. However, other SQL statements can be passed. If the “MULTI_DATASRC_OPT= LIBNAME Option” on page 160 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= LIBNAME Option” on page 160 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.

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 this example, the CUSTBASE Oracle table 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 and generates this 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

General Guidelines for WHERE Clauses

Follow the general guidelines in this table for writing efficient WHERE clauses.

Table 5.1 *Efficient WHERE Clause Guidelines*

Guideline	Inefficient	Efficient
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`**

Guideline	Inefficient	Efficient
Use DBKEY=, DBINDEX=, and MULTI_DATASRC_OPT= when appropriate. For details about these options, see "Using the DBINDEX=, DBKEY=, and MULTI_DATASRC_OPT= Options" on page 53.		

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: Optimizing Your SQL Usage" on page 45.

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, but SAS does not.

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

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 that they can be passed to the DBMS.

In this 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;
```

Generated SQL that the DBMS processes would be similar to this code.

```
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 rows from the DBMS and applies the WHERE clause.

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

- Aster nCluster on page 425
- DB2 under UNIX and PC Hosts on page 452
- DB2 under z/OS on page 492
- Greenplum on page 525
- HP Neoview on page 566
- Informix on page 585
- Microsoft SQL Server on page 603

- MySQL on page 616
- Netezza on page 632
- ODBC on page 665
- OLE DB on page 687
- Oracle on page 715
- Sybase on page 745
- Sybase IQ on page 767
- Teradata on page 796

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, SAS performs the join. Normally, this processing involves individual queries to each data source that belonged to the join, and SAS performs the join internally. 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 let you retrieve a subset of the DBMS data into SAS for the join.

When you specify MULTI_DATASRC_OPT=IN_CLAUSE for DBMS data sources in a PROC SQL join operation, the procedure retrieves the unique values of the join column from the smaller table to construct an IN clause. SAS uses this IN clause when SAS retrieves data from the larger DBMS table. SAS performs the join. If you use a SAS data set, no matter how large it is, it is always in the IN_CLAUSE. For better performance, it is recommended that the SAS data set be smaller than the DBMS table. If not, 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 the DBMASTER= data set option to designate the larger table.

MULTI_DATASRC_OPT= might provide performance improvements over DBKEY=. If you specify options, DBKEY= overrides MULTI_DATASRC_OPT=.

MULTI_DATASRC_OPT= is used only when SAS is processing a join with PROC SQL. It is not used for SAS DATA step 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 does not use the MULTI_DATASRC_OPT= optimization even when specified. If PROC SQL determines it can pass the join directly to the DBMS it also does not use this option even though it is specified.

In this example, the MULTI_DATASRC_OPT= option is used to improve the performance of an 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 rows from the Dept table and applies the WHERE clause during PROC SQL processing in SAS. Processing can be both CPU-, input-, and output-intensive if the Dept Oracle 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 SAS/ACCESS creates and passes to the DBMS is similar to this one.

```
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. 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 DBCONDITION= data set option on page 281 or the SAS WHERE=data set option.

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.

To improve the performance of joins, you can also use the DBKEY= option in a SAS DATA step, with the KEY= option in the SET statement. In this case, specify a value of KEY=DBKEY. The DATA step below creates a new data file by joining the data file KEYVALUES with the DBMS table MYTABLE. It uses the variable DEPTNO with the DBKEY= option to cause SAS/ACCESS to issue a WHERE clause.

```
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 that 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. Remember 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 make sure that the columns that you specify through the DBKEY= option match the columns that you specify 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, 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: NULLCHAR= on page 329 and NULLCHARVAL= on page 330 data set options determine how SAS missing character values are handled during DBINDEX= and DBKEY= processing.

Chapter 6

Threaded Reads

Overview: Threaded Reads in SAS/ACCESS

In SAS 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, you can reduce the table read time by retrieving the result set on multiple connections between SAS and the DBMS. SAS can 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, which improves performance.

Underlying Technology of Threaded Reads

To perform a threaded Read, SAS first creates threads within the SAS session. Threads are standard operating system tasks that SAS controls. SAS then establishes a DBMS connection on each thread, 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. Here is an example.

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

ar _ALL_;
run;
```

Previous versions of SAS opened a single connection and issued:

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

Assuming that SalesData has an EmployeeNum integer column, SAS 9.1 might open two connections by issuing these two statements:

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

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

See "Autopartitioning Techniques in SAS/ACCESS" on page 63 for more information about MOD.

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. In cases where SAS cannot always generate partitioning WHERE clauses, the SAS user can supply them. In addition to WHERE clauses, other ways to partition data might also exist.

SAS/ACCESS Interfaces and Threaded Reads

Here are the SAS/ACCESS interfaces that support threaded Reads. More interfaces are expected to support threaded Reads in future releases.

- Aster nCluster on page 424
- DB2 under UNIX and PC Hosts on page 443
- DB2 under z/OS on page 474
- Greenplum on page 523
- HP Neoview on page 563
- Informix on page 583
- ODBC on page 657
- Oracle on page 707 (not supported under z/OS)
- Sybase on page 739
- Sybase IQ on page 765
- Teradata on page 789 (supports only FastExport threaded Reads on z/OS and UNIX; see Teradata documentation for details)

Threaded Reads work across all UNIX and Windows platforms where you run SAS. For details about special considerations for Teradata on z/OS, see "Autopartitioning Scheme for Teradata " on page 789.

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 to actually perform threaded Reads. If SAS cannot automatically generate a partitioning WHERE clause or otherwise perform threaded Reads, the user can code an option that supplies partitioning. To determine whether SAS can automatically generate a partitioning WHERE clause, use the SASTRACE= on page 385 and SASTRACELOC= on page 394 system options.

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

For threaded Reads from DBMSs, SAS/ACCESS provides the DBSLICE= on page 299 and DBSLICEPARM= on page 301 data set options.

DBSLICE= applies only to a table reference. You can use it to code your own WHERE clauses to partition table data across threads, and it is useful when you are familiar with your table data. For example, 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. Here is an example.

```
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, SAS automatically attempts threaded Reads only for SAS threaded applications, which are SAS procedures that thread input, output, 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. You can specify it as a data set option, a LIBNAME option, or 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. These 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 this example:

  ```
  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 system options that apply to threaded applications. For more information about these options, see *SAS System Options: Reference*.

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 whether threaded Reads occurred and to help assess performance. These 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 these 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 contains threading information and all SQL statements that Oracle processes.

```
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, SAS places a warning in the log. For example, the data set below uses a WHERE clause that contains all possible autopartitioning columns.

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

These warnings are displayed:

```
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 nonthreading 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 these messages:

```
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=.

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

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

Here is the resulting timing information.

```
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:  0.00167
Total seconds used by the ORACLE ACCESS engine were 7.545805
```

For more information about tracing, see the SASTRACE= system option on page 385.

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. Using 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 example, threaded Reads for DB2 under z/OS involve a tradeoff, generally reducing job elapsed time but increasing DB2 workload and CPU usage. 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 these tasks.

- Turn on SAS tracing to see whether 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.

Follow these guidelines to ensure optimal tuning of threaded Reads.

- 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.
- See the DBMS-specific reference section in this document for information and tips for your 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 can experience substantial performance gains. However, you can experience 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, SAS/ACCESS Interface to Oracle automatically partitions in accordance with Oracle physical partitions rather than using MOD. SAS/ACCESS Interface to Teradata uses FastExport, if available, which lets the FastExport Utility direct 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 that your original SQL that SAS produced is **`SELECT CHR1, CHR2 FROM DBTAB`** and that the Dbtab table contains the IntCol integer column. SAS creates two threads and issues these two statements:

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

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

The first thread retrieves rows with an even value for IntCol, and the second thread retrieves rows with an odd value for IntCol. 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 do 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 use DBSLICE= to augment autopartitioning. See the information for your DBMS for specific autopartitioning details.

- Aster nCluster on page 424
- DB2 under UNIX and PC Hosts on page 443
- DB2 under z/OS on page 474
- Greenplum on page 523
- HP Neoview on page 563
- Informix on page 583
- ODBC on page 657
- Oracle on page 707 (not supported under z/OS)
- Sybase on page 739
- Sybase IQ on page 765
- Teradata on page 789 (supports only FastExport threaded Reads on z/OS and UNIX; see Teradata documentation for details)

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 Teradata on page 789 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 contains all data read on that thread. If three threads are specified for threaded Reads, three temporary spool files are written.

As the application requests subsequent passes of data, data is read from the temporary spool files, not reread 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 ways to specify this option.

- In the SAS config file, add this 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 system option, see the *SAS System Options: Reference*.

When Threaded Reads Do Not Occur

Threading does not occur under these circumstances:

- 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 column in the table exists to which SAS can apply the MOD function (for more information, see “Autopartitioning Techniques in SAS/ACCESS” on page 63)

- if all columns within a table to which SAS can apply the MOD function are in WHERE clauses. For more information, see "Autopartitioning Techniques in SAS/ACCESS" on page 63.
- if the NOTHREADS system option is set
- if DBSLICEPARM=NONE

Summary of Threaded Reads

For large reads of table data, SAS 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

Installation Requirements

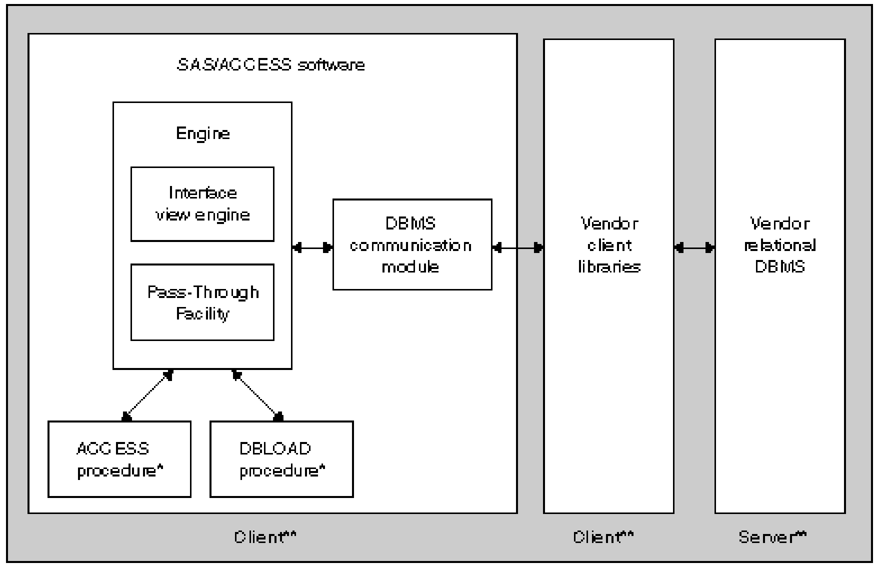
Before you use any SAS/ACCESS features, you must install Base SAS, the SAS/ACCESS interface for the DBMS that you are accessing, and any required DBMS client software. See SAS installation instructions and DBMS client installation instructions for more information.

Not all SAS/ACCESS interfaces support all features. See the DBMS-specific reference section for your SAS/ACCESS interface to determine which features are supported in your environment.

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



You can call a SAS/ACCESS relational DBMS interface by using either a LIBNAME statement on page 68 or a PROC SQL statement on page 69. (Although you can also use ACCESS and DBLOAD procedures with some of the SAS/ACCESS relational interfaces, 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

You can use SAS/ACCESS to read, update, insert, and delete data from a DBMS object as if it were a SAS data set. Here is how to do that.

1. You start 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 queries, joins, and data functions to the DBMS for processing instead of retrieving the data from the DBMS and processing it in SAS. For example, an important use of this feature is the handling of PROC SQL queries that access DBMS data. Here is 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. The DBMS processes the query or query fragment and returns the results to SAS. Any queries or query fragments that cannot be passed to the DBMS are processed in SAS.

See performance considerations on page 402 for detailed information about tasks that SAS/ACCESS can pass to the DBMS.

How the SQL Pass-Through Facility Works

When you read and update DBMS data with the SQL 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. Here is an 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. Here is an 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;

isconnect from oracle;
quit;
```

4. Terminate the connection with the DISCONNECT statement.

For more details, see the SQL pass-through facility on page 403.

How the ACCESS Procedure Works

Overview: ACCESS Procedure

When you use the ACCESS procedure to create an access descriptor, the SAS/ACCESS interface view engine requests the DBMS to execute an 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. Access descriptor information about the table and its columns is then copied into the view descriptor when it is created. It is therefore not necessary for SAS to call the DBMS when it creates a view descriptor.

Here is the process.

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 that 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 obtain 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 an 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 run the following SAS program using a view descriptor, the previous steps are executed once for the PRINT procedure and 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

Use a view descriptor, DATA step, or procedure to update DBMS data in a similar way as when you read in data. Any of these 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 an SQL INSERT statement and passes it to the DBMS. When you reference a view descriptor, 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 EXECUTE statement for the SQL pass-through facility to add, delete, or modify DBMS data directly. Literal values must be used when inserting data with the SQL pass-through facility.)
- When rows are deleted from a DBMS table, SAS constructs an 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 an 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 these 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 whether 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 an SQL CREATE TABLE statement and passes it to the DBMS.
4. SAS constructs an SQL INSERT statement for the current observation and passes it to the DBMS. New INSERT statements are constructed and executed repeatedly until all observations from the input SAS data set are passed to the DBMS. Some DBMSs have a bulk-copy capability that allows a group of observations to be inserted at once. See your DBMS documentation to determine whether your DBMS has this capability.
5. Additional non-query SQL statements that are specified in the DBLOAD procedure are executed as the user submitted them. The DBMS returns an error message if a statement does not execute successfully.
6. SAS closes the connection with the DBMS.

Chapter 8

In-Database Processing with SAS/ACCESS

Overview

When you use conventional processing to access data that is inside a DBMS, SAS asks the SAS/ACCESS engine for all table rows of the table that is being processed. The SAS/ACCESS engine generates an SQL SELECT * statement, which is passed to the DBMS. That SELECT statement fetches all rows in the table, and the SAS/ACCESS engine returns them to SAS. The number of rows in the table grows over time, so network latency grows because the amount of data that is fetched from the DBMS to SAS increases.

SAS In-Database processing integrates SAS solutions, SAS analytic processes, and third-party DBMSs. Using SAS In-Database processing, you can run scoring models, Base SAS and SAS/STAT procedures, and formatted SQL queries inside the database.

Table 8.1 *SAS Products Needed to Use In-Database Features*

In-Database Feature	Required Software	Supported DBMSs
format publishing and the SAS_PUT() function	• Base SAS • SAS/ACCESS DBMS interfaces	DB2 for UNIX and PC Hosts Netezza Teradata
scoring models	• Base SAS • SAS/ACCESS DBMS interfaces • SAS Scoring Accelerator • SAS Model Manager (optional)	Aster *n*Cluster DB2 for UNIX and PC Hosts Greenplum Netezza Teradata

In-Database Feature	Required Software	Supported DBMSs
Base SAS procedures: FREQ RANK REPORT SORT SUMMARY/MEANS TABULATE	• Base SAS • SAS/ACCESS DBMS interfaces	Aster *n*Cluster DB2 under UNIX and PC Hosts Greenplum Netezza Oracle Teradata
SAS/STAT procedures: CORR CANCORR DMDB DMINE DMREG FACTOR PRINCOMP REG SCORE TIMESERIES VARCLUS	• Base SAS (for CORR) • SAS/ACCESS Interface to Teradata • SAS/STAT (for CANCORR, FACTOR,PRINCOMP, REG, SCORE, VARCLUS) • SAS/ETS (for TIMESERIES) • SAS Enterprise Miner (for DMDB, DMINE, DMREG) • SAS Analytics Accelerator	Teradata

For more information, see *SAS In-Database Products: Administrator's Guide* and *SAS In-Database Products: User's Guide*.

Part 2

General Reference

Chapter 9

SAS/ACCESS Features by Host

Introduction

This section provides a quick reference, by host environment, of the features that are available for your SAS/ACCESS interface.

For detailed information for your particular interface, see the SAS system requirements and configuration guide documents, which are available at `http://support.sas.com`.

SAS/ACCESS Interface to Aster *n*Cluster: Supported Features

Here are the features that SAS/ACCESS Interface to Aster *n*Cluster supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.1 *Features by Host Environment for Aster nCluster*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
Linux for x86	X	X			X
Linux for x64	X	X			X
Microsoft Windows for x86	X	X			X
Microsoft Windows for x64	X	X			X

For information about these features, see “Methods for Accessing Relational Database Data” on page 4 and “Bulk Loading for Aster nCluster ” on page 427.

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

Here are the features that SAS/ACCESS Interface to DB2 under UNIX and PC Hosts supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.2 *Features by Host Environment for DB2 under UNIX and PC Hosts*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X		X	X
HP-UX	X	X		X	X

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
HP-UX for Itanium	X	X		X	X
Linux for x86	X	X		X	X
Linux for x64	X	X		X	X
Solaris for SPARC	X	X		X	X
Solaris for x64	X	X		X	X
Microsoft Windows for x86	X	X		X	X
Microsoft Windows for x64	X	X		X	X

For information about these features, see “Methods for Accessing Relational Database Data” on page 4 and “Bulk Loading for DB2 under UNIX and PC Hosts” on page 453.

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

Here are the features that SAS/ACCESS Interface to DB2 under z/OS supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.3 *Features by Host Environment for DB2 under z/OS*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
z/OS	X	X	X	X	X

For information about these features, see “Methods for Accessing Relational Database Data” on page 4 and “Bulk Loading for DB2 under z/OS” on page 496.

SAS/ACCESS Interface to Greenplum: Supported Features

Here are the features that SAS/ACCESS Interface to Greenplum supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.4 *Features by Host Environment for Greenplum*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X			X
HP-UX for Itanium	X	X			X
Linux for x86	X	X			X
Linux for x64	X	X			X
Solaris for SPARC	X	X			X
Solaris for x64	X	X			X
Microsoft Windows for x86	X	X			X

For information about these features, see “Methods for Accessing Relational Database Data” on page 4 and “Bulk Loading for Greenplum ” on page 526.

SAS/ACCESS Interface to Hadoop: Supported Features

Here are the features that SAS/ACCESS Interface to Hadoop supports.

Table 9.5 *Features by Host Environment for Hadoop*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support*
AIX	X	X			X
HP-UX for Itanium	X	X			X
Linux for x64	X	X			X
Microsoft Windows for x64	X	X			X
Microsoft Windows for x86	X	X			X
Solaris for SPARC	X	X			X
Solaris for x64	X	X			X

* SAS/ACCESS to Hadoop has no differentiation between a bulk load and a standard load process.

For information about these features, see "Methods for Accessing Relational Database Data" on page 4 and "Bulk Loading for Hadoop" on page 541.

For detailed information about the following topics, see the SAS system requirements and configuration guide documents, which are available at **`http://support.sas.com`**:

- supported Hadoop versions, character data formats support, and required JAR files, environmental variables, and patches—along with other prerequisites, required setup, and considerations for SAS/ACCESS Interface to Hadoop
- how SAS/ACCESS Interface to Hadoop interacts with Hadoop through Hive

SAS/ACCESS Interface to HP Neoview: Supported Features

Here are the features that SAS/ACCESS Interface to HP Neoview supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.6 *Features by Host Environment for HP Neoview*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X			
HP-UX	X	X			
HP-UX for Itanium	X	X			X
Linux for x86	X	X			
Solaris for SPARC	X	X			
Microsoft Windows for x86	X	X			

For information about these features, see "Methods for Accessing Relational Database Data" on page 4 and "Bulk Loading and Extracting for HP Neoview " on page 568.

SAS/ACCESS Interface to Informix: Supported Features

Here are the features that SAS/ACCESS Interface to Informix supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.7 *Features by Host Environment for Informix*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X			
HP-UX	X	X			
HP-UX for Itanium	X	X			
Linux for x64	X	X			
Solaris for SPARC	X	X			

For information about these features, see "Methods for Accessing Relational Database Data" on page 4.

SAS/ACCESS Interface to Microsoft SQL Server: Supported Features

Here are the features that SAS/ACCESS Interface to Microsoft SQL Server supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.8 *Features by Host Environment for Microsoft SQL Server*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X		X	
HP-UX	X	X		X	
HP-UX for Itanium	X	X		X	
Linux for x86	X	X		X	
Linux for x64	X	X		X	
Solaris for x64	X	X		X	
Solaris for SPARC	X	X		X	

For information about these features, see "Methods for Accessing Relational Database Data" on page 4.

SAS/ACCESS Interface to MySQL: Supported Features

Here are the features that SAS/ACCESS Interface to MySQL supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

***Table 9.9** Features by Host Environment for MySQL*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X			
HP-UX	X	X			
HP-UX for Itanium	X	X			
Linux for x86	X	X			
Linux for x64	X	X			
Solaris for SPARC	X	X			
Solaris for x64	X	X			
Microsoft Windows for x86	X	X			
Microsoft Windows for x64	X	X			

For information about these features, see "Methods for Accessing Relational Database Data" on page 4.

SAS/ACCESS Interface to Netezza: Supported Features

Here are the features that SAS/ACCESS 9.2 Interface to Netezza supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

***Table 9.10** Features by Host Environment for Netezza*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X			X

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
HP-UX	X	X			X
HP-UX for Itanium	X	X			X
Linux for x86	X	X			X
Linux for x64	X	X			X
Solaris for SPARC	X	X			X
Microsoft Windows for x86	X	X			X
Microsoft Windows for x64	X	X			X

For information about these features, see "Methods for Accessing Relational Database Data" on page 4 and "Bulk Loading and Unloading for Netezza " on page 634.

SAS/ACCESS Interface to ODBC: Supported Features

Here are the features that SAS/ACCESS Interface to ODBC supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.11 *Features by Host Environment for ODBC*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X		X	
HP-UX	X	X		X	
HP-UX for Itanium	X	X		X	
Linux for x86	X	X		X	

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
Linux for x64	X	X		X	
Solaris for SPARC	X	X		X	
Solaris for x64	X	X		X	
Microsoft Windows for x86	X	X		X	X*
Microsoft Windows for x64	X	X		X	X*

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

For information about these features, see "Methods for Accessing Relational Database Data" on page 4 and "Bulk Loading for ODBC" on page 666.

SAS/ACCESS Interface to OLE DB: Supported Features

Here are the features that SAS/ACCESS Interface to OLE DB supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.12 *Features by Host Environment for OLE DB*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
Microsoft Windows for x86	X	X			X
Microsoft Windows for x64	X	X			X

For information about these features, see "Methods for Accessing Relational Database Data" on page 4 and "Bulk Loading for OLE DB" on page 688.

SAS/ACCESS Interface to Oracle: Supported Features

Here are the features that SAS/ACCESS Interface to Oracle supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.13 *Features by Host Environment for Oracle*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X	X	X	X
HP-UX	X	X	X	X	X
HP-UX for Itanium	X	X	X	X	X
Linux for x86	X	X	X	X	X
Linux for x64	X	X	X	X	X
Solaris for SPARC	X	X	X	X	X
Solaris for x64	X	X	X	X	X
z/OS	X	X	X	X	X
Microsoft Windows for x86	X	X	X	X	X
Microsoft Windows for x64	X	X	X	X	X

For information about these features, see “Methods for Accessing Relational Database Data” on page 4 and “Bulk Loading for Oracle” on page 716.

SAS/ACCESS Interface to Sybase: Supported Features

Here are the features that SAS/ACCESS Interface to Sybase supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.14 *Features by Host Environment for Sybase*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X	X	X	X
HP-UX	X	X	X	X	X
HP-UX for Itanium	X	X	X	X	X
Linux for x86	X	X	X	X	X
Linux for x64	X	X	X	X	X
Solaris for SPARC	X	X	X	X	X
Solaris for x64	X	X	X	X	X
Microsoft Windows for x86	X	X	X	X	X
Microsoft Windows for x64	X	X	X	X	X

For information about these features, see “Methods for Accessing Relational Database Data” on page 4 and the BULKLOAD= data set option on page 274.

SAS/ACCESS Interface to Sybase IQ: Supported Features

Here are the features that SAS/ACCESS Interface to Sybase IQ supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

Table 9.15 *Features by Host Environment for Sybase IQ*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X			X
HP-UX	X	X			X
HP-UX for Itanium	X	X			X
Linux for x86	X	X			X
Linux for x64	X	X			X
Solaris for SPARC	X	X			X
Solaris for x64	X	X			X
Microsoft Windows for x86	X	X			X
Microsoft Windows for x64	X	X			X

For information about these features, see “Methods for Accessing Relational Database Data” on page 4 and “Bulk Loading for Sybase IQ ” on page 768.

SAS/ACCESS Interface to Teradata: Supported Features

Here are the features that SAS/ACCESS Interface to Teradata supports. To find out which versions of your DBMS are supported, see your system requirements documentation.

***Table 9.16** Features by Host Environment for Teradata*

Platform	SAS/ACCESS LIBNAME Statement	SQL Pass-Through Facility	ACCESS Procedure	DBLOAD Procedure	Bulk-Load Support
AIX	X	X			X
HP-UX	X	X			X
HP-UX for Itanium	X	X			X
Linux for x86	X	X			X
Linux for x64	X	X			X
Solaris for SPARC	X	X			X
Solaris for x64	X	X			X
z/OS	X	X			X
Microsoft Windows for x86	X	X			X
Microsoft Windows for x64	X	X			X

For information about these features, see “Methods for Accessing Relational Database Data” on page 4 and “Maximizing Teradata Load Performance ” on page 801.

Chapter 10

The LIBNAME Statement for Relational Databases

Overview: 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. You can use it to read from and write to a DBMS object as if 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. This section specifies the syntax of the SAS/ACCESS LIBNAME statement and provides examples. For details about the syntax, see “LIBNAME Statement Syntax for Relational Databases” on page 95.

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. DBMS format 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. Also, when you sort DBMS data, 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

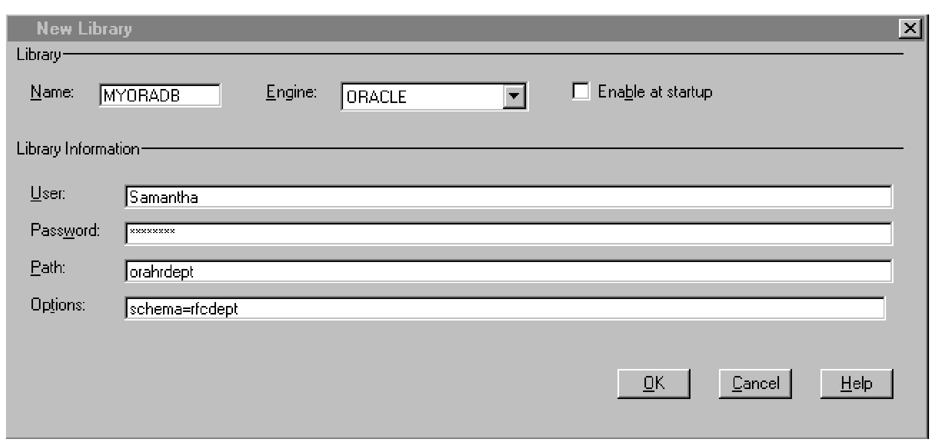
When you use librefs that refer to DBMS data with SAS functions, some functions might return a value that differs from what is returned when you use the functions with normal SAS data sets. For example, the PATHNAME function might return a blank value. For a normal SAS libref, a blank value means that 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-library* argument. When you use the LIBNAME function to assign or unassign a libref that refers to DBMS data, you omit this argument. For full details about how to use SAS functions, see the *SAS Functions and CALL Routines: Reference*.

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. You can also open the window by clicking the file cabinet icon in the SAS Explorer toolbar. Below, the user Samantha assigns a libref MYORADB to an Oracle database that the SQL*Net alias ORAHRDEPT references. By using the SCHEMA= LIBNAME option, Samantha can access database objects that another user owns.

Display 10.1 *New Library Window*



Here is 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 lets you 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.

Dictionary

LIBNAME Statement Syntax for Relational Databases

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

Valid in: anywhere

See: "Overview: LIBNAME Statement for Relational Databases " on page 93

Syntax

Form 1: **LIBNAME** *libref engine-name*
<*SAS/ACCESS-connection-options*>
<*SAS/ACCESS-LIBNAME-options*>;

Form 2: **LIBNAME** *libref* CLEAR | _ALL_ CLEAR;

Form 3: **LIBNAME** *libref* LIST | _ALL_ LIST;

Required 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. 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 DBMS-specific reference section for details.

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 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.

Optional Arguments

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. For detailed information about connection options for your DBMS, see the reference section for your SAS/ACCESS interface on page 99.

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 lets you 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. For a list of the LIBNAME options that are available for your DBMS, see the reference section for your SAS/ACCESS interface on page 99. For more information about LIBNAME options, see “LIBNAME Options for Relational Databases” on page 99.

Details

Form 1: Using Data from a DBMS

You can use a LIBNAME statement to read from and write to a DBMS table or view as if 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.

Form 2: Disassociating a Libref from a SAS Library

To disassociate or clear a libref from a DBMS, use a LIBNAME statement. Specify the libref (for example, MYDBLIB) and the CLEAR option as shown here:

```
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.

Form 3: 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 libraries to the SAS log. Specify *libref* to list the attributes of a single SAS/ACCESS library or SAS library, as shown below.

```
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 a LIBNAME statement in an SQL view that contains all information that is 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 unassigns the libref.

In this example, an SQL view of the Oracle table DEPT is created. Whenever you use this view in a SAS program, the ORALIB library is assigned. The library uses 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: You can use the USING LIBNAME syntax 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

This statement creates a libref, MYDBLIB, that uses the SAS/ACCESS interface to DB2.

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

The statement below associates the SAS libref MYDBLIB with an Oracle database that uses the SQL*Net alias AIRDB_REMOTE. You specify the SCHEMA= option in 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 SQL procedure statement below, 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

This statement uses the DBPROMPT= on page 136 LIBNAME 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 in the LIBNAME statement.

You can add the DEFER= on page 141 NO LIBNAME 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 lets you 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. The data remains on the remote machine.

This access is given to data stored in many types of SAS files. Examples include external databases (through the SAS/ACCESS LIBNAME statement and views that are created with it) and SAS data views (views that are created with PROC SQL, the DATA step, and SAS/ACCESS software). RLS lets you access SAS data sets, SAS views, and relational DBMS data that SAS/ACCESS LIBNAME statements define. 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 next 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;
```

DBMS Interface Specifics for the LIBNAME Statement

For the engine name, connection options, and LIBNAME options for your SAS/ACCESS interface, select your DBMS.

- Aster nCluster on page 416
- DB2 under UNIX and PC Hosts on page 434
- DB2 under z/OS on page 467
- Greenplum on page 516
- Hadoop on page 536
- HP Neoview on page 556
- Informix on page 576
- Microsoft SQL Server on page 594
- MySQL on page 610
- Netezza on page 624
- ODBC on page 646
- OLE DB on page 672
- Oracle on page 698
- Sybase on page 732
- Sybase IQ on page 758
- Teradata on page 779

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 “About the Data Set Options for Relational Databases” on page 207. For general information about the LIBNAME statement, see “LIBNAME Statement Syntax for Relational Databases” on page 95.

Many LIBNAME options are also available for use with the SQL pass-through facility. See the section on the SQL pass-through facility in the documentation for your SAS/ACCESS interface to determine which LIBNAME options are available in the SQL pass-through facility for your DBMS. For general information about SQL pass-through, see “SQL Pass-Through Facility” on page 401.

For a list of the LIBNAME options available in your SAS/ACCESS interface, see “DBMS Interface Specifics for the LIBNAME Statement” on page 99.

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: SAS/ACCESS LIBNAME statement

Default: none

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, Hadoop, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

Syntax

ACCESS=READONLY

Syntax Description

READONLY
specifies that you can read but not update tables and views.

Details

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

ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS= LIBNAME Option

Specifies whether to adjust the lengths of CHAR or VARCHAR data type columns that byte semantics specify.

Valid in: SAS/ACCESS LIBNAME statement

Default: conditional (see “Syntax Description”)

Data source: Oracle

See: ADJUST_NCHAR_COLUMN_LENGTHS= LIBNAME option, DBCLIENT_MAX_BYTES= LIBNAME option, DBSERVER_MAX_BYTES= LIBNAME option

Syntax

ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=YES | NO

Syntax Description

YES
indicates that column lengths are divided by the DBSERVER_MAX_BYTES= value and then multiplied by the DBCLIENT_MAX_BYTES= value. So if DBCLIENT_MAX_BYTES is greater than 1, ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=YES.

NO
indicates that any column lengths that byte semantics specify on the server are used as is on the client. So if DBCLIENT_MAX_BYTES=1, ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=NO.

Example: Adjust Client-Encoded Column Lengths

When ADJUST_BYTE_SEMANTICS_COLUMN_LENGTHS=YES, column lengths that byte semantics creates are adjusted with client encoding, as shown in this example.

```
libname x3 &engine &connopt ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=YES;
proc contents data=x3.char_sem; run;
proc contents data=x3.nchar_sem; run;
proc contents data=x3.byte_sem; run;
proc contents data=x3.mixed_sem; run;
```

In this example, various options have different settings.

```
libname x5 &engine &connopt ADJUST_NCHAR_COLUMN_LENGTHS=NO
ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=NO DBCLIENT_MAX_BYTES=3;
proc contents data=x5.char_sem; run;
proc contents data=x5.nchar_sem; run;
proc contents data=x5.byte_sem; run;
proc contents data=x5.mixed_sem; run;
```

This example also uses different settings for the various options.

```
libname x6  &engine &connopt ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=YES
ADJUST_NCHAR_COLUMN_LENGTHS=YES DBCLIENT_MAX_BYTES=3;
proc contents data=x6.char_sem; run;
proc contents data=x6.nchar_sem; run;
proc contents data=x6.byte_sem; run;
proc contents data=x6.mixed_sem; run;
```

ADJUST_NCHAR_COLUMN_LENGTHS= LIBNAME Option

Specifies whether to adjust the lengths of CHAR or VARCHAR data type columns.

Valid in: SAS/ACCESS LIBNAME statement

Default: YES

Data source: Oracle

See: ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS= LIBNAME option, DBCLIENT_MAX_BYTES= LIBNAME option, DBSERVER_MAX_BYTES= LIBNAME option

Syntax

ADJUST_NCHAR_COLUMN_LENGTHS=YES | NO

Syntax Description

YES

indicates that column lengths are multiplied by the DBSERVER_MAX_BYTES= value.

NO

indicates that column lengths that NCHAR or NVARCHAR columns specify are multiplied by the maximum number of bytes per character value of the national character set for the database.

Example: No Adjustment for Client-Encoded Column Lengths

NCHAR column lengths are no longer adjusted to client encoding when ADJUST_NCHAR_COLUMN_LENGTHS=NO, as shown in this example.

```
libname x2 &engine &connopt ADJUST_NCHAR_COLUMN_LENGTHS=NO;
proc contents data=x2.char_sem; run;
proc contents data=x2.nchar_sem; run;
proc contents data=x2.byte_sem; run;
proc contents data=x2.mixed_sem; run;
```

In this example, various options have different settings.

```
libname x5 &engine &connopt ADJUST_NCHAR_COLUMN_LENGTHS=NO
ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=NO DBCLIENT_MAX_BYTES=3;
proc contents data=x5.char_sem; run;
proc contents data=x5.nchar_sem; run;
proc contents data=x5.byte_sem; run;
proc contents data=x5.mixed_sem; run;
```

This example also uses different settings for the various options.

```
libname x6  &engine &connopt ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=YES
ADJUST_NCHAR_COLUMN_LENGTHS=YES DBCLIENT_MAX_BYTES=3;
proc contents data=x6.char_sem; run;
proc contents data=x6.nchar_sem; run;
proc contents data=x6.byte_sem; run;
proc contents data=x6.mixed_sem; run;
```

AUTHDOMAIN= LIBNAME Option

Allows connection to a server by specifying the name of an authentication domain metadata object.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Requirement: For data sources with the SERVER= option, if you specify AUTHDOMAIN=, you must also specify SERVER=. However, the authentication domain references credentials, so you need not explicitly specify USER= and PASSWORD=. An example is **`authdomain=MyServerAuth`**.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, Hadoop, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

Syntax

AUTHDOMAIN=*authentication-domain*

Syntax Description

authentication-domain
name of an authentication domain metadata object.

Details

An administrator creates authentication domain definitions while creating a user definition with the User Manager in SAS Management Console. The authentication domain is associated with one or more login metadata objects, which provide access to the server. The authentication domain is resolved when the DBMS engine calls the SAS Metadata Server and returns the authentication credentials.

The authentication domain and the associated login definition must be stored in a metadata repository and the metadata server must be running to resolve the metadata object specification.

For complete information about creating and using authentication domains, see the credential management topic in *SAS Intelligence Platform: Security Administration Guide*.

AUTHID= LIBNAME Option

Allows qualified table names with an authorization ID, a user ID, or a group ID.

Valid in: SAS/ACCESS LIBNAME statement

Alias: SCHEMA=

Default: none

Data source: DB2 under z/OS

See: AUTHID= data set option

Syntax

AUTHID=*authorization-ID*

Syntax Description

authorization-ID
cannot exceed eight 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.

AUTOCOMMIT= LIBNAME Option

Indicates whether updates are committed immediately after they are submitted.

Valid in: SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. See the DBMS-specific reference section for details.

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Sybase, Sybase IQ

See: AUTOCOMMIT= data set option

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 to update data.

Informix, MySQL: The default is YES.

Netezza: The default is YES for PROC PRINT but NO for updates and for the main LIBNAME connection. For read-only connections and the SQL pass-through facility, the default is YES.

BL_KEEPIDENTITY= LIBNAME Option

Determines whether the identity column that is created during bulk loading is populated with values that Microsoft SQL Server generates or with values that the user provides.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: OLE DB

See: BL_KEEPIDENTITY= data set option, BULKLOAD= LIBNAME option

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 valid only when you use the Microsoft SQL Server provider.

BL_KEEPNULLS= LIBNAME Option

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

Valid in:	SAS/ACCESS LIBNAME statement
Default:	YES
Requirement:	To specify this option, you must first set BULKLOAD=YES.
Data source:	OLE DB
See:	BL_KEEPNULLS= data set option, BULKLOAD= LIBNAME option

Syntax

BL_KEEPNULLS=YES | NO

Syntax Description

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

NO
specifies that Microsoft SQL Server replaces NULL values that the OLE DB interface inserts with a default value, as specified in the DEFAULT constraint.

Details

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

BL_LOG= LIBNAME Option

Specifies the name of the error file where all errors are written when BULKLOAD=YES.

Valid in:	SAS/ACCESS LIBNAME statement
Default:	none
Restriction:	This option is valid only for connections to Microsoft SQL Server.
Requirement:	To specify this option, you must first set BULKLOAD=YES.
Data source:	Microsoft SQL Server, ODBC
See:	BL_LOG= data set option, BULKLOAD= LIBNAME option

Syntax

BL_LOG=*filename*

Details

If you do not specify BL_LOG=, errors are not recorded during bulk loading.

BL_NUM_ROW_SEPS= LIBNAME Option

Specifies the number of newline characters to use as the row separator for the load or extract data stream.

Valid in: SAS/ACCESS LIBNAME statement

Default: 1

Requirements: To specify this option, you must first set BULKLOAD=YES.
You must specify an integer for this option that is greater than 0.

Data source: HP Neoview

See: BL_NUM_ROW_SEPS= data set option, BULKEXTRACT= LIBNAME option, BULKEXTRACT= data set option, BULKLOAD= LIBNAME option, BULKLOAD= data set option

Syntax

BL_NUM_ROW_SEPS=<*integer*>

Details

If your character data contains newline characters and you want to avoid parsing issues, you can specify a greater number for BL_NUM_ROW_SEPS=. This corresponds to the *records separated by* clause in the HP Neoview Transporter control file.

BL_OPTIONS= LIBNAME Option

Passes options to the DBMS bulk-load facility, which affects how it loads and processes data.

Valid in: SAS/ACCESS LIBNAME statement

Default: not specified

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: ODBC, OLE DB

See: BL_OPTIONS= data set option, BULKLOAD= LIBNAME option, BULKLOAD= data set option, UPDATE_LOCK_TYPE= LIBNAME option

Syntax

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

Details

You can use BL_OPTIONS= to pass options to the DBMS bulk-load facility when it is called, 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. See the Microsoft SQL Server documentation for more information about bulk copy options.

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

ODBC: 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.

BL_PORT= LIBNAME Option

Specifies the port number for writing table data to the Hadoop cluster.

Valid in: SAS/ACCESS LIBNAME statement

Alias: BLPORT=, BULKLOAD_PORT=

Default: 8020

Restriction: This option is required only if Hadoop HDFS streaming is running on a port other than the default port 8020.

Data source: Hadoop

Tip: You can use this option without setting BULKLOAD=YES.

See: PORT= connection option

Syntax

BL_PORT=<*port*>

Syntax Description

port
specifies the port number to use to write table data.

Details

To create a new table or append to an existing table, use this option to specify only the port number that bulk load uses to write data to the Hadoop cluster using the HDFS streaming server.

BULKEXTRACT= LIBNAME Option

Rapidly retrieves (fetches) a large number of rows from a data set.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: HP Neoview

See: BULKEXTRACT= data set option, HP Neoview bulk extracting

Syntax

BULKEXTRACT=YES | NO

Syntax Description

YES
calls the HP Neoview Transporter to retrieve data from HP Neoview.

NO
uses standard HP Neoview result sets to retrieve data from HP Neoview.

Details

Using BULKEXTRACT=YES is the fastest way to retrieve large numbers of rows from an HP Neoview table.

BULKLOAD= LIBNAME Option

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

Valid in: SAS/ACCESS LIBNAME statement

Alias: FASTLOAD= [Teradata]

Default: NO

Data source: Hadoop, ODBC, OLE DB, Teradata

See: BULKUNLOAD= LIBNAME option, BULKLOAD= data set option, BULKUNLOAD= data set option

Syntax

BULKLOAD=YES | NO

Syntax Description

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

NO
does not call the DBMS bulk-load facility.

BULKUNLOAD= LIBNAME Option

Rapidly retrieves (fetches) a large number of rows from a data set.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Netezza

See: BULKUNLOAD= data set option, BULKLOAD= LIBNAME option, Netezza bulk unloading

Syntax

BULKUNLOAD=YES | NO

Syntax Description

YES
: calls the Netezza Remote External Table interface to retrieve data from the Netezza Performance Server.

NO
: uses standard Netezza result sets to retrieve data from the DBMS.

Details

Using BULKUNLOAD=YES is the fastest way to retrieve large numbers of rows from a Netezza table.

CAST= LIBNAME Option

Specifies whether SAS or the Teradata DBMS server performs data conversions.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Teradata

See: CAST= data set option, CAST_OVERHEAD_MAXPERCENT= LIBNAME option, CAST_OVERHEAD_MAXPERCENT= data set option

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 suitable for performing this work. This is especially true when running SAS on z/OS, 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.

These Teradata types are candidates for casting.

- INTEGER
- BYTEINT
- SMALLINT
- DECIMAL
- DATE

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

You can alter the casting rules by using the CAST= or CAST_OVERHEAD_MAXPERCENT= LIBNAME option. With CAST_OVERHEAD_MAXPERCENT=, you can change the 20% 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= applies only when you are reading Teradata tables into SAS, not when you are writing Teradata tables from SAS.

Also, CAST= applies only 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. Data conversions are therefore forced to occur in Teradata instead of SAS.

Examples

Example 1: Force Casting for All Tables

This example demonstrates the use of the CAST= option in a LIBNAME statement to force casting for all referenced tables.

```
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;
```

Example 2: Turn Casting Off for a Specific Table Reference

This example demonstrates the use of the CAST= option in a table reference to turn casting off for that table.

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

CAST_OVERHEAD_MAXPERCENT= LIBNAME Option

Specifies the overhead limit for data conversions to perform in Teradata instead of SAS.

Valid in: SAS/ACCESS LIBNAME statement

Default: 20

Data source: Teradata

See: CAST= LIBNAME option, CAST= data set option, CAST_OVERHEAD_MAXPERCENT= data set option

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. Either Teradata or SAS can perform conversions. When Teradata performs the conversion, the row size that is transmitted to SAS using the Teradata CAST operator can increase. CAST_OVERHEAD_MAXPERCENT= limits the allowable increase, also called *conversion overhead*.

Example: Increase the Allowable Overhead

This example demonstrates the use of CAST_OVERHEAD_MAXPERCENT= to increase the allowable overhead to 40%.

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

CELLPROP= LIBNAME Option

Modifies the metadata and content of a result data set that the MDX command defines.

Valid in: SAS/ACCESS LIBNAME statement

Default: VALUE

Data source: OLE DB

See: "Accessing OLE DB for OLAP Data"

Syntax

CELLPROP=VALUE | FORMATTED_VALUE

Syntax Description

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

FORMATTED_VALUE
specifies that the SAS/ACCESS engine returns formatted data values. All 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.

CHAR_AS_NCHAR= LIBNAME Option

Specifies the default character type to use for table columns.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Netezza

See: DBTYPE= data set option on page 303

Syntax

CHAR_AS_NCHAR=YES | NO

Syntax Description

YES
specifies that NCHAR or NVARCHAR be used as the default column type.

NO
specifies that CHAR or VARCHAR be used as the default column type.

Details

Use this option when you cannot use the DBTYPE= data set option for table columns that contain multilingual character data.

Example: Set Multilingual Data as the Default

The SAS data set, local_cust, contains multilingual data in this example.

```
libname net netezza server=netsrv database=test uid=net1 pwd=net1 CHAR_AS_NCHAR=YES;

data net.customers;
set sas.local_cust;
run;
```

COMMAND_TIMEOUT= LIBNAME Option

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

Valid in: SAS/ACCESS LIBNAME statement

Default: 0

Data source: OLE DB

See: COMMAND_TIMEOUT= data set option

Syntax

COMMAND_TIMEOUT=*number-of-seconds*

Syntax Description

number-of-seconds
an integer greater than or equal to 0, where 0 represents no time-out.

CONNECTION= LIBNAME Option

Specifies whether operations on a single or multiple librefs share a connection to the DBMS.

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS-specific

Interaction: For DBMSs that default to CONNECTION=UNIQUE, the LIBNAME connection can fail when you use SQL_FUNCTIONS= for that same DBMS to store the external SQL dictionary.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: ACCESS= LIBNAME option, CONNECTION_GROUP= LIBNAME option, DEFER= LIBNAME option

Syntax

CONNECTION=SHAREDREAD | UNIQUE | SHARED | GLOBALREAD | GLOBAL

Syntax Description

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 [not valid for MySQL]

specifies that *all* operations that access DBMS tables *in a single libref* share a single connection.

Use this option with caution. When you use a single SHARED connection for multiple table opens, performing a commit or rollback on one table that is being updated also applies to all other tables that are open for update. Even if you open a table only for READ, its READ cursor might be resynchronized due to 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 exists in the same database or table space. This happens only in certain output processing situations and is the only recommended for use with CONNECTION=SHARED.

Note: The CONNECTION= option influences only connections that you use to open tables with a libref. When you set CONNECTION=SHARED, it has no influence on utility connections or explicit pass-through connections.

GLOBALREAD

specifies that all READ operations that access DBMS tables *in multiple librefs* share a single connection if the participating librefs that LIBNAME statements create specify identical values for these options:

- CONNECTION=, CONNECTION_GROUP=, DBCONINIT=, DBCONTERM=, DBLIBINIT=, and DBLIBTERM= options
- any DBMS connection options

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

GLOBAL [not valid for MySQL]

specifies that *all* operations that access DBMS tables *in multiple librefs* share a single connection if all participating librefs that LIBNAME statements create specify identical values for these options:

- CONNECTION=, CONNECTION_GROUP=, DBCONINIT=, DBCONTERM=, DBLIBINIT=, and DBLIBTERM= options
- any DBMS connection options

One connection is shared for all tables that any libref references for which you specify CONNECTION=GLOBAL.

Use this option with caution. When you use a GLOBAL connection for multiple table opens, performing a commit/rollback on one table that is being updated also applies to all other tables that are open for update. Even if you open a table only for

READ, its READ cursor might be 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.

When you set CONNECTION=GLOBAL, any pass-through code that you include after the LIBNAME statement can share the connection. For details, see the CONNECT statement example on page 408 for the pass-through facility.

Details

SAS/ACCESS interfaces that support single or multiple, simultaneous connections to the DBMS support this option. Not all values are valid for all SAS/ACCESS interfaces.

For most SAS/ACCESS interfaces, there must be a connection, also known as an *attach*, to the DBMS server before they can access any data. 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 lets you 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=.

Aster n*Cluster, MySQL:* The default value is UNIQUE.

Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, Sybase IQ: 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: For channel-attached systems (z/OS), the default is SHAREDREAD; for network attached systems (UNIX and PC platforms), the default is UNIQUE.

Examples

Example 1: Use SHAREDREAD

In this 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.

```
/* connection 1 */
libname mydblib oracle user=testuser
        pw=testpass path='myorapath'
        connection=sharedread;
/* connection 2 */
libname mydblib2 oracle user=testuser
        pw=testpass path='myorapath'
        connection=sharedread;
proc print data=mydblib.tab…
/* connection 3 */
proc sql;
   update mydblib.tab…
libname mydblib clear;
libname mydblib2 clear;
```

Example 2: Use GLOBALREAD

In this 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 prints the data from MYDBLIB.TAB while a second connection updates MYDBLIB.TAB. The second connection is closed at the end of the step. The first connection is closed with the final LIBNAME statement.

```
/* connection 1 */
libname mydblib oracle user=testuser
        pw=testpass path='myorapath'
        connection=globalread;
libname mydblib2 oracle user=testuser
        pw=testpass path='myorapath'
        connection=globalread;
proc print data=mydblib.tab…
/* connection 2 */
proc sql;
   update mydblib.tab…
/* does not close connection 1 */
libname mydblib clear;
/* closes connection 1 */
libname mydblib2 clear;
```

Example 3: Use UNIQUE

In this example, the MYDBLIB libref 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 update 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;
```

Example 4: Use SHARED

In this SHARED example, DB2DATA.NEW is created in the database TEST. The DB2DATA.OLD table exists in the same database. So the CONNECTION=SHARED option lets the DB2 engine share the connection for reading the old table and also creating and loading the new table.

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

If you did not use the CONNECTION= option in this case, you would deadlock in DB2 and receive this 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.
```

Example 5: Use GLOBAL

In this example for DB2 under z/OS, both PROC DATASETS invocations appropriately report “no members in directory.” This happens because SESSION.B, as a temporary table, has no entry in the SYSIBM.SYSTABLES system catalog. 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 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;
/*  Access the temp table through the global libref. */
data _null_;
set x.b;
put _all_;
run;
/*  Access 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 this 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 this last example, you would deadlock in DB2 and receive this 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.
```

CONNECTION_GROUP= LIBNAME Option

Causes operations on multiple librefs and on multiple SQL pass-through facility CONNECT statements to share a connection to the DBMS.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: CONNECTION= LIBNAME option

Syntax

CONNECTION_GROUP=*connection-group-name*

Syntax Description

connection-group-name
name of a connection group.

Details

This option causes a DBMS connection to be shared by all READ operations on multiple librefs if all participating librefs that LIBNAME statements create specify these items:

- the same value for the CONNECTION_GROUP= option
- 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.

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

Informix: The CONNECTION_GROUP option enables multiple librefs or multiple SQL pass-through facility CONNECT statements to share a connection to the DBMS. This overcomes the Release 8.2 limitation where users were unable to access scratch tables across step boundaries as a result of new connections being established with every procedure.

Example: Share a Connection with Identical Connection Options

In this 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 and also for updating MYDBLIB.TAB. The third connection is closed at the end of the step. 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.

```
/* connection 1 */
libname mydblib oracle user=testuser
        pw=testpass
        connection_group=mygroup;
libname mydblib2 oracle user=testuser
        pw=testpass
        connection_group=mygroup;
/* connection 2 */
libname mydblib3 oracle user=testuser
        pw=testpass
        connection_group=abc;
proc print data=mydblib.tab…
/* connection 1 */
proc sql;
   update mydblib.tab…
/* does not close connection 1*/
libname mydblib clear;
/* closes connection 1 */
libname mydblib2 clear;
/* closes connection 2 */
libname mydblib3 clear;
```

CONNECTION_TIMEOUT= LIBNAME Option

Specifies the number of seconds to wait before a connection times out.

Valid in: SAS/ACCESS LIBNAME statement

Alias: CON_TIMEOUT=

Default: 0

Data source: HP Neoview

Syntax

CONNECTION_TIMEOUT=*number-of-seconds*

Syntax Description

number-of-seconds
a number greater than or equal to 0. It represents the number of seconds that SAS/ACCESS Interface to HP Neoview waits for any operation on the connection to complete before returning to SAS. When the value is 0 (the default), no time-out occurs.

CURSOR_TYPE= LIBNAME Option

Specifies the cursor type for read-only and updatable cursors.

Valid in: SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. See the DBMS-specific reference section for details.

Default: DBMS- and operation-specific

Data source: DB2 under UNIX and PC Hosts, Microsoft SQL Server, ODBC, OLE DB, Sybase IQ

See: CURSOR_TYPE= data set option

Syntax

CURSOR_TYPE=DYNAMIC | FORWARD_ONLY | KEYSET_DRIVEN | STATIC

Syntax Description

DYNAMIC
specifies that the cursor reflects all 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 under UNIX and PC Hosts, Microsoft SQL Server, and ODBC interfaces.

FORWARD_ONLY [not valid for OLE DB]
specifies that the cursor functions like a DYNAMIC cursor except that it supports only sequential fetching of rows.

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. See your database documentation for more information.

When no options have been set yet, here are the initial DBMS-specific defaults.

DB2 under UNIX and PC Hosts	Microsoft SQL Server	ODBC	OLE DB	Sybase IQ
KEYSET_DRIVEN	DYNAMIC	FORWARD_ONLY	FORWARD_ONLY	DYNAMIC

Here are the operation-specific defaults.

Operation	DB2 under UNIX and PC Hosts	Microsoft SQL Server	ODBC	OLE DB	Sybase IQ
insert (UPDATE_SQL=NO)	KEYSET_DRIVEN	DYNAMIC	KEYSET_DRIVEN	FORWARD_ONLY	KEYSET_DRIVEN

Operation	DB2 under UNIX and PC Hosts	Microsoft SQL Server	ODBC	OLE DB	Sybase IQ
read (such as PROC PRINT)	driver default			driver default (FORWARD_ONLY)	FORWARD_ONLY
update (UPDATE_SQL=NO)	KEYSET_DRIVEN	DYNAMIC	KEYSET_DRIVEN	FORWARD_ONLY	KEYSET_DRIVEN
CONNECTION=GLOBAL CONNECTION=SHARED		DYNAMIC		DYNAMIC	FORWARD_ONLY

OLE DB: Here are the OLE DB properties that are applied to an open rowset. For details, see your OLE DB programmer reference documentation.

CURSOR_TYPE=	OLE DB Properties Applied
FORWARD_ONLY or DYNAMIC (see “Details”)	DBPROP_OTHERINSERT=TRUE, DBPROP_OTHERUPDATEDELETE=TRUE
KEYSET_DRIVEN	DBPROP_OTHERINSERT=FALSE, DBPROP_OTHERUPDATEDELET=TRUE
STATIC	DBPROP_OTHERINSERT=FALSE, DBPROP_OTHERUPDATEDELETE=FALSE

DB_LENGTH_SEMANTICS_BYTE= LIBNAME Option

Indicates whether CHAR and VARCHAR2 column lengths are specified in bytes or characters when creating an Oracle table.

Valid in: SAS/ACCESS LIBNAME statement

Default: YES

Data source: Oracle

See: DBSERVER_MAX_BYTES= LIBNAME option

Syntax

DB_LENGTH_SEMANTICS_BYTE=YES | NO

Syntax Description

YES
specifies that CHAR and VARCHAR2 column lengths are specified in bytes when creating an Oracle table. The byte length is derived by multiplying the number of characters in SAS with DBSERVER_MAX_BYTES= value.

NO
specifies that CHAR and VARCHAR2 column lengths are specified in characters when creating an Oracle table. The CHAR keyword is also added next to the length value to indicate that this is the character (not byte) length. For fixed-width encoding, the number of characters is derived by dividing the byte length in SAS for the variable by the value in DBCLIENT_MAX_BYTES=. For variable-width encoding, the number of characters remains the same as the number of bytes.

Details

This option is appropriate only when creating Oracle tables from SAS. It is therefore ignored in other contexts, such as reading or updating tables.

Length values chosen for variable-width encodings might be more than what is actually needed.

DB_OBJECTS= LIBNAME Option

Specifies which database objects to return with PROC DATASETS or in SAS Explorer.

Valid in: SAS/ACCESS LIBNAME statement

Default: (TABLES VIEWS)

Restriction: Because SHOW_SYNONYMS=YES overrides DB_OBJECTS= and is available only for backward compatibility, you should instead use DB_OBJECT=SYNONYMS.

Data source: Oracle

See: SHOW_SYNONYMS= LIBNAME option

Syntax

DB_OBJECTS=(TABLES | VIEWS | SYNONYMS | ALL)

Syntax Description

TABLES
returns only database table names.

VIEWS
returns only database view names.

SYNONYMS
returns only database synonym names.

ALL
returns all database object names, which can slow performance. Specify ALL by itself. It always overrides any multiple values that you specify.

Example: Specify Multiple Objects

```
DB_OBJECTS=(VIEWS SYNONYMS)
```

DBCLIENT_MAX_BYTES= LIBNAME Option

Specifies the maximum number of bytes per single character in the database client encoding, which matches SAS encoding.

Valid in: SAS/ACCESS LIBNAME statement

Default: always set to match the maximum bytes per single character of SAS session encoding

Data source: Oracle

See: ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS= LIBNAME option, , ADJUST_NCHAR_COLUMN_LENGTHS= LIBNAME option, DBSERVER_MAX_BYTES= LIBNAME option

Syntax

DBCLIENT_MAX_BYTES=*max-client-bytes*

Details

Use this option as the multiplying factor to adjust column lengths for CHAR and NCHAR columns for client encoding. In most cases, you need not set this option because the default is sufficient.

Examples

Example 1: Use Default Values for All Options

This example uses default values for all options.

```
libname x1 &engine &connopt
proc contents data=x1.char_sem; run;
proc contents data=x1.nchar_sem; run;
proc contents data=x1.byte_sem; run;
proc contents data=x1.mixed_sem; run;
```

Example 2: Specify Different Settings for Various Options

Here is one example of this.

```
libname x5 &engine &connopt ADJUST_NCHAR_COLUMN_LENGTHS=NO
ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=NO DBCLIENT_MAX_BYTES=3;
proc contents data=x5.char_sem; run;
proc contents data=x5.nchar_sem; run;
proc contents data=x5.byte_sem; run;
proc contents data=x5.mixed_sem; run;
```

This next example also uses different settings for the various options.

```
libname x6  &engine &connopt ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=YES
ADJUST_NCHAR_COLUMN_LENGTHS=YES DBCLIENT_MAX_BYTES=3;
```

```
proc contents data=x6.char_sem; run;
proc contents data=x6.nchar_sem; run;
proc contents data=x6.byte_sem; run;
proc contents data=x6.mixed_sem; run;
```

DBCOMMIT= LIBNAME Option

Causes an automatic COMMIT (permanently writing data to the DBMS) after processing a specified number of rows.

Valid in: SAS/ACCESS LIBNAME statement

Alias: CHECKPOINT= [Teradata]

Default: 1000 when a table is created and rows are inserted in a single step (DATA STEP); 0 when rows are inserted, updated, or deleted from an existing table (PROC APPEND or PROC SQL inserts, updates, or deletes)

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Informix, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: BULKLOAD= data set option, DBCOMMIT= data set option, ERRLIMIT= data set option, “Maximizing Teradata Load Performance ”, ML_CHECKPOINT= data set option, “Using FastLoad ”

Syntax

DBCOMMIT=*n*

Syntax Description

n

specifies 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. Usually, when you set DBCOMMIT=0, COMMIT is issued only once: after a procedure or DATA step completes. However, the commit is performed after each statement when you use the SQL procedure.

If you explicitly set this option, SAS/ACCESS fails any update with a WHERE clause.

Note: If you specify both DBCOMMIT= and ERRLIMIT= and these options collide during processing, COMMIT is issued first and ROLLBACK is issued second. Because COMMIT is issued (through the DBCOMMIT= option) before ROLLBACK (through the ERRLIMIT= option), DBCOMMIT= overrides ERRLIMIT=.

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

Teradata: See the FastLoad description in the Teradata section for the default behavior of this option. DBCOMMIT= and ERRLIMIT= are disabled for MultiLoad to prevent any conflict with ML_CHECKPOINT= data set option.

DBCONINIT= LIBNAME Option

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

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBCONTERM= LIBNAME option

Syntax

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

Syntax Description

DBMS-user-command
any valid command that the SAS/ACCESS engine can execute 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, a disconnection 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

Example 1: Apply the SET Statement to Every Connection

In this 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;
```

Example 2: Pass a Stored Procedure

In this 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.

DBCONTERM= LIBNAME Option

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

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBCONINIT= LIBNAME option

Syntax

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

Syntax Description

DBMS-user-command

any valid command that the SAS/ACCESS engine can execute 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, SAS provides a warning message, but unassigning the library and disconnecting from the DBMS 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

Example 1: Drop a Table Before Disconnecting

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

```
libname mydblib db2 user=testuser using=testpass
        datasrc=invoice dbconterm='drop table q1_sales';
```

Example 2: Execute a Stored Procedure at Each DBMS Connection

In this 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 datasrc=sales
        dbconinit='exec salestab_stored_proc'
        dbconterm='drop table bonuses';
```

DBCREATE_TABLE_EXTERNAL= LIBNAME Option

Specifies whether to include the EXTERNAL keyword when creating a new table.

Valid in: SAS/ACCESS LIBNAME statement

Alias: DBCREATE_EXTERNAL=,DBCREATE_EXT=

Default: NO

Interaction: You can specify this option, the DBCREATE_TABLE_LOCATION= option, or both.

Data source: Hadoop

Tip: This option determines only the disposition of a file upon delete.

See: DBCREATE_TABLE_EXTERNAL= data set option, DBCREATE_TABLE_LOCATION= data set option, DBCREATE_TABLE_OPTS= LIBNAME option, DBCREATE_TABLE_OPTS= data set option

Syntax

DBCREATE_TABLE_EXTERNAL=YES | NO

Syntax Description

YES
specifies that the EXTERNAL keyword is to be inserted between the CREATE and TABLE keywords when creating a new table.

NO
specifies that the Hive table and its associated data files are deleted.

Details

Use this option to indicate how to dispose of data files that comprise the Hive table when that table is deleted using a DROP TABLE tabname Hive command. When set to YES and the Hive table is deleted, its associated data files are not deleted.

Example: Creating a File in an Alternative Hive Depository

Both DBCREATE_TABLE_EXTERNAL= and DBCREATE_TABLE_LOCATION= options are set in this example.

```
LIBNAME db HADOOP SERVER=myserver USER=myuser DB=myschema;
DATA db.mytab (
       DBCREATE_TABLE_EXTERNAL=YES
       DBCREATE_TABLE_LOCATION="/mydir/mytab");
  SET mydata;
RUN;
```

DBCREATE_TABLE_OPTS= LIBNAME Option

Specifies DBMS-specific syntax to add to the CREATE TABLE statement.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBCREATE_TABLE_EXTERNAL= LIBNAME option, DBCREATE_TABLE_EXTERNAL= data set option, DBCREATE_TABLE_LOCATION= data set option, DBCREATE_TABLE_OPTS= data set option

Syntax

DBCREATE_TABLE_OPTS='*DBMS-SQL-clauses*'

Required Argument

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

Details

You can use DBCREATE_TABLE_OPTS= 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.

DBGEN_NAME= LIBNAME Option

Specifies how SAS automatically renames to valid SAS variable names any DBMS columns that contain characters that SAS does not allow.

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBGEN_NAME= data set option, VALIDVARNAME= system option

Syntax

DBGEN_NAME=DBMS | SAS

Syntax Description

DBMS
specifies that SAS renames DBMS columns to valid SAS variable names. SAS converts to underscores any characters that it does not allow. If it converts a column to a name that already exists, it appends a sequence number at the end of the new name.

SAS
specifies that SAS converts DBMS columns that contain characters that SAS does not allow into valid SAS variable names. SAS uses the format _COL*n*, where *n* is the column number, starting with 0. If SAS converts a name to a name that already exists, it appends a sequence number at the end of the new name.

Details

SAS retains column names when it reads 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 for the conversion of kanji to English characters. English characters that are converted from kanji are often those that SAS does not allow. Although this option works for the single-byte character set (SBCS) version of SAS, SAS ignores it in the double-byte character set (DBCS) version. So if you have the DBCS version, you must first set VALIDVARNAME=ANY before using your language characters as column variables.

Each of the various SAS/ACCESS interfaces handled name collisions differently in SAS 6. Some interfaces appended at the end of the name, some replaced one or more final characters in the name, some used a single sequence number, and others used unique counters. When you specify VALIDVARNAME=V6, SAS handles name collisions as it did in SAS 6.

Example

If you specify DBGEN_NAME=SAS, SAS renames a DBMS column named **Dept $Amt** to **_COL*n***. If you specify DBGEN_NAME=DBMS, SAS renames the **Dept$Amt** column to **Dept_Amt**.

DBINDEX= LIBNAME Option

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

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS-specific

Restriction: *Oracle:* Use this option only when the object is a TABLE, not a VIEW. Use DBKEY= when you do not know whether the object is a TABLE.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBINDEX= data set option, "Using the DBINDEX=, DBKEY=, and MULTI_DATASRC_OPT= Options"

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 process 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.

DBLIBINIT= LIBNAME Option

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

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: CONNECTION= LIBNAME option, DBLIBTERM= LIBNAME option on page 131, DEFER= LIBNAME option

Syntax

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

Syntax Description

DBMS-user-command
any DBMS command that the SAS/ACCESS engine can execute 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, a disconnection 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: Allow Only One LIBNAME Statement to Connect

In this 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';
```

DBLIBTERM= LIBNAME Option

Specifies a user-defined termination command to execute once, before the DBMS that is associated with the first connection made by the LIBNAME statement or libref disconnects.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: CONNECTION= LIBNAME option, DBLIBINIT= LIBNAME option, DEFER= LIBNAME option

Syntax

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

Syntax Description

DBMS-user-command
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, SAS provides a warning message, but unassigning the library and disconnecting from the DBMS still occur. 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: Allow Only One LIBNAME Statement to Connect

In this 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';
```

DBLINK= LIBNAME Option

Specifies a link from your 1) local database to database objects on another server [Oracle] or 2) default database to another database on the server to which you are connected [Sybase].

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Oracle, Sybase

See: DBMASTER= data set option

Syntax

DBLINK=*database-link*

Details

Oracle: A link is a database object that you use to identify an object stored in a remote database. It contains stored path information. It might also contain user name and password information for connecting to the remote database. If you specify a link, SAS uses it to access remote objects. If you omit this option, SAS accesses objects in only the local database.

Sybase: This option lets you link to another database within the same server to which you are connected. If you omit this option, SAS can access objects in only your default database.

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: SAS/ACCESS LIBNAME statement

Default: 1024

Restriction: This option applies to appending and updating rows in an existing table. It does not apply when creating a table.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, MySQL, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ

See: DBMAX_TEXT= data set option

Syntax

DBMAX_TEXT=<*integer*>

Syntax Description

integer
an integer between 1 and 32,767.

Details

Examples of a DBMS data type are the Sybase TEXT data type or the Oracle CLOB (character large object) data type.

Oracle: This option applies for CLOB, BLOB, LONG, and LONG RAW data types.

DBMSTEMP= LIBNAME Option

Specifies whether SAS creates temporary or permanent tables.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Requirement: To specify this option, you must first specify CONNECTION=GLOBAL—except for Microsoft SQL Server, which defaults to UNIQUE.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase IQ, Teradata

See: CONNECTION= LIBNAME option, "Temporary Table Support for SAS/ACCESS "

Syntax

DBMSTEMP=YES | NO

Syntax Description

YES
specifies that SAS creates one or more temporary tables.

NO
specifies that SAS creates permanent tables.

Details

To significantly improve performance, you must also set DBCOMMIT=0. The value for SCHEMA= is ignored. You can then access and use the DBMS temporary tables using SAS/ACCESS engine librefs that share the global connection that SAS used to create those tables.

To join a temporary and a permanent table, you need a libref for each table and these librefs must successfully share a global connection.

DB2 under z/OS, Oracle, and Teradata: Set INSERTBUFF=1000 or higher to significantly improve performance.

ODBC: This engine supports DB2, MS SQL Server, or Oracle if you are connected to them.

Example: Create and Join a Permanent and a Temporary Table

This example shows how to use this option to create a permanent and temporary table and then join them in a query. The temporary table might not exist beyond a single PROC step. However, this might not be true for all DBMSs.

```
options sastrace=(,,d,d) nostsuffix sastraceloc=saslog;
LIBNAME permdata DB2 DB=MA40 SCHEMA=SASTDATA connection=global
   dbcommit=0 USER=sasuser PASSWORD=xxx;
LIBNAME tempdata DB2 DB=MA40 SCHEMA=SASTDATA connection=global
   dbcommit=0 dbmstemp=yes  USER=sasuser  PASSWORD=xxx;
proc sql;
create table tempdata.ptyacc as
     (
      select pty.pty_id
      from permdata.pty_rb pty,
           permdata.PTY_ARNG_PROD_RB acc
      where acc.ACC_PD_CTGY_CD = 'LOC'
        and acc.pty_id = pty.pty_id
      group by pty.pty_id having count(*) > 5
     );
create table tempdata.ptyacloc as
```

```
     (
      select ptyacc.pty_id,
             acc.ACC_APPSYS_ID,
             acc.ACC_CO_NO,
             acc.ACCNO,
             acc.ACC_SUB_NO,
             acc.ACC_PD_CTGY_CD
      from tempdata.ptyacc ptyacc,
          perm data.PTY_ARNG_PROD_RB acc
      where ptyacc.pty_id = acc.pty_id
        and acc.ACC_PD_CTGY_CD = 'LOC'
     );
create table tempdata.righttab as
     (
      select ptyacloc.pty_id
      from permdata.loc_acc loc,
           tempdata.ptyacloc ptyacloc
      where
          ptyacloc.ACC_APPSYS_ID  = loc.ACC_APPSYS_ID
      and ptyacloc.ACC_CO_NO      = loc.ACC_CO_NO
      and ptyacloc.ACCNO          = loc.ACCNO
      and ptyacloc.ACC_SUB_NO     = loc.ACC_SUB_NO
      and ptyacloc.ACC_PD_CTGY_CD = loc.ACC_PD_CTGY_CD
      and loc.ACC_CURR_LINE_AM - loc.ACC_LDGR_BL > 20000
     );
select * from tempdata.ptyacc
except
select * from tempdata.righttab;
drop table tempdata.ptyacc;
drop table tempdata.ptyacloc;
drop table tempdata.righttab;
quit;
```

DBNULLKEYS= LIBNAME Option

Controls the format of the WHERE clause when you use the DBKEY= data set option.

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase IQ

See: DBKEY= data set option, DBNULLKEYS= data set option

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= data set option, use DBNULLKEYS=YES.

This is the default for most interfaces. When you specify DBNULLKEYS=YES and also a column that is not defined as NOT NULL in DBKEY=, 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 this syntax:

```
WHERE ((COLUMN = ?) OR ((COLUMN IS NULL) AND (? IS NULL)))
```

With this syntax SAS can prepare the statement once and use it for any (NULL or NOT NULL) value in the column. This syntax can potentially be much less efficient than the shorter form of the WHERE clause below. When you specify DBNULLKEYS=NO or a column that DBKEY= defines as NOT NULL, SAS generates a simple WHERE clause.

If you know that there are no NULL values in transaction or master tables for the columns that you specify in the DBKEY= option, you can use DBNULLKEYS=NO. This is the default for the Informix interface. If you specify DBNULLKEYS=NO and DBKEY=COLUMN, SAS generates a shorter form of the WHERE clause, regardless of whether the column that is specified in DBKEY= is defined as NOT NULL.

```
WHERE (COLUMN = ?)
```

DBPROMPT= LIBNAME Option

Specifies whether SAS displays a window that prompts the user to enter DBMS connection information before connecting to the DBMS in interactive mode.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Interaction: DEFER= LIBNAME option

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, Oracle, Sybase, Sybase IQ, Teradata

See: DBPROMPT= data set option, DEFER= LIBNAME option

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 use the LIBNAME statement to specify connection options and DBPROMPT=YES, connection option values display in the window—except the password value, which appears as a series of asterisks. You can override all of these values interactively.

The DBPROMPT= option interacts with the DEFER= LIBNAME option to determine when the prompt window appears. If DEFER=NO, the DBPROMPT window appears when the LIBNAME statement is executed. If DEFER=YES, the DBPROMPT window appears when you first open a table or view. The DEFER= option normally defaults to NO, but it 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 such cases, the DBPROMPT window appears 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: You can enter 30 characters for the USERNAME and PASSWORD and up to 70 characters for the PATH, depending on your platform.

Teradata: You can enter up to 30 characters for the USERNAME and PASSWORD.

Examples

Example 1: Preventing a Prompt Window from Opening

In this example, the DBPROMPT window does not open when the LIBNAME statement is submitted because DEFER=YES. The DBPROMPT window appears 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;
```

Example 2: Allow a Prompt Window to Open Only Once

In this example, the DBPROMPT window appears while the LIBNAME statement is processing. The DBPROMPT window does not open in subsequent statements because the DBPROMPT window appears only once per LIBNAME statement.

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

Example 3: Allow Values to Appear in a Prompt Window

In this 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 the user can edit and confirm them. The password value appears in the DBPROMPT window as a series of asterisks, so the user can also edit it.

```
libname mydblib oracle
        user=testuser pw=testpass
        path='ABC_server' dbprompt=yes defer=no;
```

DBSASLABEL= LIBNAME Option

Specifies the column labels an engine uses.

Valid in: SAS/ACCESS LIBNAME statement

Default: COMPAT

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBSASLABEL= data set option

Syntax

DBSASLABEL=COMPAT | NONE

Syntax Description

COMPAT
: specifies that the labels returned should be compatible with what the application normally receives—meaning that engines exhibit their normal behavior.

NONE
: specifies that the engine does not return a column label. The engine returns blanks for the column labels.

Details

By default, the SAS/ACCESS interface for your DBMS generates column labels from the column names instead of from the real column labels.

You can use this option to override the default behavior. It is useful for when PROC SQL uses column labels as headers instead of column aliases.

Example: Return Blank Labels for Aliases in Headings

This example shows how to use DBSASLABEL= as a LIBNAME option o return blank column labels so that PROC SQL can use the column aliases as the column headings.

```
libname x oracle user=scott pw=tiger;
proc sql;
     select deptno as Department ID, loc as Location
            from mylib.dept(dbsaslabel=none);
```

Without DBSASLABEL=NONE, aliases are ignored, and DEPTNO and LOC are used as column headings in the result set.

DBSERVER_MAX_BYTES= LIBNAME Option

Specifies the maximum number of bytes per single character in the database server encoding.

Valid in: SAS/ACCESS LIBNAME statement

Default: usually 1

Data source: DB2 under UNIX and PC Hosts, Oracle, Sybase

See: ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS= LIBNAME option, ADJUST_NCHAR_COLUMN_LENGTHS= LIBNAME option, DBCLIENT_MAX_BYTES= LIBNAME option, DB_LENGTH_SEMANTICS_BYTE= LIBNAME option

Syntax

DBSERVER_MAX_BYTES=*max-server-bytes*

Details

Use this option to derive (adjust the value of) the number of characters from the client column lengths that byte semantics initially creates. Although the default is usually 1, you can use this option to set it to another value if this information is available from the Oracle server.

Sybase: You can use this option to specify different byte encoding between the SAS client and the Sybase server. For example, if the client uses double-byte encoding and the server uses multibyte encoding, specify DBSERVER_MAX_BYTES=3. In this case, the SAS/ACCESS engine evaluates this option only if you specify a value that is greater than 2. Otherwise, it indicates that both client and server use the same encoding scheme.

Example: Adjust Specific Column Lengths

Only the lengths that you specify with DBSERVER_MAX_BYTES= affect column lengths that byte semantics created initially.

```
libname x4 &engine &connopt DBSERVER_MAX_BYTES=4 DBCLIENT_MAX_BYTES=1
ADJUST_NCHAR_COLUMN_LENGTHS=no;
proc contents data=x4.char_sem; run;
proc contents data=x4.nchar_sem; run;
proc contents data=x4.byte_sem; run;
proc contents data=x4.mixed_sem; run;
```

In this example, various options have different settings.

```
libname x5 &engine &connopt ADJUST_NCHAR_COLUMN_LENGTHS=NO
ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=NO DBCLIENT_MAX_BYTES=3;
proc contents data=x5.char_sem; run;
proc contents data=x5.nchar_sem; run;
proc contents data=x5.byte_sem; run;
proc contents data=x5.mixed_sem; run;
```

This example also uses different settings for the various options.

```
libname x6  &engine &connopt ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=YES
ADJUST_NCHAR_COLUMN_LENGTHS=YES DBCLIENT_MAX_BYTES=3;
proc contents data=x6.char_sem; run;
proc contents data=x6.nchar_sem; run;
proc contents data=x6.byte_sem; run;
proc contents data=x6.mixed_sem; run;
```

DBSLICEPARM= LIBNAME Option

Controls the scope of DBMS threaded Reads and the number of threads.

Valid in: SAS/ACCESS LIBNAME statement (also available as a SAS configuration option, SAS invocation option, global SAS option, or data set option)

Default: THREADED_APPS,2 (DB2 under z/OS, Oracle, Teradata), THREADED_APPS,2*or* 3 (DB2 under UNIX and PC Hosts, HP Neoview, Informix, Microsoft SQL Server, ODBC, Sybase, Sybase IQ)

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, HP Neoview, Informix, Microsoft SQL Server, ODBC, Oracle, Sybase, Sybase IQ, Teradata

See: DBSLICE= data set option, DBSLICEPARM= data set option, DBSLICEPARM= system option, SLEEP= LIBNAME option, SLEEP= data set option, TENACITY= LIBNAME option, TENACITY= data set option

Syntax

DBSLICEPARM=NONE | THREADED_APPS | ALL

DBSLICEPARM=(NONE | THREADED_APPS | ALL<*max-threads*>)

DBSLICEPARM=(NONE | THREADED_APPS | ALL<, *max-threads*>)

Syntax Description

NONE
disables DBMS threaded Read. SAS reads tables on a single DBMS connection, as it did with SAS 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
a positive integer value that specifies the maximum number of connections per table read. The second parameter of the option determines the number of threads to read the table in parallel. The number of partitions on the table determine the number of connections made to the Oracle server for retrieving rows from the table. A partition or portion of the data is read on each connection. The combined rows across all partitions are the same regardless 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.

If the database table is not partitioned, SAS creates *max-threads* number of connections with *WHERE MOD()*... predicates and the same number of threads.

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. See the DBMS-specific reference section for details about partitioned reads before using this parameter.

Details

You can use DBSLICEPARM= in numerous locations. The usual rules of option precedence apply: A table option has the highest precedence, then a LIBNAME option, and so on. 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 whether threaded Reads are actually generated, turn on SAS tracing and run a program, as shown in this 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.

DB2 under UNIX and PC Hosts, Informix, Microsoft SQL Server, ODBC, Sybase, Sybase IQ
The default thread number depends on whether an application passes in the number of threads (CPUCOUNT=) and whether the data type of the column that was selected for purposes of data partitioning is binary.

Examples

Example 1: Disable Threaded Read for All SAS Users

Here is how to use DBSLICEPARM= in a SAS configuration file entry in Windows to turn off threaded Reads for all SAS users.

```
-dbsliceparm NONE
```

Example 2: Enable Threaded Reads for Read-Only References

Here is how you can use DBSLICEPARM= as a z/OS invocation option to turn on threaded Reads for read-only references to DBMS tables throughout a SAS job.

```
sas o(dbsliceparm=ALL)
```

Example 3: Increase Maximum Threads (as a SAS Global Option)

In this example, you can 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 applications.

```
option dbsliceparm=(threaded_apps,3);
```

Example 4: Enable Threaded Reads for References Using a Particular Libref

You can use DBSLICEPARM= as a LIBNAME option to turn on threaded Reads for read-only table references that use this particular libref, as shown in this example.

```
libname dblib oracle user=scott password=tiger dbsliceparm=ALL;
```

Example 5: Enable Threaded Reads as a Table-Level Option

Here is 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;
```

DEFER= LIBNAME Option

Specifies when the connection to the DBMS occurs.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: CONNECTION= LIBNAME option, DBPROMPT= LIBNAME option

Syntax

DEFER=YES| NO

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.

HP Neoview, Microsoft SQL Server, Netezza, ODBC: When you set DEFER=YES, you must also set the PRESERVE_TAB_NAMES= and PRESERVE_COL_NAMES= options to the values that you want. 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.

DEGREE= LIBNAME Option

Determines whether DB2 uses parallelism.

Valid in: SAS/ACCESS LIBNAME statement

Default: ANY

Data source: DB2 under z/OS

See: DEGREE= data set option

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, although this is perhaps slower, it uses less system resources.

DELETE_MULT_ROWS= LIBNAME Option

Indicates whether to let SAS delete multiple rows from a data source, such as a DBMS table.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Aster *n*Cluster, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, OLE DB, Sybase IQ

See: UPDATE_MULT_ROWS= LIBNAME option

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 this DBMS SQL statement well and therefore delete more than the current row:

```
DELETE…WHERE CURRENT OF CURSOR
```

DIMENSION= LIBNAME Option

Specifies whether the database creates dimension tables or fact tables.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Aster *n*Cluster

See: DIMENSION= data set option, PARTITION_KEY= LIBNAME option, PARTITION_KEY= data set option

Syntax

DIMENSION=YES | NO

Syntax Description

YES
specifies that the database creates dimension tables.

NO
specifies that the database creates fact tables.

DIRECT_EXE= LIBNAME Option

Lets an SQL delete statement be passed directly to a DBMS with pass-through.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Aster *n*, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBIDIRECTEXEC= system option

Syntax

DIRECT_EXE=DELETE

Syntax Description

DELETE
specifies that an SQL delete statement is passed directly to the DBMS for processing.

Details

Performance improves significantly by using DIRECT_EXE=. This is because the SQL delete statement is passed directly to the DBMS instead of having SAS read the entire result set and delete one row at a time.

Example: 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 this:

Output 10.1 *SAS Log Output*

```
ORACLE_9: Executed:
delete from dbi_dft
```

DIRECT_SQL= LIBNAME Option

Specifies whether generated SQL is passed to the DBMS for processing.

Valid in: SAS/ACCESS LIBNAME statement

Default: YES

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: SQL_FUNCTIONS= LIBNAME option

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 does not attempt to pass any multiple outer joins to the DBMS for processing. Other join statements might 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 SQL processing.

If you specify DIRECT_SQL=NOGENSQL, 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 SQL_FUNCTIONS= LIBNAME option. If you specify SQL_FUNCTIONS=ALL and DIRECT_SQL=NONE, no functions are passed.

Examples

Example 1: Prevent a DBMS from Processing a Join

This example prevents the DBMS from processing a join between two tables by setting DIRECT_SQL=NOGENSQL. SAS processes the join instead.

```
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;
```

Example 2: Prevent a DBMS from Processing a SAS Function

```
libname mydblib oracle user=testuser password=testpass direct_sql=nofunctions;
proc print data=mydblib.tab1;
   where lastname=soundex ('Paul');
```

ENABLE_BULK= LIBNAME Option

Lets the connection process bulk copy when loading data into a Sybase table.

Valid in: SAS/ACCESS LIBNAME statement

Default: YES

Data source: Sybase

Note: In SAS 7 and previous releases, this option was called BULKCOPY=. In SAS 8 and later, an error is returned if you specify BULKCOPY=.

See: BULK_BUFFER= data set option, BULKLOAD= data set option

Syntax

ENABLE_BULK=YES | NO

Syntax Description

NO
disables bulk copy ability for the libref.

YES
lets the connection perform 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, ENABLE_BULK= LIBNAME option and the BULKLOAD=data set option, values for both options must be the same or an error is returned. However, because ENABLE_BULK=YES is the default value, you need not specify ENABLE_BULK= to use the BULKLOAD= data set option.

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)

Default: 1 million

Data source: Teradata

See: ERRLIMIT= data set option, DBCOMMIT= LIBNAME option, DBCOMMIT= data set option, ML_CHECKPOINT= data set option

Syntax

ERRLIMIT=*integer*

Syntax Description

integer
specifies a positive integer that represents the number of errors after which SAS stops loading data.

Details

SAS stops loading data when it reaches the specified number of errors and Fastload pauses. When Fastload pauses, you cannot use the table that is being loaded. Restart capability for Fastload is not yet supported, so you must manually delete the error tables before SAS can reload the table.

Example

In this example, SAS stops processing and pauses Fastload when it encounters 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;
```

ESCAPE_BACKSLASH= LIBNAME Option

Specifies whether backslashes in literals are preserved during data copy from a SAS data set to a table.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: MySQL

See: ESCAPE_BACKSLASH= data set option

Syntax

ESCAPE_BACKSLASH=YES | NO

Syntax Description

YES
specifies that an additional backslash is inserted in every literal value that already contains a backslash.

NO
specifies that backslashes that exist in literal values are not preserved. An error results.

Details

MySQL uses the backslash as an escape character. When data that is copied from a SAS data set to a MySQL table contains backslashes in literal values, the MySQL interface can preserve these if ESCAPE_BACKSLASH=YES.

FASTEXPORT= LIBNAME Option

Specifies whether the SAS/ACCESS engine uses the TPT API to read data.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Teradata

See: BULKLOAD= LIBNAME option, BULKLOAD= data set option, DBSLICEPARM= LIBNAME option, DBSLICEPARM= data set option, DBSLICEPARM= system option, LOGDB= LIBNAME option, Maximizing Teradata Load Performance, MULTILOAD= data set option, QUERY_BAND= LIBNAME option, QUERY_BAND= data set option, SLEEP= LIBNAME option, SLEEP= data set option, TENACITY= LIBNAME option, TENACITY= data set option

Syntax

FASTEXPORT=YES | NO

Syntax Description

YES
specifies that the SAS/ACCESS engine uses the Teradata Parallel Transporter (TPT) API to read data from a Teradata table.

NO
specifies that the SAS/ACCESS engine does not use the TPT API to read data from a Teradata table.

Details

By using the TPT API, you can read data from a Teradata table without working directly with the stand-alone Teradata FastExport utility. When FASTEXPORT=YES, SAS uses the TPT API export driver for bulk reads. If SAS cannot use the TPT API—due to an error or because it is not installed on the system—it still tries to read the data. However, it does not produce an error. To check whether SAS used the TPT API to read data, look for this message in the SAS log:

```
NOTE:  Teradata connection:  TPT FastExport has read n row(s).
```

When you specify a query band on this option, you must set the DBSLICEPARM= LIBNAME option. The query band is passed as a SESSION query band to the FastExport utility.

To see whether threaded Reads are actually generated, turn on SAS tracing by setting OPTIONS SASTRACE=”,,,d” in your program.

Example

In this example, the TPT API reads SAS data from a Teradata table. SAS still tries to read data even if it cannot use the TPT API.

```
Libname tera Teradata user=testuser pw=testpw FASTEXPORT=YES;
/* Create data */
Data tera.testdata;
Do i=1 to 100;
    Output;
End;
Run;
/* Read using FastExport TPT.  This note appears in the SAS log if SAS uses TPT.
NOTE:  Teradata connection:  TPT FastExport has read


n row(s).*/
Data work.testdata;
Set tera.testdata;
Run;
```

FETCH_IDENTITY= LIBNAME Option

Returns the value of the last inserted identity value.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: DB2 under UNIX and PC Hosts

See: FETCH_IDENTITY= data set option

Syntax

FETCH_IDENTITY=YES | NO

Syntax Description

YES
returns the value of the last inserted identity value.

NO
disables this option.

Details

You can use this option instead of issuing a separate SELECT statement after an INSERT statement. If FETCH_IDENTITY=YES and the INSERT that is executed is a single-row INSERT, the engine calls the DB/2 identity_val_local() function and places the results into the SYSDB2_LAST_IDENTITY macro variable. Because the DB2 engine default is multirow inserts, you must set INSERTBUFF=1 to force a single-row INSERT.

IGNORE_READ_ONLY_COLUMNS= LIBNAME Option

Specifies whether to ignore or include columns where data types are read-only when generating an SQL statement for inserts or updates.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, OLE DB, Sybase IQ

See: IGNORE_READ_ONLY_COLUMNS= data set option

Syntax

IGNORE_READ_ONLY_COLUMNS=YES | NO

Syntax Description

YES
specifies that the SAS/ACCESS engine ignores columns where 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 where 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 the data type of the Microsoft SQL Server timestamp. Several databases also have properties that allow certain data types to be read-only, such as the Microsoft SQL Server identity property.

When IGNORE_READ_ONLY_COLUMNS=NO 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 this 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;
```

IN= LIBNAME Option

Lets you specify the database and table space in which you want to create a new table.

Valid in: SAS/ACCESS LIBNAME statement

Alias: TABLESPACE=

Default: none

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS

See: IN= data set option

Syntax

IN='*database-name.tablespace-name*'| 'DATABASE *database-name*'

Syntax Description

database-name.tablespace-name
specifies the names of the database and table space, 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, 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, implicitly creating a simple table space.

INSERT_SQL= LIBNAME Option

Determines the method to use to insert rows into a data source.

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS-specific

Data source: Microsoft SQL Server, ODBC, OLE DB

See: INSERT_SQL= data set option, INSERTBUFF= LIBNAME option, INSERTBUFF= data set option

Syntax

INSERT_SQL=YES | NO

Syntax Description

YES
: specifies that SAS/ACCESS uses the SQL insert method for the data source 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. Other databases might have inferior insert performance or might fail with this setting. You should therefore experiment to determine the optimal setting to meet your needs.

HP Neoview: The default is YES.

Microsoft SQL Server: 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 your driver supports it.

Netezza: The default is YES.

ODBC: The default is YES, except for Microsoft Access, where the default is 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 your driver supports it.

OLE DB: By default, the OLE DB interface tries 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. Used when this option is set to NO, the alternate OLE DB method uses the OLE DB IRowsetChange interface.

INSERTBUFF= LIBNAME Option

Specifies the number of rows in a single DBMS insert.

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase IQ

See: INSERTBUFF= data set option, DBCOMMIT= LIBNAME option, DBCOMMIT= data set option, INSERT_SQL= LIBNAME option, INSERT_SQL= data set option

Syntax

INSERTBUFF=*positive-integer*

Syntax Description

positive-integer
specifies the number of rows to insert.

Details

SAS allows the maximum number of rows that the DBMS allows. 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.

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 set the DBCOMMIT= option with a value that is less than the value of INSERTBUFF=, then DBCOMMIT= overrides INSERTBUFF=.

When you insert rows with 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.

Additional driver-specific restrictions might apply.

DB2 under UNIX and PC Hosts: Before you can use this option, you must first set INSERT_SQL=YES. If one row in the insert buffer fails, all rows in the insert buffer fail. The default is calculated based on the row length of your data.

HP Neoview, Netezza: The default is automatically calculated based on row length.

Microsoft SQL Server: Before you can use this option, you must first set INSERT_SQL=YES. The default is 1.

MySQL: The default is 0. Values greater than 0 activate the INSERTBUFF= option, and the engine 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.

ODBC: The default is 1.

OLE DB: The default is 1.

Oracle: When REREAD_EXPOSURE=YES, the (forced) default value is 1. Otherwise, the default is 10.

INTERFACE= LIBNAME Option

Specifies the name and location of the interfaces file that is searched when you connect to the Sybase server.

Valid in:	SAS/ACCESS LIBNAME statement
Default:	none
Data source:	Sybase

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: SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. See the DBMS-specific reference section for details.

Default: 0

Data source: Microsoft SQL Server, ODBC

See: CURSOR_TYPE= LIBNAME option, KEYSET_SIZE= data set option

Syntax

KEYSET_SIZE=*number-of-rows*

Syntax Description

number-of-rows
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.

If KEYSET_SIZE=0, the entire cursor is keyset-driven. If you specify a value greater than 0 for KEYSET_SIZE=, that value indicates the number of rows within the cursor that functions as a keyset-driven cursor. When you scroll beyond the bounds that KEYSET_SIZE= specifies, 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 you specify a value between 1 and the number of rows in the cursor, the cursor is considered to be a mixed cursor: Part of it functions as a keyset-driven cursor and part functions as a dynamic cursor.

LOCATION= LIBNAME Option

Allows further qualification of exactly where a table resides.

Valid in: SAS/ACCESS LIBNAME statement

Alias: LOC=

Default: none

Requirement: If you specify LOCATION=, you must also specify the AUTHID= LIBNAME option.

Data source: DB2 under z/OS

See: AUTHID= LIBNAME option, LOCATION= data set option, REMOTE_DBTYPE= LIBNAME option [to access a database server on Linux, UNIX, or Windows]

Syntax

LOCATION=*location*

Details

The location name maps to the location in the SYSIBM.LOCATION catalog in the communication database.

In SAS/ACCESS Interface to DB2 under z/OS, the location is converted to the first level of a three-level table name: *location.authid.table*. The DB2 Distributed Data Facility (DDF) makes the connection implicitly to the remote DB2 subsystem when DB2 receives a three-level 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.

LOCKTABLE= LIBNAME Option

Places exclusive or shared locks on tables.

Valid in: SAS/ACCESS LIBNAME statement

Default: no locking

Data source: Informix

See: LOCKTABLE= data set option

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 can lock tables only if you are the owner or have been granted the necessary privilege.

LOCKTIME= LIBNAME Option

Specifies the number of seconds to wait until rows are available for locking.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Requirement: You must specify LOCKWAIT=YES for LOCKTIME= to have an effect.

Data source: Informix

See: LOCKWAIT= LIBNAME option

Syntax

LOCKTIME=*positive-integer*

Details

If you omit the LOCKTIME= option and use LOCKWAIT=YES, SAS suspends your process indefinitely until a lock can be obtained.

LOCKWAIT= LIBNAME Option

Specifies whether to wait indefinitely until rows are available for locking.

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS-specific

Data source: Informix, Oracle

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 to an alternate database-specific table that FastExport creates or MultiLoad uses.

Valid in: DATA and PROC steps, wherever you use FastExport or MultiLoad

Default: default Teradata database for the libref

Data source: Teradata

Tip: You can also use LOGDB= with TPT options.

See: FASTEXPORT= LIBNAME option, MULTILOAD= data set option, TPT= LIBNAME option, TPT= data set option, "Using MultiLoad"

Syntax

LOGDB=<*database-name*>

Syntax Description

database-name
the name of the Teradata database.

Details

Teradata FastExport utility: The FastExport restart capability is not yet supported. When you use this option with FastExport, FastExport creates restart log tables in an alternate database. You must have the necessary permissions to create tables in the specified database, and FastExport creates only restart tables in that database.

Teradata MultiLoad utility: To specify this option, you must first specify MULTILOAD=YES. When you use this option with the Teradata MultiLoad utility, MultiLoad redirects the restart table, the work table, and the required error tables to an alternate database.

Examples

Example 1: Create Restart Log Tables

In this example, PROC PRINT calls the Teradata FastExport utility, if it is installed. FastExport creates restart log tables in the ALTDB database.

```
libname mydblib teradata user=testuser pw=testpass logdb=altdb;
proc print data=mydblib.mytable(dbsliceparm=all);
run;
```

Example 2: Create Restart, Work, and Error Tables

In this next example, MultiLoad creates the restart table, work table, and error tables in the alternate database that LOGDB= specifies.

```
/* Create work tables in zoom database,
where I have create & drop privileges. */
   libname x teradata user=prboni pw=xxxxx logdb=zoom;
   data x.testload(multiload=YES);
     do i=1 to 100;
       output;
     end;
   run;
```

Example 3: Create the Work Table in a Different Database

Using MultiLoad with the TPT API, this example provides a different name for the work table and redirects the table to the AUDATA00_work database.

```
   libname tera teradata user=testuser pw=testpw logdb=audata00_work;
   data tera.testdata(MULTILOAD=YES TPT_WORK_TABLE=work);
   i=1;output; i=2;output;
   run;
```

LOGIN_TIMEOUT= LIBNAME Option

Specifies the default login time-out for connecting to and accessing data sources in a library.

Valid in: SAS/ACCESS LIBNAME statement

Default: 0

Data source: Aster *n*Cluster, HP Neoview, Netezza, ODBC, Sybase IQ

Syntax

LOGIN_TIMEOUT=*numeric-value*

MAX_CONNECTS= LIBNAME Option

Specifies the maximum number of simultaneous connections that Sybase allows.

Valid in: SAS/ACCESS LIBNAME statement

Default: 25

Data source: Sybase

Syntax

MAX_CONNECTS=*numeric-value*

Details

If you omit MAX_CONNECTS=, the default for the maximum number of connections is 25. Increasing the number of connections has a direct impact on memory.

MODE= LIBNAME Option

Specifies whether the connection to Teradata uses the ANSI or Teradata mode.

Valid in: SAS/ACCESS LIBNAME statement

Default: ANSI

Data source: Teradata

See: "SQL Pass-Through Facility Specifics for Teradata "

Syntax

MODE=TERADATA | ANSI

Syntax Description

TERADATA
specifies that SAS/ACCESS opens Teradata connections in Teradata mode.

ANSI
specifies that SAS/ACCESS opens Teradata connections in ANSI mode.

Details

This option allows Teradata connections to open in the specified mode. Connections that open with MODE=TERADATA use Teradata mode rules for all SQL requests that are passed to the Teradata DBMS. This impacts transaction behavior and can cause case insensitivity when processing data.

During data insertion, not only is each inserted row committed implicitly, but rollback is not possible when the error limit is reached if you also specify ERRLIMIT=. Any update or delete that involves a cursor does not work.

ANSI mode is recommended for all features that SAS/ACCESS supports, and Teradata mode is recommended only for reading data from Teradata.

Example

This example does not work because it requires the use of a cursor.

```
libname x teradata user=prboni pw=XXXX mode=teradata;
 /* Fails with "ERROR:  Cursor processing is
not allowed in Teradata mode." */
 proc sql;
update x.test
set i=2;
quit;
```

This example works because the DBIDIRECTEXEC= system option sends the delete SQL directly to the database without using a cursor.

```
libname B teradata user=prboni pw=XXX mode=Teradata;
options dbidirectexec;
proc sql;
delete from b.test where i=2;
quit;
```

MULTI_DATASRC_OPT= LIBNAME Option

Used in place of DBKEY to improve performance when processing a join between two data sources.

Valid in: SAS/ACCESS LIBNAME statement

Default: NONE

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBMASTER= data set option

Syntax

MULTI_DATASRC_OPT=NONE | IN_CLAUSE

Syntax Description

NONE
turns off option functionality.

IN_CLAUSE
specifies use of an IN clause that contains values that were read from a smaller table. The clause is used to retrieve matching values in a larger table based on a key column that was designated in an equijoin.

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, if the SAS data set is larger than the DBMS table, the SAS data set is still used in the IN clause.

When SAS processes 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 the DBMASTER= data set option to prevent the SELECT COUNT (*) from being issued.

The IN clause currently has a limit of 4,500 unique values.

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 settings for DIRECT_SQL= prevent a WHERE clause from being passed.

Oracle: Oracle can handle an IN clause of only 1,000 values. It therefore 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 produces 4 IN clauses, each of which contains 1,000 values. A single result is produced, *as if* all 4,000 values were processed as a whole.

OLE DB: OLE DB restricts the number of values allowed in an IN clause to 255.

Examples

Example 1: Build and Pass an IN Clause for a Join

This 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;
```

Example 2: Prevent Build and Pass of an IN Clause for a Join

This example prevents the building and passing of the IN clause to the DBMS. It requires all rows from the DBMS table to be brought into SAS to process 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;
```

MULTISTMT= LIBNAME Option

Specifies whether insert statements are sent to Teradata one at a time or in a group.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Restriction: You currently cannot use MULTISTMT= with ERRLIMIT=.

Data source: Teradata

See: MULTISTMT= data set option

Syntax

MULTISTMT=YES | NO

Syntax Description

YES
tries to send as many inserts to Teradata that can fit in a 64K buffer. If multistatement inserts are not possible, processing reverts to single-row inserts.

NO
send inserts to Teradata one row at a time.

Details

When you request multistatement inserts, SAS first determines how many insert statements it can send to Teradata. Several factors determine the actual number of statements that SAS can send—for example, how many:

- SQL insert statements can fit in a 64K buffer.
- data rows can fit in the 64K data buffer.
- inserts the Teradata server chooses to accept.

When you need to insert large volumes of data, you can significantly improve performance by using MULTISTMT= instead of inserting only single-row.

If you also specify DBCOMMIT=, SAS uses the smaller of these: the DBCOMMIT= value and the number of insert statements that can fit in a buffer as the number of insert statements to send together at one time.

OR_BINARY_DOUBLE= LIBNAME Option

Specifies the default data type to use for numeric table columns.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Requirement: Oracle Server 10g or above

Data source: Oracle

See: DBTYPE= data set option

Syntax

OR_BINARY_DOUBLE =YES | NO

Syntax Description

YES
specifies BINARY_DOUBLE as the default.

NO
specifies NUMBER as the default.

Details

Use this option when you want a wider range of numbers than when compared to the NUMBER. You can override this option with the DBTYPE= data set option.

OR_ENABLE_INTERRUPT= LIBNAME Option

Allows interruption of any long-running SQL processes on the DBMS server.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Oracle

Syntax

OR_ENABLE_INTERRUPT=YES | NO

Syntax Description

YES
allows interruption of long-running SQL processes on the DBMS server.

NO
disables interruption of long-running SQL processes on the DBMS server.

Details

You can use this option to interrupt these statements:

- any SELECT SQL statement that was submitted by using the *SELECT * FROM CONNECTION* as a pass-through statement
- any statement other than the SELECT SQL statement that you submitted by using the EXECUTE statement as a pass-through statement

OR_UPD_NOWHERE= LIBNAME Option

Specifies whether SAS uses an extra WHERE clause when updating rows with no locking.

Valid in: SAS/ACCESS LIBNAME statement

Alias: ORACLE_73_OR_ABOVE=

Default: YES

Data source: Oracle

Note: Due to the published Oracle bug 440366, an update on a row sometimes fails even if the row has not changed. Oracle offers this solution: When you create a table, increase the number of INITRANS to at least 3 for the table.

See: "Locking in the Oracle Interface", OR_UPD_NOWHERE= data set option, UPDATE_LOCK_TYPE= LIBNAME option

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. It lets you avoid extra WHERE-clause processing and potential WHERE-clause floating-point precision problems.

PACKETSIZE= LIBNAME Option

Allows specification of the packet size for Sybase to use.

Valid in: SAS/ACCESS LIBNAME statement

Default: current server setting

Data source: Sybase

Syntax

PACKETSIZE=*numeric-value*

Syntax Description

numeric-value
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.

PARTITION_KEY= LIBNAME Option

Specifies the column name to use as the partition key for creating fact tables.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Requirements: To create a data set in Aster *n*Cluster without error, you must either set DIMENSION= YES (LIBNAME or data set option) or specify a partition key in the PARTITION_KEY= (LIBNAME or data set) option.
You must enclose the column name in quotation marks.

Data source: Aster *n*Cluster

See: DIMENSION= LIBNAME option, , DIMENSION= data set option, PARTITION_KEY= data set option

Syntax

PARTITION_KEY='*column-name*'

Details

Aster *n*Cluster uses dimension and fact tables.

Example: Create a Dimension Table

This first example shows how you can use the SAS data set, SASFLT. flightschedule, to create an Aster *n*Cluster dimension table, flightschedule by using the DIMENSION= DATA step option.

```
LIBNAME sasflt 'SAS-library';
LIBNAME net_air ASTER user=louis pwd=fromage server=air2 database=flights;
data net_air.flightschedule(dimension=yes);
  set sasflt. flightschedule;
run;
```

You can create the same Aster *n*Cluster dimension table by setting DIMENSION=YES in the LIBNAME statement.

```
LIBNAME sasflt 'SAS-library';
LIBNAME net_air ASTER user=louis pwd=fromage server=air2
        database=flights dimension=yes;
data net_air.flightschedule;
  set sasflt. flightschedule;
run;
```

If you do not set DIMENSION=YES by using either the LIBNAME or data set option, the Aster *n*Cluster engine tries to create an Aster *n*Cluster fact table. To do this, however, you must set the PARTITION_KEY= LIBNAME or data set option, as shown in this example.

```
LIBNAME sasflt 'SAS-library';
LIBNAME net_air ASTER user=louis pwd=fromage server=air2 database=flights;
data net_air.flightschedule(dbtype=(flightnumber=integer)
     partition_key='flightnumber');
  set sasflt. flightschedule;
run;
```

You can create the same Aster *n*Cluster fact table by using the PARTITION_KEY= LIBNAME option.

```
LIBNAME sasflt 'SAS-library';
LIBNAME net_air ASTER user=louis pwd=fromage server=air2 database=flights
        partition_key='flightnumber';
data net_air.flightschedule(dbtype=(flightnumber=integer));
  set sasflt. flightschedule;
run;
```

The above examples use the DBTYPE= data set option so that the data type of the partition-key column meets the limitations of the Aster *n*Cluster partition-key column.

PREFETCH= LIBNAME Option

Enables the PreFetch facility on tables that the libref (defined with the LIBNAME statement) accesses.

Valid in: SAS/ACCESS LIBNAME statement

Default: not enabled

Data source: Teradata

See: "Using the PreFetch Facility"

Syntax

PREFETCH='*unique_storename*, [*#sessions*,*algorithm*]'

Syntax Description

unique_storename

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. SEQUENTIAL is currently the only valid value.

Details

Before using PreFetch, see the description for it in the Teradata section for more detailed information. This information includes 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: SAS/ACCESS LIBNAME statement (when you create DBMS tables)

Alias: PRESERVE_NAMES=

Default: DBMS-specific

Restrictions: This option applies only when you use SAS/ACCESS to create a new DBMS table.
PRESERVE_COL_NAMES= does not apply to the SQL pass-through facility.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase IQ, Teradata

See: PRESERVE_COL_NAMES= data set option, “SAS Names and Support for DBMS Names”, VALIDVARNAME= system option

Syntax

PRESERVE_COL_NAMES=YES | NO

Syntax Description

NO
specifies that column names that are used to create DBMS tables are derived from SAS variable names (VALIDVARNAME= system option) by using the SAS variable name normalization rules. However, the database applies 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 is preserved.

Details

When you create a table, you assign the column names by using one of these methods.

- To control the case of the DBMS column names, specify variables using the case that you want and set PRESERVE_COL_NAMES=YES. If you use special symbols or blanks, you must set VALIDVARNAME= to ANY and use N-literals. For more information, see the SAS/ACCESS naming topic in the DBMS-specific reference section for your interface in this document and also *SAS Data Set Options: Reference*.
- 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.

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.

To save some time when coding, specify the PRESERVE_NAMES= alias if you plan to specify both the PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options in your LIBNAME statement.

To use column names in your SAS program that are not valid SAS names, you must use one of these techniques.

- Use the DQUOTE= option in PROC SQL and reference your columns using double quotation marks. Here is an 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. Here is an example.

  ```
  proc print data=mydblib.mytable;
     format 'Total$Cost'n 22.2;
  ```

If you are *creating* a table in PROC SQL, you must also include the PRESERVE_COL_NAMES=YES option in your LIBNAME statement. Here is an example.

```
libname mydblib oracle user=testuser password=testpass
        preserve_col_names=yes;
   proc sql dquote=ansi;
      create table mydblib.mytable ("my$column" int);
```

PRESERVE_TAB_NAMES= LIBNAME Option

Preserves spaces, special characters, and case sensitivity in DBMS table names.

Valid in: SAS/ACCESS LIBNAME statement

Alias: PRESERVE_NAMES=

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase IQ, Teradata

See: PRESERVE_TAB_NAMES= data set option, DBINDEX= data set option, SAS/ACCESS naming, SCHEMA= LIBNAME option. See also naming conventions for DB2 under UNIX and PC Hosts, DB2 under z/OS, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB , Oracle, Teradata

Syntax

PRESERVE_TAB_NAMES=YES | NO

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), tables with names that do not conform to SAS member name normalization rules do not appear in output. In SAS line mode, here is how SAS indicates the number of tables that are not displayed from PROC DATASETS because of this restriction:

```
Due to the PRESERVE_TAB_NAMES=NO LIBNAME option setting,
12 table(s) have not been displayed.
```

You do not receive this warning when you use 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

For more information, see the SAS/ACCESS naming topic in the DBMS-specific reference section for your interface in this document.

To use table names in your SAS program that are not valid SAS names, use one of these techniques.

- Use the PROC SQL option DQUOTE= and place double quotation marks around the table name. The libref must specify PRESERVE_TAB_NAMES=YES. Here is an 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. Here is an example.

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

```
proc print data=mydblib.'my table'n;
run;
```

To save some time when coding, specify the PRESERVE_NAMES= alias if you plan to specify both the PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options in your LIBNAME statement.

Oracle: Unless you specify PRESERVE_TAB_NAMES=YES, the table name that you enter for SCHEMA= LIBNAME option 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, 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 a form that is normalized for Oracle. 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, the PROC DATASETS output includes all three table names: My_Table, MY_TABLE, and MY TABLE.

To apply this option to an individual data set, see the naming in your DBMS interface for the PRESERVE_TAB_NAMES= data set option.

QUALIFIER= LIBNAME Option

Allows identification of such database objects as tables and views with the specified qualifier.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: HP Neoview, Microsoft SQL Server, Netezza, ODBC, OLE DB

See: QUALIFIER= data set option

Syntax

QUALIFIER=<*qualifier-name*>

Details

If you omit this option, the default is the default DBMS qualifier name, if any. You can use QUALIFIER= for any DBMS that allows three-part identifier names, such as *qualifier.schema.object*.

MySQL: The MySQL interface does not support three-part identifier names, so a two-part name is used (such as *qualifier.object*).

Example

In this LIBNAME statement, the QUALIFIER= option causes ODBC to interpret any reference to mydblib.employee in SAS as mydept.scott.employee.

```
 libname mydblib odbc dsn=myoracle
         password=testpass schema=scott
         qualifier=mydept;
```

In this example, the QUALIFIER= option causes OLE DB to interpret any reference in SAS to mydblib.employee 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;
```

QUALIFY_ROWS= LIBNAME Option

Uniquely qualifies all member values in a result set.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: OLE DB

See: “Accessing OLE DB for OLAP Data”

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. This can occur because it forces the OLE DB interface to search through various schemas to gather the information needed to created unique qualified names.

QUERY_BAND= LIBNAME Option

Specifies whether to set a query band for the current session.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Teradata

See: QUERY_BAND= data set option, BULKLOAD= LIBNAME option, BULKLOAD= data set option, FASTEXPORT= LIBNAME option, "Maximizing Teradata Load Performance ", MULTILOAD= data set option

Syntax

QUERY_BAND="*pair-name=pair_value*" FOR SESSION;

Syntax Description

pair-name=pair_value
specifies a name and value pair of a query band for the current session.

Details

Use this option to set unique identifiers on Teradata sessions and to add them to the current session. The Teradata engine uses this syntax to pass the name-value pair to Teradata.

```
SET QUERY_BAND="org=Marketing;report=Mkt4Q08;" FOR SESSION;
```

For more information about this option and query-band limitations, see *Teradata SQL Reference: Data Definition Statements*.

QUERY_TIMEOUT= LIBNAME Option

Specifies the number of seconds of inactivity to wait before canceling a query.

Valid in: SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. See your DBMS for details.

Default: 0

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, Sybase IQ

See: QUERY_TIMEOUT= data set option

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.

QUOTE_CHAR= LIBNAME Option

Specifies which quotation mark character to use when delimiting identifiers.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Aster *n*Cluster, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, OLE DB, Sybase IQ

Syntax

QUOTE_CHAR=*character*

Syntax Description

character
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 and the DBMS allow for this character, the QUOTE_CHAR= option overrides the character that the provider returns.

Microsoft SQL Server: QUOTE_CHAR= overrides the Microsoft SQL Server default.

ODBC: This option is mainly for the ODBC interface to Sybase, and you should use it 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).

Examples

Example 1: Specify a Single Quotation Mark

Here is what to specify if you want your quotation character to be a single quotation mark.

```
libname x odbc dsn=mydsn pwd=mypassword quote_char="'";
```

Example 2: Specify a Double Quotation Mark

Here is what to specify if you want your quotation character to be a double quotation mark.

```
libname x odbc dsn=mydsn pwd=mypassword quote_char='"';
```

QUOTED_IDENTIFIER= LIBNAME Option

Allows specification of table and column names with embedded spaces and special characters.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Sybase

See: PRESERVE_COL_NAMES= LIBNAME option, PRESERVE_TAB_NAMES= LIBNAME option

Syntax

QUOTED_IDENTIFIER=YES | NO

Details

You use this option in place of the PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= LIBNAME options. They have no effect on the Sybase interface because it defaults to case sensitivity.

READBUFF= LIBNAME Option

Specifies the number of rows of DBMS data to read into the buffer.

Valid in: SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. See the DBMS-specific reference section for details.

Aliases: ROWSET_SIZE= [DB2 under UNIX and PC Hosts, DB2 under z/OS, HP Neoview, Microsoft SQL Server, Netezza, ODBC, OLE DB, Sybase, Sybase IQ]

ROWSET= [Sybase IQ]

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ

See: READBUFF= data set option

Syntax

READBUFF=*integer*

Syntax Description

integer
: the positive number of rows to hold in memory. SAS allows the maximum number that the DBMS allows.

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, 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 under UNIX and PC Hosts: If you do not specify this option, the buffer size is automatically calculated based on the row length of your data and the SQLExtendedFetch API call is used (this is the default).

DB2 under z/OS: For SAS 9.2 and above, the default is 1 and the maximum value is 32,767.

Microsoft SQL Server, ODBC: If you do not specify this option, the SQLFetch API call is used and no internal SAS buffering is performed (this is the default). When you set READBUFF=1 or greater, the SQLExtendedFetch API call is used.

HP Neoview, Netezza: The default is automatically calculated based on row length.

OLE DB: The default is 1.

Oracle: The default is 250.

Sybase: The default is 100.

READ_ISOLATION_LEVEL= LIBNAME Option

Defines the degree of isolation of the current application process from other concurrently running application processes.

Valid in: SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. See the DBMS-specific reference section for details.

Default: DBMS-specific

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, Informix, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: READ_ISOLATION_LEVEL= data set option, READ_LOCK_TYPE= LIBNAME option

[DBMS-specific locking information] DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

Syntax

READ_ISOLATION_LEVEL=*DBMS-specific value*

Syntax Description

See the documentation for your SAS/ACCESS interface for 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.
- update activity of other concurrently executing application processes can affect the current application.

DB2 under UNIX and PC Hosts, ODBC: This option is ignored if you do not set the READ_LOCK_TYPE= LIBNAME option to ROW.

See the locking topic for your interface in the DBMS-specific reference section for details.

READ_LOCK_TYPE= LIBNAME Option

Specifies how data in a DBMS table is locked during a READ transaction.

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS-specific

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: CONNECTION= LIBNAME option, READ_ISOLATION_LEVEL= LIBNAME option, READ_LOCK_TYPE= data set option

[DBMS-specific locking information] DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

Syntax

READ_LOCK_TYPE=ROW | PAGE | TABLE | NOLOCK | VIEW

Syntax Description

ROW [valid for DB2 under UNIX and PC Hosts, Microsoft SQL Server, ODBC, Oracle, Sybase IQ]
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= LIBNAME option.

PAGE [valid for Sybase]
locks a page of data, which is a DBMS-specific number of bytes. (This value is valid in the Sybase interface.)

TABLE [valid for DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, Oracle, Sybase IQ, Teradata]
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).

NOLOCK [valid for Microsoft SQL Server, ODBC with Microsoft SQL Server driver, OLE DB, Oracle, Sybase]
does not lock the DBMS table, pages, or rows during a read transaction.

VIEW [valid for Teradata]
locks the entire DBMS view.

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 locking topic for your interface in the DBMS-specific reference section for details.

Example

In this example, the libref MYDBLIB uses SAS/ACCESS Interface to Oracle to connect to an Oracle database. USER=, PASSWORD=, and PATH= are SAS/ACCESS connection options. The LIBNAME options specify to use row-level locking when data is read or updated.

```
libname mydblib oracle user=testuser password=testpass
        path=myorapath read_lock_type=row update_lock_type=row;
```

READ_MODE_WAIT= LIBNAME Option

During SAS/ACCESS Read operations, specifies whether Teradata should wait to acquire a lock or fail the request when a different user has already locked the DBMS resource.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Teradata

See: READ_MODE_WAIT= data set option, "Locking in the Teradata Interface"

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 Teradata fails 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, the executing SAS step fails. SAS/ACCESS continues processing 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 the lock that you want. 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.

REMOTE_DBTYPE= LIBNAME Option

Specifies whether the libref points to a database server on z/OS or to one on Linux, UNIX, or Windows.

Valid in: SAS/ACCESS LIBNAME statement

Default: ZOS

Restriction: This option is ignored if you do not use is with either the SERVER= CONNECT statement option or the LOCATION= LIBNAME option.

Requirement: Use this option with the SERVER= CONNECT statement option or the LOCATION= LIBNAME option.

Data source: DB2 under z/OS

See: LOCATION= LIBNAME option, SERVER= CONNECT statement option (SQL Pass-Through Facility Specifics for DB2 under z/OS - Key Information)

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 (LUW).

ZOS
specifies that the database server that is accessed through the libref resides on z/OS.

Details

Specifying REMOTE_DBTYPE= in the LIBNAME statement ensures that the SQL that some SAS procedures use to access the DB2 catalog tables is generated properly and is based on the database server type. It also lets such special catalog calls as DBMS::Indexes function properly when the target database does not reside on a mainframe computer.

If the target data source is a DB2 LUW or another DB2 database on z/OS, the SQL dictionary is loaded when you specify this option.

Example

This example uses REMOTE_DBTYPE= with the SERVER= option.

```
libname mylib db2 ssid=db2a server=db2_udb remote_dbtype=luw;
proc datasets lib=mylib;
quit;
```

By specifying REMOTE_DBTYPE=LUW, this SAS code lets the catalog call 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;
```

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: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: UPDATE_ISOLATION_LEVEL= LIBNAME option, UPDATE_LOCK_TYPE= LIBNAME option

Syntax

REREAD_EXPOSURE=YES | NO

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, you read a different row the next time you read row 5. 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.

HP Neoview, Netezza, ODBC, OLE DB: If you set this option to YES, it is advisable to set UPDATE_ISOLATION_LEVEL=S (serializable) to avoid data integrity problems.

Oracle: If you set this option to YES, it is advisable to set UPDATE_LOCK_TYPE=TABLE to avoid data integrity problems.

RESULTS= LIBNAME Option

Determines where to store query results.

Valid in: SAS/ACCESS LIBNAME statement

Default: MEMORY

Data source: MySQL

Syntax

RESULTS=MEMORY | SERVER | DISK

Syntax Description

MEMORY
stores query results in client memory.

SERVER
stores query results on the server.

DISK
stores query results in a temporary disk file on the client computer.

Details

Multiple concurrent connections to the server are not supported. Therefore, when RESULTS=SERVER, the entire query must be one that you can push to the server. If not, this message appears in the log:

```
Commands out of sync; you can't run this command now.
```

RESULTS=DISK lets you run complex queries with result sets that would typically cause an out-of-memory error. Result-set size is limited only to the free space on the drive that is used for temporary files.

Example: Sending Results to a Temporary Disk File

```
libname spooled mysql…results=disk;
```

SCHEMA= LIBNAME Option

Allows reading of such database objects as tables and views in the specified schema.

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: SCHEMA= data set option, PRESERVE_TAB_NAMES= LIBNAME option

Syntax

SCHEMA=*schema-name*

Syntax Description

schema-name
specifies the name that is assigned to a logical classification of objects in a relational database.

Details

For this option to work, you must have the appropriate privileges to the specified schema.

If you do not specify this option, 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.

Aster n*Cluster:* The default is **`none`**, which uses the database user's default schema. However, the user name is used instead when the user's default scheme is the user name. An example is when SQLTables is called to obtain a table listing using PROC DATASETS or SAS Explorer.

Oracle: 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.

Sybase: You cannot use the SCHEMA= option when you use UPDATE_LOCK_TYPE=PAGE to update a table.

Teradata: 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 lets you 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.) For more information about changing the default database, see the DATABASE statement in your Teradata documentation.

Example

In this example, SCHEMA= causes DB2 to interpret any reference in SAS to mydb.employee as scott.employee.

```
libname mydb db2 SCHEMA=SCOTT;
```

To access an Oracle object in another schema, use the SCHEMA= option, as in this example. The schema name is typically a user name or ID.

```
libname mydblib oracle user=testuser
        password=testpass path='hrdept_002' schema=john;
```

In this 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, specify the schema in the SCHEMA= option and then put in the *libref.table* the DATA statement for the procedure.

```
libname cargo oracle schema=airports user=testuser password=testpass
        path="myorapath";
proc print data=cargo.schedule;
run;
```

In this Teradata example, the testuser user 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;
```

SESSIONS= LIBNAME Option

Specifies how many Teradata sessions to be logged on when using FastLoad, FastExport, or Multiload.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Teradata

See: FASTEXPORT= LIBNAME option, SESSIONS= data set option, “Using MultiLoad”

Syntax

SESSIONS=*number-of-sessions*

Syntax Description

number-of-sessions
specifies a numeric value that indicates the number of sessions to be logged on.

Details

When you read data with FastExport or load data with FastLoad or MultiLoad, you can request multiple sessions to increase throughput. Using large values might not necessarily increase throughput due to the overhead associated with session management. Check whether your site has any recommended value for the number of sessions to use. See your Teradata documentation for details about using multiple sessions.

Example

This example uses SESSIONS= in a LIBNAME statement to request five sessions for loading data with FastLoad.

```
libname x teradata user=prboni pw=prboni SESSIONS=2;
proc datasets library=x;
  delete test;run;
data x.test(FASTLOAD=YES);
i=5;
run;
```

SHOW_SYNONYMS= LIBNAME Option

Specifies whether PROC DATASETS shows synonyms, tables, views, or materialized views for the current user and schema if you specified the SCHEMA= option.

Valid in: SAS/ACCESS LIBNAME statement

Default: YES

Restriction:

Data source: Oracle

See: DBLINK= LIBNAME option, DB_OBJECTS= LIBNAME option

Syntax

SHOW_SYNONYMS=<YES | NO>

Syntax Description

YES
specifies that PROC DATASETS shows only synonyms that represent tables, views, or materialized views for the current user.

NO
specifies that PROC DATASETS shows only tables, views, or materialized views for the current user.

Details

Rather than submit PROC DATASETS, you can select the libref in SAS Explorer to obtain this same information. By default, no PUBLIC synonyms are displayed unless you specify SCHEMA=PUBLIC.

When you specify only the SCHEMA option, the current schema is always displayed with the appropriate privileges.

Tables, views, materialized views, or synonyms on the remote database always are displayed when you specify the DBLINK= LIBNAME option. If a synonym represents an object on a remote database that you might not be able to read, you might receive an Oracle error. An example is a synonym representing a sequence.

Synonyms, tables, views, and materialized views in a different schema are also displayed.

SLEEP= LIBNAME Option

Specifies the number of minutes that FASTEXPORT=, FastLoad, or MultiLoad waits before trying again to log on to Teradata.

Valid in: SAS/ACCESS LIBNAME statement

Default: 6

Data source: Teradata

Tip: The data set option has precedence over the LIBNAME option.

See: DBSLICEPARM= LIBNAME option, DBSLICEPARM= data set option, DBSLICEPARM= system option, FASTEXPORT= LIBNAME option, Maximizing Teradata Load Performance, MULTILOAD= data set option, SLEEP= data set option, TENACITY= LIBNAME option, TENACITY= data set option, Using the TPT API

Syntax

SLEEP=*number-of-minutes*

Syntax Description

number-of-minutes
the number of minutes to wait before trying again to log on to Teradata.

Details

Use this option to indicate to FASTEXPORT=, FastLoad, or 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 on the database server setting.) The default value for SLEEP= is 6 minutes. The value that you specify for SLEEP= must be greater than 0.

Use SLEEP= with TENACITY=. TENACITY= specifies the time in hours that FASTEXPORT=, FastLoad, or MultiLoad must continue to try the logon operation. SLEEP= and TENACITY= function very much like the SLEEP and TENACITY run-time options of the native Teradata FastExport, FastLoad, or MultiLoad utility.

SPOOL= LIBNAME Option

Specifies whether SAS creates a utility spool file during read transactions that read data more than once.

Valid in: SAS/ACCESS LIBNAME statement

Default: YES

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: CONNECTION= LIBNAME option

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 reread from the DBMS table. This guarantees that the rowset 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 rowset is the same for each pass through the data.

DBMS [valid only for Oracle]
specifies that the required rows for all passes of the data are read from the DBMS table. However, additional enforcements are made on the DBMS server side to ensure that the rowset is the same for every pass through the data. This setting causes 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, you must set CONNECTION=UNIQUE or 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 input and output 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.

MySQL: Do not use SPOOL=NO with the MySQL interface.

Teradata: SPOOL=NO requires SAS/ACCESS to issue identical SELECT statements to Teradata twice. In addition, because the Teradata table can be modified between passes, SPOOL=NO can cause data integrity problems. Use SPOOL=NO with discretion.

SQL_FUNCTIONS= LIBNAME Option

Customizes the in-memory SQL dictionary function list for this particular LIBNAME statement.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Restrictions: Informix and OLE DB support only SQL_FUNCTIONS=ALL.

You must specify a two-part data set name, such as <*libref.member*> or an error results.

<*libref.member*> must be a SAS data set. No check is performed to ensure that it is assigned to the default Base SAS engine.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: SQL_FUNCTIONS_COPY= LIBNAME option

Syntax

SQL_FUNCTIONS=ALL | "<*libref.member*>" | "EXTERNAL_APPEND=<*libref.member*>"

Syntax Description

ALL
customizes the in-memory SQL dictionary function list for this particular LIBNAME statement by adding the set of all existing functions, even those that might be risky or untested.

EXTERNAL_REPLACE=<*libref.member*> [not valid for Informix, OLE DB]
indicates a user-specified, external SAS data set from which the complete function list in the SQL dictionary is to be built. The assumption is that the user has already issued a LIBNAME statement to the directory where the SAS data set exists.

EXTERNAL_APPEND=<*libref.member*> [not valid for Informix, OLE DB]
indicates a user-specified, external SAS data set from which additional functions are to be added to the existing function list in the SQL dictionary. The assumption is that the user has already issued a LIBNAME statement to the directory where the SAS data set exists.

Details

Using this option can cause unexpected results, especially if you use it for NULL processing and for handling date, time, and timestamp. For example, when executed without SQL_FUNCTIONS= enabled, this SAS code returns the SAS date 15308.

```
proc sql;
   select distinct DATE () from x.test;
quit;
```

However, with SQL_FUNCTIONS=ALL, the same code returns 2001-1-29, which is an ODBC date format. So you should exercise care when you use this option.

Functions that are passed are different for each DBMS. See the DBMS-specific reference section for your SAS/ACCESS interface for list of functions that it supports.

Here are additional details to keep in mind when you add to or modify the SAS data set.

Variable	Required*	Optional**	Read-Only**	Valid Values
SASFUNCNAME	X			Truncated to 32 characters if length is greater than 32
SASFUNCNAMELEN	X			Must correctly reflect the length of SASFUNCNAME
DBMSFUNCNAME	X			Truncated to 50 characters if length is greater than 50
DBMSFUNCNAMELEN	X			Must correctly reflect the length of DBMSFUNCNAME
FUNCTION_ CATEGORY		X		AGGREGATE , CONSTANT, SCALAR
FUNC_USAGE_ CONTEXT		X		SELECT_LIST, WHERE_ORDERBY
FUNCTION_ RETURNTYP		X		BINARY, CHAR, DATE, DATETIME, DECIMAL, GRAPHIC, INTEGER, INTERVAL, NUMERIC, TIME, VARCHAR
FUNCTION_NUM_ ARGS		X		0
CONVERT_ARGSS			X	Must be set to 0 for a newly added function.
ENGINEINDEX			X	Must remain unchanged for existing functions. Set to 0 for a newly added function.

* An error results when a value is missing.

** For new and existing functions.

Examples

Example 1: Include and Replace Existing Functions

You can use EXTERNAL_APPEND= to include one or more existing functions to the in-memory function list and EXTERNAL_REPLACE= to replace them. In this example, the DATEPART function in a SAS data set of Oracle functions by appending the function to an existing list of SAS functions.

```
proc sql;
create table work.append as select *
       from work.allfuncs where sasfuncname='DATEPART';
quit;
libname mydblib oracle sql_functions="EXTERNAL_APPEND=work.append"
    sql_functions_copy=saslog;
```

Example 2: Replace All SAS Functions with the Oracle Equivalent

In this example, the equivalent Oracle functions in a SAS data set replace all SAS functions that contain the letter I.

```
proc sql;
create table work.replace as select *
       from work.allfuncs where sasfuncname like '%I%';
quit;
libname mydblib oracle sql_functions="EXTERNAL_REPLACE=work.replace"
   sql_functions_copy=saslog;
```

Example 3: Add a New Function

```
data work.newfunc;
SASFUNCNAME = "sasname";
SASFUNCNAMELEN = 7;
DBMSFUNCNAME = "DBMSUDFName";
DBMSFUNCNAMELEN = 11;
 FUNCTION_CATEGORY = "CONSTANT";
 FUNC_USAGE_CONTEXT = "WHERE_ORDERBY";
 FUNCTION_RETURNTYP = "NUMERIC";
 FUNCTION_NUM_ARGS = 0;
 CONVERT_ARGS = 0;
 ENGINEINDEX = 0;
output;
run;
/* Add function to existing in-memory function list */
libname mydblib oracle sql_functions="EXTERNAL_APPEND=work.newfunc"
   sql_functions_copy=saslog;
```

See Also

- "Passing SAS Functions to Aster nCluster" on page 425
- "Passing SAS Functions to DB2 under UNIX and PC Hosts" on page 452
- "Passing SAS Functions to DB2 under z/OS" on page 492
- "Passing SAS Functions to Greenplum" on page 525
- "Passing SAS Functions to HP Neoview" on page 566
- "Passing SAS Functions to Informix" on page 585

- "Passing SAS Functions to Microsoft SQL Server" on page 603
- "Passing SAS Functions to MySQL" on page 616
- "Passing SAS Functions to Netezza" on page 632
- "Passing SAS Functions to ODBC" on page 665
- "Passing SAS Functions to OLE DB" on page 687
- "Passing SAS Functions to Oracle" on page 715
- "Passing SAS Functions to Sybase" on page 745
- "Passing SAS Functions to Sybase IQ" on page 767
- "Passing SAS Functions to Teradata" on page 796

SQL_FUNCTIONS_COPY= LIBNAME Option

Writes the function associated with this particular LIBNAME statement to a SAS data set or the SAS log.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Restrictions: You must specify a two-part data set name, such as <*libref.member*> or an error results.

<*libref.member*> must be a SAS data set. It is not checked to make sure that it is assigned to the default Base SAS engine.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, MySQL, Netezza, ODBC, Oracle, Sybase, Sybase IQ, Teradata

See: SQL_FUNCTIONS= LIBNAME option\

Syntax

SQL_FUNCTIONS_COPY=<*libref.member*> | SASLOG

Syntax Description

<libref.member>
writes the current in-memory function list to a user-specified SAS data set for this particular LIBNAME statement.

SASLOG
writes the current in-memory function list to the SAS log for this particular LIBNAME statement.

SQL_OJ_ANSI= LIBNAME Option

Specifies whether to pass ANSI outer-join syntax through to the database.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Restriction: Sybase can process SQL outer joins only if the version of the Adaptive Server Enterprise (ASE) database is 12.5.2 or higher.

Data source: Sybase

Syntax

SQL_OJ_ANSI=YES | NO

Syntax Description

YES
specifies that ANSI outer-join syntax is passed through to the database.

NO
disables pass-through of ANSI outer-joins.

SQLGENERATION= LIBNAME Option

Specifies whether and when SAS procedures generate SQL for in-database processing of source data.

Valid in: SAS/ACCESS LIBNAME statement

Default: NONE DBMS='ASTER DB2 GREENPLM NETEZZA ORACLE TERADATA'

Restriction: You must specify NONE and DBMS, which indicate the primary state.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, Netezza, Oracle, Teradata

See: SQLGENERATION= system option, Table 12.1, *SAS In-Database Products: Administrator's Guide*

Syntax

SQLGENERATION=<(>NONE | DBMS <DBMS='*engine1 engine2…enginen*'>
<<EXCLUDEDB=*engine* | '*engine1 engine2…enginen*'>>
<<EXCLUDEPROC="*engine*='
proc1 proc2…procn' *enginen*='*proc1 proc2…procn*' "> <)>>

SQLGENERATION=" "

Syntax Description

NONE
prevents those SAS procedures that are enabled for in-database processing from generating SQL for in-database processing. This is a primary state.

DBMS
allows SAS procedures that are enabled for in-database processing to generate SQL for in-database processing of DBMS tables through supported SAS/ACCESS engines. This is a primary state.

DBMS='*engine1 engine2…enginen*'
specifies one or more SAS/ACCESS engines. It modifies the primary state.

Restriction: The maximum length of an engine name is 8 characters.

EXCLUDEDB=*engine* | '*engine1 engine2…enginen*'
prevents SAS procedures from generating SQL for in-database processing for one or more specified SAS/ACCESS engines.

Restriction: The maximum length of an engine name is 8 characters.

EXCLUDEPROC="*engine*='*proc1 proc2…procn*' *enginen*='*proc1 proc2…procn*' "
identifies engine-specific SAS procedures that do not support in-database processing.

Restrictions:
The maximum length of a procedure name is 16 characters.

An engine can appear only once, and a procedure can appear only once for a given engine.

" "
resets the value to the default that was shipped.

Details

Use this option with such procedures as PROC FREQ to indicate what SQL is generated for in-database processing based on the type of subsetting that you need and the SAS/ACCESS engines that you want to access the source table.

The maximum length of the option value is 4096. Also, parentheses are required when this option value contains multiple keywords.

Not all procedures support SQL generation for in-database processing for every engine type. If you specify an unsupported setting, an error message indicates the level of SQL generation that is not supported, and the procedure can reset to the default so that source table records can be read and processed within SAS. If this is not possible, the procedure ends and sets SYSERR= as needed.

You can specify different SQLGENERATION= values for the DATA= and OUT= data sets by using different LIBNAME statements for each of these two data sets.

STRINGDATES= LIBNAME Option

Specifies whether to read date and time values from the database as character strings or as numeric date values.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, OLE DB, Sybase IQ

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

Use STRINGDATES=NO for SAS 6 compatibility.

TEMPORAL_QUALIFIER= LIBNAME Option

Specifies time-dimension criteria for retrieving data from Teradata.

Valid in: SAS/ACCESS LIBNAME statement

Default: CURRENT VALIDTIME [valid-time column], CURRENT TRANSACTIONTIME [transaction-time column]

Interaction: Specifying values in a DATA step overrides LIBNAME values.

Data source: Teradata

See: TEMPORAL_QUALIFIER= data set option

Syntax

TEMPORAL_QUALIFIER=CURRENT VALIDTIME
| VALIDTIME AS OF PERIOD '*period*' | SEQUENCED VALIDTIME
| NONSEQUENCED VALIDTIME '*period*'

TEMPORAL_QUALIFIER=CURRENT TRANSACTIONTIME
| TRANSACTIONTIME AS OF PERIOD '*period*'
NONSEQUENCED TRANSACTIONTIME

Syntax Description

CURRENT VALIDTIME
selects rows that are valid at the current time.

VALIDTIME AS OF PERIOD '*period*'
selects rows with valid-time periods that overlap the specified AS OF period. For the period, you can specify either a single date or a time period (date range) by specifying a start date and an end date.

SEQUENCED VALIDTIME PERIOD '*period*'
selects history, current, or future rows that are valid for the specified time period.

NONSEQUENCED VALIDTIME PERIOD '*period*'
treats the table as nontemporal.

CURRENT TRANSACTIONTIME
selects rows that are open in transaction time.

TRANSACTIONTIME AS OF '*period*'
selects rows with transaction-time periods that overlap the specified AS OF period. For the period, you can specify either a single date or a time period (date range) by specifying a start date and an end date.

NONSEQUENCED TRANSACTIONTIME PERIOD '*period*'
treats the table as nontemporal.

Details

Use temporal qualifiers to specify time criteria for selecting data from temporal tables.

Temporal qualifiers that you specify in a LIBNAME statement apply only to that Teradata session and are implemented through session commands that are issued at connect time. For example, if you specify TEMPORAL_QUALIFIER='AS OF PERIOD '(1999-01-01, 2099-01-05)' ' in a LIBNAME statement, below is the Teradata SET SESSION command that is issued at connect time. The SQL is submitted as usual.

```
.SET SESSION ASOF PERIOD '(1999-01-01, 2099-01-05)'
```

Example: Select Valid-Time Rows for a Specific Date for the Current Session

In this example, valid-time rows are selected for a specific date from the mytest data set.

```
/* Consider data as of 1995-01-01. */
libname x teradata user=testuser pw=testpw server=td1310
        TEMPORAL_QUALIFIER='VALIDTIME AS OF DATE '1995-01-01' '

/* .SET SESSION VALIDTIME ASOF DATE'1995-01-01'; is issued
   before submitting the SQL "Select * from mytest" */
proc print data=x.mytest(DBSLICEPARM=ALL);
run;
```

TENACITY= LIBNAME Option

Specifies how many hours that FASTEXPORT=, FastLoad, or MultiLoad continues to try log on again to Teradata if the maximum number of Teradata utilities are already running.

Valid in: SAS/ACCESS LIBNAME statement

Default: 0 [FastLoad], 4 [FASTEXPORT=, MultiLoad]

Data source: Teradata

Tip: The data set option has precedence over the LIBNAME option.

See: DBSLICEPARM= LIBNAME option, DBSLICEPARM= data set option, DBSLICEPARM= system option, FASTEXPORT= LIBNAME option, Maximizing Teradata Load Performance, MULTILOAD= data set option, SLEEP= LIBNAME option, SLEEP= data set option, TENACITY= data set option, Using the TPT API

Syntax

TENACITY=*number-of-hours*

Syntax Description

number-of-hours
specifies the number of hours to continue to try again to log on to Teradata.

Details

Use this option to indicate to FASTEXPORT=, FastLoad, or 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 on the database server setting.) The default value for

TENACITY= is 4 hours. The value specified for TENACITY= must be greater than zero.

Use TENACITY= with SLEEP=. SLEEP= specifies the number of minutes that FASTEXPORT=, FastLoad, or 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 FastExport, FastLoad, or MultiLoad utility.

Here is an example of the message that is written to the SAS log if the time period that TENACITY= specifies is exceeded.

```
ERROR:  MultiLoad failed unexpectedly with returncode 12
```

Check the FASTEXPORT=, FastLoad, or MultiLoad log for more information about the cause of the FASTEXPORT=, FastLoad, or MultiLoad failure. SAS does not receive any informational messages from Teradata in either of these situations:

- when the currently run FASTEXPORT=, FastLoad, or MultiLoad process waits because the maximum number of utilities are already running
- if FASTEXPORT=, FastLoad, or MultiLoad is terminated because the time limit that TENACITY= specifies has been exceeded

The native Teradata FastExport, FastLoad, or MultiLoad utility sends messages associated with SLEEP= and TENACITY= only to the FastExport, FastLoad, or MultiLoad log. Therefore, nothing is written to the SAS log.

TPT= LIBNAME Option

Specifies whether SAS uses the Teradata Parallel Transporter (TPT) API to load data when SAS requests a Fastload, MultiLoad, or Multi-Statement insert.

Valid in: SAS/ACCESS LIBNAME statement

Default: YES

Data source: Teradata

See: BULKLOAD= LIBNAME option, BULKLOAD= data set option, LOGDB= LIBNAME option, “Maximizing Teradata Load Performance ”, MULTILOAD= data set option, MULTISTMT= data set option, TPT= data set option, TPT_APPL_PHASE= data set option, TPT_BUFFER_SIZE= data set option, TPT_CHECKPOINT= data set option, TPT_DATA_ENCRYPTION= data set option, TPT_ERROR_TABLE_1= data set option, TPT_ERROR_TABLE_2= data set option, TPT_LOG_TABLE= data set option, TPT_MAX_SESSIONS= data set option, TPT_MIN_SESSIONS= data set option, TPT_PACK= set option, TPT_PACKMAXIMUM= data set option, TPT_RESTART= data set option, TPT_TRACE_LEVEL= data set option, TPT_TRACE_LEVEL_INF= data set option, TPT_TRACE_OUTPUT= data set option, TPT_WORK_TABLE= data set option, “Using the TPT API ”

Syntax

TPT=YES | NO

Syntax Description

YES

specifies that SAS uses the TPT API when Fastload, MultiLoad, or Multi-Statement insert is requested.

NO
specifies that SAS does not use the TPT API when Fastload, MultiLoad, or Multi-Statement insert is requested.

Details

By using the TPT API, you can load data into a Teradata table without working directly with such stand-alone Teradata utilities as Fastload, MultiLoad, or TPump. When TPT=NO, SAS uses the TPT API load driver for FastLoad, the update driver for MultiLoad, and the stream driver for Multi-Statement insert.

When TPT=YES, sometimes SAS cannot use the TPT API due to an error or because it is not installed on the system. When this happens, SAS does not produce an error, but it still tries to load data using the requested load method (Fastload, MultiLoad, or Multi-Statement insert). To check whether SAS used the TPT API to load data, look for a similar message to this one in the SAS log.

```
NOTE: Teradata connection: TPT FastLoad/MultiLoad/MultiStatement
insert has read n row(s).
```

Example: Load SAS Data into Teradata

In this example, SAS data is loaded into Teradata using the TPT API. This is the default method of loading when Fastload, MultiLoad, or Multi-Statement insert are requested. SAS still tries to load data even if it cannot use the TPT API.

```
libname tera teradata user=testuser pw=testpw TPT=YES;
/* Create data */
data testdata;
do i=1 to 100;
    output;
end;
run;

/* Load using MultiLoad TPT.  This note appears in the SAS
   log if SAS uses TPT.  NOTE:  Teradata connection:
   TPT MultiLoad has inserted 100 row(s).*/
data tera.testdata(MULTILOAD=YES);
set testdata;
run;
```

TR_ENABLE_INTERRUPT= LIBNAME Option

Allows interruption of any long-running SQL processes that are involved in creating the result set or spool file.

Valid in:	SAS/ACCESS LIBNAME statement
Default:	NO
Restriction:	Valid for only Windows and UNIX platforms
Data source:	Teradata

Syntax

TR_ENABLE_INTERRUPT=YES | NO

Syntax Description

YES
allows interruption of long-running SQL processes that are involved in creating the result set or spool file.

NO
disables the interrupt processing code path.

Details

When set to YES, here is how you can use this option:

- to interrupt any SELECT SQL statement that was submitted by using the SELECT * FROM CONNECTION as a pass-through statement
- by using the **Interrupt/Attention** button to interrupt a query that you submitted through a DATA STEP, through PROC SQL implicit pass-through, or through explicit pass-through

Once the result set or spool file forms on the Teradata server and SAS is fetching the results, it is likely that the interrupt might no longer be available and you must wait until all results are fetched. The interrupt works in only one of these cases:

- when the Teradata server is building the result set or spool file.
- if the Teradata server is in a wait state before building the result set or spool file because of locking.

Example

```
libname x teradata user=sasias pass=sasias
  TR_ENABLE_INTERRUPT=YES server=tera2650;

data _NULL_; set x.paul_test; run;

proc datasets lib=x; quit;

proc sql;
create table work.a as select * from x.td_cancel, x.td_cancel;
quit;

proc sql; connect to teradata (user=sasias pass=sasias
  TR_ENABLE_INTERRUPT=YES server=tera2650);
select * from connection to teradata
  ( select * From td_cancel a , td_cancel b );
quit;
```

TRACE= LIBNAME Option

Specifies whether to turn on tracing information for use in debugging.

Valid in: SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. See the DBMS-specific reference section for details.

Default: NO

Restriction: This option is not supported on UNIX platforms.

Data source: Aster *n*Cluster, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, Sybase IQ

See: TRACEFILE= LIBNAME option

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 TRACEFILE= specifies.

NO
specifies that tracing is not turned on.

TRACEFILE= LIBNAME Option

Specifies the filename to which the DBMS driver manager writes trace information.

Valid in: SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. See the DBMS-specific references section for details.

Default: none

Restrictions: This option is not supported on UNIX platforms.
TRACEFILE= is used only when TRACE=YES.

Data source: Aster *n*Cluster, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, Sybase IQ

See: TRACE= LIBNAME option

Syntax

TRACEFILE= *filename* | <'>*path-and-filename*<'>

Details

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.

UPDATE_ISOLATION_LEVEL= LIBNAME Option

Defines the degree of isolation of the current application process from other concurrently running application processes.

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS-specific

Restriction: This option is ignored in the interfaces to ODBC and DB2 under UNIX and PC Hosts if you do not set UPDATE_LOCK_TYPE=ROW.

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: UPDATE_ISOLATION_LEVEL= data set option, UPDATE_LOCK_TYPE= LIBNAME optionUPDATE_LOCK_TYPE= data set option

[DBMS-specific locking details] DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Teradata

Syntax

UPDATE_ISOLATION_LEVEL=*DBMS-specific-value*

Syntax Description

The values for this option are DBMS-specific. See the DBMS-specific reference section for details.

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.
- update activity of other concurrently executing application processes can affect the current application.

This option is ignored in the interfaces to DB2 under UNIX and PC Hosts and ODBC if you do not set UPDATE_LOCK_TYPE=ROW.

See the locking topic for your interface in the DBMS-specific reference section for details.

See Also

- "Locking in the DB2 under UNIX and PC Hosts Interface" on page 456
- "Locking in the DB2 under z/OS Interface" on page 502
- "Locking in the Microsoft SQL Server Interface" on page 603
- "Locking in the ODBC Interface" on page 666
- "Locking in the OLE DB Interface" on page 689
- "Locking in the Oracle Interface" on page 718
- "Locking in the Sybase Interface" on page 747
- "Locking in the Sybase IQ Interface" on page 769
- "Locking in the Teradata Interface" on page 814

UPDATE_LOCK_TYPE= LIBNAME Option

Specifies how data in a DBMS table is locked during an update transaction.

Valid in: SAS/ACCESS LIBNAME statement

Default: DBMS-specific

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: UPDATE_LOCK_TYPE= data set option, SCHEMA= LIBNAME option

Syntax

UPDATE_LOCK_TYPE=ROW | PAGE | TABLE | NOLOCK |VIEW

Syntax Description

ROW [valid for DB2 under UNIX and PC Hosts, Microsoft SQL Server, ODBC, Oracle]
locks a row if any of its columns are to be updated.

PAGE [valid for Sybase]
locks a page of data, which is a DBMS-specific number of bytes. This value is not valid for the Sybase interface when you use the .

TABLE [valid for DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, Oracle, Sybase IQ, Teradata]
locks the entire DBMS table.

NOLOCK [valid for Microsoft SQL Server, ODBC with Microsoft SQL Server driver, OLE DB, Oracle, Sybase]
does not lock the DBMS table, page, or any rows when reading them for update.

VIEW [valid for Teradata]
locks the entire DBMS view.

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 locking topic for your interface in the DBMS-specific reference section for details.

See Also

- "Locking in the DB2 under UNIX and PC Hosts Interface" on page 456
- "Locking in the DB2 under z/OS Interface" on page 502
- "Locking in the Informix Interface" on page 586
- "Locking in the Microsoft SQL Server Interface" on page 603
- "Locking in the ODBC Interface" on page 666
- "Locking in the OLE DB Interface" on page 689
- "Locking in the Oracle Interface" on page 718
- "Locking in the Sybase Interface" on page 747
- "Locking in the Sybase IQ Interface" on page 769
- "Locking in the Teradata Interface" on page 814

UPDATE_MODE_WAIT= LIBNAME Option

Specifies during SAS/ACCESS Update operations whether Teradata should wait to acquire a lock or fail the request when a different user has locked the DBMS resource.

Valid in: SAS/ACCESS LIBNAME statement

Default: none

Data source: Teradata

See: "Locking in the Teradata Interface"UPDATE_MODE_WAIT= data set option

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 that Teradata fails the lock request if the specified DBMS resource is locked.

Details

If you specify UPDATE_MODE_WAIT=NO and a different user holds a restrictive lock, 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 the lock that you want. 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 the standard Teradata standard locking is undesirable.

UPDATE_MULT_ROWS= LIBNAME Option

Indicates whether SAS updates multiple rows from a data source, such as a DBMS table.

Valid in: SAS/ACCESS LIBNAME statement

Default: NO

Data source: Aster *n*Cluster, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, OLE DB, Sybase IQ

See: DELETE_MULT_ROWS= LIBNAME option

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 this DBMS SQL statement well and therefore update more than the current row with this statement:

```
UPDATE…WHERE CURRENT OF CURSOR
```

UPDATE_MULT_ROWS= lets SAS/ACCESS 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

Default: YES [except for the Oracle drivers from Microsoft and Oracle]

Restriction: The default for the Oracle drivers from Microsoft and Oracle is NO because these drivers do not support CURRENT OF CURSOR operations.

Data source: Microsoft SQL Server, ODBC

See: UPDATE_SQL= data set option, , INSERT_SQL= LIBNAME option

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= LIBNAME option. The default for the Oracle drivers from Microsoft and Oracle is NO because these drivers do not support Current-Of-Cursor operations.

UPDATEBUFF= LIBNAME Option

Specifies the number of rows that are processed in a single DBMS Update or Delete operation.

Valid in: SAS/ACCESS LIBNAME statement

Default: 1

Data source: Oracle

See: UPDATEBUFF= data set option

Syntax

UPDATEBUFF=*positive-integer*

Syntax Description

positive-integer
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 one observation at a time. They do this because they use record-level locking by default and therefore lock only the observation that is currently being edited.

USE_ODBC_CL= LIBNAME Option

Indicates whether the Driver Manager uses the ODBC Cursor Library.

Valid in: SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. See the DBMS-specific reference section for details.

Default: NO

Data source: Aster *n*Cluster, HP Neoview, Microsoft SQL Server, Netezza, ODBC

See: For more information about the ODBC Cursor Library, see your vendor-specific documentation.

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.

UTILCONN_TRANSIENT= LIBNAME Option

Enables utility connections to maintain or drop, as needed.

Valid in: SAS/ACCESS LIBNAME statement and some DBMS-specific connection options. See the DBMS-specific reference section for details.

Default: YES (DB2 under z/OS), NO (Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLEDB, Oracle, Sybase, Sybase IQ, Teradata)

Restriction: UTILCONN_TRANSIENT= has no effect on engines that do not support utility connections.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DELETE_MULT_ROWS= LIBNAME option

Syntax

UTILCONN_TRANSIENT=YES | NO

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

A utility connection is used for engines that can lock system resources as a result of such operations as DELETE or RENAME, or as a result of queries on system tables or table indexes. This connection prevents COMMIT statements that are issued to unlock system resources from being submitted on the same connection that is being used for table processing. Keeping COMMIT statements from table processing connection alleviates such problems that they can cause as invalidating cursors and committing pending updates on the tables that are being processed.

Because a utility connection exists for each LIBNAME statement, the number of connections to a DBMS can be large as multiple librefs are assigned across multiple SAS sessions. Setting UTILCONN_TRANSIENT=YES keeps these connections from existing when they are not being used. This setting reduces the number of current connections to the DBMS at any given point in time.

Chapter 11

Data Set Options for Relational Databases

About the Data Set Options for Relational Databases

Overview

You can specify SAS/ACCESS data set options on a SAS data set when you access DBMS data with the SAS/ACCESS LIBNAME statement on page 93. 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. For options that you can assign to a group of relational DBMS tables or views, see "LIBNAME Options for Relational Databases" on page 99.

Here is an example of how you can SAS/ACCESS data set options.

```
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 896. Here is an example.

```
proc print mylib.myviewd(data-set-option=value)
```

You *cannot* use most data set options 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. SAS/ACCESS interfaces do not support the REPLACE= SAS data set option. For information about using SAS data set options, see *SAS Data Set Options: Reference*.

The information in this section explains all applicable data set options. The information includes DBMS support and the corresponding LIBNAME options, and refers you to documentation for your SAS/ACCESS interface when appropriate. For a list of the data set options available in your SAS/ACCESS interface with default values, see the reference section for your DBMS.

- Aster nCluster on page 420
- DB2 under UNIX and PC Hosts on page 438
- DB2 under z/OS on page 470
- Greenplum on page 519
- HP Neoview on page 559
- Informix on page 579
- Microsoft SQL Server on page 598
- MySQL on page 612
- Netezza on page 627
- ODBC on page 651
- OLE DB on page 679
- Oracle on page 720
- Sybase on page 735

- Sybase IQ on page 762
- Teradata on page 784

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: Optimizing Your SQL Usage" on page 45.

Dictionary

AUTHID= Data Set Option

Lets you 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)

Alias: SCHEMA=

Default: LIBNAME setting

Data source: DB2 under z/OS

See: AUTHID= LIBNAME option

Syntax

AUTHID=*authorization-ID*

Syntax Description

authorization-ID
limited to eight 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 you do not specify AUTHID=, the table name is not qualified before being 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.

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)

Default: LIBNAME setting

Data source: MySQL, Sybase

See: AUTOCOMMIT= LIBNAME option, DBCOMMIT= data set option

Syntax

AUTOCOMMIT=YES | NO

Syntax Description

YES
specifies that SAS commits all updates, inserts, and deletes immediately after they are executed and that no rollback is possible.

NO
specifies that SAS commits after processing the number of rows that DBCOMMIT= specifies or the default number of rows if you do not specify DBCOMMIT=.

BL_ALLOW_READ_ACCESS= Data Set Option

Specifies that the original table data is still visible to readers during bulk load.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Default:	NO
Requirement:	To specify this option, you must first set BULKLOAD=YES.
Data source:	DB2 under UNIX and PC Hosts
See:	BL_ALLOW_WRITE_ACCESS= data set option, BULKLOAD= data set option

Syntax

BL_ALLOW_READ_ACCESS=YES | NO

Syntax Description

YES
specifies that the original (unchanged) data in the table is still visible to readers while bulk load is in progress.

NO
specifies that readers cannot view the original data in the table while bulk load is in progress.

Details

For more information about using this option, see the SQLU_ALLOW_READ_ACCESS parameter in the *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

BL_ALLOW_WRITE_ACCESS= Data Set Option

Specifies that table data is still accessible to readers and writers while import is in progress.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Default:	NO
Requirement:	To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under UNIX and PC Hosts

See: BL_ALLOW_READ_ACCESS= data set option, BULKLOAD= data set option

Syntax

BL_ALLOW_WRITE_ACCESS=YES | NO

Syntax Description

YES
specifies that table data is still visible to readers and writers during data import.

NO
specifies that readers and writers cannot view table data during data import.

Details

For more information about using this option, see the SQLU_ALLOW_WRITE_ACCESS parameter in the *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

BL_BADDATA_FILE= Data Set Option

Specifies where to put records that failed to process internally.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: creates a data file in the current directory or with the default file specifications

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.

Data source: HP Neoview

See: BL_DISCARDS= data set option, BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_BADDATA_FILE=*filename*

Syntax Description

filename
name of the file that contains records that failed to process internally.

Details

For bulk load, these are source records that failed internal processing before they were written to the database. For example, a record might contain only six fields, but eight fields were expected. Load records are in the same format as the source file.

For extraction, these are records that were retrieved from the database that could not be properly written into the target format. For example, a database value might be a string of ten characters, but a fixed-width format of only eight characters was specified for the target file.

BL_BADFILE= Data Set Option

Identifies a file that contains records that were rejected during bulk load.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: creates a data file in the current directory or with the default file specifications

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle

See: BULKLOAD= data set option

Syntax

BL_BADFILE=*path-and-filename*

Syntax Description

path-and-filename

an SQL*Loader file to which rejected rows of data are written. On most platforms, the default filename takes the form BL_<*table*>_<*unique-ID*>.bad:

table

specifies the table name

unique-ID

specifies 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.

Either Oracle or the SQL*Loader can reject records. 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 details about overriding this, see the information about SQL*Loader file attributes in the SQL*Loader chapter in your Oracle user's guide for z/OS.

BL_CLIENT_DATAFILE= Data Set Option

Specifies the client view of the data file that contains DBMS data for bulk load.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: the current directory

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Sybase IQ

See: BULKLOAD= data set option

Syntax

BL_CLIENT_DATAFILE=*path-and-data-filename*

Syntax Description

path-and-data-filename

specifies the file that contains the rows of data to load or append into a DBMS table during bulk load. On most platforms, the default filename takes the form BL_*<table>*_*<unique-ID>*.*dat*:

table

specifies the table name.

unique-ID

specifies 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.

dat

specifies the .DAT file extension for the data file.

BL_CODEPAGE= Data Set Option

Identifies the code page that the DBMS engine uses to convert SAS character data to the current database code page during bulk load.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: the code page ID of the current window

Requirements: The value for this option must never be 0. If you do not wish any code page conversions to take place, use the BL_OPTIONS= option to specify 'FORCEIN'. Code page conversions occur for only DB2 character data types.

To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under UNIX and PC Hosts

See: BL_OPTIONS= data set option, BULKLOAD= data set option

Syntax

BL_CODEPAGE=*numeric-code-page-ID*

Syntax Description

numeric-code-page-ID

a numeric value that represents a character set that is used to interpret multibyte character data and determine the character values.

BL_CONTROL= Data Set Option

Identifies the file that contains control statements.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: FE_EXECNAME [Teradata]

Default: DBMS-specific

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle, Teradata

See: BL_DATAFILE= data set option, BL_DELETE_DATAFILE= data set option, BL_DELETE_ONLY_DATAFILE= data set option, BULKLOAD= data set option, DBSLICEPARM= LIBNAME option, DBSLICEPARM= data set option

Syntax

BL_CONTROL=*path-and-control-filename* [Oracle]

BL_CONTROL=*path-and-data-filename* [Teradata]

Syntax Description

***path-and-control-filename* [Oracle]**
specifies the SQL*Loader file (where SQLLDR control statements are written) that describe the data to include in bulk load.

***path-and-data-filename* [Teradata]**
specifies the name of the control file to generate for extracting data with SAS/ACCESS Interface to Teradata using FastExport multithreaded Read.

BL_*<table>*_*<unique-ID>*.ctl [Oracle, Teradata]
the default filename on most platforms, where:

table
specifies the table name

unique-ID
specifies 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

Oracle: The Oracle interface 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). By default it creates a control file in the current directory or with the default file specifications. If you do not specify this option and a control file does not already 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 Oracle interface reuses the file and replaces the contents with the new control statements.

Teradata: To specify this option, you must first set DBSLICEPARM=ALL as a LIBNAME or data set option for threaded Reads. By default SAS creates a data file in the current directory or with a platform-specific name. If you do not specify this option and a control file does not exist, SAS creates a script file in the current directory or with the default file specifications. If you do not specify this option and a control file already exists, the DATA step. SAS/ACCESS Interface to Teradata creates the control file by using information from the input data and SAS/ACCESS options. The file contains FastExport Language 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 FastExport should interpret the data that you are loading from the DATA (.DAT) file. Because the script file that SAS generates for FastExport must contain login information in clear text, it is recommended that you secure the script file by specifying a directory path that is protected.

Example: Generate Teradata Script Files

This example generates a Teradata script file, C:\protdir\fe.ctl on Windows.

```
DATA test;
SET teralib.mydata(DBSLICEPARM=ALL BL_CONTROL="C:\protdir\fe.ctl");
run;
```

This example generates a Teradata script file, /tmp/fe.ctl, on UNIX.

```
DATA test;
SET teralib.mydata(DBSLICEPARM=ALL BL_CONTROL="/tmp/fe.ctl");
run;
```

This example generates a script file, USERID.SECURE.SCR.CTL, by appending CTL and prepending the user ID.

```
DATA test;
SET teralib.mydata(DBSLICEPARM=ALL BL_CONTROL="SECURE.SCR");
run;
```

BL_COPY_LOCATION= Data Set Option

Specifies the directory to which DB2 saves a copy of the loaded data.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES and BL_RECOVERABLE=YES.

Data source: DB2 under UNIX and PC Hosts

See: BL_RECOVERABLE= data set option, BULKLOAD= data set option

Syntax

BL_COPY_LOCATION=*pathname*

BL_CPU_PARALLELISM= Data Set Option

Specifies the number of processes or threads to use when building table objects.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under UNIX and PC Hosts

See: BL_DATA_BUFFER_SIZE= data set option, BL_DISK_PARALLELISM= data set option, BULKLOAD= data set option

Syntax

BL_CPU_PARALLELISM=*number of processes or threads*

Syntax Description

number of processes or threads
specifies the number of processes or threads that the load utility uses to parse, convert, and format data records when building table objects.

Details

This option exploits intrapartition parallelism and significantly improves load performance. It is particularly useful when loading presorted data, because record order in the source data is preserved.

The maximum allowable number is 30. If the value is 0 or has not been specified, the load utility selects an intelligent default. This default is based on the number of available CPUs on the system at run time. If there is insufficient memory to support the specified value, the utility adjusts the value.

When BL_CPU_PARALLELISM is greater than 1, the flushing operations are asynchronous, permitting the loader to exploit the CPU. If tables include either LOB or LONG VARCHAR data, parallelism is not supported and the value is set to 1—regardless of the number of system CPUs or the specified value.

Although use of this parameter is not restricted to symmetric multiprocessor (SMP) hardware, you might not obtain any discernible performance benefit from using it in non-SMP environments.

For more information about using BL_CPU_PARALLELISM=, see the CPU_PARALLELISM parameter in the *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

BL_DATA_BUFFER_SIZE= Data Set Option

Specifies the total amount of memory to allocate for the bulk load utility to use as a buffer for transferring data.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under UNIX and PC Hosts

See: BL_CPU_PARALLELISM= data set option, BL_DISK_PARALLELISM= data set option, BULKLOAD= data set option

Syntax

BL_DATA_BUFFER_SIZE=*buffer-size*

Syntax Description

buffer-size
: specifies the total amount of memory (in 4KB pages)—regardless of the degree of parallelism—that is allocated for the bulk load utility to use as buffered space for transferring data within the utility.

Details

If you specify a value that is less than the algorithmic minimum, the minimum required resource is used and no warning is returned. This memory is allocated directly from the utility heap, the size of which you can modify through the util_heap_sz database configuration parameter. If you do not specify a valued, the utility calculates an intelligent default at run time that is based on a percentage of the free space that is available in the utility heap at the time of instantiation of the loader, as well as some characteristics of the table.

It is recommended that the buffer be several extents in size. An *extent* is the unit of movement for data within DB2, and the extent size can be one or more 4KB pages. The DATA BUFFER parameter is useful when you are working with large objects (LOBs) because it reduces input and output waiting time. The data buffer is allocated from the utility heap. Depending on the amount of storage available on your system, you should consider allocating more memory for use by the DB2 utilities. You can modify the database configuration parameter util_heap_sz accordingly. The default value for the Utility Heap Size configuration parameter is 5000 4KB pages. Because load is only one of several utilities that use memory from the utility heap, it is recommended that no more than 50% of the pages defined by this parameter be made available for the load utility, and that the utility heap be defined large enough.

For more information about using this option, see the DATA BUFFER parameter in the *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

BL_DATAFILE= Data Set Option

Identifies the file that contains DBMS data for bulk load.

Valid in: DATA and PROC steps (when accessing data using SAS/ACCESS software)

Default: DBMS-specific

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Netezza, Oracle, Sybase IQ

See: BL_CONTROL= data set option, BL_DELETE_DATAFILE= data set option, BL_DELETE_ONLY_DATAFILE= data set option, BL_PROTOCOL= data set option, BL_USE_PIPE= data set option, BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_DATAFILE=*path-and-data-filename*

Syntax Description

path-and-data-filename

specifies the file that contains the rows of data to load or append into a DBMS table during bulk load. On most platforms, the default filename takes the form BL_*<table>*_*<unique-ID>*.*ext*:

table

specifies the table name.

unique-ID

specifies 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.

ext

specifies the file extension (.DAT or .IXF) for the data file.

Details

DB2 under UNIX and PC Hosts: The default is the current directory.

Greenplum: This option specifies the name of the external file to load. It is meaningful only when BL_PROTOCOL= is set to gpfdist or file. If you do not specify this option, the filename is generated automatically. When you specify the filename with a full path, the path overrides the value of the GPLOAD_HOME environment variable. However, bulk load might fail if the path does not match the base directory that the gpfdist utility used.

HP Neoview, Netezza: You can use this option only when BL_USE_PIPE=NO. By default the SAS/ACCESS engine creates a data file from the input SAS data set in the current directory or with the default file specifications 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.

Oracle: 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. If you do not specify this option and a data file does not exist, the file is created 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. SAS/ACCESS Interface to Oracle on z/OS is the exception: The data file is never reused because the interface causes bulk load to fail instead of reusing a data file.

Sybase IQ: By default, the SAS/ACCESS engine creates a data file with a .DAT file extension in the current directory or with the default file specifications. Also, by default, the data file is deleted after the load is completed. To override this behavior, specify BL_DELETE_DATAFILE=NO.

BL_DATAFILE= Data Set Option [Teradata only]

Identifies the file that contains control statements.

Valid in: DATA and PROC steps (when accessing data using SAS/ACCESS software)

Default: creates a MultiLoad script file in the current directory or with a platform-specific name

Requirement: To specify this option, you must first set MULTILOAD=YES.

Data source: Teradata

See: BL_CONTROL= data set option, MULTILOAD= data set option

Syntax

BL_DATAFILE=*path-and-data-filename*

Syntax Description

path-and-data-filename
specifies the name of the control file to generate for loading data with SAS/ACCESS Interface to Teradata using MultiLoad. On most platforms, the default filename takes the form BL_*<table>*_*<unique-ID>.ctl*:

table
specifies the table name.

unique-ID
specifies 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

The file contains MultiLoad Language definitions that specify the location of the data and how the data corresponds to the database table. It specifies exactly how MultiLoad should interpret the data that you are loading. Because the script file that SAS generates for MultiLoad must contain login information in clear text, you should secure the script file by specifying a protected directory path.

Example: Generate Teradata Script Files

This example generates a Teradata script file, C:\protdir\ml.ctl, on Windows.

```
DATA teralib.test(DBSLICEPARM=ALL BL_DATAFILE="C:\protdir\ml.ctl");
SET teralib.mydata;
run;
```

This next example generates a Teradata script file, fe.ctl, for FastExport and ml.ctl for MultiLoad.

```
data teralib.test1(MULTILOAD=YES TPT=NO BL_DATAFILE="ml.ctl");
SET teralib.test2(DBSLICEPARM=ALL BL_CONTROL="fe.ctl");
run;
```

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)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2CURSOR='SELECT * from *filename*'

Details

You can use it to load DB2 tables directly from other DB2 and objects that are not DB2. However, before you can select data from a remote location, your database administrator must first populate the communication database with the appropriate entries.

BL_DB2DATACLAS= Data Set Option

Specifies a data class for a new SMS-managed data set.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BL_DB2MGMTCLAS= data set option, BL_DB2STORCLAS= data set option [contains sample code], BL_DB2UNITCOUNT= data set option, BULKLOAD= data set option

Syntax

BL_DB2DATACLAS=*data-class*

Details

This option applies to the control file (BL_DB2IN=), the input file (BL_DB2REC=), and the output file (BL_DB2PRINT=) for the bulk loader. Use this option to specify a data class for a new SMS-managed data set. SMS ignores this option if you specify it for a data set that SMS does not support. If SMS is not installed or active, the operating environment ignores any data class that BL_DB2DATACLAS= passes. Your site storage administrator defines the data class names that you can specify when you use this option.

BL_DB2DEVT_PERM= Data Set Option

Specifies the unit address or generic device type to use for permanent data sets that the LOAD utility creates—also SYSIN, SYSREC, and SYSPRINT when SAS allocates them.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: SYSDA

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2DEVT_PERM=*unit-specification*

BL_DB2DEVT_TEMP= Data Set Option

Specifies the unit address or generic device type to use for temporary data sets that the LOAD utility creates (Pnch, Copy1, Copy2, RCpy1, RCpy2, Work1, Work2).

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: SYSDA

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2DEVT_TEMP=*unit-specification*

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)

Default: a generated data set name

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2DISC=*data-set-name*

Details

The DSNUTILS procedure with DISP=(NEW,CATLG,CATLG) allocates this option. This option must be the name of a nonexistent data set, except on a RESTART because it would already have been created. The LOAD utility allocates it as DISP=(MOD,CATLG,CATLG) on a RESTART. The default is a generated data set name, which appears in output that is written to the DB2PRINT location.

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)

Default: a generated data set name

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2ERR=*data-set-name*

Details

The DSNUTILS procedure with DISP=(NEW,CATLG,CATLG) allocates this option. This option must be the name of a nonexistent data set, except on a RESTART because it would already have been created. The LOAD utility allocates it as DISP=(MOD,CATLG,CATLG) on a RESTART. The default is a generated data set name, which appears in output that is written to the DB2PRINT location.

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)

Default: a generated data set name

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2IN=*data-set-name*

Details

This option is allocated based on the value of BL_DB2LDEXT=. It is initially allocated as SPACE=(trk,(10,1),rlse) with the default being a generated data set name, which appears in the DB2PRINT output, with these DCB attributes:

DSORG=PS LRECL=516
RECFM=VB BLKSZE=23476

It supports these DCB attributes for existing data sets:

DSORG=PS
RECFM=F, FB, FS, FBS, V, VB, VS, or VBS
LRECL=any valid value for RECFM, which is < 32,760
BLKSIZE=any valid value for RECFM, which is < 32,760

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)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2LDCT1='*string*'

Details

This option specifies a string that contains a segment of the Load Utility Control Statement between 'LOAD DATA' and 'INTO TABLE'. Valid control statement options include but are not limited to RESUME, REPLACE, LOG, and ENFORCE.

You can use DB2 bulk-load control options (BL_DB2LDCT1=, BL_DB2LDCT2=, and BL_DB2DCT3= options to specify sections of the control statement, which the engine incorporates into the control statement that it generates. These options have no effect when BL_DB2LDEXT=USERUN. You can use these options as an alternative to specifying BL_DB2LDEXT=GENONLY and editing the control statement to include options that the engine cannot generate. In some cases it is necessary to specify at least one of these options—for example, if you run the utility on an existing table where you must specify either RESUME or REPLACE.

The LOAD utility requires that the control statement be in uppercase—except for objects such as table or column names, which must match the table. You must specify values for DB2 bulk-load control options using the correct case. SAS/ACCESS Interface to DB2 under z/OS cannot convert the entire control statement to uppercase because it might contain table or column names that must remain in lower case.

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)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2LDCT2='*string*'

Details

Valid control statement options include but are not limited to PART, PREFORMAT, RESUME, REPLACE, and WHEN.

BL_DB2LDCT3= Data Set Option

Specifies a string in the LOAD utility control statement after (*field-specification*).

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2LDCT3='*string*'

Details

This option handles any options that might be defined for this location in later versions of DB2.

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)

Default: GENRUN

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2LDEXT=GENRUN | GENONLY | USERUN

Syntax Description

GENRUN
generates the control (SYSIN) file and the data (SYSREC) file, and runs the utility with them.

GENONLY
generates the control (SYSIN) file and the data (SYSREC) file but does not run the utility. Use this method when you need to edit the control file or to 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. Existing files can be from a previous run or from previously run batch utility jobs. Use this method when you restart a previously stopped run of the utility.

All valid data sets that the utility accepts are supported when BL_DB2LDEXT=USERUN. However, syntax errors from the utility can occur because no parsing is done when reading in the SYSIN data set. Specifically, neither embedded comments (beginning with a double hyphen, '--') nor columns 73 through 80 of RECFM=FB LRECL=80 data sets are stripped from the control statement. The solution is to remove embedded comments and columns 73 through 80 of RECFM=FB LRECL=80 data sets from the data set. However, this is not an issue when you use engine-generated SYSIN data sets because they are RECFM=VB and therefore have no embedded comments.

Details

This option specifies the mode of execution for the DB2 LOAD utility, which involves creating data sets that the utility needs and to call the utility.

BL_DB2MGMTCLAS= Data Set Option

Specifies a management class for a new SMS-managed data set.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BL_DB2DATACLAS= data set option, BL_DB2STORCLAS= data set option [contains sample code], BL_DB2UNITCOUNT= data set option, BULKLOAD= data set option

Syntax

BL_DB2MGMTCLAS=*management-class*

Details

This option applies to the control file (BL_DB2IN), the input file (BL_DB2REC), and the output file (BL_DB2PRINT) for the bulk loader. Use this option to specify a management class for a new SMS-managed data set. If SMS is not installed or active, the operating environment ignores any management class that BL_DB2MGMTCLAS=

passes. Your site storage administrator defines the management class names that you can specify when you use this option.

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)

Default: a generated data set name

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2MAP=*data-set-name*

Details

The DSNUTILS procedure with DISP=(NEW,CATLG,CATLG) allocates this option. This option must be the name of a nonexistent data set, except on a RESTART because it would already have been created. The LOAD utility allocates it as DISP=(MOD,CATLG,CATLG) on a RESTART. The default is a generated data set name, which appears in output that is written to the DB2PRINT location.

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)

Default: a generated data set name

Requirement: To specify this option, you must first set BULKLOAD=YES. You must also specify BL_DB2PRNLOG=YES so that you can see the generated data set name in the SAS log.

Data source: DB2 under z/OS

See: BL_DB2PRNLOG= data set option [for generated data set name in the SAS log], BULKLOAD= data set option

Syntax

BL_DB2PRINT=*data-set-name*

Details

This option is allocated with DISP=(NEW,CATLG,DELETE) and SPACE=(trk,(10,1),rlse). The default is a generated data set name, which appears in the DB2PRINT dsn, with these DCB attributes:

DSORG=PS LRECL=258

RECFM=VBA BLKSIZE=262 – 32760

BL_DB2PRNLOG= Data Set Option

Determines whether to write SYSPRINT output to the SAS log.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: YES

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option, DB2 under z/OS Bulk Loading

Syntax

BL_DB2PRNLOG=YES | NO

Syntax Description

YES
specifies that SYSPRINT output is written to the SAS log.

NO
specifies that SYSPRINT output is not written to the SAS log.

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)

Default: a generated data set name

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2REC=*data-set-name*

Details

The value of BL_DB2LDEXT= allocates this option. It is initially allocated as SPACE=(cyl,(BL_DB2RECSP, 10%(BL_DB2RECSP)),rlse). The default is a generated data set name, which appears in output that is written to the DB2PRINT data set name. It supports these DCB attributes for existing data sets:

DSORG=PS LRECL=any valid value for RECFM
RECFM=FB BLKSIZE=any valid value for RECFM

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)

Default: 10

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2RECSP=*primary-allocation*

Details

The secondary allocation is 10% of the primary allocation.

BL_DB2RSTRT= Data Set Option

Tells the LOAD utility whether the current load is a restart and, if so, indicates where to begin.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Requirements: To specify this option, you must first set BULKLOAD=YES.

When you specify a value other than NO for BL_DB2RSTRT=, you must also specify BL_DB2TBLXST=YES and BL_DB2LDEXT=USERUN.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2RSTRT=NO | CURRENT | PHASE

Syntax Description

NO
specifies a new run (not restart) of the LOAD utility.

CURRENT
specifies to restart at the last commit point.

PHASE
specifies to restart at the beginning of the current phase.

BL_DB2SPC_PERM= Data Set Option

Determines the number of cylinders to specify as the primary allocation for permanent data sets that the LOAD utility creates.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 10

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2SPC_PERM=*primary-allocation*

Details

Permanent data sets are Disc, Maps, and Err. The DSNUTILS procedure controls the secondary allocation, which is 10% of the primary allocation.

BL_DB2SPC_TEMP= Data Set Option

Determines the number of cylinders to specify as the primary allocation for temporary data sets that the LOAD utility creates.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 10

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option, DB2 under z/OS Bulk Loading

Syntax

BL_DB2SPC_TEMP=*primary-allocation*

BL_DB2STORCLAS= Data Set Option

Specifies a storage class for a new SMS-managed data set.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BL_DB2DATACLAS= data set option, BL_DB2DEVT_PERM= data set option, BL_DB2MGMTCLAS= data set option, BL_DB2UNITCOUNT= data set option, BULKLOAD= data set option

Syntax

BL_DB2STORCLAS=*storage-class*

Details

A storage class contains the attributes that identify a storage service level that SMS uses for storage of the data set. It replaces any storage attributes that you specify in BL_DB2DEVT_PERM=.

This option applies to the control file (BL_DB2IN), the input file (BL_DB2REC), and the output file (BL_DB2PRINT) for the bulk loader. Use this option to specify a management class for a new SMS-managed data set. If SMS is not installed or active, the operating environment ignores any storage class that BL_DB2MGMTCLAS= passes. Your site storage administrator defines the storage class names that you can specify when you use this option.

Example: Generate SMS-Managed Control and Data Files

This example generates SMS-managed control and data files. It does not create the table, and you need not run the utility to load it.

```
libname db2lib db2 ssid=db2a;
data db2lib.customers (bulkload=yes
      bl_db2ldext=genonly
      bl_db2in='testuser.sysin'
      bl_db2rec='testuser.sysrec'
      bl_db2tblxst=yes
       bl_db2ldct1='REPLACE'
      bl_db2dataclas='STD'
      bl_db2mgmtclas='STD'
      bl_db2storclas='STD');
set work.customers;
run;
```

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)

Default: NO

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

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" on page 230.)

NO
specifies that the LOAD utility does not run against an existing table.

Details

SAS/ACCESS does not currently support table replacement. You cannot simply create a new copy of an existing table to replace the original table. Instead, you must delete the table and then create a new version of it.

The DB2 LOAD utility does not create tables—it loads data into existing tables. The DB2 under z/OS interface creates a table before loading data into it—whether you use SQL INSERT statements or start the LOAD utility.

You might want to start the utility for an existing table that the DB2 engine did not create. If so, specify BL_DB2TBLXST=YES to tell the engine that the table already exists. When BL_DB2TBLXST=YES, the engine neither verifies that the table does not already exist, which eliminates the NO REPLACE error, nor creates the table. BULKLOAD= is not valid for update opening of tables, which includes appending to an existing table. Therefore, to accomplish appending, use either BL_DB2TBLXST= with an output open (normally creates the table) or the LOAD utility against a previously created table. You can also use BL_DB2TBLXST= with BL_DB2LDEXT=GENONLY if the table does not yet exist and you do not want to create or load it yet. In this case the control and data files are generated but the table is neither created nor loaded.

Because the table might be empty or might contain rows, specify the appropriate LOAD utility control statement values for REPLACE, RESUME, or both by using BL_DB2LDCT1, BL_DB2LDCT2 , or both.

The data to be loaded into the existing table must match the table column types. The engine does not try to verify input data with the table definition. The LOAC utility flags any incompatible differences.

BL_DB2UNITCOUNT= Data Set Option

Specifies the number of volumes on which data sets can be extended.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Restriction: An error is returned if you specify a value for this option that exceeds the maximum number of volumes for the unit.

Requirements: To specify this option, you must first set BULKLOAD=YES.

This option applies only to the input file (BL_DB2REC data set), which is the file that must be loaded into the DB2 table.

Data source: DB2 under z/OS

See: BL_DB2DATACLAS= data set option, BL_DB2DEVT_PERM= data set option, BL_DB2STORCLAS= data set option [contains sample code], BL_DB2STORCLAS= data set option, BULKLOAD= data set option

Syntax

BL_DB2UNITCOUNT=*number-of-volumes*

Syntax Description

number-of-volumes
specifies the number of volumes across which data sets can be extended. It must be an integer between 1 and 59. This option is ignored if the value is greater than 59. See the details in this section.

Details

You must specify an integer from 1–59 as a value for this option. This option is ignored if the value is greater than 59. However, the value depends on the unit name in BL_DB2DEVT_PERM=. At the operating environment level an association exists that defines the maximum number of volumes for a unit name. Ask your storage administrator for this number.

The data class determines whether SMS-managed data sets can be extended on multiple volumes. When you specify both BL_DB2DATACLAS= and BL_DB2UNITCOUNT=, BL_DB2UNITCOUNT= overrides the unit count values for the data class.

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)

Default: user ID and second level DSN qualifier

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under z/OS

See: BULKLOAD= data set option

Syntax

BL_DB2UTID=*utility-ID*

Syntax Description

utility-ID
a character string up to 16 bytes long.

Details

By default, the value for this option is the user ID concatenated with the second-level data set name qualifier. The generated ID appears in output that is written to the DB2PRINT data set name. This name generation makes it easy to associate all information for each utility execution and to separate it from other executions.

BL_DBNAME= Data Set Option

Specifies the database name to use for bulk loading.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirements: To specify this option, you must first set BULKLOAD=YES.
You must enclose the database name in quotation marks.

Data source: Aster *n*Cluster

See: BL_HOST= data set option, BL_PATH= data set option, BULKLOAD= data set option

Syntax

BL_DBNAME='*database-name*'

Syntax Description

database-name
specifies the database name to use for bulk loading.

Details

Use this option to pass the database name to the DBMS bulk-load facility.

BL_DEFAULT_DIR= Data Set Option

Specifies where bulk load creates all intermediate files.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: <*database-name*>

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle

See: BULKLOAD= data set option

Syntax

BL_DEFAULT_DIR=<*host-specific-directory-path*>

Required Argument

<host-specific-directory-path>
specifies the host-specific directory path where intermediate bulk-load files (CTL, DAT, LOG, BAD, DSC) are created

Details

The value that you specify for this option is prepended to the filename. Be sure to provide the complete, host-specific directory path, including the file and directory separator character to accommodate all platforms.

Example: Create All Files in a Temporary Directory

In this example, bulk load creates all related files in the C:\temp directory.

```
data x.test (bulkload=yes BL_DEFAULT_DIR="c:\temp\" bl_delete_files=no);
c1=1;
run;
```

BL_DELETE_DATAFILE= Data Set Option

Specifies whether to delete only the data file or all files that the SAS/ACCESS engine creates for the DBMS bulk-load facility.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: BL_DELETE_FILES= [Oracle]

Default: YES

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Netezza, Oracle, Sybase IQ

See: BULKEXTRACT= data set option, BL_CONTROL= data set option, BL_DATAFILE= data set option, BL_DELETE_ONLY_DATAFILE= data set option, BL_USE_PIPE= data set option, BULKLOAD= data set option

Syntax

BL_DELETE_DATAFILE=YES | NO

Syntax Description

YES
deletes all (data, control, and log) files that the SAS/ACCESS engine creates for the DBMS bulk-load facility.

NO
does not delete these files.

Details

DB2 under UNIX and PC Hosts: Setting BL_DELETE_DATAFILE=YES deletes only the temporary data file that SAS/ACCESS creates after the load completes.

Greenplum: When BL_DELETE_DATAFILE=YES, the external data file is deleted after the load completes.

HP Neoview, Netezza: You can use this option only when BL_USE_PIPE=NO.

Oracle: When BL_DELETE_DATAFILE=YES, all files (DAT, CTL, and LOG) are deleted.

Examples

Example 1: Delete All Files

In this example the default is YES, so all files are deleted.

```
libname x oracle &connopts
proc datasets library=x;
  delete test1;run;
data x.test1 ( bulkload=yes );
c1=1;
run;
x dir BL_TEST1*.*;
```

Example 2: Retain All Files

No files are deleted in this example.

```
libname x oracle &connopts
proc datasets library=x;
  delete test2;run;
data x.test2 ( bulkload=yes bl_delete_files=no );
c1=1;
run;
x dir BL_TEST2*.*;
```

BL_DELETE_ONLY_DATAFILE= Data Set Option

Specifies whether to delete the data file that the SAS/ACCESS engine creates for the DBMS bulk-load facility.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle

See: BL_CONTROL= data set option, BL_DATAFILE= data set option, option, BULKLOAD= data set option

Syntax

BL_DELETE_ONLY_DATAFILE=YES | NO

Syntax Description

YES
deletes only the data file that the SAS/ACCESS engine creates for the DBMS bulk-load facility.

NO
does not delete the data file.

Details

Setting this option overrides the BL_DELETE_DATAFILE= option.

Examples

Example 1: Delete Only the Control and Log Files

BL_DELETE_DATAFILE=YES is the default in this example, so only the control and log files are deleted.

```
proc datasets library=x;
  delete test3;run;
data x.test3 ( bulkload=yes bl_delete_only_datafile=no );
c1=1;
run;
x dir BL_TEST3*.*;
```

Example 2: Retain All Files

Both options are set to NO in this example, so no files are deleted.

```
proc datasets library=x;
  delete test4;run;
data x.test4 ( bulkload=yes bl_delete_only_datafile=no bl_delete_files=NO );
c1=1;
run;
x dir BL_TEST4*.*;
```

Example 3: Delete Only the Data File

Only the data file is deleted in this example.

```
proc datasets library=x;
  delete test5;run;
data x.test5 ( bulkload=yes bl_delete_only_datafile=YES );
c1=1;
run;
x dir BL_TEST5*.*;
```

The same is true in this example.

```
proc datasets library=x;
  delete test6;run;
run;
data x.test6 ( bulkload=yes bl_delete_only_datafile=YES  bl_delete_files=NO );
c1=1;
run;
x dir BL_TEST6*.*;
```

BL_DELIMITER= Data Set Option

Specifies override of the default delimiter character for separating columns of data during data transfer or retrieval during bulk load or bulk unload.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: DELIM, DELIMIT [Hadoop]

Default: DBMS-specific

Requirement: To specify this option, you must first set BULKEXTRACT=YES or BULKLOAD=YES.

Data source: Aster *n*Cluster, Greenplum, Hadoop, HP Neoview, Netezza

See: BL_DATAFILE= data set option, BL_DELETE_DATAFILE= data set option, BL_FORCE_NOT_NULL= data set option, BL_FORMAT= data set option, BL_NULL= data set option, BL_OPTIONS= data set option, BL_QUOTE= data set option, BL_USE_PIPE= data set option, BULKEXTRACT= data set option on page 274, BULKLOAD= data set option, BULKUNLOAD= LIBNAME option, BULKUNLOAD= data set option

Syntax

BL_DELIMITER='<*any-single-character*>'

Details

Here is when you might want to use this option:

- to override the default delimiter character that the interface uses to separate columns of data that it transfers to or retrieves from the DBMS during bulk load (or bulk unload for Netezza)
- if your character data contains the default delimiter character, to avoid any problems while parsing the data stream

Aster n*Cluster:* The default is /t (the tab character).

Greenplum, Netezza: The default is the pipe symbol (|).

Hadoop: The default is \001 (Ctrl-A). To change the default delimiter, specify a value as either a single character or three-digit decimal ASCII value between 001 and 127 that represents the ASCII value of the delimiter that you want to use. You cannot use other typical SAS or UNIX formats such as '\001', 0x01 or '01'x because these do not work. Also, for such procedures as APPEND, SQL, or INSERT, the existing delimiter of the base table—the one being appended to—overrides any specified value for the DELIMITER= option. Otherwise, data corruption would result because the original and appended parts of the resulting table would use different delimiters.

HP Neoview: The default is the pipe symbol (|). Valid characters that you can use are a comma (,), a semicolon (;), or any ASCII character that you specify as an octal number except for these:

- upper- and lowercase letters (A through Z and a through z)
- decimal digits 0 through 9
- a carriage return (\015)
- a linefeed (\012)

For example, specify `BL_DELIMITER='\174'` to use the pipe symbol (| or \174 in octal representation) as a delimiter. You must specify octal numbers as three digits even if the first couple of digits would be 0. For example, use `'\003'` or `'\016'`, not `'\3'` or `'\16'`, or `'\136'` if you want to use the caret (^) symbol.

Sybase IQ: The default is | (the pipe symbol). You can specify the delimiter as a single printable character (such as |), or you can use hexadecimal notation to specify any single 8-bit hexadecimal ASCII code. For example, to use the tab character as a delimiter, you can specify BL_DELIMITER='\x09'.

Examples

Example 1: Specify the Default Delimiter

Data in this example contains the pipe symbol.

```
data work.testdel;
col1='my|data';col2=12;
run;
```

Example 2: Override the Default Delimiter

This example shows how you can override this default when BULKLOAD=YES.

```
/* Use a comma to delimit data */
proc append base=netlib.mydat(BULKLOAD=YES BL_DELIMITER=',')
data=work.testdel;
run;
```

Example 3: Override the Default Hadoop Delimiter

```
data db.joeapp (delim=007); set db.JoeTable2; run;
data db.joeapp (delim="127"); set db.JoeTable2; run;
data db.joeapp (delimiter=#); set db.JoeTable2; run;
data db.joeapp (delimit="#"); set db.JoeTable2; run;

proc sql;
  create table db.joeapp (delim='#') as select * from db.JoeTable2;
quit;
```

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)

Default: YES

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle

See: BULKLOAD= data set option

Syntax

BL_DIRECT_PATH=YES | NO

Syntax Description

YES
: sets the Oracle SQL*Loader option DIRECT to TRUE, letting the SQL*Loader use Direct Path Load to insert rows into a table.

NO
: sets the Oracle SQL*Loader option DIRECT to FALSE, letting the SQL*Loader use Conventional Path Load to insert rows into a table.

Details

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.

Direct Path Load creates data blocks that are already in the Oracle database block format. 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.

BL_DISCARDFILE= Data Set Option

Identifies the file that contains records that were filtered from bulk load because they did not match the criteria as specified in the CONTROL file.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: creates a file in the current directory or with the default file specifications

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle

See: BL_BADFILE= data set option [to set the name and location of the file that contains *rejected* rows], BULKLOAD= data set option

Syntax

BL_DISCARDFILE=*path-and-discard-filename*

Syntax Description

path-and-discard-filename

an SQL*Loader discard file containing rows that did not meet the specified criteria. On most platforms, the default filename takes the form BL_<*table*>_<*unique-ID*>.dsc:

table

specifies the table name

unique-ID

specifies 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

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 and replaces 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 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 such a case, see the section about SQL*Loader file attributes in the SQL*Loader chapter in the Oracle user's guide for z/OS.

BL_DISCARDS= Data Set Option

Specifies whether and when to stop processing a job, based on the number of discarded records.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 1000

Restriction: This option is ignored for extraction.

Requirements: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.
Enter 0 to disable this option.

Data source: HP Neoview

See: BL_BADDATA_FILE= data set option, BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_DISCARDS=*number-of-discarded-records*

Syntax Description

number
specifies whether and when to stop processing a job.

Details

Job processing stops when the number of records in the bad data file for the job reaches the specified number of discarded records.

BL_DISK_PARALLELISM= Data Set Option

Specifies the number of processes or threads to use when writing data to disk.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under UNIX and PC Hosts

See: BL_CPU_PARALLELISM= data set option, BL_DATA_BUFFER_SIZE= data set option, BULKLOAD= set option

Syntax

BL_DISK_PARALLELISM=*number-of-processes-or-threads*

Syntax Description

number of processes or threads
specifies the number of processes or threads that the load utility uses to write data records to the table-space containers.

Details

This option exploits the available containers when it loads data and significantly improves load performance.

The maximum number that is allowed is the greater of 50 or four times the BL_CPU_PARALLELISM value, which the load utility actually uses. By default, BL_DISK_PARALLELISM is equal to the sum of the table–space containers on all table spaces that contain objects for the table that is being loaded except where this value exceeds the maximum number that is allowed.

If you do not specify a value, the utility selects an intelligent default that is based on the number of table-space containers and the characteristics of the table.

For more information about using this option, see the DISK_PARALLELISM parameter in the *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

BL_ENCODING= Data Set Option

Specifies the character set encoding to use for the external table.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Default:	DEFAULT
Requirement:	To specify this option, you must first set BULKLOAD=YES.
Data source:	Greenplum
See:	BULKLOAD= data set option

Syntax

BL_ENCODING=*character-set-encoding*

Syntax Description

character-set-encoding
specifies the character set encoding to use for the external table. Specify a string constant (such as 'SQL_ASCII'), an integer-encoding number, or DEFAULT to use the default client encoding.

BL_ERRORS= Data Set Option

Specifies whether and when to stop processing a job based on the number of failed records.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Default:	1000
Requirements:	To specify this option, you must first set BULKLOAD=YES.

Enter 0 to disable this option.

Data source: HP Neoview

See: BL_FAILED_DATA= data set option, BULKLOAD= data set option

Syntax

BL_ERRORS=*number-of-failed-records*

Syntax Description

number-of-failed-records
specifies whether and when to stop processing a job. When the number of records in the failed data file for the job reaches the specified number of failed records, job processing stops.

BL_ESCAPE= Data Set Option

Specifies the single character to use for C escape sequences.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: \

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Greenplum

See: BL_FAILED_DATA= data set option, BULKLOAD= data set option

Syntax

BL_ESCAPE='<*any-single-character*>'

Details

Use this option to specify the single character to use for C escape sequences. These can be \n, \t, or \100. It can also be for escape data characters that might otherwise be used as row or column delimiters. Be sure to choose one that is not used anywhere in your actual column data.

Although the default is \ (backslash), you can specify any other character. You can also specify OFF to disable the use of escape characters. This is very useful for Web log data that contains numerous embedded backslashes that are not intended as escape characters.

BL_EXECUTE_CMD= Data Set Option

Specifies the operating system command for segment instances to run.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Restriction: Only for Web tables

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Greenplum

See: BL_EXECUTE_LOCATION= data set option, BL_EXTERNAL_WEB= data set option, BULKLOAD= set option

Syntax

BL_EXECUTE_CMD=*command* | *script*

Syntax Description

command
specifies the operating system command for segment instances to run.

script
specifies a script that contains one or more operating system commands for segment instances to run.

Details

Output is Web table data at the time of access. Web tables that you define with an EXECUTE clause run the specified script or shell command on the specified hosts. By default, all active segment instances on all segment hosts run the command. For example, if each segment host runs four primary segment instances, the command is executed four times per host. You can also limit the number of segment instances that execute the command.

BL_EXECUTE_LOCATION= Data Set Option

Specifies which segment instances runs the given command.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Greenplum

See: BL_EXECUTE_CMD= data set option, BL_EXTERNAL_WEB= data set option, data set option, BULKLOAD= data set option

Syntax

BL_EXECUTE_LOCATION=ALL | MASTER
| HOST [*segment-hostname*], *number-of-segments* | SEGMENT <*segmentID*>

Syntax Description

ALL
specifies that all segment instances run the given command or script.

MASTER
specifies that the master segment instance runs the given command or script.

HOST [*segment-hostname*], *number-of-segments*
indicates that the specified number of segments on the specified host runs the given command or script.

SEGMENT <*segmentID*>
indicates that the specified segment instance runs the given command or script.

Details

For more information about valid values for this option, see the *Greenplum Database Administrator Guide*.

BL_EXCEPTION= Data Set Option

Specifies the exception table into which rows in error are copied.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Default:	none
Requirement:	To specify this option, you must first set BULKLOAD=YES.
Data source:	DB2 under UNIX and PC Hosts, Greenplum
See:	BULKLOAD= data set option, “Capturing Bulk-Load Statistics into Macro Variables”

Syntax

BL_EXCEPTION=*exception-table-name*

Syntax Description

exception table-name
specifies the exception table into which rows in error are copied.

Details

DB2 under UNIX and PC Hosts: Any row that is in violation of a unique index or a primary key index is copied. DATALINK exceptions are also captured in the exception table. If you specify an unqualified table name, the table is qualified with the CURRENT SCHEMA. Information that is written to the exception table is not written to the dump file. In a partitioned database environment, you must define an exception table for those partitions on which the loading table is defined. However, the dump file contains rows that cannot be loaded because they are not valid or contain syntax errors.

Greenplum: Formatting errors are logged when running in single-row, error-isolation mode. You can then examine this error table to determine whether any error rows were not loaded. The specified error table is used if it already exists. Otherwise, it is generated automatically.

For more information about using this option with DB2 under UNIX and PC Hosts, see the FOR EXCEPTION parameter in the *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*. For more information about the load exception table, see the load exception table topics in the *IBM DB2 Universal Database Data Movement Utilities Guide and Reference* and the *IBM DB2 Universal Database SQL Reference, Volume 1*.

BL_EXTERNAL_WEB= Data Set Option

Specifies whether the external data set accesses a dynamic data source.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Greenplum

See: Accessing Dynamic Data in Web Tables, BL_EXECUTE_CMD= data set option, BL_EXECUTE_LOCATION= data set option, BULKLOAD= data set option

Syntax

BL_EXTERNAL_WEB=YES | NO

Syntax Description

YES
specifies that the external data set is not a dynamic data source that resides on the Web.

NO
specifies that the external data set is a dynamic data source that resides on the Web.

Details

The external data set can access a dynamic data source on the Web, or it can run an operating system command or script. For more information about external Web tables, see the *Greenplum Database Administrator Guide*.

Example

```
libname sasflt 'SAS-library';
libname mydblib sasiogpl user=iqusr1 password=iqpwd1 dsn=greenplum;
proc sql;
create table mydblib.flights98
               (bulkload=yes
                bl_external_web='yes'
                bl_execute_cmd='/var/load_scripts/get_flight_data.sh'
                bl_execute_location='HOST'
                bl_format='TEXT'
                bl_delimiter='|')
    as select * from _NULL_;
quit;
libname sasflt 'SAS-library';
libname mydblib sasiogpl user=iqusr1 password=iqpwd1 dsn=greenplum;
proc sql;
create table mydblib.flights98
               (bulkload=yes
                bl_external_web='yes'
                bl_location_protocol='http'
```

```
                bl_datafile='intranet.company.com/expense/sales/file.csv'
                bl_format='CSV')
    as select * from _NULL_;
quit;
```

BL_FAILEDDATA= Data Set Option

Specifies where to put records that could not be written to the database.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: creates a data file in the current directory or with the default file specifications

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.

Data source: HP Neoview

See: BL_ERRORS= data set option, BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_FAILEDDATA=*filename*

Syntax Description

filename
specifies where to put source records that have a valid format but could not be written to the database. For example, a record might fail a data conversion step or violate a uniqueness constraint. These records are in the same format as the source file.

BL_FORCE_NOT_NULL= Data Set Option

Specifies how to process CSV column values.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Aster *n*Cluster, Greenplum

See: BL_DELIMITER= data set option, BL_FORMAT= data set option, BL_NULL= data set option, BL_QUOTE=set option, BULKLOAD= data set option

Syntax

BL_FORCE_NOT_NULL=YES | NO

Syntax Description

YES
specifies that each specified column is processed as if it is enclosed in quotation marks and is therefore not a null value.

NO
specifies that each specified column is processed as if it is a null value.

Details

You can use this option only when BL_FORMAT=CSV. For the default null string, where no value exists between two delimiters, missing values are evaluated as zero-length strings.

BL_FORMAT= Data Set Option

Specifies the format of the external or Web table data.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: TEXT

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Greenplum

See: BL_DELIMITER= data set option, BL_FORCE_NOT_NULL= data set option, set option, BL_QUOTE= data set option, BULKLOAD= data set option

Syntax

BL_FORMAT=TEXT | CSV

Syntax Description

TEXT
specifies plain text format.

CSV
specifies a comma-separated value format.

BL_HEADER= Data Set Option

Indicates whether to skip or load the first record in the input data file.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Restriction: You can use this option only when loading a table using an external Web source.

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Greenplum

See: BULKLOAD= data set option

Syntax

BL_HEADER=YES | NO

Syntax Description

YES
indicates that the first record is skipped (not loaded).

NO
indicates that the first record is loaded.

Details

When the first record of the input data file contains the name of the columns to load, you can indicate that it should be skipped during the load process.

BL_HOST= Data Set Option

Specifies the host name or IP address of the server where the external data file is stored.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: DBMS-specific

Requirements: To specify this option, you must first set BULKLOAD=YES.
You must enclose the name in quotation marks.

Data source: Aster *n*Cluster, Greenplum

See: BL_DBNAME= data set option, BL_PATH= data set option, BULKLOAD= data set option

Syntax

BL_HOST='*hostname*' [Aster *n*Cluster]

BL_HOST='*localhost*' [Greenplum]

Syntax Description

localhost
specifies the IP address of the server where the external data file is stored.

Details

Use this option to pass the IP address to the DBMS bulk-load facility.

Greenplum: The default is 127.0.0.1. You can use the GPLOAD_HOST environment variable to override the default.

BL_HOSTNAME= Data Set Option

Specifies the unqualified host name of the HP Neoview machine.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.

Data source: HP Neoview

See: BL_PORT= data set option, BL_STREAMS= data set option, BULKEXTRACT= LIBNAME option, BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_HOSTNAME=*hostname*

BL_INDEX_OPTIONS= Data Set Option

Lets you specify SQL*Loader Index options with bulk loading.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: SQLLDR_INDEX_OPTION=

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle

See: BULKLOAD= data set option

Syntax

BL_INDEX_OPTIONS=*any valid SQL*Loader Index optionsegment-name*

Syntax Description

any valid SQL*Loader Index option
The value that you specify for this option must be a valid SQL*Loader index option, such as one of those below. Otherwise, an error occurs.

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 lets the SQL*Loader 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, see the Oracle utilities documentation.

Example: Specify SQL*Loader Index Options

This 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

Indicates which scheme the DB2 load utility should use for index maintenance.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Default:	AUTOSELECT
Requirement:	To specify this option, you must first set BULKLOAD=YES.
Data source:	DB2 under UNIX and PC Hosts
See:	BULKLOAD= data set option

Syntax

BL_INDEXING_MODE=AUTOSELECT | REBUILD | INCREMENTAL | DEFERRED

Syntax Description

AUTOSELECT
The load utility automatically decides between REBUILD or INCREMENTAL mode.

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.

Details

For more information about using the values for this option, see the *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

BL_KEEPIDENTITY= Data Set Option

Determines whether the identity column that is created during bulk load is populated with values that Microsoft SQL Server generates or with values that the user provides.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: LIBNAME setting

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

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: OLE DB

See: BL_KEEPIDENTITY= LIBNAME option, BULKLOAD= data set option

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.

BL_KEEPNULLS= Data Set Option

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

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: LIBNAME setting

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

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: OLE DB

See: BL_KEEPNULLS= LIBNAME option, BULKLOAD= data set option

Syntax

BL_KEEPNULLS=YES | NO

Syntax Description

YES
preserves null values that the OLE DB interface inserts.

NO
replaces null values that the OLE DB interface inserts with a default value, as specified in the DEFAULT constraint.

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)

Default: INSERT when loading an empty table; APPEND when loading a table that contains data

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle

See: BULKLOAD= data set option

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 loads new rows from the data file.

TRUNCATE
uses the *SQL truncate* command to achieve the best possible performance. You must first disable the referential integrity constraints of the DBMS table.

Details

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. See your Oracle utilities documentation for information about using the TRUNCATE and REPLACE load methods.

BL_LOAD_REPLACE= Data Set Option

Specifies whether DB2 appends or replaces rows during bulk loading.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under UNIX and PC Hosts

See: BULKLOAD= data set option

Syntax

BL_LOAD_REPLACE=YES | NO

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_LOCATION= Data Set Option

Specifies the location of a file on a Web server for segment hosts to access.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Greenplum

See: BL_EXECUTE_LOCATION= data set option, BL_HOST= data set option, BULKLOAD= data set option

Syntax

BL_LOCATION=http://*file-location*

BL_LOG= Data Set Option

Identifies a log file that contains information for bulk load, such as statistics and errors.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: DBMS-specific

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under UNIX and PC Hosts, Oracle, Teradata

See: BL_LOG= LIBNAME option, BULKLOAD= data set option, DB2 under UNIX and PC Hosts Bulk Loading, Maximizing Teradata Load Performance (Teradata bulk loading)

Syntax

BL_LOG=*path-and-log-filename*

Syntax Description

path-and-log-filename
a file where information about the loading process is written.

Details

See the reference section for your SAS/ACCESS interface for additional details about specifying this option.

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.

DB2 under UNIX and PC Hosts: If BL_LOG= is not specified, the log file is deleted automatically after a successful operation. For more information, see the bulk-load topic in the DB2 under UNIX and PC Hosts bulk loading section.

Oracle: 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 already exist, it is created in the current directory or with the default file specifications. If a log file does already exist, the Oracle bulk loader reuses the file, replacing the contents with information from the new load. On most platforms, the default filename takes the form BL_<*table*>_<*unique-ID*>.log:

table
specifies the table name

unique-ID
specifies 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.

Teradata: For more information, see the bulk-loading topic for Teradata.

BL_METHOD= Data Set Option

Specifies the bulk-load method to use for DB2.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: Before you can use the CLI LOAD interface, you must first set BULKLOAD=YES and then specify this option.

Data source: DB2 under UNIX and PC Hosts

See: BULKLOAD= data set option

Syntax

BL_METHOD=CLILOAD

Syntax Description

CLILOAD
enables the CLI LOAD interface to the LOAD utility.

BL_NULL= Data Set Option

Specifies the string that represents a null value.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: '\N' [TEXT mode], unquoted empty value [CSV mode]

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Greenplum

See: BL_DELIMITER= data set option, BL_FORCE_NOT_NULL= data set option, BL_FORMAT= set option, BL_QUOTE= data set option, BULKLOAD= data set option

Syntax

BL_NULL='\N' | *empty-value*

Details

You might prefer an empty string even in TEXT mode for cases where you do not want to distinguish nulls from empty strings. When you use this option with external and Web tables, any data item that matches this string is considered a null value.

BL_NUM_ROW_SEPS= Data Set Option

Specifies the number of newline characters to use as the row separator for the load or extract data stream.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 1

Requirement: To specify this option, you must first set BULKEXTRACT=YES, BULKLOAD=YES.

Data source: HP Neoview

See: BL_NUM_ROW_SEPS= LIBNAME option, BULKEXTRACT= LIBNAME option, BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_NUM_ROW_SEPS=<*integer*>

Details

You must specify an integer that is greater than 0 for this option.

If your character data contains newline characters and you want to avoid parsing issues, you can specify a greater number for BL_NUM_ROW_SEPS=. This corresponds to the *records separated by* clause in the HP Neoview Transporter control file.

BL_OPTIONS= Data Set Option

Passes options to the DBMS bulk-load facility, which affects how it loads and processes data.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: DBMS-specific

Requirements: To specify this option, you must first set BULKLOAD=YES.

You must separate multiple options with commas and enclose the entire string of options in single quotation marks.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Netezza, OLE DB, Oracle, Sybase IQ

See: BL_OPTIONS= LIBNAME option, BULKLOAD= LIBNAME option, BULKLOAD= data set option

Syntax

BL_OPTIONS='<*option*...,*option*>' [DB2 under UNIX and PC Hosts, OLE DB, Oracle]

BL_OPTIONS='<<*option*> > <<*value*> … '> [Aster *n*Cluster, Netezza, Sybase IQ]

Syntax Description

option
specifies an option from the available options that are specific to each SAS/ACCESS interface. See the details in this section.

Details

You can use BL_OPTIONS= to pass options to the DBMS bulk-load facility when it is called, which affects how data is loaded and processed.

Aster n*Cluster:* By default, no options are specified.

DB2 under UNIX and PC Hosts: 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 *IBM DB2 Universal Database Data Movement Utilities Guide and Reference*.

Netezza: Any text that you enter for this option is appended to the USING clause of the CREATE EXTERNAL TABLE statement—namely, any external_table_options in the *Netezza Database User's Guide*.

OLE DB: By default, no options are specified. This option is valid only when you are using the Microsoft 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; you can use it to improve performance if the file is sorted according to the clustered index on the table. See the Microsoft SQL Server documentation for a complete list of supported bulk copy options.

Oracle: This option lets you specify the SQL*Loader options ERRORS= and LOAD=. The ERRORS= option specifies the number of insert errors that terminates 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. See your Oracle utilities documentation for a complete list of SQL*Loader options that you can specify in BL_OPTIONS=.

Sybase IQ: By default, no options are specified. Any text that you enter for this option is appended to the LOAD TABLE command that the SAS/ACCESS interface uses for the bulk-load process.

Examples

Example 1: Specify the Number of Permitted Errors

In this Oracle example BL_OPTIONS= specifies the number of errors that are permitted during a load of 2,000 rows of data, where all listed options are enclosed in quotation marks.

```
bl_options='ERRORS=999,LOAD=2000'
```

Example 2: Specify External Table Options

This Netezza example shows you how to use BL_OPTIONS= to specify two different external table options, ctrlchars and logdir:

```
data netlib.mdata(bulkload=yes bl_options="ctrlchars true logdir 'c:\temp'");
set saslib.transdata;
run;
```

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)
Default:	none
Requirement:	To specify this option, you must first set BULKLOAD=YES.
Data source:	Oracle
Tip:	The parse file is deleted at the end of SQL*Loader processing.
See:	BULKLOAD= data set option

Syntax

BL_PARFILE=<*parse-file*>

Syntax Description

parse-file

the name that you give the file that contains the SQL*Loader command line options. It can also specify the path. If you do not specify a path, the file is created in the current directory.

Details

This option prompts the SQL*Loader to use the PARFILE= option. This SQL*Loader option lets you specify SQL*Loader command-line options in a file instead of as command-line options. Here is an example of how you can call the SQL*Loader by specifying user ID and control options.

```
sqlldr userid=scott/tiger control=example.ctl
```

You can also call it by using the PARFILE = option.

```
sqlldr parfile=example.par
```

Example.par now contains the USERID= and CONTROL= options. Security is a major advantage of using the BL_PARFILE= option because the user ID and password are stored in a separate file.

Permissions on the file default to operating system defaults. Create the file in a protected directory to prevent unauthorized users from accessing its contents.

To display the contents of the parse file in the SAS log, use the **SASTRACE=",,,d"** option. The password is blocked out and replaced with **xxxx**, however.

Example: Invoke SQL*Loader Using BL_PARFILE=

This example demonstrates how SQL*Loader invocation is different when you specify the BL_PARFILE= option.

```
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 how SQL*Loader is invoked in this
      DATA step, which uses BL_PARFILE=.     */
  sqlldr  parfile=test.par
   /* In this case all options are written to the test.par file.  */
data x.bltst2 ( bulkload=yes bl_parfile='test.par');
 c1=1;
run;
```

BL_PATH= Data Set Option

Specifies the path to use for bulk loading.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirements: To specify this option, you must first set BULKLOAD=YES.
You must enclose the entire path in quotation marks.

Data source: Aster *n*Cluster

See: BL_DBNAME= data set option, BL_HOST= data set option, BULKLOAD= data set option

Syntax

BL_PATH='*path*'

Syntax Description

path
specifies the path to use for bulk loading.

Details

Use this option to pass the path to the DBMS bulk-load facility.

BL_PORT= Data Set Option

Specifies the port number to use.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Alias:	BULKLOAD_PORT= [Hadoop]
Default:	DBMS-specific
Restriction:	*Hadoop:* This option is required only if Hadoop HDFS streaming is running on a port other than the default port of 8020.
Requirement:	To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.
Data source:	Greenplum, HP Neoview
See:	BL_HOSTNAME= data set option, BL_STREAMS= data set option, BULKEXTRACT= LIBNAME option, BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_PORT=<*port*>

Syntax Description

port
specifies the port number to use.

Details

Greenplum: Use this option to specify the port number that bulk load uses to communicate with the server where the input data file resides. There is no default.

HP Neoview: Use this option to specify the port number to which the HP Neoview machine listens for connections. The default is 8080.

BL_PORT_MAX= Data Set Option

Sets the highest available port number for concurrent uploads.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirements: To specify this option, you must first set BULKLOAD=YES.

To reserve a port range, you must specify values for this option and also the BL_PORT_MIN= option.

Data source: DB2 under UNIX and PC Hosts

See: BL_PORT_MIN= data set option, BULKLOAD= data set option

Syntax

BL_PORT_MAX=<*integer*>

Syntax Description

integer

specifies a positive integer that represents the highest available port number for concurrent uploads.

BL_PORT_MIN= Data Set Option

Sets the lowest available port number for concurrent uploads.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirements: To specify this option, you must first set BULKLOAD=YES.

To reserve a port range, you must specify values for both this option and the BL_PORT_MAX= option.

Data source: DB2 under UNIX and PC Hosts

See: BL_PORT_MAX= data set option, BULKLOAD= data set option

Syntax

BL_PORT_MIN=<*integer*>

Syntax Description

integer

specifies a positive integer that represents the lowest available port number for concurrent uploads.

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)

Default: NO

Restriction: This option is not supported on z/OS.

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle

See: BULKLOAD= data set option

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 set this option to YES only for CHAR columns. Do not 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.

BL_PROTOCOL= Data Set Option

Specifies the protocol to use.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: gpfdist

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Greenplum

See: BL_DATAFILE= data set option, BL_HOST= data set option, BULKLOAD= data set option, “Using Protocols to Access External Tables”, “Using the file:// Protocol”

Syntax

BL_PROTOCOL='*gpfdist*' | '*file*' | '*http*'

Syntax Description

gpfdist
specifies the Greenplum file distribution program.

file
specifies external tables on a segment host.

http
specifies Web address of a file on a segment host. This value is valid only for external Web tables.

BL_QUOTE= Data Set Option

Specifies the quotation character for CSV mode.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: " (double quote)

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Aster *n*Cluster, Greenplum

See: BL_DELIMITER= data set option, BL_FORCE_NOT_NULL= data set option, BL_FORMAT= set option, BL_NULL= data set option, BULKLOAD= data set option

Syntax

BL_QUOTE="

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)

Default: DBMS-specific

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under UNIX and PC Hosts, Oracle

See: BULKLOAD= data set option

Syntax

BL_RECOVERABLE=YES | NO

Syntax Description

YES
specifies that the LOAD process is recoverable. For DB2, YES also specifies that BL_COPY_LOCATION= should specify the copy location for the data.

NO
specifies that the LOAD process is not recoverable.

Details

DB2 under UNIX and PC Hosts: The default is NO.

Oracle: The default is YES. Set this option to NO to improve direct load performance. Specifying NO adds the UNRECOVERABLE keyword before the LOAD keyword in the control file.

CAUTION:

Be aware that an unrecoverable load does not log loaded data into the redo log file. Therefore, 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 documentation.

Example: Specify a Load as Unrecoverable

This Oracle example shows how to use BL_RECOVERABLE= to specify that the load is unrecoverable.

```
data x.recover_no (bulkload=yes bl_recoverable=no); c1=1; run;
```

BL_REJECT_LIMIT= Data Set Option

Specifies the reject limit count.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Default:	none
Requirement:	To specify this option, you must first set BULKLOAD=YES and then set BL_REJECT_TYPE=.
Data source:	Greenplum
See:	BL_REJECT_TYPE= data set option, BULKLOAD= data set option

Syntax

BL_REJECT_LIMIT=*number*

Syntax Description

number

specifies the reject limit count either as a percentage (1 to 99) of total rows or as a number of rows.

Details

When BL_REJECT_TYPE=PERCENT, the percentage of rows per segment is calculated based on the Greenplum database configuration parameter (gp_reject_percent_threshold). The default value for this parameter is 300.

Input rows with format errors are discarded if the reject limit count is not reached on any Greenplum segment instance during the load operation.

Constraint errors result when violations occur to such constraints as NOT NULL, CHECK, or UNIQUE. A single constraint error causes the entire external table operation to fail. If the reject limit is not reached, rows without errors are processed and rows with errors are discarded.

BL_REJECT_TYPE= Data Set Option

Indicates whether the reject limit count is a number of rows or a percentage of total rows.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: ROWS

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Greenplum

See: BL_REJECT_LIMIT= data set option, BULKLOAD= data set option

Syntax

BL_REJECT_TYPE=ROWS | PERCENT

Syntax Description

ROWS
specifies the reject limit count as a number of rows.

PERCENT
specifies the reject limit count as a percentage (1 to 99) of total rows.

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)

Default: none

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under UNIX and PC Hosts

See: BL_SERVER_DATAFILE= data set option (specifies the path from the server), BULKLOAD= data set option, DB2 under z/OS Bulk Loading

Syntax

BL_REMOTE_FILE=*pathname-and-base-filename*

Syntax Description

pathname-and-base-filename
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, BL_<*table*>_<*unique-ID*> is the form of the base filename.

table
specifies the table name.

unique-ID
specifies a number that prevents collisions in the event of two or more simultaneous bulk loads of a particular table. The SAS/ACCESS engine generates this number.

Details

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 SAS 6.1 might cause the table space to become unusable in the event of a load error. A load error might affect tables other than the table being loaded.

When you specify this option, the DB2 LOAD command is used instead of the IMPORT command. For more information about these commands, see the bulk-load topic in the DB2 under z/OS section.

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.

BL_RETRIES= Data Set Option

Specifies the number of attempts to make for a job.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 3

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.

Data source: HP Neoview

See: BL_TENACITY= data set option, BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_RETRIES=*number-of-attempts*

Syntax Description

YES
specifies the number of attempts to try to establish a database connection, to open a JMS source, or to open a named pipe for a job.

NO
specifies that job entries in a specific job are processed serially.

BL_RETURN_WARNINGS_AS_ERRORS= Data Set Option

Specifies whether SQL*Loader (bulk-load) warnings should be displayed in SAS through the SYSERR macro as warnings or errors.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle

See: BULKLOAD= data set option

Syntax

BL_RETURN_WARNINGS_AS_ERRORS=YES | NO

Syntax Description

YES
specifies to return all SQLLDER warnings as errors, which SYSERR reflects.

NO
specifies to return all SQLLDER warnings as warnings.

BL_ROWSETSIZE= Data Set Option

Specifies the number of records to exchange with the database.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Restriction: This option is ignored for extraction.

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES. Enter 0 to disable this option.

Data source: HP Neoview

See: BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_ROWSETSIZE=*number-of-records*

Syntax Description

number-of-records
specifies the number of records in each batch of rows to exchange with the database.

Details

The value for this option must be an integer from 1 to 100,000. If you do not specify this option, an optimized value is chosen based on the SQL table or query.

BL_SERVER_DATAFILE= Data Set Option

Specifies the name and location of the data file that the DBMS server instance sees.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: BL_DATAFILE

Default: creates a data file in the current directory or with the default file specifications (same as for BL_DATAFILE=)

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: DB2 under UNIX and PC Hosts, Sybase IQ

See: BL_CLIENT_DATAFILE= data set option [Sybase IQ], BL_DATAFILE= data set option [DB2 for UNIX and PC Hosts], BL_REMOTE_FILE= data set option, BULKLOAD= data set option, "Bulk Loading for DB2 under UNIX and PC Hosts"

Syntax

BL_SERVER_DATAFILE=*path-and-data-filename*

Syntax Description

pathname-and-data-filename

specifies the fully qualified pathname and filename of the data file to load, as seen by the DBMS server instance. By default, the base filename takes the form BL_<*table*>_<*unique-ID*>:

table

specifies the table name.

unique-ID

specifies 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

DB2 under UNIX and PC Hosts: You must also specify a value for BL_REMOTE_FILE=. If the path to the data file from the DB2 server instance is different from 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 that BL_DATAFILE= specifies, this option facilitates use of the DB2 LOAD command. For more information about the LOAD command, see the bulk-load topic in the DB2 under z/OS section. To specify the path from the client, see the BL_DATAFILE= data set option.

Sybase IQ: To specify the path from the client, see the BL_CLIENT_DATAFILE= data set option, which is the client view of the data file.

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)

Default: SQLLDR

Restriction: This option is ignored on z/OS.

Requirement: To specify this option, you must first set BULKLOAD=YES.

Data source: Oracle

See: BULKLOAD= data set option

Syntax

BL_SQLLDR_PATH=*pathname*

Syntax Description

pathname
specifies the full pathname to the SQLLDR executable file so that the SAS/ACCESS Interface for Oracle can call 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.

BL_STREAMS= Data Set Option

Specifies the value for the HP Neoview Transporter parallel streams option.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 4 (for extracts), none (for loads)

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.

Data source: HP Neoview

See: BL_HOSTNAME= data set option, BL_PORT= data set option, BULKEXTRACT= LIBNAME option, BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_STREAMS=<*number*>

Syntax Description

number
specifies the value for the HP Neoview Transporter parallel streams option.

Details

For source data, this option specifies the number of threads to use when reading data and therefore the number of data files or pipes to create. For target data, the value for this option is passed to the HP Neoview Transporter to control the number of internal connections to use in the HP Neoview Transporter.

BL_SUPPRESS_NULLIF= Data Set Option

Indicates whether to suppress the NULLIF clause for the specified columns to increase performance when a table is created.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Requirement: To specify this option, you must first set BULKLOAD=YES. If you specify more than one column name, you must separate the names with spaces.

Data source: Oracle

See: BULKLOAD= data set option

Syntax

BL_SUPPRESS_NULLIF=<_ALL_=YES | NO>
| (<*column-name-1*=YES | NO><…><*column-name-n*=YES | NO>)

Syntax Description

***column-name-1*=YES**
specifies that the NULLIF clause should be suppressed for the specified column in the table.

***column-name-1*=NO**
specifies that the NULLIF clause should not be suppressed for the specified column in the table.

ALL
specifies that the YES or NO value applies to all columns in the table.

Details

This option processes values from left to right. If you specify a column name twice or use the _ALL_ value, the last value overrides the first value that you specified for the column.

CAUTION:
If you set this option to YES and try to insert null values, unpredictable values are inserted into the column.

Examples

Example 1: Suppress NULLIF for Specific Table Columns

In this example, BL_SUPPRESS_NULLIF= in the DATA step suppresses the NULLIF clause for columns C1 and C5 in the table.

```
data x.suppressnullif2_yes (bulkload=yes BL_SUPPRESS_NULLIF=(c1=yes c5=yes));
run;
```

Example 2: Suppress NULLIF for All Table Columns

In this example, BL_SUPPRESS_NULLIF= in the DATA step suppresses the NULLIF clause for all 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_SYNCHRONOUS= Data Set Option

Specifies how to process source file record sets.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: YES

Restriction: This option is ignored for extraction.

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.

Data source: HP Neoview

See: BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_SYNCHRONOUS=YES | NO

Syntax Description

YES
specifies that source file record sets can be processed in a different order for increased performance and parallelism.

NO
specifies that source file record sets are processed serially (in the order in which they appear in the source file).

BL_SYSTEM= Data Set Option

Specifies the unqualified name of the primary segment on an HP Neoview system.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.

Data source: HP Neoview

See: BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_SYSTEM=*unqualified-systemname*

Syntax Description

unqualified-systemname
specifies the unqualified name of the primary segment on an HP Neoview system.

BL_TENACITY= Data Set Option

Specifies how long the HP Neoview Transporter waits before trying again.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 15

Restriction: This option is ignored for extraction.

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES. Enter 0 to disable this option.

Data source: HP Neoview

See: BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_TENACITY=*number-of-seconds*

Syntax Description

number-of-seconds
specifies how long the HP Neoview Transporter waits (in seconds) between attempts to establish a database connection, open a JMS source, or open a named pipe before retrying. The value can be 0 or a positive integer.

BL_TRIGGER= Data Set Option

Specifies whether to enable triggers on a table when loading jobs.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: YES

Restriction: This option is ignored for extraction.

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES. Enter 0 to disable this option.

Data source: HP Neoview

See: BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_TRIGGER=YES | NO

Syntax Description

YES
specifies that triggers on a table are enabled when loading jobs.

NO
specifies that triggers on a table are disabled when loading jobs.

BL_TRUNCATE= Data Set Option

Specifies whether the HP Neoview Transporter truncates target tables (when loading) or target data files (when extracting) before job processing begins.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Requirement: To specify this option, you must first set BULKEXTRACT= YES or BULKLOAD=YES.

Data source: HP Neoview

See: BULKEXTRACT= data set option, BULKLOAD= data set option

Syntax

BL_TRUNCATE=YES | NO

Syntax Description

YES
specifies that the HP Neoview Transporter deletes data from the target before job processing begins.

NO
specifies that the HP Neoview Transporter does not delete data from the target before job processing begins.

BL_USE_PIPE= Data Set Option

Specifies a named pipe for data transfer.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: DBMS-specific

Restriction: Not available for Oracle on z/OS

Requirement: To specify this option, you must first set BULKEXTRACT= YES [HP Neoview], BULKLOAD=YES [HP Neoview, Netezza, Oracle, Sybase IQ], or BULKUNLOAD=YES [Netezza, Oracle].

Data source: HP Neoview, Netezza, Oracle, Sybase IQ

See: BL_DATAFILE= data set option, BULKEXTRACT= LIBNAME option, BULKEXTRACT= data set option, BULKLOAD= data set option, BULKUNLOAD= LIBNAME option, BULKUNLOAD= data set option

Syntax

BL_USE_PIPE=YES | NO

Syntax Description

YES
specifies that a named pipe is used to transfer data between SAS/ACCESS interfaces and the DBMS client interface.

NO
specifies that a flat file is used to transfer data.

Details

By default, the DBMS interface uses a named pipe interface to transfer large amounts of data between SAS and the DBMS when using bulk load or bulk unload. If you prefer to use a flat data file that you can save for later use or examination, specify BL_USE_PIPE=NO.

HP Neoview: The default is YES. This option determines how the sources section of the control file are set up and the method that is used to transfer or receive data from the HP Neoview Transporter. In particular, its setting helps you choose which specific source to select.

Netezza: The default is YES.

Oracle: The default is NO.

Sybase IQ: The default is YES.

BL_WARNING_COUNT= Data Set Option

Specifies the maximum number of row warnings to allow before the load fails.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Default:	2147483646
Requirement:	To specify this option, you must first set BULKLOAD=YES and also specify a value for BL_REMOTE_FILE=.
Data source:	DB2 under UNIX and PC Hosts
See:	BL_REMOTE_FILE= data set option, BULKLOAD= data set option

Syntax

BL_WARNING_COUNT=*warning-count*

Syntax Description

warning-count
specifies the maximum number of row warnings to allow before the load fails.

Details

Use this option to limit the maximum number of rows that generate warnings. See the log file for information about why the rows generated warnings.

BUFFERS= Data Set Option

Specifies the number of shared memory buffers to use for transferring data from SAS to Teradata.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

Default: 2

Data source: Teradata

See: MBUFSIZE= data set option, MULTILOAD= data set option

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 you use the MULTILOAD= data set option, 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. Specify the MBUFSIZE= data set option to vary the size of the shared memory buffers from the size of each data row up to 1MB.

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)

Default: 100

Data source: Sybase

See: BULKLOAD= data set option, ENABLE_BULK= LIBNAME option

Syntax

BULK_BUFFER=*numeric-value*

Syntax Description

numeric-value
specifies 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.

BULKEXTRACT= Data Set Option

Rapidly retrieves (fetches) large number of rows from a data set.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Data source: HP Neoview

See: BULKEXTRACT= LIBNAME option, BL_DATAFILE= data set option, option, BL_DELIMITER= data set option, BL_USE_PIPE= data set option, BULKLOAD= data set option, extracting

Syntax

BULKEXTRACT=YES | NO

Syntax Description

YES
calls the HP Neoview Transporter to retrieve data from HP Neoview.

NO
uses standard HP Neoview result sets to retrieve data from HP Neoview.

Details

Using BULKEXTRACT=YES is the fastest way to retrieve large numbers of rows from an HP Neoview table.

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)

Alias: BL_DB2LDUTIL= [DB2 under z/OS], FASTLOAD= [Teradata]

Default: NO

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

Tip: Using BULKLOAD=YES is the fastest way to insert rows into a DBMS table.

See: BULKEXTRACT= LIBNAME option, BULKEXTRACT= data set option, BULKUNLOAD= LIBNAME option, BULKUNLOAD= data set option, DBCOMMIT= data set option, ENABLE_BULK= LIBNAME option [Sybase], ERRLIMIT= data set option

[DBMS-specific details] Aster nCluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Netezza, ODBC, OLE DB , Oracle, Sybase, Sybase IQ, Teradata

Syntax

BULKLOAD=YES | NO

Syntax Description

YES
calls a DBMS-specific bulk-load facility 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.

BULKUNLOAD= Data Set Option

Rapidly retrieves (fetches) large number of rows from a data set.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Data source: Netezza

See: BULKUNLOAD= LIBNAME option, BL_DATAFILE= data set option, BL_DELIMITER= data set option, BL_DELIMITER= data set option, BL_USE_PIPE= data set option, BULKLOAD= data set option, Bulk Unloading for Netezza

Syntax

BULKUNLOAD=YES | NO

Syntax Description

YES
calls the Netezza Remote External Table interface to retrieve data from the Netezza Performance Server.

NO
uses standard Netezza result sets to retrieve data from the DBMS.

Details

Using BULKUNLOAD=YES is the fastest way to retrieve large numbers of rows from a Netezza table.

CAST= Data Set Option

Specifies whether SAS or the Teradata DBMS server should perform data conversions.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Restriction: CAST= applies only when you are reading Teradata tables into SAS, not when you are writing Teradata tables from SAS. It also applies only 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 in Teradata instead of SAS.

Data source: Teradata

See: CAST= LIBNAME option, CAST_OVERHEAD_MAXPERCENT= LIBNAME option, CAST_OVERHEAD_MAXPERCENT= data set option

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 several 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 points (Teradata FLOAT). SAS/ACCESS can use the Teradata CAST= function to cause Teradata to perform numeric conversions. The parallelism of Teradata makes it suitable for performing this work, particularly if you are running SAS on z/OS, 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.

These Teradata types are candidates for casting:

- INTEGER
- BYTEINT
- SMALLINT
- DECIMAL
- DATE

SAS/ACCESS limits data expansion for CAST= to 20% to trade rapid data conversion by Teradata for extra data transmission. If casting does not exceed a 20% data increase, all candidate columns are cast. If the increase exceeds this limit, 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. With CAST_OVERHEAD_MAXPERCENT=, you can change the 20% 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_OVERHEAD_MAXPERCENT= Data Set Option

Specifies the overhead limit for data conversions to perform in Teradata instead of SAS.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 20

Data source: Teradata

See: CAST= LIBNAME option, CAST= data set option, CAST_OVERHEAD_MAXPERCENT= LIBNAME option

Syntax

CAST_OVERHEAD_MAXPERCENT=<*n*>

Syntax Description

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

Details

Teradata INTEGER, BYTEINT, SMALLINT, and DATE columns require conversion when read in to SAS. Either Teradata or SAS can perform conversions. When Teradata performs the conversion, the row size that is transmitted to SAS using the Teradata CAST operator can increase. CAST_OVERHEAD_MAXPERCENT= limits the allowable increase, also called *conversion overhead*.

For more information about conversions, conversion overhead, and casting, see the CAST= LIBNAME option.

Example: Increase the Allowable Overhead

This example demonstrates the use of CAST_OVERHEAD_MAXPERCENT= to increase the allowable overhead to 40%.

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

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)

Default: LIBNAME setting

Data source: OLE DB

See: COMMAND_TIMEOUT= LIBNAME option

Syntax

COMMAND_TIMEOUT=*number-of-seconds*

Syntax Description

number-of-seconds
an integer greater than or equal to 0, where 0 represents no time-out.

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)

Default: LIBNAME setting

Data source: DB2 under UNIX and PC Hosts, Microsoft SQL Server, ODBC, OLE DB, Sybase IQ

See: COMMAND_TIMEOUT= LIBNAME option, KEYSET_SIZE= data set option [only Microsoft SQL Server and ODBC]

Syntax

CURSOR_TYPE=DYNAMIC | FORWARD_ONLY | KEYSET_DRIVEN | STATIC

Syntax Description

DYNAMIC
specifies that the cursor reflects all 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 under UNIX and PC Hosts, Microsoft SQL Server, and ODBC interfaces. For OLE DB details, see “Details.”

FORWARD_ONLY [not valid for OLE DB]
specifies that the cursor functions like a DYNAMIC cursor except that it supports only sequential fetching of rows.

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.

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

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. See your database documentation for more information.

When no options have been set yet, here are the initial DBMS-specific defaults.

DB2 for UNIX and PC	Microsoft SQL Server	ODBC	OLE DB	Sybase IQ
KEYSET_DRIVEN	DYNAMIC	FORWARD_ONLY	FORWARD_ONLY	DYNAMIC

Here are the operation-specific defaults.

Operation	DB2 for UNIX and PC	Microsoft SQL Server	ODBC	OLE DB	Sybase IQ
insert (UPDATE_ SQL=NO)	KEYSET_ DRIVEN	DYNAMIC	KEYSET_ DRIVEN	FORWARD_ ONLY	
read (such as PROC PRINT)	driver default			driver default (FORWARD_ ONLY)	
update (UPDATE_ SQL=NO)	KEYSET_ DRIVEN	DYNAMIC	KEYSET_ DRIVEN	FORWARD_ ONLY	
CONNECTION=GLOBAL CONNECTION=SHARED		DYNAMIC		DYNAMIC	

OLE DB: Here are the OLE DB properties that are applied to an open rowset. For details, see your OLE DB programmer reference documentation.

CURSOR_TYPE=	OLE DB Properties Applied
FORWARD_ONLY or DYNAMIC (see "Details")	DBPROP_OTHERINSERT=TRUE, DBPROP_OTHERUPDATEDELETE=TRUE

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

DB_ONE_CONNECT_PER_THREAD= Data Set Option

Specifies whether to limit the number of connections to the DBMS server for a threaded Read.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: YES

Data source: Oracle

See: Autopartitioning Scheme for Oracle

Syntax

DB_ONE_CONNECT_PER_THREAD=YES | NO

Syntax Description

YES
enables this option, allowing only one connection per partition.

NO
disables this option.

Details

Use this option if you want to have only one connection per partition. By default, the number of connections is limited to the maximum number of allowed threads. If the value of the maximum number of allowed threads is less than the number of partitions on the table, a single connection reads multiple partitions.

DBCOMMIT= Data Set Option

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

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: CHECKPOINT= [Teradata]

Default: the current LIBNAME setting

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: BULKLOAD= LIBNAME option, BULKLOAD= data set option, DBCOMMIT= LIBNAME option, ERRLIMIT= LIBNAME option, ERRLIMIT= data set option, INSERT_SQL= LIBNAME option, INSERT_SQL= data set option, INSERTBUFF= LIBNAME option, INSERTBUFF= data set option, ML_CHECKPOINT= data set option, , "Using FastLoad "

Syntax

DBCOMMIT=*n*

Syntax Description

n

specifies 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, COMMIT is issued only once—after the procedure or DATA step completes.

If you explicitly set the DBCOMMIT= option, SAS/ACCESS fails any update with a WHERE clause.

If you specify both DBCOMMIT= and ERRLIMIT= and these options collide during processing, COMMIT is issued first and ROLLBACK is issued second. Because COMMIT is issued (through the DBCOMMIT= option) before ROLLBACK (through the ERRLIMIT= option), DBCOMMIT= overrides ERRLIMIT=.

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

Teradata: For the default behavior of this option, see FastLoad description in the Teradata section. DBCOMMIT= and ERRLIMIT= are disabled for MultiLoad to prevent any conflict with ML_CHECKPOINT=.

Example: Specify the Number of Row to Process

A commit is issued after every 10 rows are processed in this example:

```
data oracle.dept(dbcommit=10);
   set myoralib.staff;
run;
```

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)

Default: none

Restrictions: The DBKEY= and DBINDEX= options are ignored when you use DBCONDITION=.

DBCONDITION= is ignored if it specifies ORDER BY and you also use a BY statement.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBINDEX= data set option, DBKEY= data set option

Syntax

DBCONDITION="*DBMS-SQL-query-clause*"

Syntax Description

DBMS-SQL-query-clause
specifies a DBMS-specific SQL query clause, such as WHERE, GROUP BY, HAVING, or ORDER BY.

Details

You can use this option 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.

Example: Return Only Condition-Specific Rows

In this example, the function that is passed to the DBMS with the DBCONDITION= option causes the DBMS to return to SAS only those 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;
```

DBCONSTRAINT= Data Set Option

Provides table-level definitions to specify when a table is created.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: None

Requirement: You must specify constraints within the CREATE TABLE statement parentheses.

Data source: Teradata

See: DBCREATE_TABLE_OPTS= data set option

Syntax

DBCONSTRAINT='DBMS-SQL-clauses'

Syntax Description

DBMS-SQL-clauses
indicates one or more clauses that are specific to Teradata that must be specified when creating a table but that must appear inside the CREATE TABLE parentheses.

Details

Use this option to add table-level definitions in the CREATE TABLE statement. DBCREATE_TABLE_OPTS= is similar to this option except that it lets you add DBMS-specific text outside (to the right) of the parentheses.

Example: Specify Primary Key Columns for a Table

In this example, DBCONSTRAINT= specifies a table-level constraint that columns x and y are primary key columns.

```
libname x teradata user=testuser pw=testpw;

/*
 * Submits this SQL with table-level constraints.
 *
 * CREATE MULTISET TABLE "test"
 * ("x" FLOAT NOT NULL ,
 *  "y" FLOAT NOT NULL ,
 *   CONSTRAINT test PRIMARY KEY(X,Y)
 * );
 */
data x.test(DBCONSTRAINT='CONSTRAINT test PRIMARY KEY(X,Y)' DBNULL=(_ALL_=NO));
x=1;y=1;
run;
```

DBCREATE_TABLE_EXTERNAL= Data Set Option

Specifies whether to include the EXTERNAL keyword when creating a new table.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: DBCREATE_EXTERNAL=,DBCREATE_EXT=

Default: NO

Interaction: You can specify this option, the DBCREATE_TABLE_LOCATION= option, or both.

Data source: Hadoop

Tip: This option determines only the disposition of a file upon delete.

See: DBCREATE_TABLE_EXTERNAL= LIBNAME option, DBCREATE_TABLE_LOCATION= data set option, DBCREATE_TABLE_OPTS= LIBNAME option, DBCREATE_TABLE_OPTS= data set option

Syntax

DBCREATE_TABLE_EXTERNAL=YES | NO

Syntax Description

YES
specifies that the EXTERNAL keyword is to be inserted between the CREATE and TABLE keywords when creating a new table.

NO
specifies that the Hive table and its associated data files are deleted.

Details

Use this option to indicate how to dispose of data files that comprise the Hive table when that table is deleted using a DROP TABLE tabname Hive command. When set to YES and the Hive table is deleted, its associated data files are not deleted.

Example: Creating a File in an Alternative Hive Depository

Both DBCREATE_TABLE_EXTERNAL= and DBCREATE_TABLE_LOCATION= options are set in this example.

```
LIBNAME db HADOOP SERVER=myserver USER=myuser DB=myschema;
DATA db.mytab (
       DBCREATE_TABLE_EXTERNAL=YES
       DBCREATE_TABLE_LOCATION="/mydir/mytab");
  SET mydata;
RUN;
```

DBCREATE_TABLE_LOCATION= Data Set Option

Identifies the HDFS location of the root directory for storing table data.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: DBCREATE_LOCATION=,DBCREATE_LOC=,DBCREATE_PATH=

Default: /user/hive/warehouse/*tabname* [with the default schema], /user/hive/warehouse/*schema*.db/*tabname* [with a nondefault schema]

Interaction: You can specify this option, the DBCREATE_TABLE_EXTERNAL= option, or both.

Data source: Hadoop

Tip: This option determines only the physical location of a file.

See: DBCREATE_TABLE_EXTERNAL= LIBNAME option, DBCREATE_TABLE_EXTERNAL= data set option, DBCREATE_TABLE_OPTS= LIBNAME option, DBCREATE_TABLE_OPTS= data set option

Syntax

DBCREATE_TABLE_LOCATION='*path*'

Syntax Description

'path'
specifies the HDFS location of the root directory for storing table data.

Details

Use this option to specify an alternative HDFS location, which adds the LOCATION keyword to the CREATE TABLE DDL.

Example: Creating a File in an Alternative Hive Depository

Both DBCREATE_TABLE_EXTERNAL= and DBCREATE_TABLE_LOCATION= options are set in this example.

```
LIBNAME db HADOOP SERVER=myserver USER=myuser DB=myschema;
DATA db.mytab (
       DBCREATE_TABLE_EXTERNAL=YES
       DBCREATE_TABLE_LOCATION="/mydir/mytab");
  SET mydata;
RUN;
```

DBCREATE_TABLE_OPTS= Data Set Option

Specifies DBMS-specific syntax to add to the CREATE TABLE statement.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: the current LIBNAME setting

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, Hadoop, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBCREATE_TABLE_EXTERNAL= LIBNAME option, DBCREATE_TABLE_EXTERNAL= data set option, DBCREATE_TABLE_LOCATION= data set option, DBCREATE_TABLE_OPTS= LIBNAME option, DBTYPE= data set option

Syntax

DBCREATE_TABLE_OPTS=*'DBMS-SQL-clauses'*

Syntax Description

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

Details

You can use this option to add DBMS-specific clauses at the end of the SQL CREATE TABLE statement. The SAS/ACCESS engine passes the SQL CREATE TABLE statement and its clauses to the DBMS. The DBMS then 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.

If you are already using the DBTYPE= data set option within an SQL CREATE TABLE statement, you can also use it to include column modifiers.

Example

In this 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;
```

When you use this data set option to create the DB2 table, the SAS/ACCESS interface to DB2 passes this DB2 SQL statement:

```
CREATE TABLE TEMP (X DOUBLE) PARTITIONING
   KEY (X) USING HASHING
```

DBFORCE= Data Set Option

Specifies whether to force data truncation during insert processing.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

Syntax

DBFORCE=YES | NO

Syntax Description

YES
specifies that rows that 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 that 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. DBFORCE= works only when you create a DBMS table with the DBTYPE= data set option—namely, you must specify both DBTYPE= and this option. DBFORCE= does not work for inserts or updates. Therefore, to insert or update a DBMS table, you cannot use the DBFORCE= option—you must instead specify the options that are available with SAS procedures. For example, specify the FORCE= data set option in SAS with PROC APPEND.

FORCE= overrides DBFORCE= when you use FORCE= with PROC APPEND or the PROC SQL UPDATE statement. PROC SQL UPDATE does not warn you before it truncates data.

Example: Truncate Data during Insert Processing

In this example, two librefs are associated with Oracle databases, and it does not specify databases and schemas because it uses the defaults. In the DATA step, MYDBLIB.DEPT is created from the Oracle data that MYORALIB.STAFF references. The LASTNAME variable is a character variable of length 20 in MYORALIB.STAFF. When MYDBLIB.DEPT is created, 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;
```

DBGEN_NAME= Data Set Option

Specifies how SAS automatically renames columns (when they contain characters that SAS does not allow, such as $) to valid SAS variable names.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: DBMS

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBGEN_NAME= LIBNAME option, VALIDVARNAME= system option

Syntax

DBGEN_NAME=DBMS | SAS

Syntax Description

DBMS
specifies that SAS renames DBMS columns to valid SAS variable names. SAS converts any disallowed characters to underscores. If it converts a column to a name that already exists, it appends a sequence number at the end of the new name.

SAS
specifies that SAS converts DBMS columns with disallowed characters into valid SAS variable names. SAS uses the format _COL*n*, where *n* is the column number, starting with 0. If SAS converts a name to a name that already exists, it appends a sequence number at the end of the new name.

Details

SAS retains column names when it reads 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 converting kanji to English characters. English characters that are converted from kanji are often those that SAS does not allow. Although this option works for the single-byte character set (SBCS) version of SAS, SAS ignores it in the double-byte character set (DBCS) version. So if you have the DBCS version, you must first set VALIDVARNAME=ANY before using your language characters as column variables.

In SAS 6, each SAS/ACCESS interface handled name collisions differently: Some appended at the end of the name, some replaced one or more final characters in the name, some used a single sequence number, and still others used unique counters. When you specify VALIDVARNAME=V6, SAS handles name collisions as it did in SAS 6.

Example

If you specify DBGEN_NAME=SAS, SAS renames a DBMS column named **Dept $Amt** to **_COL*n***. If you specify DBGEN_NAME=DBMS, SAS renames the **Dept$Amt** column to **Dept_Amt**.

DBINDEX= Data Set Option

Detects and verifies that indexes exist on a DBMS table.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBINDEX= LIBNAME option, DBKEY= data set option, MULTI_DATASRC_OPT= LIBNAME option

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 SAS/ACCESS finds a usable index, it passes the join WHERE clause to the DBMS for processing. A usable index should have at least the same attributes as the join column.

NO
indicates that no automated index search is performed.

index-name
verifies the index name that is specified for the index columns on the DBMS table. It requires the same type of call as when DBINDEX=YES is used.

Details

If indexes exist on a DBMS table and are of the correct type, you can use this option to potentially improve performance when you are processing a join query that involves a large DBMS table and a relatively small SAS data set that is passed to the DBMS.

CAUTION:

Improper use of this option can impair performance. See “Using the DBINDEX=, DBKEY=, and MULTI_DATASRC_OPT= Options” on page 53 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.

You can enter the DBINDEX= option as a LIBNAME option, SAS data set option, or an option with PROC SQL. Here is the order in which the engine processes it:

1. DATA step or PROC SQL specification.
2. LIBNAME statement specification

Specifying the DBKEY= data set option takes precedence over DBINDEX=.

Examples

Example 1

Here is the SAS data set that is used in these examples.

```
data s1;
   a=1; y='aaaaa'; output;
   a=2; y='bbbbb'; output;
   a=5; y='ccccc'; output;
run;
```

Example 2: Use DBINDEX= in a 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 that 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.

Example 3: Use DBINDEX= in a SAS DATA Step

```
data a;
set 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, a pass-down occurs. Otherwise, the join takes place in SAS.

Example 4: Use 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;
```

DBKEY= Data Set Option

Specifies a key column to optimize DBMS retrieval.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBINDEX= data set option

Syntax

DBKEY=(<'> *column-1*<'> <… <'>*column-n*<'>>)

Syntax Description

column

SAS uses this 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 that the WHERE clause designates. However, if the DBKEY= column does NOT match the key column in the WHERE clause, DBKEY= is not used.

Details

You can use this option to potentially improve performance when you are processing a join that involves a large DBMS table and a small SAS data set or DBMS table.

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. For detailed information about using this option, see the DBINDEX= LIBNAME option on page 53.

Examples

Example 1: Using DBKEY= with MODIFY=

This 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;
```

Example 2: Using More Than One DBKEY= Value

To use more than one value for DBKEY=, you must include the second value as a join on the WHERE clause. In the next example 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;
```

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)

Default: NO

Restriction: This option is valid only for creating DBMS tables.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

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.

Example: Specify a Variable Label

In this 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. By setting DBLABEL=YES, the label can 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 local database to database objects on another server [Oracle]. Specifies a link from your default database to another database on the server to which you are connected [Sybase].

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: LIBNAME setting

Data source: Oracle, Sybase

See: DBLINK= LIBNAME option

Syntax

DBLINK=*database-link*

Details

This option operates differently in each DBMS.

Oracle: 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: You can use this option to link to another database within the same server to which you are connected. If you omit DBLINK=, SAS can access objects only in your default database.

Example: Specify an Oracle Link

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;
```

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)

Default: none

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: MULTI_DATASRC_OPT= LIBNAME option

Syntax

DBMASTER=YES

Syntax Description

YES
designates which of two tables references in a join operation is the larger table.

Details

You can use this option with MULTI_DATASRC_OPT= to specify which table reference in a join is the larger table. This can improve performance by eliminating the processing that is normally performed to determine this information. However, this option is ignored when outer joins are processed.

Example: Join Two Tables

In this 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;
```

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)

Default: 1024

Restriction: This option applies to appending and updating rows in an existing table. It does not apply when creating a table.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ

See: DBMAX_TEXT= LIBNAME option

Syntax

DBMAX_TEXT=*integer*

Syntax Description

integer
is a number between 1 and 32,767.

Details

DBMAX_TEXT= is usually used with a very long DBMS character data type, such as the Sybase TEXT data type or the Oracle CLOB data type.

Oracle: This option applies for CHAR, VARCHAR2, CLOB, and LONG data types.

For SAS 9 or higher, this option applies for CHAR, VARCHAR2, CLOB, LONG, and LOB data types. However, because PROC ACCESS and PROC DBLOAD behavior has not changed since SAS 8, only LONG and LOB data types are valid if you use this option with those procedures.

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)

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: NULLCHAR= data set option, NULLCHARVAL= data set option

Syntax

DBNULL=<_ALL_=YES | NO >
| (<*column-name-1*=YES | NO> <…<*column-name-n*=YES | NO>>)

Syntax Description

ALL [valid only for Informix, Oracle, Sybase, Teradata]
specifies that the YES or NO applies to all columns in the table.

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, you must separate them with spaces.

The DBNULL= option processes values from left to right. If you specify a column name twice or if you use the _ALL_ value, the last value overrides the first value that you specified for the column.

Examples

Example 1: Prevent Specific Columns from Accepting Null Values

In this example, you can use the DBNULL= option to prevent the EMPID and JOBCODE columns in the new MYDBLIB.MYDEPT2 table 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;
```

Example 2: Prevent All Columns from Accepting Null Values

In this 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;
```

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)

Default: LIBNAME setting

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase IQ

See: DBKEY= data set option, DBNULLKEYS= LIBNAME option

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. When you specify DBNULLKEYS=YES and specify a column that the DBKEY= data set option defines as NOT NULL, SAS generates a WHERE clause to find NULL values. For example, if you specify DBKEY=COLUMN and COLUMN is not defined as NOT NULL, SAS generates a WHERE clause with this 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. This syntax has the potential to be much less efficient than the shorter form of the following WHERE clause. 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, you can use DBNULLKEYS=NO. If you specify DBNULLKEYS=NO and DBKEY=COLUMN, SAS generates a shorter form of the WHERE clause, regardless of whether the column that is specified in DBKEY= is defined as NOT NULL.

```
WHERE (COLUMN = ?)
```

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)

Default: NO

Data source: Aster *n*Cluster, Greenplum, HP Neoview, MySQL, Netezza, Oracle, Sybase, Sybase IQ

See: DBPROMPT= LIBNAME option

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:
In the Oracle interface, you can enter 30 characters each for USERNAME and PASSWORD and up to 70 characters for PATH, depending on your platform and terminal type.

Examples

Example 1: Use the Default Value (No Prompt)

In this 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;
```

Example 2: Prompt for Connection Information

In the next example, the DBPROMPT window appears 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;
```

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)

Default: COMPAT

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBSASLABEL= LIBNAME option

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 returns blanks for the column labels.

Details

By default, the SAS/ACCESS interface for your DBMS generates column labels from column names instead of from the real column labels.

You can use this option to override the default behavior. It is useful for when PROC SQL uses column labels as headings instead of column aliases.

Example: Return Blank Labels for Aliases in Headings

This example shows how to use DBSASLABEL= to return blank column labels so that PROC SQL can use the column aliases as the column headings.

```
proc sql;
     select deptno as Department ID, loc as Location
     from mylib.dept(dbsaslabel=none);
```

When DBSASLABEL=NONE, PROC SQL ignores the aliases, and it uses DEPTNO and LOC as column headings 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)

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase IQ, Teradata

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, which can be CHAR(*n*), NUMERIC, DATETIME, DATE, TIME. See the DBMS-specific reference section 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. In that case, SAS prints an error to the log.

Examples

Example 1: Override the Default Data Type

In this example, DBSASTYPE= specifies a data type to use for the MYCOLUMN column when SAS prints ODBC data. SAS can print the values if the data in this DBMS column is stored in a format that SAS does not support, such as SQL_DOUBLE(20).

```
proc print data=mylib.mytable
   (dbsastype=(mycolumn='CHAR(20)'));
run;
```

Example 2: Convert Column Length

In the next example, data that is 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 use only PROC PRINT on the DBMS table, the data might be rounded or are displayed as a missing value. So you could use DBSASTYPE= instead to convert the column so that the length of the character field is 21. The DBMS performs the conversion before the data is brought into SAS, so precision is preserved.

```
proc print data=mylib.specprod
   (dbsastype=(fibersize='CHAR(21)'));
run;
```

Example 3: Append Tables to Match Data Types

The next example uses DBSASTYPE= to append one table to another when the data types cannot be compared. If the EMPID variable in the SAS data set is defined as CHAR(20) and the EMPID column in the DBMS table is 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)

Default: none

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, HP Neoview, Informix, Microsoft SQL Server, ODBC, Oracle, Sybase, Sybase IQ, Teradata

See: DBSLICEPARM= LIBNAME option, DBSLICEPARM= data set option

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

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, this DBSLICE= specification omits rows, creating an*incorrect* result set:

```
DBSLICE=("EMPNUM<1000" "EMPNUM>=1000")
```

Here is a correct form:

```
DBSLICE=("EMPNUM<1000" "EMPNUM>=1000" "EMPNUM IS NULL")
```

In this example, DBSLICE= creates an *incorrect* set by duplicating SALES with a value of 0.

```
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. Use this to obtain the best possible read performance so that your SAS thread can connect directly to the node that contains the data partition that corresponds to your WHERE clause. See the DBMS-specific reference section for your interface for details.

- DB2 under UNIX and PC Hosts on page 443
- ODBC on page 657

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. This partitions the table with the WHERE clauses that 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 usually outperforms 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 starts the FastExport Utility for automatic partitioning. If DBSLICE= overrides this action, WHERE clauses are generated instead. Even with well planned WHERE clauses, performance is degraded because FastExport is considerably 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

Example 1: Partition a Column (Two Threads)

In this example, DBSLICE= partitions on the GENDER column can have only the values **`m`**, **`M`**, **`f`**, and **`F`**. This DBSLICE= clause does 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;
```

Example 2: Partition a Column (Three Threads)

The next 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)

Default: THREADED_APPS,2 [DB2 under z/OS, Oracle, and Teradata] THREADED_APPS,2 or THREADED_APP,3 [DB2 under UNIX and PC Hosts, HP Neoview, Informix, Microsoft SQL Server, ODBC, and Sybase, Sybase IQ]

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, HP Neoview, Informix, Microsoft SQL Server, ODBC, Oracle, Sybase, Sybase IQ, Teradata

See: DBSLICE= data set option, DBSLICEPARM= LIBNAME option, DBSLICEPARM= system option, SLEEP= LIBNAME option, SLEEP= data set option, TENACITY= LIBNAME option, TENACITY= data set option

Syntax

DBSLICEPARM=NONE | THREADED_APPS | ALL

DBSLICEPARM=(NONE | THREADED_APPS | ALL<*max-threads*>)

DBSLICEPARM=(NONE | THREADED_APPS | ALL<, *max-threads*>)

Syntax Description

NONE

disables DBMS threaded Reads. SAS reads tables on a single DBMS connection, as it did with SAS 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. It includes SAS threaded applications, the SAS DATA step, and numerous SAS procedures.

max-threads
a positive integer value that specifies the maximum number of connections per table read. The second parameter of the option determines the number of threads to read the table in parallel. The number of partitions on the table determine the number of connections made to the Oracle server for retrieving rows from the table. A partition or portion of the data is read on each connection. The combined rows across all partitions are the same regardless 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. See the DBMS-specific reference section about threaded Reads for your interface before using this parameter.

Details

You can use DBSLICEPARM= in numerous locations. The usual rules of option precedence apply: A table option has the highest precedence, followed by 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 determine whether threaded Reads are actually generated, turn on SAS tracing and run a program, as shown in this 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.

DB2 under UNIX and PC Hosts, HP Neoview, Informix, Microsoft SQL Server, ODBC, Sybase, Sybase IQ:
The default thread number depends on whether an application passes in the number of threads (CPUCOUNT=) and whether the data type of the column that was selected for purposes of data partitioning is binary.

Examples

Example 1: Disable Threaded Reads for All SAS Users

Here is how to use DBSLICEPARM= in a SAS configuration file entry in Windows to turn off threaded Reads for all SAS users.

```
-dbsliceparm NONE
```

Example 2: Enable Threaded Reads for Read-Only References

Here is how you can use DBSLICEPARM= as a z/OS invocation option to turn on threaded Reads for read-only references to DBMS tables throughout a SAS job.

```
sas o(dbsliceparm=ALL)
```

Example 3: Increase Maximum Threads (as a SAS Global Option)

In this example, you can 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 applications.

```
option dbsliceparm=(threaded_apps,3);
```

Example 4: Enable Threaded Reads for References Using a Particular Libref

You can use DBSLICEPARM= as a LIBNAME option to turn on threaded Reads for read-only table references that use this particular libref, as shown in this example

```
libname dblib oracle user=scott password=tiger dbsliceparm=ALL;
```

Example 5: Enable Threaded Reads as a Table-Level Option

Here is 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;
```

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)

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: “DBCREATE_TABLE_OPTS= Data Set Option”, “DBFORCE= Data Set Option”, “DBNULL= Data Set Option”

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 DBMS-specific reference section 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 it outputs 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.

You can also use this option to specify column modifiers. The allowable syntax for these modifiers is generally DBMS-specific. For more information, see the SQL reference for your database.

MySQL:
All text strings are passed as is to the MySQL server. MySQL truncates text strings to fit the maximum length of the field without generating an error message.

Teradata:
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

Example 1: Specify Data Types for Columns

In this example, DBTYPE= specifies the data types to use when you create columns in the DBMS table.

```
data mydblib.newdept(dbtype=(deptno='number(10,2)' city='char(25)'));
   set mydblib.dept;
run;
```

Example 2: Specify Data Types for Columns in a New Table

This 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;
```

Example 3: Specify a Data Type for a Column in a New Table

This 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 single quotation marks that the Teradata FORMAT attribute requires. 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;
```

Example 4: Create a Primary Key for a New Table

Where x indicates the Oracle engine, this example creates a new table, ALLACCTX, and uses DBTYPE= to create the primary key, ALLACCT_PK.

```
data x.ALLACCTX  ( dbtype=(
SourceSystem  = 'varchar(4)'
acctnum  = 'numeric(18,5) CONSTRAINT "ALLACCT_PK" PRIMARY KEY'
accttype  = 'numeric(18,5)'
balance  = 'numeric(18,5)'
clientid  = 'numeric(18,5)'
closedate  = 'date'
opendate  = 'date'
primary_cd  = 'numeric(18,5)'
status  = 'varchar(1)'
)  );
set work.ALLACCT ;
format CLOSEDATE date9.;
format OPENDATE  date9.;
run;
```

The code generates this CREATE TABLE statement.

Output 11.1 *Output from a CREATE TABLE Statement That Uses DBTYPE= to Specify a Column Modifier*

```
CREATE TABLE ALLACCTX(SourceSystem varchar(4),
cctnum numeric(18,5) CONSTRAINT "ALLACCT_PK" PRIMARY KEY,
ccttype numeric(18,5),balance numeric(18,5),clientid numeric(18,5),
losedate date,opendate date,primary_cd numeric(18,5),status varchar(1))
```

DEGREE= Data Set Option

Determines whether DB2 uses parallelism.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: ANY

Data source: DB2 under z/OS

See: DEGREE= LIBNAME option

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, although this is perhaps slower, it uses fewer system resources.

DIMENSION= Data Set Option

Specifies whether the database creates dimension tables or fact tables.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Data source: Aster *n*Cluster

See: DIMENSION= LIBNAME option, PARTITION_KEY= LIBNAME option, PARTITION_KEY= data set option

Syntax

DIMENSION=YES | NO

Syntax Description

YES
specifies that the database creates dimension tables.

NO
specifies that the database creates fact tables.

DISTRIBUTE_ON= Data Set Option

Specifies a column name to use in the DISTRIBUTE ON clause of the CREATE TABLE statement.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: DISTRIBUTE= [Netezza]

Default: none

Data source: Aster *n*Cluster, Netezza

Syntax

DISTRIBUTE_ON='*column-1* <…,*column-n*>' | RANDOM

Syntax Description

column-name
specifies a DBMS column name.

RANDOM
specifies that data is distributed evenly. For Netezza, the Netezza Performance Server does this across all SPUs. This is known as *round-robin distribution*.

Details

You can use this option to specify a column name to use in the DISTRIBUTE ON= clause of the CREATE TABLE statement. Each table in the database must have a distribution key that consists of one to four columns. If you do not specify this option, the DBMS selects a distribution key.

Examples

Example 1: Create a Distribution Key on a Single Column

```
proc sql;
create table netlib.customtab(DISTRIBUTE_ON='partno')
   as select partno, customer, orderdat from saslib.orders;
quit;
```

Example 2: Create a Distribution Key on Many Columns

For more than one column, separate the columns with commas.

```
data netlib.mytab(DISTRIBUTE_ON='col1,col2');
col1=1;col2=12345;col4='mytest';col5=98.45;
run;
```

Example 3: Use the RANDOM Keyword

```
data netlib.foo(distribute_on=random);
mycol1=1;mycol2='test';
run;
```

DISTRIBUTED_BY= Data Set Option

Uses one or multiple columns to distribute table rows across database segments.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: RANDOMLY DISTRIBUTED

Data source: Greenplum

Syntax

DISTRIBUTED_BY='*column-1* <…,*column-n*>' | DISTRIBUTED RANDOMLY

Syntax Description

column-name
specifies a DBMS column name.

DISTRIBUTED RANDOMLY
determines the column or set of columns that the Greenplum database uses to distribute table rows across database segments. This is known as round-robin distribution.

Details

For uniform distribution—namely, so that table records are stored evenly across segments (machines) that are part of the database configuration—the distribution key should be as unique as possible.

Example: Create a Table By Specifying a Distribution Key

```
libname x sasiogpl user=myuser password=mypwd dsn=Greenplum;
data x.sales (dbtype=(id=int qty=int amt=int) distributed_by='distributed by (id)');
          id = 1;
          qty = 100;
          sales_date = '27Aug2009'd;
          amt = 20000;
run;
```

It creates the SALES table.

```
CREATE TABLE SALES
(id int,
 qty int,
 sales_date double precision,
 amt int
) distributed by (id)
```

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)

Default: 1

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: ERRLIMIT= LIBNAME option, DBCOMMIT= LIBNAME option, DBCOMMIT= data set option, ML_CHECKPOINT= data set option

Syntax

ERRLIMIT=*integer*

Syntax Description

integer
specifies a positive integer that represents the number of errors after which SAS stops processing and issues a rollback.

Details

SAS ends the step abnormally and calls the DBMS to issue a rollback after a specified number of errors while processing inserts, deletes, updates, and appends. If ERRLIMIT=0, SAS processes all rows no matter how many errors occur. The SAS log displays the total number of rows that SAS processed and the number of failed rows, if applicable.

If the step ends abnormally, any rows that SAS successfully processed after the last commit are rolled back and are therefore lost. Unless DBCOMMIT=1, it is very likely that rows will be lost. The default value is 1000.

Note: A significant performance impact can result if you use this option from a SAS client session in SAS/SHARE or SAS/CONNECT environments to create or populate a newly created table. To prevent this, use the default setting, ERRLIMIT=1.

Teradata:
DBCOMMIT= and ERRLIMIT= are disabled for MultiLoad to prevent any conflict with ML_CHECKPOINT=.

Example: Specify the Number of Allowable Errors

In this 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;
```

ESCAPE_BACKSLASH= Data Set Option

Specifies whether backslashes in literals are preserved during data copy from a SAS data set to a table.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Data source: MySQL

See: ESCAPE_BACKSLASH= LIBNAME option

Syntax

ESCAPE_BACKSLASH=YES | NO

Syntax Description

YES
specifies that an additional backslash is inserted in every literal value that already contains a backslash.

NO
specifies that backslashes that exist in literal values are not preserved. An error results.

Details

MySQL uses the backslash as an escape character. When data that is copied from a SAS data set to a MySQL table contains backslashes in literal values, the MySQL interface can preserve them if ESCAPE_BACKSLASH=YES.

Examples

Example 1: Preserve Backslashes

In this example, SAS preserves the backslashes for **x** and **y** values.

```
libname out mysql user=dbitest pw=dbigrp1
        server=striper database=test port=3306;
data work.test;
   length x y z $10;
   x = "ABC";
   y = "DEF\";
   z = 'GHI\';
run;
data out.test(escape_backslash=yes);
set work.test;
run;
```

The code successfully generates this INSERT statement.

```
INSERT INTO 'test' ('x','y','z')  VALUES ('ABC','DEF\\','GHI\\')
```

Example 2: Use the Default Value (Do Not Preserve Backslashes)

For the prior example, here is the error that is displayed if ESCAPE_BACKSLASH=NO.

```
ERROR: Execute error: You have an error in your SQL syntax;
check the manual that corresponds to your MySQL server version for the
right syntax to use near 'GHI\')' at line 1
```

FETCH_IDENTITY= Data Set Option

Returns the value of the last inserted identity value.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Data source: DB2 under UNIX and PC Hosts

See: FETCH_IDENTITY= LIBNAME option

Syntax

FETCH_IDENTITY=YES | NO

Syntax Description

YES
returns the value of the last inserted identity value.

NO
disables this option.

Details

You can use this option instead of issuing a separate SELECT statement after an INSERT statement. If FETCH_IDENTITY=YES and the INSERT that is executed is a single-row INSERT, the engine calls the DB/2 identity_val_local() function and places the results into the SYSDB2_LAST_IDENTITY macro variable. Because the DB2 engine default is multirow inserts, you must set INSERTBUFF=1 to force a single-row INSERT.

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)

Default: NO

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, OLE DB, Sybase IQ

See: IGNORE_READ_ONLY_COLUMNS= LIBNAME option

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 the Microsoft SQL Server timestamp data type. Several databases also have properties that allow certain data types to be read-only, such as the Microsoft SQL Server identity property.

When IGNORE_READ_ONLY_COLUMNS=NO (the default) and a DBMS table contains a column that is read-only, an error is returned that the data could not be modified for that column.

Example: Insert Data into a Table

For this 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;
```

When IGNORE_READ_ONLY_COLUMNS=NO (the default), the database returns an error because the ID column cannot be updated. However, if you set the option to YES and execute a PROC APPEND, the append succeeds and the generated SQL statement 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;
```

IN= Data Set Option

Lets you specify the database or table space in which you want to create a new table.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: TABLESPACE=

Default: LIBNAME setting

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS

See: IN= LIBNAME option

Syntax

IN='*database-name.tablespace-name*'|'DATABASE *database-name*'

Syntax Description

database-name.tablespace-name
specifies the names of the database and table space, which are separated by a period.

DATABASE *database-name*
specifies only the database name. In this case, you specify the word DATABASE, 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 table space.

INSERT_SQL= Data Set Option

Determines the method to use to insert rows into a data source.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: LIBNAME setting

Data source: Microsoft SQL Server, ODBC, OLE DB

See: INSERTBUFF= data set option, INSERT_SQL= LIBNAME option

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 have generally improved insert performance when INSERT_SQL=NO. Other databases might have inferior insert performance or might fail with this setting. Therefore, you should experiment to determine the optimal setting for your situation.

Microsoft SQL Server:
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 your ODBC driver supports it.

ODBC:
The default for ODBC 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 your ODBC driver supports it.

OLE DB:
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.

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)

Default: LIBNAME setting

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase IQ

See: DBCOMMIT= LIBNAME option, DBCOMMIT= data set option, INSERTBUFF= LIBNAME option, INSERT_SQL LIBNAME option, INSERT_SQL data set option

Syntax

INSERTBUFF=*positive-integer*

Syntax Description

positive-integer
specifies the number of rows to insert. SAS allows the maximum that the DBMS allows.

Details

SAS allows the maximum number of rows that the DBMS allows. 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.

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 set the DBCOMMIT= option with a value that is less than the value of INSERTBUFF=, then DBCOMMIT= overrides INSERTBUFF=.

When you insert rows with the VIEWTABLE window or the FSEDIT or FSVIEW procedure, use INSERTBUFF=1 to prevent the engine from trying to insert multiple rows. These features do not support inserting more than one row at a time.

Additional driver-specific restrictions might apply.

DB2 under UNIX and PC Hosts:
To use this option, you must first set INSERT_SQL=YES. If one row in the insert buffer fails, all rows in the insert buffer fail. The default is calculated based on the row length of your data.

HP Neoview, Netezza:
The default is automatically calculated based on row length.

Microsoft SQL Server, Greenplum:
To use this option, you must set INSERT_SQL=YES.

MySQL:
The default is 0. Values greater than 0 activate the INSERTBUFF= option, and the engine calculates how many rows it can insert at one time, based on row size. If one

row in the insert buffer fails, all rows in the insert buffer might fail, depending on your storage type.

ODBC:
The default is 1.

OLE DB:
The default is 1.

Oracle:
When REREAD_EXPOSURE=YES, the (forced) default value is 1. Otherwise, the default is 10.

KEYSET_SIZE= Data Set Option

Specifies the number of rows in the cursor that the keyset drives.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Default:	LIBNAME setting
Requirement:	This option is valid only when CURSOR_TYPE=KEYSET_DRIVEN.
Data source:	Microsoft SQL Server, ODBC
See:	KEYSET_SIZE= LIBNAME option

Syntax

KEYSET_SIZE=*number-of-rows*

Syntax Description

number-of-rows
specifies a positive integer from 0 through the number of rows in the cursor.

Details

If KEYSET_SIZE=0, the entire cursor is keyset-driven.

If you specify a value greater than 0, that value indicates the number of rows within the cursor that function as a keyset-driven cursor. When you scroll beyond the bounds that KEYSET_SIZE= specifies, the cursor becomes dynamic and new rows might be included in the cursor. This results in a new keyset, where the cursor functions as a keyset-driven cursor again.

When you specify a value between 1 and the number of rows in the cursor, the cursor is considered to be a mixed cursor. Part of the cursor functions as a keyset-driven cursor, and another part of the cursor functions as a dynamic cursor.

LOCATION= Data Set Option

Lets you further specify exactly where a table resides.

Valid in:	DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)
Alias:	LOC=

Default: LIBNAME setting

Requirement: If you specify LOCATION=, you must also specify the AUTHID= data set option.

Data source: DB2 under z/OS

See: AUTHID= data set option, LOCATION= LIBNAME option

Syntax

LOCATION=*location-name*

Details

The location name maps to the location in the SYSIBM.LOCATIONS catalog in the communication database.

In SAS/ACCESS Interface to DB2 under z/OS, the location is converted to the first level of a three-level table name: *location-name*.AUTHID.TABLE. The DB2 Distributed Data Facility (DDF) makes the connection implicitly to the remote DB2 subsystem when DB2 receives a three-level name in an SQL statement.

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)

Default: LIBNAME setting

Restriction: If you omit LOCKTABLE=, no locking occurs.

Requirement: You can lock tables only if you are the owner or have been granted the necessary privilege.

Data source: Informix

See: LOCKTABLE= LIBNAME option

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. It allows other users or processes to read data from the tables but preventing users from updating data.

MBUFSIZE= Data Set Option

Specifies the size of the shared memory buffers to use for transferring data from SAS to Teradata.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

Default: 64K

Requirement: To specify this option, you must first set MULTILOAD=YES.

Data source: Teradata

Tip: Use BUFFERS= to vary the number of buffers for data transfer from 1 to 8. Use MBUFSIZE= to vary the size of the shared memory buffers from the size of each data row up to 1MB.

See: BUFFERS= data set option, MULTILOAD= data set option "Using MultiLoad"

Syntax

MBUFSIZE=*size-of-shared-memory-buffers*

Syntax Description

size-of-shared-memory-buffers
specifies a numeric value (between the size of a row being loaded and 1MB) that specifies the buffer size.

Details

This option 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 you use MULTILOAD=, data transfers from SAS to Teradata using shared memory segments. The default shared memory buffer size is 64K. The default number of shared memory buffers that are used for the transfer is 2.

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)

Default: none (see "Details")

Requirement: To specify this option, you must first set MULTILOAD=YES.

Data source: Teradata

See: DBCOMMIT= LIBNAME option, DBCOMMIT= data set option, MULTILOAD= data set option [using checkpoints, restarting MultiLoad jobs], "Using MultiLoad"

Syntax

ML_CHECKPOINT=*checkpoint-rate*

Syntax Description

checkpoint-rate
a numeric value that specifies the interval between checkpoint operations in minutes.

Details

If you do not specify a value for ML_CHECKPOINT=, the Teradata Multiload default of 15 applies. If ML_CHECKPOINT= is between 1 and 59 inclusive, checkpoints are taken at the specified intervals, in minutes. If ML_CHECKPOINT= is greater than or equal to 60, a checkpoint occurs after a multiple of the specified rows is loaded.

ML_CHECKPOINT= functions very similarly to CHECKPOINT in the native Teradata MultiLoad utility. However, it differs from DBCOMMIT=. Both DBCOMMIT= and its alias, CHECKPOINT, are disabled to prevent any conflict.

See the Teradata documentation on the MultiLoad utility for more information about using MultiLoad checkpoints.

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)

Default: none

Restriction: Do not use ML_ERROR1 with the ML_LOG= data set option. ML_LOG= provides a common prefix for all temporary tables that the Teradata MultiLoad utility uses.

Requirements: To specify this option, you must first set MULTILOAD=YES.

When you restart a failed MultiLoad job, you must specify the same acquisition table from the earlier run so that the MultiLoad job can restart correctly. Using ML_RESTART=,ML_ERROR2=, and ML_WORK=, you must also specify the same log table, application error table, and work table upon restarting.

Data source: Teradata

See: ML_LOG= data set option, ML_ERROR2= data set option, ML_RESTART= data set option, ML_WORK= data set option, MULTILOAD= data set option, “Using MultiLoad”

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

Use this option to specify the name of a table to use for tracking errors that were generated during the acquisition phase of the MultiLoad bulk-load operation. By default, the acquisition error table is named SAS_ML_ET_*randnum*, where *randnum* is a random number.

For more information about temporary table names that the Teradata MultiLoad utility uses and what is stored in each, see the Teradata MultiLoad reference.

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)

Default: none

Restriction: Do not use ML_ERROR2 with ML_LOG=, which provides a common prefix for all temporary tables that the Teradata MultiLoad utility uses.

Requirements: To specify this option, you must first set MULTILOAD=YES.

When you restart a failed MultiLoad job, you must specify the same application table from the earlier run so that the MultiLoad job can restart correctly. Using ML_RESTART=,ML_ERROR1=, and ML_WORK=, you must also specify the same log table, acquisition error table, and work table upon restarting.

Data source: Teradata

See: ML_ERROR1= data set option, ML_LOG= data set option, , ML_RESTART= data set option, ML_WORK= data set option, MULTILOAD= data set option, "Using MultiLoad"

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

Use this option to specify the name of a table to use for tracking errors that were generated during the application phase of the MultiLoad bulk-load operation. By default, the application error table is named SAS_ML_UT_*randnum*, where *randnum* is a random number.

For more information about temporary table names that the Teradata MultiLoad utility uses and what is stored in each, see the Teradata MultiLoad reference.

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)

Default: none

Restriction: Do not use ML_LOG= with ML_RESTART=,ML_ERROR1=,ML_ERROR2=, or ML_WORK= because ML_LOG= provide specific names to the temporary files.

Requirement: To specify this option, you must first set MULTILOAD=YES.

Data source: Teradata

See: ML_ERROR1= data set option, ML_ERROR2= data set option, ML_RESTART= data set option, ML_RESTART= data set option, ML_WORK= data set option, MULTILOAD= data set option, "Using MultiLoad"

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

You can use this option to specify a prefix for the temporary table names that the MultiLoad utility uses during the load process. The MultiLoad utility uses a log table, two error tables, and a work table while loading data to the target table. By default, here are the names for these tables, where *randnum* is a random number.

Temporary Table	Table Name
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*

To override the default names, here are the table names that would be generated if **ML_LOG=MY_LOAD**, for example.

Temporary Table	Table Name
Restart table	MY_LOAD_RS
Acquisition error table	MY_LOAD_ET
Application error table	MY_LOAD_UT
Work table	MY_LOAD_WT

SAS/ACCESS automatically deletes the error tables if no errors are logged. If there are errors, the tables are retained, and SAS/ACCESS issues a warning message that includes the names of the tables in error.

For more information about temporary table names that the Teradata MultiLoad utility uses and what is stored in each, see the Teradata MultiLoad reference.

ML_RESTART= Data Set Option

Specifies the name of a temporary table that MultiLoad uses to track checkpoint information.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

Default: none

Restriction: Do not use ML_RESTART= with ML_LOG=, which provides a common prefix for all temporary tables that the Teradata MultiLoad utility uses.

Requirements: To specify this option, you must first set MULTILOAD=YES.

When you restart a failed MultiLoad job, you must specify the same application table from the earlier run so that the MultiLoad job can restart correctly. Using ML_RESTART=,ML_ERROR1=, and ML_WORK=, you must also specify the same log table, acquisition error table, and work table upon restarting.

Data source: Teradata

See: ML_ERROR1= data set option, ML_ERROR2 data set option, ML_LOG= data set option, ML_WORK= data set option, MULTILOAD= data set option, “Using MultiLoad”

Syntax

ML_RESTART=*temporary-table-name*

Syntax Description

temporary-table-name
specifies the name of the temporary table that the Teradata MultiLoad utility uses to track checkpoint information.

Details

Use this option 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 you used for tracking checkpoint information in the earlier failed run.

For more information about the temporary table names that the Teradata MultiLoad utility uses, see the Teradata documentation on the MultiLoad utility.

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)

Default: none

Restriction: Do not use ML_WORK= with ML_LOG=, which provides a common prefix for all temporary tables that the Teradata MultiLoad utility uses.

Requirement: To specify this option, you must first set MULTILOAD=YES.

Data source: Teradata

See: ML_ERROR1= data set option, ML_ERROR2 data set option, ML_LOG= data set option, ML_RESTART= data set option, MULTILOAD= data set option, "Using MultiLoad"

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 the MultiLoad utility receives during a bulk-load operation.

Details

Use this option to specify the name of the table to use for tracking intermediate data that the MultiLoad utility received during a bulk-load operation. When you restart the job, use ML_WORK= to specify the name of the table for tracking intermediate data during a previously failed MultiLoad job.

For more information about temporary table names that the MultiLoad utility uses and what is stored in each, see the Teradata MultiLoad reference.

MULTILOAD= Data Set Option

Specifies whether Teradata Insert and Append operations should use the Teradata MultiLoad utility.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

Default: NO

Restriction: You cannot use ML_LOG with any of these options: ML_ERROR1,ML_ERROR2,ML_WORK, and ML_RESTART.

Data source: Teradata

See: BUFFERS= data set option, BULKLOAD= LIBNAME option, BULKLOAD= data set option, DBCOMMIT= LIBNAME option, DBCOMMIT= data set option, FASTEXPORT= LIBNAME option, "Maximizing Teradata Load Performance ", MBUFSIZE= data set option, ML_CHECKPOINT= data set option, ML_ERROR1= data set option, ML_ERROR2= data set option, ML_LOG= data set option, ML_RESTART= data set option, ML_WORK= data set option, QUERY_BAND= LIBNAME option, QUERY_BAND= data set option, SLEEP= data set option, , TENACITY= data set option, "Using MultiLoad"

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.

To determine whether threaded Reads are actually generated, turn on SAS tracing by setting OPTIONS SASTRACE=",,,d" in your program.

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. You can use BUFFERS= to vary the number of buffers for data transfer from 1 to 8. You can use MBUFSIZE= 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 it performs the bulk-load operation. It 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, where *randnum* represents a random number. To specify a different name for these tables, use ML_RESTART=, ML_ERROR1=, ML_ERROR2=, and ML_WORK=, respectively.

Temporary Table	Table Name
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*

You can use ML_LOG= to specify a prefix for the temporary table names that MultiLoad uses.

Here is the order that is used for naming the temporary tables that MultiLoad uses:

1. If you set ML_LOG=, the prefix that you specified is used when naming temporary tables for MultiLoad.
2. If you do not specify ML_LOG=, the values that you specified for ML_ERROR1, ML_ERROR2, ML_WORK, ML_RESTART are used.
3. If you do not specify any table naming options, temporary table names are generated by default.

Restarting MultiLoad

The MultiLoad bulk-load operation (or MultiLoad job) works in phases. The first is the *acquisition phase*, during which data is transferred from SAS to Teradata work tables. The second is 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 —when data is loaded onto the target tables from the work table—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.

You can use ML_CHECKPOINT= to specify the checkpoint rate. Specify a value for ML_CHECKPOINT= if you want restart capability. 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, and no checkpoints are recoded if you use the default.

If ML_CHECKPOINT is between 1 and 59 inclusive, checkpoints are recorded at the specified interval in minutes. If ML_CHECKPOINT is greater than or equal to 60, a checkpoint occurs after a multiple of the specified rows are loaded.

ML_CHECKPOINT= functions very much like the Teradata MultiLoad utility checkpoint, but it differs from the DBCOMMIT= data set option.

These restrictions apply when you restart a failed MultiLoad job.

- The failed MultiLoad job must have specified a checkpoint rate other than 0 using the ML_CHECKPOINT= data set option. Otherwise, restarting begins from the first record of the source data.

  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, this 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 because the target table already exists. It is also necessary to identify the work tables, restart table, and the error tables used in the original job.

For example, assume that the DATA step shown above failed with this 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 data set 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,
use the value (7278+n-1) for FIRSTOBS in the restart.  Otherwise,
use FIRSTOBS=7278.  Sometimes the FIRSTOBS value that is used
on the restart can be an earlier position than the last checkpoint
because restart is block-oriented and not record-oriented.
```

After you fix the error, you must restart the job as an append process. You must also specify the same work, error, and restart tables as you used in the earlier run. You use a FIRSTOBS= value 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 you used ML_LOG= in the run that failed, you can specify the same value for ML_LOG= on restart. Therefore, you need not specify four data set options to identify the temporary tables that MultiLoad uses.

  For example, assume that this is how the original run used ML_LOG=:

```
data trlib.mlfloat(MultiLoad=yes ML_CHECKPOINT=1000 ML_LOG=MY_ERRORS);
  set work.testdata;
run;
```

  If this DATA step fails with this error, the restart capability needs only ML_LOG= to identify all 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 data set 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, use
the value (7278+n-1) for FIRSTOBS in the restart.  Otherwise use
FIRSTOBS=7278.  Sometimes the FIRSTOBS value used on the restart can 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. You must restart a MultiLoad job outside of SAS using the script that is written to the SAS log. See your Teradata documentation on the MultiLoad utility for instructions on how to run MultiLoad scripts. Here is an example of a script that is written in the SAS log.

```
=-=-= 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 restart script 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 from the SAS log error message to calculate the value for FIRSTOBS= upon restart. These examples show how to do this.

```
/* 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;
```

Assume 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, use the value (7278+n-1) for FIRSTOBS in the restart.
Otherwise use FIRSTOBS=7278.  Sometimes the FIRSTOBS value specified on the
restart can 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;
```

Please keep these considerations in mind.

- 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 that is 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. You do this because the data that is needed to complete the load has already been transferred.
- The restart capability in MultiLoad is block-oriented, not record-oriented. For example, if a checkpoint was taken at 5000 records, you might need to restart from an earlier record, such as record 4000. You do this because the block of data containing record 5001 might have started at record 4000. The exact record where restart should occur is displayed in the SAS log.

Examples

Example 1: Load SAS Data to an Alternate Database

This example uses MultiLoad to load SAS data to an alternate database. 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;
```

Example 2: Extract Data from One Table to Another

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;
```

MULTISTMT= Data Set Option

Specifies whether insert statements are sent to Teradata one at a time or in a group.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Restriction: You cannot currently use MULTISTMT= with the ERRLIMIT= option.

Data source: Teradata

See: DBCOMMIT= LIBNAME option, DBCOMMIT= data set option, ERRLIMIT= LIBNAME option, ERRLIMIT= data set option, MULTILOAD= data set option, MULTISTMT= LIBNAME option

Syntax

MULTISTMT=YES | NO

Syntax Description

YES
tries to send as many inserts to Teradata that can fit in a 64K buffer. If multistatement inserts are not possible, processing reverts to single-row inserts.

NO
send inserts to Teradata one row at a time.

Details

When you request multistatement inserts, SAS first determines how many insert statements that it can send to Teradata. Several factors determine the actual number of statements that SAS can send—for example:

- how many SQL insert statements can fit in a 64K buffer
- how many data rows can fit in the 64K data buffer
- how many inserts the Teradata server chooses to accept

When you need to insert large volumes of data, you can significantly improve performance by using MULTISTMT= instead of inserting only single-row.

When you also specify the DBCOMMIT= option, SAS determines the number of insert statements to send together at one time. It uses the smaller of the DBCOMMIT= value and the number of insert statements that can fit in a buffer.

Examples

Example 1: Send and Insert Statements One at a Time

This example shows how to send insert statements one at a time to Teradata.

```
libname user teradata user=zoom pw=XXXXXX server=dbc;
proc datasets library=user;
  delete testdata;run;
data user.testdata(DBTYPE=(I="INT") MULTISTMT=NO);
     do i=1 to 50;
         output;
     end;
run;
```

Example 2: Send 100 Rows at a Time

In this example, DBCOMMIT=100. Therefore, SAS issues a commit after every 100 rows, sending only 100 rows at a time.

```
libname user teradata user=zoom pw=XXXXX server=dbc;
proc datasets library=user;
```

```
  delete testdata;run;
proc datasets library=user;
  delete testdata;run;
data user.testdata(MULTISTMT=YES DBCOMMIT=100);
do i=1 to 1000;
   output;
   end;
run;
```

Example 3: Send a Specified Group of Rows at a Time

In this example, DBCOMMIT=1000, which is much higher than in the previous example. SAS sends as many rows as it can fit in the buffer at a time (up to 1000), and it issues a commit after every 1000 rows. If only 600 can fit, 600 are sent to the database. It is followed by the remaining 400—the difference between 1000 and the initial 600 that were already sent. SAS then commits all rows.

```
libname user teradata user=zoom pw=XXXXX server=dbc;
proc datasets library=user;
  delete testdata;run;
proc datasets library=user;
  delete testdata;run;
data user.testdata(MULTISTMT=YES DBCOMMIT=1000);
do i=1 to 10000;
   output;
   end;
run;
```

Example 4: Use a Global Options to Store a Temporary Table

This example sets CONNECTION=GLOBAL for all tables, creates a global temporary table, and stores the table in the current database schema.

```
libname user teradata user=zoom pw=XXXXX server=dbc connection=global;
proc datasets library=user;
  delete temp1;run;

proc sql;
   connect to teradata(user=zoom pw=XXXXXXX server=dbc connection=global);
   execute (CREATE GLOBAL TEMPORARY TABLE temp1 (col1 INT )
            ON COMMIT PRESERVE ROWS) by teradata;
   execute (COMMIT WORK) by teradata;
quit;
data work.test;
  do col1=1 to 1000;
     output;
  end;
run;
proc append data=work.test base=user.temp1(multistmt=yes);
run;
```

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)

Default: SAS

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: BULKLOAD= data set option, DBINDEX= data set option, DBKEY= data set option, DBNULL= data set option, NULLCHARVAL= data set option

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 these. 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 these. 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 NULL values for the column.

Details

This option affects insert and update processing. It also applies when you use the DBINDEX= and DBKEY= data set options.

It works with the NULLCHARVAL= data set option, which determines what is inserted when NULL values are not allowed. The DBMS treats all missing SAS numeric values (represented in SAS as '.') as NULLvalues.

Oracle: For interactions between NULLCHAR= and BULKLOAD=ZX`11, see the bulk-load topic in the Oracle section.

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)

Default: a blank character

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBFORCE= data set option, DBINDEX= data set option, DBKEY= data set option, DBNULL= data set option, NULLCHAR= data set option

Syntax

NULLCHARVAL=*'character-string'*

Details

This option affects insert and update processing and also applies when you use the DBINDEX= and DBKEY= data set options.

It also works with the NULLCHAR= option to determine whether a missing SAS character value is treated as a NULL value. If NULLCHARVAL= is longer than the maximum column width, one of these things happens:

- The string is truncated if DBFORCE=YES.
- The operation fails if DBFORCE=NO.

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)

Default: none

Restriction: The partition name must be valid or an error occurs.

Data source: Oracle

Tip: 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.

Syntax

OR_PARTITION=*name of a partition in a partitioned Oracle table*

Syntax Description

name of a partition in a partitioned Oracle table
specifies the partition name.

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 boosts performance because you are limiting your access to only one partition of a table instead of the entire table.

Examples

Example 1: Read, Update, Delete, Load, and Insert from a Partitioned Table

This example shows one way that you can use this option.

```
libname x oracle user=scott pw=tiger path=oraclev9;
proc datasets library=x;
  delete 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;
```

Example 2: Create and Manipulate a Partitioned Table

Here 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 examples depend.

```
libname x oracle user=scott pw=tiger path=oraclev9;
proc datasets library=x;
  delete 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;
```

In this example, only the P4 partition is read.

```
proc print data=x.orparttest ( or_partition-p4 );
run;
```

Next, rows that belong to only the single P4 partition are updated.

```
proc sql;
update x.orparttest ( or_partition=p4 ) set num=35;
quit;
```

Although this shows how a particular partition can be updated, 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, this next 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;
```

In this example, all rows in the P4 partition are deleted.

```
proc sql;
delete from x.orparttest ( or_partition=p4 );
quit;
```

Next, 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;
```

This 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)

Alias: ORACLE_73_OR_ABOVE=

Default: LIBNAME setting

Requirement: Due to the published Oracle bug 440366, sometimes an update on a row fails even if the row has not changed. Oracle offers this solution: When you create a table, increase the number of INITRANS to at least 3 for the table.

Data source: Oracle

See: "Locking in the Oracle Interface", OR_UPD_NOWHERE= LIBNAME option, SASTRACE= system option, SASTRACELOC= system option, UPDATE_LOCK_TYPE= data set option

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 later) 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 so that you can avoid problems with extra WHERE clause processing and potential WHERE clause floating-point precision.

Specify OR_UPD_NOWHERE=NO for compatibility when you are updating a SAS 6 view descriptor.

Example: Create and Update a Table

In this example, you create a small Oracle table, TEST. You then update it once by using the default setting (OR_UPD_NOWHERE=YES) and once by 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 SASTRACE= and SASTRACELOC= system options to send the output to the SAS log.

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)

Default: none

Data source: Oracle

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.

Example: Pass a Hint

This example runs a SAS procedure on DBMS data and SAS converts the procedure to one or more SQL queries. ORHINTS= lets you 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 this example, SAS sends the Oracle hint **`'/*+ ALL_ROWS */'`** to Oracle as part of this statement:

```
SELECT /*+ ALL_ROWS */ * FROM PAYROLL
```

PARTITION_KEY= Data Set Option

Specifies the column name to use as the partition key for creating fact tables.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Requirements: To create a data set in Aster *n*Cluster without error, you must either set DIMENSION= YES (LIBNAME or data set option) or specify a partition key in the PARTITION_KEY= (LIBNAME or data set) option.
You must enclose the column name in quotation marks.

Data source: Aster *n*Cluster

See: DIMENSION= LIBNAME option, DIMENSION= data set option, PARTITION_KEY= LIBNAME option [contains examples]

Syntax

PARTITION_KEY='*column-name*'

Details

Aster *n*Cluster uses two table types, dimension and fact tables.

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).

Alias: PRESERVE_NAMES= (see "Details")

Default: LIBNAME setting

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase IQ, Teradata

See: PRESERVE_COL_NAMES= LIBNAME option,VALIDVARNAME= system option. See also SAS Names and Support for DBMS Names, plus naming conventions for Aster nCluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata.

Syntax

PRESERVE_COL_NAMES=YES | NO

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. (For more information see the VALIDVARNAME= system option.) 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 these 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 naming topic in this document and also *SAS Data Set Options: Reference*.
- So that the DBMS to normalize the column names according to its naming conventions, specify variables with any case and set PRESERVE_COLUMN_NAMES=NO.

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 SAS/ACCESS naming topic in the DBMS-specific reference section for your interface.

To save some time when coding, specify the PRESERVE_NAMES= alias if you plan to specify both the PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options in your LIBNAME statement.

To use column names in your SAS program that are not valid SAS names, you must use one of these techniques.

- Use the DQUOTE= option in PROC SQL and reference your columns using double quotation marks. Here is an example.

```
proc sql dquote=ansi;
   select "Total$Cost" from mydblib.mytable;
```

- Specify the global VALIDVARNAME=ANY system option and use name literals in the SAS language. Here is an example.

```
proc print data=mydblib.mytable;
   format 'Total$Cost'n 22.2;
```

If you are *creating* a table in PROC SQL, you must also include the PRESERVE_COL_NAMES=YES option. Here is an 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.

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)

Default: LIBNAME setting

Data source: HP Neoview, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB

See: QUALIFIER= LIBNAME option

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*.

QUERY_BAND= Data Set Option

Specifies whether to set a query band for the current transaction.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Data source: Teradata

See: BULKLOAD= LIBNAME option, BULKLOAD= data set option, FASTEXPORT= LIBNAME option, Maximizing Teradata Load Performance, MULTILOAD= data set option, QUERY_BAND= LIBNAME option

Syntax

QUERY_BAND="*pair-name=pair_value*" FOR TRANSACTION;

Syntax Description

pair-name=pair_value
specifies a name and value pair of a query band for the current transaction.

Details

Use this option to set unique identifiers on Teradata transactions and to add them to the current transaction. The Teradata engine uses this syntax to pass the name-value pair to Teradata.

```
SET QUERY_BAND="org=Marketing;report=Mkt4Q08;" FOR TRANSACTION;
```

For more information about this option and query-band limitations, see *Teradata SQL Reference: Data Definition Statements*.

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)

Default: LIBNAME setting

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, Sybase IQ

See: QUERY_TIMEOUT= LIBNAME option

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.

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)

Alias: ROWSET_SIZE= [DB2 under UNIX and PC Hosts, Microsoft SQL Server, Netezza, ODBC, OLE DB]

Default: LIBNAME setting

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ

See: READBUFF= LIBNAME option

Syntax

READBUFF=*integer*

Syntax Description

integer
specifies 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, 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 under UNIX and PC Hosts:
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.

Greenplum, Microsoft SQL Server, Netezza, ODBC, Sybase IQ:
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.

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)

Default: DBMS-specific

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Teradata

See: READ_ISOLATION_LEVEL= LIBNAME option, READ_LOCK_TYPE= data set option. See also locking information for DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata.

Syntax

READ_ISOLATION_LEVEL=*DBMS-specific-value*

Syntax Description

dbms-specific-value
See the DBMS-specific reference section for your interface for this value.

Details

The degree of isolation defines the degree to which these items are affected:

- rows that are read and updated by the current application are available to other concurrently executing applications

- update activity of other concurrently executing application processes can affect the current application

`DB2 under UNIX and PC Hosts, Netezza, ODBC`: This option is ignored if you do not set READ_LOCK_TYPE=ROW.

See the locking topic for your interface in the DBMS-specific reference section for details.

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)

Default: DBMS-specific

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: CONNECTION= LIBNAME option, READ_LOCK_TYPE= LIBNAME option. See also locking information for DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata.

Syntax

READ_LOCK_TYPE=ROW | PAGE | TABLE | NOLOCK | VIEW

Syntax Description

PAGE [valid only for Sybase]
locks a page of data, which is a DBMS-specific number of bytes.

TABLE [valid only for DB2 under UNIX and PC Hosts, DB2 under z/OS, ODBC, Oracle, Microsoft SQL Server, Teradata]
locks the entire DBMS table. If you specify READ_LOCK_TYPE=TABLE, you must also specify the 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).

NOLOCK [valid only for Microsoft SQL Server, Oracle, Sybase, and ODBC with the Microsoft SQL Server driver]
does not lock the DBMS table, pages, or any rows during a read transaction.

VIEW [valid only for Teradata]
locks the entire DBMS view.

Details

If you omit READ_LOCK_TYPE=, you receive 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 locking topic for your interface in the DBMS-specific reference section for details.

READ_MODE_WAIT= Data Set Option

Specifies during SAS/ACCESS Read operations whether Teradata waits to acquire a lock or fails your request when a different user has locked the DBMS resource.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: LIBNAME setting

Data source: Teradata

See: READ_MODE_WAIT= LIBNAME option. See also "Locking in the Teradata Interface".

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 locking topic in the Teradata section.

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)

Default: DBMS-specific

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DBSASTYPE= data set option

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 SAS column date format does not match the date format of the corresponding DBMS column, convert the SAS date values to the appropriate DBMS date values. Use the SASDATEFMT= option 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 these 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 DBMS-specific reference section 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. However, 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:
It is recommended that you use the DBSASTYPE= data set option instead of SASDATEFMT=.

Examples

Example 1: Change the Date Format in Oracle

In this 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;
```

Example 2: Change a SAS Date Format to a Teradata Format

In the next 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 );
```

Example 3: Change a Teradata Date Format to a SAS Format

In this 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

Allows reading of such database objects as tables and views in the specified schema.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Alias: DATABASE= [Teradata]

Default: LIBNAME option [Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, HP Neoview, Informix, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ], AUTHID= [DB2 under z/OS]

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: PRESERVE_TAB_NAMES= LIBNAME option, SCHEMA= LIBNAME option

Syntax

SCHEMA=*schema-name*

Syntax Description

schema-name
specifies the name that is 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.

If you do not specify this option, you connect to the default schema for your DBMS.

The values for SCHEMA= are usually case sensitive, so be careful when you specify this option.

Aster nCluster: The default is **none**, which uses the database user's default schema. When the user's default scheme is the user name, the user name is used instead. An example is when SQLTables is called to obtain a table listing using PROC DATASETS or SAS Explorer.

Oracle: The default is the LIBNAME setting. 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.

Sybase: You cannot use the SCHEMA= option when you use UPDATE_LOCK_TYPE=PAGE to update a table.

Teradata: The default is the LIBNAME setting. 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 lets you 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.) For more information about changing the default database, see the DATABASE statement in your Teradata documentation.

Example

In this example, SCHEMA= causes DB2 to interpret MYDB.TEMP_EMPS as SCOTT.TEMP_EMPS.

```
proc print data=mydb.temp_emps
     schema=SCOTT;
run;
```

In this next 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 this 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;
```

SEGMENT_NAME= Data Set Option

Lets you control the segment in which you create a table.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Data source: Sybase

Syntax

SEGMENT_NAME=*segment-name*

Syntax Description

segment-name
specifies the name of the segment in which to create a table.

SESSIONS= Data Set Option

Specifies how many Teradata sessions to be logged on when using FastLoad, FastExport, or Multiload.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

Default: none

Data source: Teradata

See: SESSIONS= LIBNAME option

Syntax

SESSIONS=*number-of-sessions*

Syntax Description

number-of-sessions
specifies a numeric value that indicates the number of sessions to be logged on.

Details

When reading data with FastExport or loading data with FastLoad and MultiLoad, you can request multiple sessions to increase throughput. Using large values might not necessarily increase throughput due to the overhead associated with session management. Check whether your site has any recommended value for the number of sessions to use. See your Teradata documentation for details about using multiple sessions.

Example: Request Sessions to Load a Table

This example uses SESSIONS= in a LIBNAME statement to request that five sessions be used to load data with FastLoad.

```
libname x teradata user=prboni pw=prboni;
proc datasets library=x;
  delete test;run;
data x.test(FASTLOAD=YES SESSIONS=2);
i=5;
run;
```

SET= Data Set Option

Specifies whether duplicate rows are allowed when creating a table.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

Alias: TBLSET

Default: NO

Data source: Teradata

Syntax

SET=YES | NO

Syntax Description

YES
specifies that no duplicate rows are allowed.

NO
specifies that duplicate rows are allowed.

Details

Use the SET= data set option to specify whether to allow duplicate rows when creating a table. This option overrides the default Teradata MULTISET characteristic.

Example: Create a Table without Duplicate Rows

This example creates a Teradata table of type SET that does not allow duplicate rows.

```
libname trlib teradata user=testuser pw=testpass;
options sastrace=',,,d' sastraceloc=saslog;
proc datasets library=x;
  delete test1;run;
data x.test1(TBLSET=YES);
i=1;output;
run;
```

SLEEP= Data Set Option

Specifies the number of minutes that FASTEXPORT=, FastLoad, or MultiLoad waits before trying again to log on to Teradata.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

Default: 6

Data source: Teradata

Tip: The data set option has precedence over the LIBNAME option.

See: DBSLICEPARM= LIBNAME option, DBSLICEPARM= data set option, DBSLICEPARM= system option, FASTEXPORT= LIBNAME option, Maximizing Teradata Load Performance, MULTILOAD= data set option, SLEEP= LIBNAME option, TENACITY= LIBNAME option, TENACITY= data set option, Using the TPT API

Syntax

SLEEP=*number-of-minutes*

Syntax Description

number-of-minutes
the number of minutes to wait before trying again to log on to Teradata.

Details

Use this option to indicate to FASTEXPORT=, FastLoad, or 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 on the database server setting.) The default value for SLEEP= is 6 minutes. The value that you specify for SLEEP= must be greater than 0.

Use SLEEP= with TENACITY=, which specifies the time in hours that FASTEXPORT=, FastLoad, or MultiLoad must continue to try the logon operation. SLEEP= and TENACITY= function very much like the SLEEP and TENACITY run-time options of the native Teradata FastExport, FastLoad, or MultiLoad utility.

TEMPORAL_QUALIFIER= Data Set Option

Specifies time-dimension criteria for retrieving data from Teradata.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: CURRENT VALIDTIME [valid-time column], CURRENT TRANSACTIONTIME [transaction-time column]

Interaction: Specifying values in a DATA step overrides LIBNAME values.

Data source: Teradata

See: TEMPORAL_QUALIFIER= LIBNAME option

Syntax

TEMPORAL_QUALIFIER=CURRENT VALIDTIME
| VALIDTIME AS OF PERIOD '*period*' | SEQUENCED VALIDTIME
| NONSEQUENCED VALIDTIME '*period*'

TEMPORAL_QUALIFIER=CURRENT TRANSACTIONTIME
| TRANSACTIONTIME AS OF PERIOD '*period*'
| NONSEQUENCED TRANSACTIONTIME

Syntax Description

CURRENT VALIDTIME
selects rows that are valid at the current time.

VALIDTIME AS OF PERIOD '*period*'
selects rows with valid-time periods that overlap the specified AS OF period. For the period, you can specify either a single date or a time period (date range) by specifying a start date and an end date.

SEQUENCED VALIDTIME PERIOD '*period*'
selects history, current, or future rows that are valid for the specified time period.

NONSEQUENCED VALIDTIME PERIOD '*period*'
treats the table as nontemporal.

CURRENT TRANSACTIONTIME
selects rows that are open in transaction time.

TRANSACTIONTIME AS OF '*period*'
selects rows with transaction-time periods that overlap the specified AS OF period. For the period, you can specify either a single date or a time period (date range) by specifying a start date and an end date.

NONSEQUENCED TRANSACTIONTIME PERIOD '*period*'
treats the table as nontemporal.

Details

Use temporal qualifiers to specify time criteria for selecting data from temporal tables.

To use them, before the SQL add a value that you specify for one or more temporal qualifiers for a data set. For example, if you specify **`TEMPORAL_QUALIFIER='AS OF PERIOD '(1999-01-01, 2099-01-05)' '`** in a DATA step, **`'AS OF PERIOD '(1999-01-01, 2099-01-05)' '`** is added before the SQL to select the data.

Examples

Example 1: Select Valid-Time Rows at the Current Time

```
/* Consider data as of 1995-01-01. */
libname x teradata user=testuser pw=testpw server=td1310
          TEMPORAL_QUALIFIER='VALIDTIME AS OF DATE '1995-01-01' '

/* ASOF PERIOD '(1999-01-01, 2099-01-05)' select * from mytest is submitted. */
proc print data=x.mytest(TEMPORAL_QUALIFIER='CURRENT VALIDTIME');
run;
```

Example 2: Select Valid-Time Rows for a Specific Date

This example extracts salary details for employees who worked on January 1, 2000.

Employment data from this employee table contains the ValidTime data column, JobDuration.

```
EName   E_Id    Dept    Job_duration
Sania   1001    Dept1   1990-01-01, 2003-01-01
Sania   1001    Dept3   2003-01-01, UNTIL_CHANGED
Ash     1002    Dept1   1995-01-01, 2000-01-01
Ash     1002    Dept2   1999-01-01, 2010-01-01
```

Salary data is from the ValidTime column, SalaryPeriod.

```
E_Id    Sal     SalaryPeriod
1001    10000   1990-01-01, 2003-01-01
1001    20000   2003-01-01, 2010-01-01
1001    30000   2010-01-01, UNTIL_CHANGED
1002    25000   1995-01-01, 2010-01-01
```

Here is the query.

```
VALIDTIME AS OF DATE'2000-01-01'
SELECT E.EName as Name, S.Sal as Salary
FROM Employee E, Salary S
WHERE E.E_Id = S.E_Id;
```

It produces this data as the result.

Name	Salary
Sania	10000
Ash	25000

Example 3: Select Transaction-Time Rows for a Specific Date and Time

This example extracts stock details as of a specific timestamp.

Data from this stock table contains a transaction-time dimension: the TransactionTime data column, RecordedTime.

```
StockName  StockValue  RecordedTime
Teradata   38          2006-01-01 10:00:00.000000+00:00,
                       2006-01-01 12:10:10.000000+00:00
Teradata   37          2006-01-01 12:10:10.000000+00:00,
                       2006-01-03 10:00:00.000000+00:00
Teradata   40          2006-01-03 10:00:00.000000+00:00, UNTIL_CLOSED
```

Here is the query.

```
TRANSACTIONTIME AS OF TIMESTAMP'2006-01-02 12:10:10.000000+00:00'
SELECT * FROM Stock;
```

It produces this data as the result.

StockName	StockValue
Teradata	37

TENACITY= Data Set Option

Specifies how many hours that FASTEXPORT=, FastLoad, or MultiLoad continues to try to log on again 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)

Default: 0 [FastLoad], 4 [FASTEXPORT=, MultiLoad]

Data source: Teradata

Tip: The data set option has precedence over the LIBNAME option.

See: DBSLICEPARM= LIBNAME option, DBSLICEPARM= data set option, DBSLICEPARM= system option, FASTEXPORT= LIBNAME option, Maximizing Teradata Load Performance, MULTILOAD= data set option, SLEEP= LIBNAME option, SLEEP= data set option, Using the TPT API

Syntax

TENACITY=*number-of-hours*

Syntax Description

number-of-hours
specifies the number of hours to continue to try again to log on to Teradata.

Details

Use this option to indicate to FASTEXPORT=, FastLoad, or 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 on the database server setting.) The default value for TENACITY= is four hours. The value specified for TENACITY= must be greater than zero.

Use TENACITY= with SLEEP=, which specifies the number of minutes that FASTEXPORT=, FastLoad, or 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 FastExport, FastLoad, or MultiLoad utility.

Here is an example of the message that is written to the SAS log if the time period that TENACITY= specifies is exceeded.

```
ERROR:  MultiLoad failed unexpectedly with returncode 12
```

Check the FASTEXPORT=, FastLoad, or MultiLoad log for more information about the cause of the FASTEXPORT=, FastLoad, or MultiLoad failure. SAS does not receive any informational messages from Teradata in either of these situations:

- when the currently run FASTEXPORT=, FastLoad, or MultiLoad process waits because the maximum number of utilities are already running
- if FASTEXPORT=, FastLoad, or MultiLoad is terminated because the time limit that TENACITY= specifies has been exceeded

The native Teradata FastExport, FastLoad, or MultiLoad utility sends messages associated with SLEEP= and TENACITY= only to the FastExport, FastLoad, or MultiLoad log. So nothing is written to the SAS log.

TPT= Data Set Option

Specifies whether SAS uses the TPT API to load data for Fastload, MultiLoad, or Multi-Statement insert requests.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: YES

Data source: Teradata

See: Using the TPT API, BULKLOAD= LIBNAME option, BULKLOAD= data set option, LOGDB= LIBNAME option, MULTILOAD= data set option, MULTISTMT= data set option, TPT= LIBNAME option, TPT_APPL_PHASE= data set option, TPT_BUFFER_SIZE= data set option, TPT_CHECKPOINT= data set option, TPT_DATA_ENCRYPTION= data set option, TPT_ERROR_TABLE_1= data set option, TPT_ERROR_TABLE_2= data set option, TPT_LOG_TABLE= data set option, TPT_MAX_SESSIONS= data set option, TPT_MIN_SESSIONS= data set option, TPT_PACK= data set option, TPT_PACKMAXIMUM= data set option, TPT_RESTART= data set option, TPT_TRACE_LEVEL= data set option, TPT_TRACE_LEVEL_INF= data set option, TPT_TRACE_OUTPUT= data set option, TPT_WORK_TABLE= data set option.

Maximizing Teradata Load Performance

Syntax

TPT=YES | NO

Syntax Description

YES
specifies that SAS uses the TPT API when Fastload, MultiLoad, or Multi-Statement insert is requested.

NO
specifies that SAS does not use the TPT API when Fastload, MultiLoad, or Multi-Statement insert is requested.

Details

By using the TPT API, you can load data into a Teradata table without working directly with such stand-alone Teradata utilities as Fastload, MultiLoad, or TPump. When TPT=NO, SAS uses the TPT API load driver for FastLoad, the update driver for MultiLoad, and the stream driver for Multi-Statement insert.

When TPT=YES, sometimes SAS cannot use the TPT API due to an error or because it is not installed on the system. When this happens, SAS does not produce an error, but it still tries to load data using the requested load method (Fastload, MultiLoad, or Multi-Statement insert). To check whether SAS used the TPT API to load data, look for a similar message to this one in the SAS log:

```
NOTE: Teradata connection: TPT FastLoad/MultiLoad/MultiStatement
insert has read n row(s).
```

Example: Load Data Using the TPT API

In this example, SAS data is loaded into Teradata using the TPT API. This is the default method of loading when Fastload, MultiLoad, or Multi-Statement insert is requested. SAS still tries to load data even if it cannot use the TPT API.

```
libname tera Teradata user=testuser pw=testpw;
/* Create data */
data testdata;
do i=1 to 100;
    output;
end;
run;
/* Load using FastLoad TPT.  This note appears in the SAS log if SAS uses TPT.
NOTE:  Teradata connection:  TPT FastLoad has inserted 100 row(s).*/
data tera.testdata(FASTLOAD=YES TPT=YES);
set testdata;
run;
```

TPT_APPL_PHASE= Data Set Option

Specifies whether a load process that is being restarted has failed in the application phase.

Valid in: PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option, TPT= LIBNAME option, TPT= data set option, TPT_CHECKPOINT_DATA= data set option, TPT_RESTART= data set option

Syntax

TPT_APPL_PHASE=YES | NO

Syntax Description

YES
specifies that the Fastload or MultiLoad run that is being restarted has failed in the application phase. This is valid only when SAS uses the TPT API.

NO
specifies that the load process that is being restarted has not failed in the application phase.

Details

To use this option, you must first set TPT=YES.

SAS can restart from checkpoints any Fastload, MultiLoad, and Multi-Statement insert that is run using the TPT API. The restart procedure varies: It depends on whether checkpoints were recorded and in which phase the step failed during the load process. Teradata loads data in two phases: the acquisition phase and the application phase. In the acquisition phase, data transfers from SAS to Teradata. After this phase, SAS has no more data to transfer to Teradata. If failure occurs after this phase, set TPT_APPL_PHASE=YES in the restart step to indicate that restart is in the application phase. (Multi-Statement insert does not have an application phase. Therefore, it does not need to be restarted if it fails after the acquisition phase.)

Use OBS=1 for the source data set when restart occurs in the application phase. When SAS encounters TPT_RESTART=YES and TPT_APPL_PHASE=YES, it initiates restart in the application phase. No data from the source data set is actually sent. If you use OBS=1 for the source data set, the SAS step completes as soon as it reads the first record. (It actually throws away the record because SAS already sent all data to Teradata for loading.)

Example: Restart after Failure

Here is a sample SAS program that failed after the acquisition phase.

```
libname x teradata user=testuser pw=testpw;
data x.test(MULTILOAD=YES TPT=YES CHECKPOINT=7);
do i=1 to 20;
output;
end;
run;
ERROR: Teradata connection:  Failure occurred after the acquisition phase.
Restart outside of SAS using checkpoint data 14.
```

Set TPT_APPL_PHASE=YES to restart when failure occurs in the application phase because SAS has already sent all data to Teradata.

```
proc append base=x.test(MULTILOAD=YES TPT_RESTART=YES
    TPT_CHECKPOINT_DATA=14 TPT_APPL_PHASE=YES) data=test(obs=1);
run;
```

TPT_BUFFER_SIZE= Data Set Option

Specifies the output buffer size in kilobytes when SAS sends data to Teradata with Fastload or MultiLoad using the TPT API.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 64

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, TPT= LIBNAME option, TPT= data set option

Syntax

TPT_BUFFER_SIZE=*integer*

Syntax Description

integer
specifies the size of data parcels in kilobytes from 1 through 64.

Details

To use this option, you must first set TPT=YES.

You can use the output buffer size to control the amount of data that is transferred in each parcel from SAS to Teradata when using the TPT API. A larger buffer size can reduce processing overhead by including more data in each parcel. See your Teradata documentation for details.

TPT_CHECKPOINT_DATA= Data Set Option

Specifies the checkpoint data to return to Teradata when restarting a failed MultiLoad or Multi-Statement step that uses the TPT API.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: none

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", MULTILOAD= data set option , MULTISTMT= data set option, TPT= LIBNAME option, BULKLOAD= data set option, TPT_APPL_PHASE= data set option, TPT_RESTART= data set option

Syntax

TPT_CHECKPOINT_DATA=*checkpoint_data_in_error_message*

Syntax Description

checkpoint_data_in_error_message
specifies the value to use to restart a failed MultiLoad or Multi-Statement step that uses the TPT API.

Details

To use this option, you must first set TPT=YES and TPT_RESTART=YES.

SAS can restart from the last checkpoint any failed Fastload, MultiLoad, and Multi-Statement insert that are run using the TPT API. Teradata returns a checkpoint value each time MultiLoad or Multi-Statement records a checkpoint. The SAS log contains this value when a load fails. SAS must provide the same value as a data set option when it tries to restart the load process.

Here are the rules that govern restart.

- The TPT API does not return a checkpoint value when FastLoad records a checkpoint. Therefore, you need not set TPT_CHECKPOINT_VALUE= when you use FastLoad. Set TPT_RESTART= instead.
- If the default error table name, work table name, or restart table name is overridden, SAS must use the same name while restarting the load process.
- Teradata loads data in two phases: the acquisition phase and the application phase. In the acquisition phase, data transfers from SAS to Teradata. After this phase, SAS has no more data to transfer to Teradata. If failure occurs after this phase, set TPT_APPL_PHASE=YES while restarting. (Multi-Statement insert does not have an application phase. Therefore, it does not need to be restarted if it fails after the

acquisition phase.) Use OBS=1 for the source data set because SAS has already sent the data to Teradata. Therefore, there is no need to send any more data.

- If failure occurred before the acquisition phase ended and the load process recorded no checkpoints, you must restart the load process from the beginning by setting TPT_RESTART=YES. However, you need not set TPT_CHECKPOINT_VALUE= because no checkpoints were recorded. The error message in the SAS log provides all needed information for restart.

Examples

Example 1

In this example, assume that the MultiLoad step that uses the TPT API fails before the acquisition phase ends and no options were set to record checkpoints.

```
libname x teradata user=testuser pw=testpw;
data test;In
do i=1 to 100;
output;
end;
run;
/* Set TPT=YES is optional because it is the default. */
data x.test(MULTILOAD=YES TPT=YES);
set test;
run;
```

This error message is sent to the SAS log. You need not set TPT_CHECKPOINT_DATA= because no checkpoints were recorded.

```
ERROR:  Teradata connection:  Correct error and restart as an APPEND
process with option TPT_RESTART=YES.  Since no checkpoints were taken,
if the previous run used FIRSTOBS=n, use the same value in the restart.
```

Example 2

Here is an example of the restart step.

```
proc append data=test base=x.test(FASTLOAD=YES TPT=YES TPT_RESTART=YES);
run;
```

Example 3

In this next example, failure occurs after checkpoints are recorded.

```
libname tera teradata user=testuser pw=testpw;
/* Create data */
data testdata;
do i=1 to 100;
    output;
end;
run;
/* Assume that this step fails after loading row 19. */
data x.test(MULTISTMT=YES CHECKPOINT=3);
set testdata;
run;
```

Here is the resulting error when it fails after loading 18 rows.

```
ERROR: Teradata connection: Correct error and restart as
an APPEND process with option TPT_RESTART=YES.  If the previous run
used FIRSTOBS=n, use the value ( n-1+ 19 ) for FIRSTOBS in the restart.
Otherwise use FIRSTOBS= 19.  Also specify TPT_CHECKPOINT_DATA= 18.
```

You can restart the failed step with this code.

```
proc append base=x.test(MULTISTMT=YES TPT_RESTART=YES
    TPT_CHECKPOINT_DATA=18) data=test(firstobs=19);
run;
```

If failure occurs after the end of the acquisition phase, you must write a custom C++ program to restart from the point where it stopped.

Example 4

Here is a sample SAS program that failed after the acquisition phase and the resulting error message.

```
libname x teradata user=testuser pw=testpw;
data x.test(MULTILOAD=YES TPT=YES CHECKPOINT=7);
do i=1 to 20;
output;
end;
run;
ERROR:  Teradata connection:  Failure occurred after the acquisition phase.
Restart outside of SAS using checkpoint data  14.
```

Set TPT_APPL_PHASE=YES to restart when failure occurs in the application phase because SAS has already sent all data to Teradata.

```
proc append base=x.test(MULTILOAD=YES TPT_RESTART=YES
    TPT_CHECKPOINT_DATA=14 TPT_APPL_PHASE=YES) data=test(obs=1);
run;
```

TPT_DATA_ENCRYPTION= Data Set Option

Specifies whether to fully encrypt SQL requests, responses, and data when SAS sends data to Teradata for Fastload, MultiLoad, or Multi-Statement insert that uses the TPT API.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option, MULTISTMT= data set option, TPT= LIBNAME option, TPT= data set option

Syntax

TPT_DATA_ENCRYPTION=YES | NO

Syntax Description

YES

specifies that all communication between the Teradata client and server is encrypted when using the TPT API.

NO
specifies that all communication between the Teradata client and server is not encrypted when using the TPT API.

Details

To use this option, you must first set TPT=YES.

You can ensure that SQL requests, responses, and data that are transferred between the Teradata client and server are encrypted when using the TPT API. See your Teradata documentation for details.

TPT_ERROR_TABLE_1= Data Set Option

Specifies the name of the first error table for SAS to use when using the TPT API with Fastload or MultiLoad.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: table_name_ET

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option, TPT= LIBNAME option, TPT= data set option, TPT_ERROR_TABLE_2= data set option, TPT_LOG_TABLE= data set option, TPT_WORK_TABLE= data set option

Syntax

TPT_ERROR_TABLE_1=*valid_teradata_table_name*

Syntax Description

valid_teradata_table_name
specifies the name of the first error table for SAS to use when using the TPT API to load data with Fastload or MultiLoad.

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

Fastload and MultiLoad require an error table to hold records that were rejected during the acquisition phase. If you do not specify an error table, Teradata appends "_ET" to the name of the target table to load and uses it as the first error table by default. You can override this name by setting TPT_ERROR_TABLE_1=. If you do this and the load step fails, you must specify the same name when restarting. For information about errors that are logged in this table, see your Teradata documentation.

The name that you specify in TPT_ERROR_TABLE_1= must be unique. It cannot be the name of an existing table unless it is in a restart scenario.

Example: Specify Different Names for Two Error Tables

In this example, a different name is provided for both the first and second error tables that Fastload and MultiLoad use with the TPT API.

```
libname tera teradata user=testuser pw=testpw;
/* Load using Fastload TPT.  Use alternate names for the error tables. */
data tera.testdata(FASTLOAD=YES TPT_ERROR_TABLE_1=testerror1
    TPT_ERROR_TABLE_2=testerror2);
i=1;output; i=2;output;
run;
```

TPT_ERROR_TABLE_2= Data Set Option

Specifies the name of the second error table for SAS to use when using the TPT API with Fastload or MultiLoad.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: table_name_UV

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option, TPT= LIBNAME option, TPT= data set option, TPT_ERROR_TABLE_1= data set option, TPT_LOG_TABLE= data set option, TPT_WORK_TABLE= data set option

Syntax

TPT_ERROR_TABLE_2=*valid_teradata_table_name*

Syntax Description

valid_teradata_table_name
specifies the name of the second error table for SAS to use when using the TPT API to load data with Fastload or MultiLoad.

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

Fastload and MultiLoad require an error table to hold records that were rejected during the acquisition phase. If you do not specify an error table, Teradata appends "_UV" to the name of the target table to load and uses it as the second error table by default. You can override this name by setting TPT_ERROR_TABLE_2=. If you do this and the load step fails, you must specify the same name when restarting. For information about errors that are logged in this table, see your Teradata documentation.

The name that you specify in TPT_ERROR_TABLE_2= must be unique. It cannot be the name of an existing table unless it is in a restart scenario.

Example: Specify Different Names for First and Second Error Tables

In this example, a different name is provided for both the first and second error tables that Fastload and MultiLoad use with the TPT API.

```
libname tera teradata user=testuser pw=testpw;
/* Load using Fastload TPT.  Use alternate names for the error tables. */
```

```
data tera.testdata(FASTLOAD=YES TPT_ERROR_TABLE_1=testerror1
    TPT_ERROR_TABLE_2=testerror2);
i=1;output; i=2;output;
run;
```

TPT_LOG_TABLE= Data Set Option

Specifies the name of the restart log table for SAS to use when using the TPT API with Fastload, MultiLoad, or Multi-Statement insert.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: table_name_RS

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option, MULTISTMT= data set option, TPT= LIBNAME option, TPT= data set option, TPT_ERROR_TABLE_1= data set option, TPT_ERROR_TABLE_2= data set option, TPT_WORK_TABLE= data set option

Syntax

TPT_LOG_TABLE=*valid_teradata_table_name*

Syntax Description

valid_teradata_table_name
specifies the name of the restart log table for SAS to use when using the TPT API to load data with Fastload or MultiLoad.

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

Fastload, MultiLoad, and Multi-Statement insert that use the TPT API require a restart log table. If you do not specify a restart log table, Teradata appends "_RS" to the name of the target table to load and uses it as the restart log table by default. You can override this name by setting TPT_LOG_TABLE=. If you do this and the load step fails, you must specify the same name when restarting.

The name that you specify in TPT_LOG_TABLE= must be unique. It cannot be the name of an existing table unless it is in a restart scenario.

Example: Specify a Different Name for the Restart Log Table

In this example, a different name is provided for the restart log table that Multi-Statement uses with the TPT API.

```
libname tera teradata user=testuser pw=testpw;
/* Load using Fastload TPT.  Use alternate names for the log table. */
data tera.testdata(MULTISTMT=YES TPT_LOG_TABLE=restarttab);
```

```
i=1;output; i=2;output;
run;
```

TPT_MAX_SESSIONS= Data Set Option

Specifies the maximum number of sessions for Teradata to use when using the TPT API with FastLoad, MultiLoad, or Multi-Statement insert.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 1 session per available Access Module Processor (AMP)

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option, MULTISTMT= data set option, TPT= LIBNAME option, TPT= data set option, TPT_MIN_SESSIONS= data set option

Syntax

TPT_MAX_SESSIONS=*integer*

Syntax Description

integer

specifies the maximum number of sessions for Teradata to use when using the TPT API to load data with FastLoad, MultiLoad, or Multi-Statement insert.

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

You can control the number of sessions for Teradata to use when using the TPT API to load data with MultiLoad. The maximum value cannot be more than the number of available Access Module Processors (AMPs). See your Teradata documentation for details.

TPT_MIN_SESSIONS= Data Set Option

Specifies the minimum number of sessions for Teradata to use when using the TPT API with FastLoad, MultiLoad, or Multi-Statement insert.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 1

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option, MULTISTMT= data set option, TPT= LIBNAME option, TPT= data set option, TPT_MAX_SESSIONS= data set option

Syntax

TPT_MIN_SESSIONS=*integer*

Syntax Description

integer
specifies the minimum number of sessions for Teradata to use when using the TPT API to load data with FastLoad, MultiLoad, or Multi-Statement insert.

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

You can control the number of sessions that are required before using the TPT API to load data with MultiLoad. This value must be greater than zero and less than the maximum number of required sessions. See your Teradata documentation for details.

TPT_PACK= Data Set Option

Specifies the number of statements to pack into a Multi-Statement insert request when using the TPT API.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 20

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", MULTISTMT= data set option, TPT= LIBNAME option, TPT= data set option, TPT_PACKMAXIMUM= data set option

Syntax

TPT_PACK=*integer*

Syntax Description

integer
specifies the number of statements to pack into a Multi-Statement insert request when using the TPT API.

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

The maximum value is 600. See your Teradata documentation for details.

TPT_PACKMAXIMUM= Data Set Option

Specifies whether to pack the maximum possible or default number of statements into Multi-Statement insert requests when using the TPT API.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Restriction: This option is valid only when using the TPT API.

Requirement: To use this option, you must first set TPT=YES.

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", MULTISTMT= data set option, TPT= LIBNAME option, TPT= data set option, TPT_PACK= data set option

Syntax

TPT_PACKMAXIMUM=*YES* | *NO*

Syntax Description

YES
specifies that the maximum possible number of statements be packed into Multi-Statement insert requests when using the TPT API.

NO
specifies that the default number of statements be packed into Multi-Statement insert requests when using the TPT API.

Details

When TPT_PACKMAXIMUM=YES, the maximum possible number of statements that can fit in a Multi-Statement request is determined dynamically. See your Teradata documentation for details.

TPT_RESTART= Data Set Option

Specifies that a failed Fastload, MultiLoad, or Multi-Statement run that used the TPT API is being restarted.

Valid in: PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option , MULTISTMT= data set option, BULKLOAD= data set option, TPT_APPL_PHASE= data set option, TPT_CHECKPOINT_DATA= data set option

Syntax

TPT_RESTART=YES | NO

Syntax Description

YES
specifies that the load process is being restarted.

NO
specifies that the load process is not being restarted.

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

SAS can restart from checkpoints any Fastload, MultiLoad, and Multi-Statement insert that are run using the TPT API. The restart procedure varies: It depends on whether checkpoints were recorded and in which phase the step failed during the load process. The error message in the log is extremely important and contains instructions on how to restart.

Here are the rules that govern restart.

- The TPT API does not return a checkpoint value when FastLoad records a checkpoint. Therefore, you need not set TPT_CHECKPOINT_VALUE= when you use FastLoad. Set TPT_RESTART= instead.
- If the default error table name, work table name, or restart table name is overridden, SAS must use the same name while restarting the load process.
- Teradata loads data in two phases: the acquisition phase and the application phase. In the acquisition phase, data transfers from SAS to Teradata. After this phase, SAS has no more data to transfer to Teradata. If failure occurs after this phase, set TPT_APPL_PHASE=YES while restarting. (Multi-Statement insert does not have an application phase. Therefore, it does not need to be restarted if it fails after the acquisition phase.) Use OBS=1 for the source data set because SAS has already sent the data to Teradata. TTherefore, there is no need to send any more data.
- If failure occurred before the acquisition phase ended and the load process recorded no checkpoints, you must restart the load process from the beginning by setting TPT_RESTART=YES. However, you need not set TPT_CHECKPOINT_VALUE= because no checkpoints were recorded. The error message in the SAS log provides all needed information for restart.

Examples

Example 1

In this example, assume that the MultiLoad step that uses the TPT API fails before the acquisition phase ends and no options were set to record checkpoints.

```
libname x teradata user=testuser pw=testpw;
data test;In
do i=1 to 100;
output;
end;
run;
/* Set TPT=YES is optional because it is the default. */
data x.test(MULTILOAD=YES TPT=YES);
set test;
run;
```

This error message is sent to the SAS log. You need not set TPT_CHECKPOINT_DATA= because no checkpoints were recorded.

```
ERROR:  Teradata connection:  Correct error and restart as an APPEND process
with option TPT_RESTART=YES.  Since no checkpoints were taken,
if the previous run used FIRSTOBS=n, use the same value in the restart.
```

Example 2

Here is an example of the restart step.

```
proc append data=test base=x.test(MULTILOAD=YES TPT=YES TPT_RESTART=YES);
run;
```

Example 3

In this next example, failure occurs after checkpoints are recorded.

```
libname tera teradata user=testuser pw=testpw;
/* Create data */
data testdata;
do i=1 to 100;
    output;
end;
run;
/* Assume that this step fails after loading row 19. */
data x.test(MULTISTMT=YES CHECKPOINT=3);
set testdata;
run;
```

Here is the resulting error when it fails after loading 18 rows.

```
ERROR:  Teradata connection:  Correct error and restart as an APPEND process
with option TPT_RESTART=YES. If the previous run used FIRSTOBS=n,
use the value ( n-1+ 19) for FIRSTOBS in the restart.  Otherwise use FIRSTOBS=19.
Also specify TPT_CHECKPOINT_DATA= 18.
```

You can restart the failed step with this code.

```
proc append base=x.test(MULTISTMT=YES TPT_RESTART=YES
    TPT_CHECKPOINT_DATA=18) data=test(firstobs=19);
run;
```

If failure occurs after the end of the acquisition phase, you must write a custom C++ program to restart from the point where it stopped.

Example 4

Here is a sample SAS program that failed after the acquisition phase and the resulting error message.

```
libname x teradata user=testuser pw=testpw;
data x.test(MULTILOAD=YES TPT=YES CHECKPOINT=7);
do i=1 to 20;
output;
end;
run;
ERROR:  Teradata connection:  Failure occurred after the acquisition phase.
Restart outside of SAS using checkpoint data  14.
```

Set TPT_APPL_PHASE=YES to restart when failure occurs in the application phase because SAS has already sent all data to Teradata.

```
proc append base=x.test(MULTILOAD=YES TPT_RESTART=YES
    TPT_CHECKPOINT_DATA=14 TPT_APPL_PHASE=YES) data=test(obs=1);
run;
```

You must always use TPT_CHECKPOINT_DATA= with TPT_RESTART= for MultLoad and Multi-Statement insert.

TPT_TRACE_LEVEL= Data Set Option

Specifies the required tracing level for sending data to Teradata and using the TPT API with Fastload, MultiLoad, or Multi-Statement insert.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 1

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option , MULTISTMT= data set option, TPT= LIBNAME option, TPT= data set option, TPT_TRACE_LEVEL_INF= data set option, TPT_TRACE_OUTPUT= data set option

Syntax

TPT_TRACE_LEVEL=*integer*

Syntax Description

integer
specifies the needed tracing level (1 to 9) when loading data to Teradata.

1
no tracing

2
operator-level general trace

3
operator-level command-line interface (CLI) trace

4
operator-level notify method trace

5
operator-level common library trace

6
all operator-level traces

7
Telnet API (TELAPI) layer general trace

8
PutRow/GetRow trace

9
operator log message information

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

You can perform debugging by writing diagnostic messages to an external log file when loading data to Teradata using the TPT API. If you do not specify a name in

TPT_TRACE_OUTPUT= for the log file, a default name is generated using the current timestamp. See your Teradata documentation for details.

TPT_TRACE_LEVEL_INF= Data Set Option

Specifies the tracing level for the required infrastructure for sending data to Teradata and using the TPT API with Fastload, MultiLoad, or Multi-Statement insert.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: 1

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option , MULTISTMT= data set option, TPT= LIBNAME option, TPT= data set option, TPT_TRACE_LEVEL= data set option, TPT_TRACE_OUTPUT= data set option

Syntax

TPT_TRACE_LEVEL_INF=*integer*

Syntax Description

integer
specifies the needed infrastructure tracing level (10 to 18) when loading data to Teradata.

10
no tracing

11
operator-level general trace

12
operator-level command-line interface (CLI) trace

13
operator-level notify method trace

14
operator-level common library trace

15
all operator-level traces

16
Telnet API (TELAPI) layer general trace

17
PutRow/GetRow trace

18
operator log message information

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

You can perform debugging by writing diagnostic messages to an external log file when loading data to Teradata using the TPT API. If you do not specify a name in TPT_TRACE_OUTPUT= for the log file, a default name is generated using the current timestamp. See your Teradata documentation for details.

TPT_TRACE_OUTPUT= Data Set Option

Specifies the name of the external file for SAS to use for tracing when using the TPT API with Fastload, MultiLoad, or Multi-Statement insert.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: *driver_name timestamp*

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", BULKLOAD= LIBNAME option, BULKLOAD= data set option, MULTILOAD= data set option , MULTISTMT= data set option, TPT= LIBNAME option, TPT= data set option, TPT_TRACE_LEVEL= data set option, TPT_TRACE_LEVEL_INF= data set option

Syntax

TPT_TRACE_OUTPUT=*integer*

Syntax Description

integer
specifies the name of the external file to use for tracing. The name must be a valid filename for the operating system.

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

When loading data to Teradata using Teradata PT API, diagnostic messages can be written to an external log file. If no name is specified for the log file and tracing is requested, a default name is generated using the name of the driver and a timestamp. If a name is specified using TPT_TRACE_OUTPUT, that file is used for trace messages. If the file already exists, it is overwritten. Please refer to the Teradata documentation for more details.

You can write diagnostic message to an external log file when loading data to Teradata using the TPT PT API. If you do not specify a name in TPT_TRACE_OUTPUT= for the log file and tracing is requested, a default name is generated using the name of the driver and the current timestamp. Otherwise, the name that you specify is used for tracing messages. See your Teradata documentation for details.

TPT_WORK_TABLE= Data Set Option

Specifies the name of the work table for SAS to use when using the TPT API with MultiLoad.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: table_name_WT

Data source: Teradata

See: "Maximizing Teradata Load Performance ", "Using the TPT API ", MULTILOAD= data set option , TPT= LIBNAME option, TPT= data set option, TPT_ERROR_TABLE_1= data set option, TPT_ERROR_TABLE_2= data set option, TPT_LOG_TABLE= data set option

Syntax

TPT_WORK_TABLE=*valid_teradata_table_name*

Syntax Description

valid_teradata_table_name
specifies the name of the work table for SAS to use when using the TPT API to load data with MultiLoad.

Details

To use this option, you must first set TPT=YES. This option is valid only when using the TPT API.

MultiLoad inserts that use the TPT API require a work table. If you do not specify a work table, Teradata appends "_WT" to the name of the target table to load and uses it as the work table by default. You can override this name by setting TPT_WORK_TABLE=. If you do this and the load step fails, you must specify the same name when restarting.

The name that you specify in TPT_WORK_TABLE= must be unique. It cannot be the name of an existing table unless it is in a restart scenario.

Example: Specify a Different Name for the Work Table

In this example, a different name is provided for the work table that MultiLoad uses with the TPT API.

```
libname tera teradata user=testuser pw=testpw;
/* Load using Multiload TPT.  Use alternate names for the work table. */
data tera.testdata(MULTILOAD=YES TPT_WORK_TABLE=worktab);
i=1;output; i=2;output;
run;
```

TRAP151= Data Set Option

Enables removal of columns that cannot be updated from a FOR UPDATE OF clause so that update of columns can proceed as normal.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: NO

Data source: DB2 under z/OS

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

Example 1: SAS Log for TRAP151=YES

In this example, DB2DBUG is turned on so that you can see what occurs when 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.
```

Example 2: SAS Log for TRAP151=NO

The next example features the same code with TRAP151 turned off:

```
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)

Default: LIBNAME setting

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, MySQL, ODBC, OLE DB, Oracle, Sybase, Teradata

See: UPDATE_ISOLATION_LEVEL= LIBNAME option, UPDATE_LOCK_TYPE= LIBNAME optionUPDATE_LOCK_TYPE= data set option. See also locking information for DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Teradata.

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 degree to which:

- the rows that are read and updated by the current application are available to other concurrently executing applications.
- update activity of other concurrently executing application processes can affect the current application.

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

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)

Default: LIBNAME setting

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: UPDATE_ISOLATION_LEVEL= LIBNAME option, UPDATE_ISOLATION_LEVEL= data set option, UPDATE_LOCK_TYPE= LIBNAME option. See also locking information for DB2 under UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, OLE DB, Oracle, Sybase, Teradata.

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 UNIX and PC Hosts, DB2 under z/OS, Microsoft SQL Server, ODBC, Oracle, 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

If you omit UPDATE_LOCK_TYPE=, you receive 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.

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

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 a different user has locked the DBMS resource.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: LIBNAME setting

Data source: Teradata

See: UPDATE_MODE_WAIT= LIBNAME option. See also "Locking in the Teradata Interface".

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 standard locking is undesirable.

For more information, see the locking topic in the Teradata section.

UPDATE_SQL= Data Set Option

Determines which method to use to update and delete rows in a data source.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Default: LIBNAME setting

Data source: Microsoft SQL Server, ODBC

See: INSERT_SQL= data set option, UPDATE_SQL= LIBNAME option

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= data set option.

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)

Default: LIBNAME setting

Data source: Oracle

See: UPDATEBUFF= LIBNAME option

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).

Chapter 12

Macro Variables and System Options for Relational Databases

Introduction to Macro Variables and System Options

This section describes system options on page 377 and macro variables on page 375 that you can use with SAS/ACCESS software. It describes only those components of the macro facility that depend on SAS/ACCESS engines. Most features of the SAS macro facility are portable.

For more information about the macro facility, see *SAS Macro Language: Reference*. For more information about SAS system options, see *SAS System Options: Reference*.

Macro Variables for Relational Databases

SYSDBMSG, SYSDBRC, SQLXMSG, and SQLXRC are automatic SAS macro variables. The SAS/ACCESS engine and your DBMS determine their values. Initially, SYSDBMSG and SQLXMSG are blank, and 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 these 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 variables anywhere while you are accessing DBMS data. Only one set of macro variables is provided, however. So 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 statement below, %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 this 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 this output.

Log 12.1 *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");
```

Here is an example.

```
data_null_;
msg=symget("SYSDBMSG");
put msg;
run;
```

The SQL pass-through facility generates return codes and error messages that are available to you through these SAS macro variables:

SQLXMSG
: contains DBMS-specific error messages.

SQLXRC
contains DBMS-specific error codes.

You can use SQLXMSG and SQLXRC only through explicit pass-through with the SQL pass-through facility. See Return Codes on page 403.

You can print the contents of SQLXMSG and SQLXRC in the SAS log by using the %PUT macro. SQLXMSG is reset to a blank string, and SQLXRC is reset to 0 when any SQL pass-through facility statement is executed.

System Options for Relational Databases

Available System Options

Here are the available systems options.

No SAS/ACCESS interface support is available for the REPLACE= system option. (See *SAS System Options: Reference*.)

SAS System Options	Default
DB2CATALOG=	SYSIBM
DBFMTIGNORE=	NODBFMTIGNORE
DBIDIRECTEXEC	NODBIDIRECTEXEC
DBSLICEPARM=	THREADED_APPS,2 [DB2 under z/OS, Oracle, and Teradata] THREADED_APPS,2 or THREADED_APP,3 [DB2 under UNIX and PC Hosts, HP Neoview, Informix, Microsoft SQL Server, ODBC, and Sybase, Sybase IQ]
DBSRVTP=	NONE
REPLACE=	No SAS/ACCESS interface support is available. See *SAS System Options: Reference*.
SASTRACE=	none
SASTRACELOC=	stdout
SQLGENERATION=	Specifies whether and when SAS procedures generate SQL for in-database processing of source data.
SQLMAPPUTTO=	Specifies whether the PUT function is mapped to the SAS_PUT() function for a database, possible also where the SAS_PUT() function is mapped.
VALIDVARNAME=	Controls the type of SAS variable names that can be used or created during a SAS session.

Dictionary

DB2CATALOG= System Option

Overrides the default owner of DB2 catalog tables.

Valid in: OPTIONS statement

Category: Databases: DB2

Default: SYSIBM

Restriction: This option applies to only the local DB2 subsystem. So when you set the LOCATION= or SERVER= connection option in the LIBNAME statement, the SAS/ACCESS engine always uses SYSIBM as the default value.

Data source: DB2 under z/OS

See: LOCATION= connection option, SERVER= connection option

Syntax

DB2CATALOG=SYSIBM | *catalog-owner*

Syntax Description

SYSIBM
specifies the default catalog owner.

catalog-owner
specifies a different catalog owner from the default.

Details

The default value for this option is initialized when SAS is installed. You can override the default only when these conditions are met:

- SYSIBM cannot be the owner of the catalog that you want to access.
- Your site must have a shadow catalog of tables (one to which all users have access).
- You must set DB2CATALOG= in the restricted options table and then rebuild the table.

DBFMTIGNORE= System Option

Specifies whether to ignore numeric formats.

Valid in: configuration file, SAS invocation, OPTIONS statement,SAS System Options window

Category: Files: SAS Files, Input control: Data processing

Default: NODBFMTIGNORE

Restriction: This option pertains only to SAS formats that are numeric.

Data source: Teradata

See: SQL_FUNCTIONS= LIBNAME option, *SAS In-Database Products: Administrator's Guide*

Syntax

DBFMTIGNORE | **NODBFMTIGNORE**

Syntax Description

DBFMTIGNORE
specifies that numeric formats are ignored and FLOAT data type is created.

NODBFMTIGNORE
specifies that numeric formats are used.

Details

You normally use numeric formats to specify a database data type when processing output. SAS takes all nonnumeric formats (such as date, time, datetime, and char) as hints when it processes output. So use this option to ignore numeric formats and create a FLOAT data type instead. For example, the SAS/ACCESS engine creates a table with a column type of INT for a SAS variable with a format of 5.0.

DBIDIRECTEXEC= System Option

Lets the SQL pass-through facility optimize handling of SQL statements by passing them directly to the database for execution.

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: Files: External files, System administration: Performance

Default: NODBIDIRECTEXEC

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, DB2 under z/OS, Greenplum, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Sybase IQ, Teradata

See: DIRECT_EXE= LIBNAME option

Syntax

DBIDIRECTEXEC | **NODBIDIRECTEXEC**

Syntax Description

DBIDIRECTEXEC
indicates that the SQL pass-through facility optimizes handling of SQL statements by passing them directly to the database for execution, which optimizes performance. Using this option, you can process CREATE TABLE AS SELECT and DELETE statements.

NODBIDIRECTEXEC
indicates that the SQL pass-through facility does not optimize handling of SQL statements.

Details

This option applies to all hosts and all SAS/ACCESS engines. You can use it to significantly improve CPU, input, and output performance.

Certain database-specific criteria exist for passing SQL statements to the DBMS. These criteria are the same as the criteria that exist for passing joins. For details for your DBMS, see "Passing Joins to the DBMS" on page 47 and "When Passing Joins to the DBMS Will Fail" on page 49.

When these criteria are met, a database can process the **CREATE TABLE *table-name* AS SELECT** statement in a single step instead of as three separate statements (CREATE, SELECT, and INSERT). For example, if multiple librefs point to different data sources, the statement is processed normally, regardless of how you set this option. However, when you enable it, PROC SQL sends the CREATE TABLE AS SELECT statement to the database.

You can also send a DELETE statement directly to the database for execution, which can improve CPU, input, and output performance.

Once a system administrator sets the default for this option globally, users can override it within their own configuration file.

When you specify DBIDIRECTEXEC=, PROC SQL can pass this statement directly to the database:

```
CREATE TABLE table-name AS SELECT query
```

Before an SQL statement can be processed, all librefs that are associated with the statement must reference compatible data sources. For example, a CREATE TABLE AS SELECT statement that creates an Oracle table by selecting from a SAS table is not sent to the database for execution because the data sources are not compatible. The libref must also use the same database server for all compatible data sources.

Examples

Example 1: Create a Temporary Table

This example creates a temporary table from a SELECT statement using the DBIDIRECTEXEC system option.

```
libname lib1 db2 user=andy password=andypwd datasrc=sample connection=global;
libname lib2 db2 user=mike password=mikepwd datasrc=sample
    connection=global dbmstemp=yes;
data lib1.tab1;
     a=1;
     b='one';
run;
options dbidirectexec sastraceloc=saslog;
proc sql;
     create table lib2.tab1 as
     select * from lib1.tab1;
quit;
```

Example 2: Reference One Database, Use Different Schemas

In this example, two librefs point to the same database server but use different schemas.

```
libname lib1 db2 user=henry password=henrypwd datasrc=sample;
libname lib2 db2 user=scott password=scottpwd datasrc=sample;
data lib1.tab1;
  a=1;
  b='one';
run;
options dbidirectexec sastraceloc=saslog;
proc sql;
  create table lib2.tab2 as
  select * from lib1.t1;
quit;
```

Example 3: Pass a Statement Directly to the Database

This example shows how a statement can be passed directly to the database for execution, if you specify DBIDIRECTEXEC.

```
libname company oracle user=scott pw=tiger path=mydb;
proc sql;
     create table company.hr_tab as
     select * from company.emp
  where deptid = 'HR';
quit;
```

DBSLICEPARM= System Option

Controls the scope of DBMS threaded Reads and the number of threads.

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: System administration: Performance

Default: THREADED_APPS,2 (DB2 under z/OS, Oracle, Teradata), THREADED_APPS,2*or* 3 (DB2 under UNIX and PC Hosts, HP Neoview, Informix, Microsoft SQL Server, ODBC, Sybase, Sybase IQ)

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, HP Neoview, Informix, Microsoft SQL Server, ODBC, Oracle, Sybase, Sybase IQ, Teradata

See: DBSLICE= data set option, DBSLICEPARM= LIBNAME option, DBSLICEPARM= data set option, SLEEP= LIBNAME option, SLEEP= data set option, TENACITY= LIBNAME option, TENACITY= data set option

Syntax

DBSLICEPARM=NONE | THREADED_APPS | ALL

DBSLICEPARM=(NONE | THREADED_APPS | ALL<*max-threads*>)

DBSLICEPARM=(NONE | THREADED_APPS | ALL<, *max-threads*>)

Syntax Description

NONE
disables DBMS threaded Read. SAS reads tables on a single DBMS connection, as it did with SAS 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
a positive integer value that specifies the maximum number of connections per table read. The second parameter of the option determines the number of threads to read the table in parallel. The number of partitions on the table determine the number of connections made to the Oracle server for retrieving rows from the table. A partition or portion of the data is read on each connection. The combined rows across all partitions are the same regardless 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 saved on the SAS step. See the DBMS-specific reference section for details about partitioned reads before using this parameter.

Details

You can use DBSLICEPARM= in numerous locations. The usual rules of option precedence apply: A table option has the highest precedence, then a LIBNAME option, and so on. 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 whether threaded Reads are actually generated, turn on SAS tracing and run a program, as shown in this 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.

DB2 under UNIX and PC Hosts, Informix, Microsoft SQL Server, ODBC, Sybase, Sybase IQ
The default thread number depends on whether an application passes in the number of threads (CPUCOUNT=) and whether the data type of the column that was selected for purposes of data partitioning is binary.

Examples

Example 1: Disable Threaded Read for All SAS Users

Here is how to use DBSLICEPARM= in a SAS configuration file entry in Windows to turn off threaded Reads for all SAS users.

```
-dbsliceparm NONE
```

Example 2: Enable Threaded Reads for Read-Only References

Here is how you can use DBSLICEPARM= as a z/OS invocation option to turn on threaded Reads for read-only references to DBMS tables throughout a SAS job.

```
sas o(dbsliceparm=ALL)
```

Example 3: Increase Maximum Threads (as a SAS Global Option)

```
option dbsliceparm=(threaded_apps,3);
```

Example 4: Enable Threaded Reads for References Using a Particular Libref

You can use DBSLICEPARM= as a LIBNAME option to turn on threaded Reads for read-only table references that use this particular libref, as shown in this example.

```
libname dblib oracle user=scott password=tiger dbsliceparm=ALL;
```

Example 5: Enable Threaded Reads as a Table-Level Option

Here is 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;
```

DBSRVTP= System Option

Specifies whether SAS/ACCESS engines hold or block the originating client while making performance-critical calls to the database.

Valid in: SAS invocation

Category: Communications: Networking & encryption

Default: NONE

Data source: DB2 under UNIX and PC Hosts, Informix, Netezza, ODBC, OLE DB, Oracle, Sybase, Teradata

Syntax

DBSRVTP='ALL' | 'NONE' | '(*engine-name(s)*)'

Syntax Description

ALL
indicates that SAS does not use any blocking operations for all underlying SAS/ACCESS engines that support this option.

NONE
indicates that SAS uses standard blocking operations for all SAS/ACCESS engines.

engine-name(s)
indicates that SAS does not use any blocking operations for the specified SAS/ACCESS engines. You can specify one or more engine names. If you specify more than one, separate them with blank spaces and enclose the list in parentheses.

db2 (under UNIX and PC Hosts only)

informix

netezza

odbc (indicates that SAS uses non-blocking operations for SAS/ACCESS ODBC and Microsoft SQL Server interfaces)

oledb

oracle

sybase

teradata (not supported on z/OS)

Details

This option applies only when SAS is called as a server responding to multiple clients.

You can use this option to help throughput of the SAS server because it supports multiple simultaneous execution streams, if the server uses certain SAS/ACCESS interfaces. Improved throughput occurs 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 client 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. When specifying multiple values, 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 z/OS (Oracle only). On some of these hosts, you can call SAS with the -SETJMP system option. Setting -SETJMP disables the DBSRVTP= option.

Example

Each of these examples call 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: configuration file, SAS invocation, OPTIONS statement

Default: none

Data source: DB2 under UNIX and PC Hosts, DB2 under z/OS, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Teradata

Tip: You can also use more than one SASTRACE= option at a time (for example, `SASTRACE=',,d,d'`).

See: SASTRACELOC= system option, "Generating Trace Information for Threaded Reads" on page 60

Syntax

SASTRACE= ',,,d' | ' ,,d,' | ' d,' | ',,,db' | ' ,,,s' | ' ,,,sa' | ',,t,'

Syntax Description

',,,d'

specifies that all SQL statements that are sent to the DBMS are sent to the log. Here are the applicable statements:

SELECT	DELETE
CREATE	SYSTEM CATALOG
DROP	COMMIT
INSERT	ROLLBACK
UPDATE	

For engines that do not generate SQL statements, API calls and all parameters are sent to the log.

',,d,'

specifies that all routine calls are sent to the log. All function enters, exits, and pertinent parameters and return codes are traced when you select this option. The information varies from engine to engine, however.

This option is most useful if you have 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. This information varies from engine to engine, however.

This option is most useful if you have a problem and need to send a SAS log to technical support for troubleshooting.

',,,db'

specifies that only a brief version of all SQL statements that the ',,,d' option normally generates are sent to the log.

',,,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 that is made to the DBMS is sent to the log along with a summary.

',,t,'
specifies that all threading information is sent to the log. Here is the information that it includes:

- number of threads that are spawned
- number of observations that each thread contains
- exit code of the thread, if it fails

Details

SASTRACE= and SASTRACELOC= behavior is specific to SAS/ACCESS software. SASTRACE= is a very powerful tool to use when you want to see the commands that SAS/ACCESS sent to your DBMS. SASTRACE= output is DBMS-specific. However, most SAS/ACCESS engines show you statements like SELECT or COMMIT as the DBMS processes them for the SAS application. These details can help you manage SASTRACE= output in your DBMS.

- When using SASTRACE= on PC platforms, you must also specify SASTRACELOC=.
- Here is how to turn SAS tracing off:

  ```
  options sastrace=off;
  ```

- Log output is much easier to read if you specify NOSTSUFFIX. Because this code is entered without specifying the option, the resulting log is longer and harder to decipher.

  Note: NOSTSUFFIX is not supported on z/OS.

  ```
  options sastrace=',,,d' sastraceloc=saslog;
  proc print data=mydblib.snow_birthdays;
  run;
  ```

  Here is the resulting log.

  ```
  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
  ```

  Use NOSTSUFFIX to make the log file much easier to read.

  ```
  options sastrace=',,,d' sastraceloc=saslog nostsuffix;
  proc print data=mydblib.snow_birthdays;
  run;
  ```

  Here is the resulting log.

```
ORACLE_1: Prepared:
SELECT * FROM scott.SNOW_BIRTHDAYS
12 proc print data=mydblib.snow_birthdays; run;
ORACLE_2: Executed:
Prepared statement ORACLE_1
```

Examples

Example 1: Use SQL Trace ',,,d'

These examples use 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;
```

Examples are based on this data set.

```
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;
```

Output for this ',,,d' example is written to the SAS log, as specified in the SASTRACELOC=SASLOG option.

Output 12.1 *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 datasets library=mydblib;
  delete 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: Use 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;
```

Output is written to the SAS log, as specified in the SASTRACELOC=SASLOG option.

Output 12.2 *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: Use 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;
```

Output is written to the SAS log, as specified in the SASTRACELOC=SASLOG option.

Output 12.3 *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: Use Brief SQL Trace ',,,db'

```
options sastrace=',,,db' sastraceloc=saslog nostsuffix;
libname mydblib oracle user=scott password=tiger path=oraclev9;
data mydblib.employee1;
   set mydblib.employee;
run;
```

Output is written to the SAS log, as specified in the SASTRACELOC=SASLOG option.

Output 12.4 *SAS Log Output from the SASTRACE= ',,,db' System Option*

```
ORACLE_23: Prepared: on connection 2
SELECT * FROM EMPLOYEE
 19?
ORACLE_24: Prepared: on connection 3
SELECT * FROM EMPLOYEE1
NOTE: SAS variable labels, formats, and lengths are not written to DBMS
      tables.
ORACLE_25: Executed: on connection 4
CREATE TABLE EMPLOYEE1(NAME VARCHAR2 (20),ID NUMBER (5),CITY VARCHAR2
(15),SALARY NUMBER ,DEPT NUMBER (5))
ORACLE_26: Executed: on connection 2
SELECT statement  ORACLE_23
ORACLE_27: Prepared: on connection 4
INSERT  INTO EMPLOYEE1 (NAME,ID,CITY,SALARY,DEPT) VALUES
(:NAME,:ID,:CITY,:SALARY,:DEPT)
**NOTE**: ORACLE_27 on connection 4
The Execute statements associated with
this Insert statement are suppressed due to SASTRACE brief
setting-SASTRACE=',,,bd'. Remove the 'b' to obtain full trace.
NOTE: There were 17 observations read from the data set MYDBLIB.EMPLOYEE.
```

Example 5: Use 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;
```

Output is written to the SAS log, as specified in the SASTRACELOC=SASLOG option.

Output 12.5 *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 6: Use 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;
```

Output is written to the SAS log, as specified in the SASTRACELOC=SASLOG option.

Output 12.6 *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 7: Use 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;
```

Output is written to the SAS log, as specified in the SASTRACELOC=SASLOG option.

Output 12.7 *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_BaaaaaAYS.
NOTE: PROCEDURE PRINT used (Total process time):
       real time           1.12 seconds
       cpu time            0.17 seconds
```

SASTRACELOC= System Option

Prints SASTRACE= information to a specified location.

Valid in: configuration file, SAS invocation, OPTIONS statement

Default: stdout

Data source: DB2 UNIX/PC, HP Neoview, Informix, Microsoft SQL Server, MySQL, Netezza, ODBC, OLE DB, Oracle, Sybase, Teradata

See: SASTRACE= system option

Syntax

SASTRACELOC=stdout | SASLOG | FILE *'path-and-filename'*

Details

SASTRACELOC= lets you specify where to put the trace messages that SASTRACE= generates. By default, output goes to the default output location for your operating environment. Specify SASTRACELOC=SASLOG to send output to a SAS log.

This option and its values might differ for each host.

Example: Specify a Location for the Trace Log

This example writes trace information to the TRACE.LOG file in the work directory on the C drive on a PC platform.

```
options sastrace=',,,d' sastraceloc=file 'c:\work\trace.log';
```

SQLGENERATION= System Option

Specifies whether and when SAS procedures generate SQL for in-database processing of source data.

Valid in: configuration file, SAS invocation, OPTIONS statement, SAS System Options window

Category: System administration: Performance

Default: (NONE DBMS='ASTER DB2 GREENPLM NETEZZA ORACLE TERADATA')

Restriction: For DBMS= and EXCLUDEDB= values, the maximum length of an engine name is eight characters. For the EXCLUDEPROC= value, the maximum length of a procedure name is 16 characters. An engine can appear only once, and a procedure can appear only once for a given engine.

Data source: Aster *n*Cluster, DB2 under UNIX and PC Hosts, Greenplum, Netezza, Oracle, Teradata

See: SQLGENERATION= LIBNAME option (includes examples), and also “Running In-Database Procedures” in *SAS In-Database Products: User’s Guide*

Syntax

SQLGENERATION=<(>NONE | DBMS <DBMS='*engine1 engine2…enginen* '>
<EXCLUDEDB=*engine* | '*engine1…enginen*'>
<EXCLUDEPROC="*engine*='*proc1…procn*'
enginen='*proc1…procn*' "><)>

SQLGENERATION=" "

Syntax Description

NONE
prevents those SAS procedures that are enabled for in-database processing from generating SQL for in-database processing. This is a primary state.

DBMS
allows SAS procedures that are enabled for in-database processing to generate SQL for in-database processing of DBMS tables through supported SAS/ACCESS engines. This is a primary state.

DBMS='*engine1…enginen*'
specifies one or more SAS/ACCESS engines. It modifies the primary state.

EXCLUDEDB=*engine* | '*engine1…enginen*'
prevents SAS procedures from generating SQL for in-database processing for one or more specified SAS/ACCESS engines.

EXCLUDEPROC="*engine*='*proc1…procn*' *enginen*='*proc1…procn*' "
identifies engine-specific SAS procedures that you do not want to run inside the database.

" "
resets the value to the default that was shipped.

Details

Use this option with such procedures as PROC FREQ to indicate that SQL is generated for in-database processing of DBMS tables through supported SAS/ACCESS engines.

You must specify NONE, DBMS, or both. One or both of these arguments indicates the primary state.

The maximum length of the option value is 4096. Also, parentheses are required when this option value contains multiple keywords.

Not all procedures support SQL generation for in-database processing for every engine type. If you specify a setting that is not supported, an error message indicates the level of SQL generation that is not supported, and the procedure can reset to the default so that source table records can be read and processed within SAS. If this is not possible, the procedure ends and sets SYSERR= as needed.

You can specify different SQLGENERATION= values for the DATA= and OUT= data sets by using different LIBNAME statements for each of these data sets.

Here is how SAS/ACCESS handles precedence.

Table 12.1 *Precedence of Values for SQLGENERATION= LIBNAME and System Options*

<table>
<tr><th>LIBNAME Option</th><th>PROC EXCLUDE on System Option?</th><th>Engine Type</th><th>Engine Specified on System Option</th><th>Resulting Value</th><th>From (option)</th></tr>
<tr><td>not set
NONE
DBMS</td><td>yes</td><td rowspan="5">database interface</td><td rowspan="3">NONE
DBMS</td><td>NONE
EXCLUDEDB</td><td>system</td></tr>
<tr><td>NONE</td><td rowspan="7">no</td><td>NONE</td><td rowspan="2">LIBNAME</td></tr>
<tr><td>DBMS</td><td>DBMS</td></tr>
<tr><td>not set</td><td>NONE</td><td>NONE</td><td>system</td></tr>
<tr><td rowspan="2">NONE
DBMS</td><td>DBMS</td><td>DBMS</td><td rowspan="2">LIBNAME</td></tr>
<tr><td>no SQL generated for this database host or database version</td><td rowspan="3">NONE
DBMS</td><td rowspan="3">NONE</td></tr>
<tr><td>not set</td><td rowspan="2">Base</td><td>system</td></tr>
<tr><td>NONE
DBMS</td><td>LIBNAME</td></tr>
</table>

Example

Here is the default that is shipped with the product.

```
options sqlgeneration='';
proc options option=sqlgeneration
run;
```

SAS procedures generate SQL for in-database processing for all databases except DB2 in this example.

```
options sqlgeneration='';
options sqlgeneration=(DBMS EXCLUDEDB='DB2');
proc options option=sqlgeneration;
run;
```

In this example, in-database processing occurs only for Teradata. SAS procedures that are run on other databases do not generate SQL for in-database processing.

```
options sqlgeneration='';
options SQLGENERATION=(NONE DBMS='Teradata');
proc options option=sqlgeneration;
run;
```

For this example, SAS procedures generate SQL for Teradata and Oracle in-database processing. However, no SQL is generated for PROC1 and PROC2 in Oracle.

```
options sqlgeneration='';
Options SQLGENERATION = (NONE DBMS='Teradata Oracle'
   EXCLUDEPROC="oracle='proc1 proc2'");
proc options option=sqlgeneration;
run;
```

SQLMAPPUTTO= System Option

Specifies whether the PUT function is mapped to the SAS_PUT() function for a database, possible also where the SAS_PUT() function is mapped.

Valid in: configuration file, SAS invocation, OPTIONS statement

Category: Files: SAS files

Default: SAS_PUT

Data source: DB2 under UNIX and PC Hosts, Netezza, Teradata

See: "SQL_FUNCTIONS= LIBNAME Option", *SAS In-Database Products: User's Guide*

Syntax

SQLMAPPUTTO= NONE | SAS_PUT | (*database*.SAS_PUT)

Syntax Description

NONE
specifies to PROC SQL that no PUT mapping is to occur.

SAS_PUT
specifies that the PUT function be mapped to the SAS_PUT() function.

***database*.SAS_PUT**
specifies the database name.

TIP It is not necessary that the format definitions and the SAS_PUT() function reside in the same database as the one that contains the data that you want to format. You can use the *database*.SAS_PUT argument to specify the database where the format definitions and the SAS_PUT() function have been published.

TIP The database name can be a multilevel name and it can include blanks.

Requirement: If you specify a database name, you must enclose the entire argument in parentheses.

Details

The format publishing macros deploy or publish, the PUT function implementation to the database as a new function named SAS_PUT(). The format publishing macros also publish both user-defined formats and formats that SAS supplies that you create using PROC FORMAT. The SAS_PUT() function supports the use of SAS formats, and you can use it in SQL queries that SAS submits to the database so that the entire SQL query can be processed inside the database. You can also use it in conjunction with in-database procedures.

You can use this option with the SQLREDUCEPUT=, SQLREDUCEPUTOBS, and SQLREDUCEPUTVALUES= system options. For more information about these options, see *SAS SQL Procedure User's Guide*.

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

Category: Files: SAS files

Default: V7

See: "Introduction to SAS/ACCESS Naming"

Syntax

VALIDVARNAME=V7 | UPCASE | ANY

Required Arguments

VALIDVARNAME=V7
indicates that a DBMS column name is changed to a valid SAS name, following these rules. This is the default value for SAS 7 and later.

- 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.

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

This example shows how the SQL pass-through facility works with VALIDVARNAME=V6.

```
options validvarname=v7;
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;
```

Output from this example would show that "Amount Budgeted$" becomes AMOUNT_B and "Amount Spent$" becomes AMOUNT_S.

Chapter 13
The SQL Pass-Through Facility for Relational Databases

About SQL Procedure Interactions

Overview of SQL Procedure Interactions with SAS/ACCESS

The SQL procedure implements structured query language (SQL) for SAS software. For information about PROC SQL , see the *Base SAS Procedures Guide*. Here is how you can use SAS/ACCESS software for relational databases for PROC SQL interactions.

- You can assign a libref to a DBMS using the SAS/ACCESS LIBNAME statement and reference the new libref in a PROC SQL statement to query, update, or delete DBMS data. (See "Overview: LIBNAME Statement for Relational Databases " on page 93.)
- 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 97.)
- You can send DBMS-specific SQL statements directly to a DBMS using an extension to PROC SQL called the SQL pass-through facility. (See "Syntax: SQL Pass-Through Facility for Relational Databases" on page 403.)

SQL Pass-Through Facility

The SQL pass-through facility uses SAS/ACCESS to connect to a DBMS and to send statements directly to the DBMS for execution. As an alternative to the SAS/ACCESS

LIBNAME statement, this facility lets you use the SQL syntax of your DBMS. It supports any SQL that is not ANSI-standard that your DBMS supports.

Not all SAS/ACCESS interfaces support this feature, however. To determine whether it is available in your environment, see “Introduction” on page 77.

Here are the tasks that you can complete by using the SQL pass-through facility.

- Establish and terminate connections with a DBMS using its CONNECT and DISCONNECT statements.
- Send dynamic, non-query, DBMS-specific SQL statements to a DBMS using its EXECUTE statement.
- Retrieve data directly from a DBMS using its CONNECTION TO component in the FROM clause of a PROC SQL SELECT statement.

You can use SQL 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 you use the view in a SAS program, SAS can establish the appropriate connection to the DBMS.

DBMS Interface Specifics for the SQL Pass-Through Facility

See the section for your interface for DBMS-specific details about the SQL pass-through facility.

- Aster nCluster on page 422
- DB2 under UNIX and PC Hosts on page 441
- DB2 under z/OS on page 473
- Greenplum on page 521
- Hadoop on page 422
- HP Neoview on page 562
- Informix on page 580
- Microsoft SQL Server on page 600
- MySQL on page 613
- Netezza on page 629
- ODBC on page 652
- OLE DB on page 680
- Oracle on page 704
- Sybase on page 737
- Sybase IQ on page 764
- Teradata on page 787

Syntax: SQL Pass-Through Facility for Relational Databases

Overview

The syntax section presents the syntax for the SQL 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 SQL 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 these SAS macro variables:

SQLXRC
: contains the DBMS return code that identifies the DBMS error.

SQLXMSG
: contains descriptive information about the DBMS error that the DBMS generates.

The contents of the SQLXRC and SQLXMSG macro variables are printed in the SAS log using the %PUT macro. They are reset after each SQL pass-through facility statement has been executed.

See "Macro Variables for Relational Databases" on page 375 for more information about these return codes.

Dictionary

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*>)> ;

Required Argument

dbms-name
identifies the database management system to which you want to connect. You must specify the DBMS name for your SAS/ACCESS interface on page 402. You can also specify an optional alias.

Optional Arguments

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 SQL pass-through connection.

database-connection-arguments
specifies the DBMS-specific arguments that PROC SQL needs to connect to the DBMS. These arguments are optional for most databases. However, if you include any, you must enclose them in parentheses. See the documentation for your SAS/ACCESS interface on page 402 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 let the SQL pass-through facility use some of the connection management features of the LIBNAME statement. Although these arguments are optional, if you include any, you must enclose it in parentheses.

- CONNECTION= on page 404
- CONNECTION_GROUP= on page 405
- DBCONINIT= on page 405
- DBCONTERM= on page 405
- DBGEN_NAME= on page 406
- DBMAX_TEXT= on page 406
- DBPROMPT= on page 406
- DEFER= on page 406
- VALIDVARNAME= on page 406

Note: In addition to the arguments listed here, several other LIBNAME options are available for use with the CONNECT statement. See the SQL pass-through facility reference section for your SAS/ACCESS interface to determine which LIBNAME options are available in the SQL pass-through facility for your DBMS. When used with the SQL pass-through facility CONNECT statement, these options have the same effect as they do in a LIBNAME statement.

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 SQL 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 SQL pass-through statements that use this alias share the connection. SHARED is the default value for CONNECTION=.

In this 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. 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 SQL pass-through facility.

Default: none

By specifying the name of a connection group, you can share one DBMS connection among several 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 is 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: 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 128 for more 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 before connecting to the DBMS.

  Default: NO

  Interaction: DEFER= LIBNAME option

  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= LIBNAME option to determine when the prompt window appears. If DEFER=NO, the DBPROMPT window appears when the CONNECT statement is executed. If DEFER=YES, the DBPROMPT window appears 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: NO

  If DEFER=YES, the connection to the DBMS occurs when the first SQL pass-through statement is executed. If DEFER=NO, the connection to the DBMS occurs when the CONNECT statement occurs.

VALIDVARNAME=V6
: indicates that only SAS 6 variable names are considered valid. Specify this connection argument if you want the SQL pass-through facility to operate in SAS 6 compatibility mode.

  By default, DBMS column names are changed to valid SAS names, following these 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 does in SAS 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 SQL pass-through connection.

This example shows how the SQL 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;
```

So using it as a connection argument saves you coding later.

Details

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.

Follow these steps to connect to a DBMS using the SQL pass-through facility.

1. Initiate a PROC SQL step.
2. Use the SQL pass-through facility CONNECT statement, identify the DBMS (such as Oracle or DB2), and assign an (optional) alias.
3. Specify any attributes for the connection such as SHARED or UNIQUE.
4. Specify any arguments that are needed to connect to the database.

The CONNECT statement is optional for some DBMSs. However, if you do not specify it, the default values for all database connection arguments are used.

Any return code or message that the DBMS generates is available in the SQLXRC and SQLXMSG macro variables after the statement executes. See "Macro Variables for Relational Databases" on page 375 for more information about these macro variables.

Example: CONNECT Statement Example

This example connects to a Sybase server and assigns the alias SYBCON1 to it. Sybase is a case-sensitive database, so 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 by using default settings. See the documentation for your SAS/ACCESS interface on page 402 for more information.

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)*

Required 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 on page 402 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.

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 enter 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 SQL pass-through facility. Furthermore, if the stored procedure contains more than one query, only the first query is processed.

Details

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;

You can use CONNECTION TO in any FROM clause, including those in nested queries —that is, subqueries.

You can store an SQL 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. So 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 403.) 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 on page 402 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 "SAS Names and Support for DBMS Names" on page 13 for more information.

Examples

Example 1: Send an Oracle SQL Query to the Oracle Database

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 by using the SQL syntax of your DBMS.

This example sends an Oracle SQL query (highlighted below) 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 375 for more information.

Example 2: Name and Store the Query as an SQL View

This example gives the query a name and stores it as the SQL view samples.

```
libname samples 'SAS-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;
```

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*

Required Arguments

dbms-name
specifies the database management system from which you want to disconnect. You must either specify the DBMS name for your SAS/ACCESS interface on page 402 or use an alias in the DISCONNECT statement.

Note: If you used the CONNECT statement to connect to the DBMS, the DBMS name or alias in the DISCONNECT statement must match what you specified in the CONNECT statement.

alias
specifies an alias that was defined in the CONNECT statement.

Details

The DISCONNECT statement ends the connection with the DBMS. If you do not include the DISCONNECT statement, SAS performs an implicit DISCONNECT 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 375 for more information about these macro variables.

Example

To exit the SQL pass-through facility, use the facilities DISCONNECT statement and then QUIT the PROC SQL statement. This 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*;

Required Arguments

(dbms-specific-sql-statement)
a dynamic non-query, DBMS-specific SQL statement. This argument is required and must be enclosed in parentheses. The SQL statement might be case sensitive, depending on your DBMS, and it is passed to the DBMS exactly as you enter it.

On some DBMSs, this argument can be a DBMS stored procedure. However, stored procedures with output parameters are not supported in the SQL 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 375 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 on page 402 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.)

Details

Details

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 403.) 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 on page 402 for details.

The EXECUTE statement cannot be stored as part of an SQL pass-through facility query in a PROC SQL view.

Useful Statements to Include in EXECUTE Statements

You can pass these statements to the DBMS by using the SQL pass-through facility EXECUTE statement.

Statement	Description
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.

Part 3

DBMS-Specific Reference

Chapter 14
SAS/ACCESS Interface to Aster *n*Cluster

Introduction to SAS/ACCESS Interface to Aster *n*Cluster

This section describes SAS/ACCESS Interface to Aster *n*Cluster. For a list of SAS/ACCESS features that are available for this interface, see "SAS/ACCESS Interface to Aster nCluster: Supported Features" on page 78.

LIBNAME Statement Specifics for Aster *n*Cluster

Overview

This section describes the LIBNAME statement options that SAS/ACCESS Interface to Aster *n*Cluster supports and includes examples. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing Aster *n*Cluster.

LIBNAME *libref* **aster** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`aster`
specifies the SAS/ACCESS engine name for the Aster *n*Cluster interface.

connection-options
provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. When you use the LIBNAME statement, you can connect to the Aster *n*Cluster database in several ways. Specify *only one* of these methods for each connection because they are mutually exclusive.

- SERVER=, DATABASE=, PORT=, USER=, PASSWORD=
- DSN=, USER=, PASSWORD=
- NOPROMPT=
- PROMPT=
- REQUIRED=

Here is how these options are defined.

SERVER=<'>*server-name*<'>
specifies the host name or IP address where the Aster *n*Cluster database is running. If the server name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

DATABASE=<'>*database-name*<'>
Alias: DB=

specifies the Aster *n*Cluster database that contains the tables and views that you want to access. If the database name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

PORT=*port*

specifies the port number that is used to connect to the specified Aster *n*Cluster database. If you do not specify a port, the default port 2406 is used.

USER=<'>*Aster* n*Cluster user-name*<'>

specifies the Aster *n*Cluster user name (also called the user ID) that you use to connect to your database. If the user name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

PASSWORD=<'>*Aster* n*Cluster password*<'>

Alias: PASS=, PW=, PWD=

specifies the password that is associated with your Aster *n*Cluster User ID. If the password contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

DSN=<'>*Aster* n*Cluster data-source*<'>

specifies the configured Aster *n*Cluster ODBC data source to which you want to connect. Use this option if you have existing Aster *n*Cluster ODBC data sources that are configured on your client. This method requires additional setup, either through the ODBC Administrator control panel on Windows platforms or through the odbc.ini file on UNIX platforms. So it is recommended that you use this connection method only if you have existing, functioning data sources that have been defined.

Requirement: If you use this method, you just specify the user ID and password on the LIBNAME statement, even if these are already specified in the ODBC data source.

NOPROMPT=<'>*Aster* n*Cluster ODBC-connection-options*<'>

specifies connection options for your data source or database. Separate multiple options with a semicolon. If you do not specify enough correct connection options, an error is returned. No dialog box is displayed to help you with the connection string.

PROMPT=<'> *Aster* n*Cluster ODBC-connection-options*<'>

specifies connection options for your data source or database. Separate multiple options with a semicolon. When connection succeeds, the complete connection string is returned in the SYSDBMSG macro variable. PROMPT= does not immediately try to connect to the DBMS. It instead displays a dialog box that contains the values that you entered in the PROMPT= connection string. You can edit values or enter additional values in any field before you connect to the data source. This option is not supported on UNIX platforms.

REQUIRED=<'>*Aster* n*Cluster ODBC-connection-options*<'>

specifies connection options for your data source or database. Separate multiple options with a semicolon. When connection succeeds, the complete connection string is returned in the SYSDBMSG macro variable. If you do not specify enough correct connection options, a dialog box prompts you for the connection options. REQUIRED= lets you modify only required fields in the dialog box. This option is not supported on UNIX platforms.

LIBNAME -options

define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to Aster *n*Cluster with the applicable default values. For more detail about these options, see “LIBNAME Options for Relational Databases” on page 99.

Table 14.1 SAS/ACCESS LIBNAME Options for Aster nCluster

Option	Default Value
ACCESS=	none
AUTHDOMAIN=	none
AUTOCOMMIT=	operation-specific
CONNECTION=	UNIQUE
CONNECTION_GROUP=	none
DBCOMMIT=	1000 when inserting rows; 0 when updating rows
DBCONINIT=	none
DBCONTERM=	none
DBCREATE_TABLE_OPTS=	none
DBGEN_NAME=	DBMS
DBINDEX=	YES
DBLIBINIT=	none
DBLIBTERM=	none
DBMAX_TEXT=	1024
DBMSTEMP=	NO
DBNULLKEYS=	YES
DBPROMPT=	NO
DBSASLABEL=	COMPAT
DEFER=	NO
DELETE_MULT_ROWS=	NO
DIMENSION=	NO
DIRECT_EXE=	none
DIRECT_SQL=	YES

Option	Default Value
IGNORE_READ_ONLY_COLUMNS =	NO
INSERTBUFF=	automatically calculated based on row length
LOGIN_TIMEOUT=	0
MULTI_DATASRC_OPT=	none
PARTITION_KEY=	none
PRESERVE_COL_NAMES=	see "Naming Conventions for Aster nCluster" on page 428
PRESERVE_TAB_NAMES=	see "Naming Conventions for Aster nCluster" on page 428
QUERY_TIMEOUT=	0
QUOTE_CHAR=	none
READBUFF=	automatically calculated based on row length
REREAD_EXPOSURE=	NO
SCHEMA=	none
SPOOL=	YES
SQL_FUNCTIONS=	none
SQL_FUNCTIONS_COPY=	none
SQLGENERATION=	DBMS
STRINGDATES=	NO
TRACE=	NO
TRACEFILE=	none
UPDATE_MULT_ROWS=	NO
USE_ODBC_CL=	NO
UTILCONN_TRANSIENT=	NO

Aster nCluster LIBNAME Statement Examples

In this first example, SERVER=, DATABASE=, USER=, and PASSWORD= are the connection options.

```
LIBNAME mydblib ASTER SERVER=npssrv1 DATABASE=test
     USER=netusr1 PASSWORD=netpwd1;

PROC Print DATA=mydblib.customers;
     WHERE state='CA';
run;
```

In this next example, the DSN= option, the USER= option, and the PASSWORD= option are connection options. The Aster *n*Cluster data source is configured in the ODBC Administrator Control Panel on Windows platforms. It is also configured in the odbc.ini file or a similarly named configuration file on UNIX platforms.

```
LIBNAME mydblib aster dsn=nCluster user=netusr1 password=netpwd1;

PROC Print DATA=mydblib.customers;
     WHERE state='CA';
run;
```

Here is how you can use the NOPROMPT= option.

```
libname x aster NOPROMPT="dsn=aster;";
libname x aster NOPROMPT="DRIVER=nCluster; server=192.168.28.100;
   uid=username; pwd=password; database=asterdb";
```

This example uses the PROMPT= option. Blanks are also passed down as part of the connection options. Therefore, the specified value must immediately follow the semicolon.

```
libname x aster PROMPT="DRIVER=nCluster;";
```

The REQUIRED= option is used in this example. If you enter all needed connection options, REQUIRED= does not prompt you for any input.

```
libname x aster REQUIRED="DRIVER=nCluster; server=192.168.28.100;
   uid=username;pwd=password; database=asterdb ;";
```

As shown above, when *asterdb* (contains a trailing blank) is specified as the database instead of *asterdb* (no trailing blank), this error results:

```
ERROR: CLI error trying to establish connection:
     ERROR:  Database asterdb  does not exist.
```

Data Set Options for Aster *n*Cluster

All SAS/ACCESS data set options in this table are supported for Aster *n*Cluster. Default values are provided where applicable. For details about this feature, see the Data Set Options for Relational Databases on page 207.

Table 14.2 *SAS/ACCESS Data Set Options for Aster nCluster*

Option	Default Value
BL_DATAFILE=	none
BL_DBNAME=	none
BL_DELETE_DATAFILE=	Yes

Option	Default Value
BL_DELIMITER=	\t (the tab symbol)
BL_HOST=	none
BL_OPTIONS=	none
BL_PATH=	none
BULKLOAD=	NO
DBCOMMIT=	LIBNAME option setting
DBCONDITION=	none
DBCREATE_TABLE_OPTS=	LIBNAME option setting
DBFORCE=	NO
DBGEN_NAME=	DBMS
DBINDEX=	LIBNAME option setting
DBKEY=	none
DBLABEL=	NO
DBMASTER=	none
DBMAX_TEXT=	1024
DBNULL=	Yes
DBNULLKEYS=	LIBNAME option setting
DBPROMPT=	LIBNAME option setting
DBSASTYPE=	see "Data Types for Aster nCluster " on page 428
DBTYPE=	see "Data Types for Aster nCluster " on page 428
DIMENSION=	NO
DISTRIBUTE_ON=	none
ERRLIMIT=	1
IGNORE_READ_ONLY_COLUMNS=	NO
INSERTBUFF=	LIBNAME option setting
NULLCHAR=	SAS

Option	Default Value
NULLCHARVAL=	A blank character
PARTITION_KEY=	none
PRESERVE_COL_NAMES=	LIBNAME option setting
QUERY_TIMEOUT=	LIBNAME option setting
READBUFF=	LIBNAME option setting
SASDATEFMT=	none
SCHEMA=	LIBNAME option setting

SQL Pass-Through Facility Specifics for Aster *n*Cluster

Key Information

For general information about this feature, see "Overview of SQL Procedure Interactions with SAS/ACCESS" on page 401. Aster *n*Cluster examples are available.

Here are the SQL pass-through facility specifics for the Aster *n*Cluster interface.

- The *dbms-name* is **ASTER**.
- The CONNECT statement is required.
- PROC SQL supports multiple connections to Aster *n*Cluster. If you use multiple simultaneous connections, you must use the *alias* argument to identify the different connections. If you do not specify an alias, the default **ASTER** alias is used.
- The *database-connection-arguments* for the CONNECT statement are identical to its LIBNAME connection options.

CONNECT Statement Example

This example uses the DBCON alias to connect to **mynpssrv** the Aster *n*Cluster database and execute a query. The connection alias is optional.

```
proc sql;
   connect to aster as dbcon
   (server=mynpssrv database=test user=myuser password=mypwd);
select * from connection to dbcon
   (select * from customers WHERE customer like '1%');
quit;
```

Special Catalog Queries

SAS/ACCESS Interface to Aster *n*Cluster supports the following special queries. You can use the queries to call the ODBC-style catalog function application programming interfaces (APIs). Here is the general format of the special queries:

Aster::SQLAPI*'parameter–1'*, *'parameter-n'*

Aster::
: is required to distinguish special queries from regular queries. Aster:: is not case sensitive.

SQLAPI
: is the specific API that is being called. SQLAPI is not case sensitive.

'parameter n'
: is a quoted string that is delimited by commas.

Within the quoted string, two characters are universally recognized: the percent sign (%) and the underscore (_). The percent sign matches any sequence of zero or more characters, and the underscore represents any single character. To use either character as a literal value, you can use the backslash character (\) to escape the match characters. For example, this call to SQL Tables usually matches table names such as myatest and my_test:

```
select * from connection to aster (ASTER::SQLTables
"test","","my_test");
```

Use the escape character to search only for the my_test table.

```
select * from connection to aster (ASTER::SQLTables "test","","my\_test");
```

SAS/ACCESS Interface to Aster *n*Cluster supports these special queries.

ASTER::SQLTables <*'Catalog'*, *'Schema'*, *'Table-name'*, *'Type'*>
: returns a list of all tables that match the specified arguments. If you do not specify any arguments, all accessible table names and information are returned.

ASTER::SQLColumns <*'Catalog'*, *'Schema'*, *'Table-name'*, *'Column-name'*>
: returns a list of all tables that match the specified arguments. If you do not specify any arguments, all accessible table names and information are returned.

ASTER::SQLColumns <'*Catalog'*, '*Schema'*, *'Table-name'*, *'Column-name'*>
: returns a list of all columns that match the specified arguments. If you do not specify any argument, all accessible column names and information are returned.

ASTER::SQLPrimaryKeys <*'Catalog'*, *'Schema'*, *'Table-name''Type'* >
: returns a list of all columns that compose the primary key that matches the specified table. A primary key can be composed of one or more columns. If you do not specify any table name, this special query fails.

ASTER::SQLStatistics <*'Catalog'*, *'Schema'*, *'Table-name'*>
: returns a list of the statistics for the specified table name, with options of SQL_INDEX_ALL and SQL_ENSURE set in the SQLStatistics API call. If you do not specify any table name argument, this special query fails.

ASTER::SQLGetTypeInfo
: returns information about the data types that the Aster *n*Cluster database supports.

ASTER::SQLTablePrivileges<'*Catalog*', '*Schema*', '*Table-name*'>
returns a list of all tables and associated privileges that match the specified arguments. If no arguments are specified, all accessible table names and associated privileges are returned.

Autopartitioning Scheme for Aster *n*Cluster

Overview

Autopartitioning for SAS/ACCESS Interface to Aster *n*Cluster is a modulo (MOD) function method. For general information about this feature, see "Autopartitioning Techniques in SAS/ACCESS" on page 63.

Autopartitioning Restrictions

SAS/ACCESS Interface to Aster *n*Cluster places additional restrictions on the columns that you can use for the partitioning column during the autopartitioning phase. Here is how columns are partitioned.

- SQL_INTEGER, SQL_BIT, SQL_SMALLINT, and SQL_TINYINT columns are given preference.
- You can use SQL_DECIMAL, SQL_DOUBLE, SQL_FLOAT, SQL_NUMERIC, and SQL_REAL columns for partitioning under these conditions:
  - Aster *n*Cluster supports converting these types to SQL_INTEGER by using the INTEGER cast function.
  - The precision minus the scale of the column is greater than 0 but less than 10—namely, 0<(*precision-scale*)<10.

Nullable Columns

If you select a nullable column for autopartitioning, the **OR<*column-name*>IS NULL** SQL statement is appended at the end of the SQL code that is generated for the threaded Read. This ensures that any possible NULL values are returned in the result set. Also, if the column to be used for the partitioning is SQL_BIT, the number of threads are automatically changed to two, regardless of DBSLICEPARM= option setting.

Using WHERE Clauses

Autopartitioning does not select a column to be the partitioning column if it appears in a WHERE clause. For example, this DATA step could not use a threaded Read to retrieve the data. All numeric columns in the table are in the WHERE clause:

```
DATA work.locemp;
    SET trlib.MYEMPS;
    WHERE EMPNUM<=30 and ISTENURE=0 and
       SALARY<=35000 and NUMCLASS>2;
run;
```

Using DBSLICEPARM=

SAS/ACCESS Interface to Aster *n*Cluster defaults to three threads when you use autopartitioning but do not specify a maximum number of threads in to use for the threaded Read. See DBSLICEPARM= LIBNAME option.

Using DBSLICE=

You might achieve the best possible performance when using threaded Reads by specifying the DBSLICE= data set option for Aster *n*Cluster in your SAS operation. Using DBSLICE= allows connections to individual partitions so that you can configure an Aster *n*Cluster data source for each partition. Use this option to specify both the data source and the WHERE clause for each partition.

```
proc print data=trilb.MYEMPS(DBSLICE=(DSN1='EMPNUM BETWEEN 1 AND 33'
DSN2='EMPNUM BETWEEN 34 AND 66'
DSN3='EMPNUM BETWEEN 67 AND 100'));
run;
```

Using the DATASOURCE= option is not required to use DBSLICE= option with threaded Reads.

Using DBSLICE= works well when the table that you want to read is not stored in multiple partitions. It gives you flexibility in column selection. For example, if you know that the STATE column in your employee table contains only a few distinct values, you can customize your DBSLICE= option accordingly.

```
data work.locemp;
   set trlib2.MYEMP(DBSLICE=("STATE='FL'" "STATE='GA'"
     "STATE='SC'" "STATE='VA'" "STATE='NC'"));
where EMPNUM<=30 and ISTENURE=0 and SALARY<=35000 and NUMCLASS>2;
run;
```

Passing SAS Functions to Aster *n*Cluster

SAS/ACCESS Interface to Aster *n*Cluster passes the following SAS functions to Aster *n*Cluster for processing. Where the Aster *n*Cluster function name differs from the SAS function name, the Aster *n*Cluster name appears in parentheses. For more information, see “Passing Functions to the DBMS Using PROC SQL” on page 46.

ABS	MAX
ARCOS (ACOS)	MIN
ARSIN (ASIN)	MLOWCASE (lower)INUTE (date_part)
ATAN	MOD
ATAN2	MONTH (date_part)
AVG	QTR (date_part)
BYTE (chr)	REPEAT
CEIL (ceiling)	SIGN
COALESCE	SIN
COS	SQRT
COUNT	STRIP (btrim)

DAY (date_part)	SUBSTR (substring)
EXP	SUM
FLOOR	TAN
HOUR (date_part)	TRANWRD (replace)
INDEX (strpos)	TRIMN (rtrim)
LOG (ln)	UPCASE (upper)
LOG10 (log)	YEAR (date_part)

SQL_FUNCTIONS= ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to Aster *n*Cluster. Due to incompatibility in date and time functions between Aster *n*Cluster and SAS, Aster *n*Cluster might not process them correctly. Check your results to determine whether these functions are working as expected. For more information, see "SQL_FUNCTIONS= LIBNAME Option" on page 185.

- COMPRESS (replace)
- DATE (now::date)
- DATEPART (cast)
- DATETIME (now)
- LENGTH
- ROUND
- TIME (now::time)
- TIMEPART (cast)
- TODAY (now::date)
- TRANSLATE

Passing Joins to Aster *n*Cluster

For a multiple libref join to pass to Aster *n*Cluster, all of these components of the LIBNAME statements must match exactly.

- user ID (USER=)
- password (PASSWORD=)
- server (SERVER=)
- database (DATABASE=)
- port (PORT=)
- data source (DSN=, if specified)
- SQL functions (SQL_FUNCTIONS=)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Bulk Loading for Aster *n*Cluster

Loading

Bulk loading is the fastest way to insert large numbers of rows into an Aster *n*Cluster table. To use the bulk-load facility, specify BULKLOAD=YES. The bulk-load facility uses the Aster *n*Cluster loader client application to move data from the client to the Aster *n*Cluster database.

Here are the Aster *n*Cluster bulk-load data set options. For detailed information about these options, see "About the Data Set Options for Relational Databases" on page 207.

- BL_DATAFILE=
- BL_DBNAME=
- BL_DELETE_DATAFILE=
- BL_DELIMITER=
- BL_HOST=
- BL_OPTIONS=
- BL_PATH=
- BULKLOAD=

Examples

This example shows how you can use a SAS data set, SASFLT.FLT98, to create and load a large Aster *n*Cluster table, FLIGHTS98.

```
LIBNAME sasflt 'SAS-library';
LIBNAME net_air ASTER user=louis pwd=fromage
        server=air2 database=flights dimension=yes;

PROC sql;
create table net_air.flights98
       (bulkload=YES bl_host='queen' bl_path='/home/ncluster_loader/'
          bl_dbname='beehive')
      as select * from sasflt.flt98;
quit;
```

You can use BL_OPTIONS= to pass specific Aster *n*Cluster options to the bulk-loading process.

You can create the same table using a DATA step.

```
data net_air.flights98(bulkload=YES bl_host='queen'
     bl_path='/home/ncluster_loader/'
     bl_dbname='beehive');
     set sasflt.flt98;
run;
```

You can then append the SAS data set, SASFLT.FLT98, to the existing Aster *n*Cluster table, ALLFLIGHTS. SAS/ACCESS Interface to Aster *n*Cluster to write data to a flat file, as specified in the BL_DATAFILE= option. Rather than deleting the data file, BL_DELETE_DATAFILE=NO causes the engine to retain it after the load completes.

```
PROC append base=net_air.allflights
    (BULKLOAD=YES
     BL_DATAFILE='/tmp/fltdata.dat'
     BL_HOST='queen'
     BL_PATH='/home/ncluster_loader/'
     BL_DBNAME='beehive'
     BL_DELETE_DATAFILE=NO )
data=sasflt.flt98;
run;
```

Naming Conventions for Aster *n*Cluster

Since SAS 7, most SAS names can be up to 32 characters long. SAS/ACCESS Interface to Aster *n*Cluster supports table names and column names that contain up to 32 characters. If column names are longer than 32 characters, they are truncated to 32 characters. If truncating a column name would result in identical column names, SAS generates a unique name by replacing the last character with a number. DBMS table names must be 32 characters or less. SAS does not truncate a name longer than 32 characters. If you have a table name that is greater than 32 characters, it is recommended that you create a table view.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options determine how SAS/ACCESS Interface to Aster *n*Cluster handles case sensitivity. Aster *n*Cluster is not case sensitive, so all names default to lowercase.

Aster *n*Cluster objects include tables, views, and columns. They follow these conventions.

- A name must be from 1 to 64 characters long.
- A name must begin with a letter (A through Z), diacritic marks, non-Latin characters (200-377 octal) or an underscore (_).
- To enable case sensitivity, enclose names in quotes. All references to quoted names must always be enclosed in quotes, and preserve case sensitivity.
- A name cannot begin with a _bee prefix. Leading _bee prefixes are reserved for system objects.
- A name cannot be a reserved word in Aster *n*Cluster, such as WHERE or VIEW.
- A name cannot be the same as another Aster *n*Cluster object that has the same type.

For more information, see your *Aster* n*Cluster Database User's Guide*.

Data Types for Aster *n*Cluster

Overview

Every column in a table has a name and a data type. The data type tells Aster *n*Cluster how much physical storage to set aside for the column and the form in which the data is stored. This information includes information about Aster *n*Cluster data types and data conversions.

For information about Aster *n*Cluster data types and to which data types are available for your version of Aster *n*Cluster, see the *Aster* n*Cluster Database User's Guide*.

SAS/ACCESS Interface to Aster *n*Cluster does not directly support TIMETZ or INTERVAL types. Any columns using these types are read into SAS as character strings.

String Data

CHAR(*n*)
: specifies a fixed-length column for character string data. The maximum length is 32,768 characters.

VARCHAR(*n*)
: specifies a varying-length column for character string data. The maximum length is 32,768 characters.

Numeric Data

BIGINT
: specifies a big integer. Values in a column of this type can range from –9223372036854775808 to +9223372036854775807.

SMALLINT
: specifies a small integer. Values in a column of this type can range from –32768 through +32767.

INTEGER
: specifies a large integer. Values in a column of this type can range from –2147483648 through +2147483647.

DOUBLE | DOUBLE PRECISION
: specifies a floating-point number that is 64 bits long. The double precision type typically has a range of around 1E-307 to 1E+308 with a precision of at least 15 decimal digits.

REAL
: specifies a floating-point number that is 32 bits long. On most platforms, the real type typically has a range of around 1E-37 to 1E+37 with a precision of at least 6 decimal digits.

DECIMAL | DEC | NUMERIC | NUM
: specifies a fixed-point decimal number. The precision and scale of the number determines the position of the decimal point. The numbers to the right of the decimal point are the scale, and the scale cannot be negative or greater than the precision.

Date, Time, and Timestamp Data

SQL date and time data types are collectively called datetime values. The SQL data types for dates, times, and timestamps are listed here. Be aware that columns of these data types can contain data values that are out of range for SAS.

DATE
: specifies date values. The range is 4713 BC to 5874897 AD. The default format is *YYYY-MM-DD* (for example, 1961-06-13). Aster *n*Cluster supports many other

formats for entering date data. For more information, see your *Aster* n*Cluster Database User's Guide*.

TIME
: specifies time values in hours, minutes, and seconds to six decimal positions: hh:mm:ss[.nnnnnn]. The range is 00:00:00.000000 to 24:00:00.000000. Due to the ODBC-style interface that SAS/ACCESS Interface to Aster *n*Cluster uses to communicate with the server, fractional seconds are lost in the data transfer from server to client.

TIMESTAMP
: combines a date and time in the default format of *yyyy-mm-dd hh:mm:ss[.nnnnnn]*. For example, a timestamp for precisely 2:25 p.m. on January 25, 1991, would be 1991-01-25-14.25.00.000000. Values in a column of this type have the same ranges as described for DATE and TIME.

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to Aster *n*Cluster assigns to SAS variables to read from an Aster *n*Cluster table when using the LIBNAME statement on page 93. These default formats are based on Aster *n*Cluster column attributes.

Table 14.3 *LIBNAME Statement: Default SAS Formats for Aster nCluster Data Types*

Aster *n*Cluster Data Type	SAS Data Type	Default SAS Format
CHAR(*n*)*	character	$*n*.
VARCHAR(*n*)*	character	$*n*.
INTEGER	numeric	11.
SMALLINT	numeric	6.
BIGINT	numeric	20.
DECIMAL(*p,s*)	numeric	*m.n*
NUMERIC(*p,s*)	numeric	*m.n*
REAL	numeric	none
DOUBLE	numeric	none
TIME	numeric	TIME8.
DATE	numeric	DATE9.
TIMESTAMP	numeric	DATETIME*25.6*

* *n* in Aster *n*Cluster data types is equivalent to *w* in SAS formats.

This table shows the default Aster *n*Cluster data types that SAS/ACCESS assigns to SAS variable formats during output operations when you use the LIBNAME statement.

Table 14.4 *LIBNAME Statement: Default Aster nCluster Data Types for SAS Variable Formats*

SAS Variable Format	Aster *n*Cluster Data Type
m.n	DECIMAL(*p,s*)
other numerics	DOUBLE
$*n*.	VARCHAR(*n*)*
datetime formats	TIMESTAMP
date formats	DATE
time formats	TIME

* *n* in Aster *n*Cluster data types is equivalent to *w* in SAS formats.

Chapter 15
SAS/ACCESS Interface to DB2 for UNIX and PC Hosts

Introduction to SAS/ACCESS Interface to DB2 under UNIX and PC Hosts

This section describes SAS/ACCESS Interface to DB2 under UNIX and PC Hosts. For a list of SAS/ACCESS features that are available in this interface, see “SAS/ACCESS Interface to DB2 under UNIX and PC Hosts: Supported Features” on page 78.

LIBNAME Statement Specifics for DB2 under UNIX and PC Hosts

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to DB2 under UNIX and PC Hosts supports and includes an example. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing DB2 under UNIX and PC Hosts.

LIBNAME *libref* **db2** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`db2`
specifies the SAS/ACCESS engine name for the DB2 under UNIX and PC Hosts interface.

connection-options
provides connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. When you use the LIBNAME statement, you can connect to DB2 several ways. Specify *only one* of these methods for each connection because they are mutually exclusive.

- USER=, PASSWORD=, DATASRC=
- COMPLETE=
- NOPROMPT=
- PROMPT=
- REQUIRED=

Here is how these options are defined.

USER=<'>*user-name*<'>

lets you connect to a DB2 database with a user ID that is different from the default ID. USER= is optional. If you specify USER=, you must also specify PASSWORD=. If USER= is omitted, your default user ID for your operating environment is used.

PASSWORD=<'>*password*<'>

specifies the DB2 password that is associated with your DB2 user ID. PASSWORD= is optional. If it contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. If you specify USER=, you must specify PASSWORD=.

DATASRC=<'>*data-source-name*<'>

specifies the DB2 data source or database to which you want to connect. DATASRC= is optional. If you omit it, you connect by using a default environment variable.

Alias: DATABASE=, DB=, DSN=

COMPLETE=<'>*CLI-connection-string*<'>

specifies connection information for your data source or database for PCs only. Separate multiple options with a semicolon. When a successful connection is made, the complete connection string is returned in the SYSDBMSG macro variable. If you do not specify enough correct connection options, you are prompted with a dialog box that displays the values from the COMPLETE= connection string. You can edit any field before you connect to the data source. This option is not available on UNIX platforms. See your DB2 documentation for more details.

NOPROMPT=<'>*CLI-connection-string*<'>

specifies connection information for your data source or database. Separate multiple options with a semicolon. If you do not specify enough correct connection options, an error is returned (no dialog box is displayed).

PROMPT=<'> *CLI-connection-string*<'>

specifies connection information for your data source or database for PCs only. Separate multiple options with a semicolon. When a successful connection is made, the complete connection string is returned in the SYSDBMSG macro variable. PROMPT= does not immediately attempt to connect to the DBMS. Instead, it displays a dialog box that contains the values that you entered in the PROMPT= connection string. You can edit values or enter additional values in any field before you connect to the data source.

This option is not available on UNIX platforms.

REQUIRED=<'>*CLI-connection-string*<'>

specifies connection information for your data source or database for PCs only. Separate the multiple options with semicolons. When a successful connection is made, the complete connection string is returned in the SYSDBMSG macro variable. If you do not specify enough correct connection options, a dialog box prompts you for the connection options. REQUIRED= lets you modify only required fields in the dialog box.

This option is not available on UNIX platforms.

LIBNAME-options
defines how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to DB2 under UNIX and PC Hosts, with the applicable default values. For more detail about these options, see "LIBNAME Options for Relational Databases" on page 99.

Table 15.1 *SAS/ACCESS LIBNAME Options for DB2 under UNIX and PC Hosts*

Option	Default Value
ACCESS=	none
AUTHDOMAIN=	none
AUTOCOMMIT=	varies with transaction type
CONNECTION=	SHAREDREAD
CONNECTION_GROUP=	none
CURSOR_TYPE=	operation-specific
DBCOMMIT=	1000 when inserting rows; 0 when updating rows; 10,000 when bulk-loading rows
DBCONINIT=	none
DBCONTERM=	none
DBCREATE_TABLE_OPTS=	none
DBGEN_NAME=	DBMS
DBINDEX=	YES
DBLIBINIT=	none
DBLIBTERM=	none
DBMAX_TEXT=	1024
DBMSTEMP=	NO
DBNULLKEYS=	YES
DBPROMPT=	NO
DBSLICEPARM=	THREADED_APPS,2 or 3

Option	Default Value
DEFER=	NO
DIRECT_EXE=	none
DIRECT_SQL=	YES
FETCH-IDENTITY=	NO
IGNORE_READ_ONLY_COLUMNS=	NO
IN=	none
INSERTBUFF=	automatically calculated based on row length
MULTI_DATASRC_OPT=	NONE
PRESERVE_COL_NAMES=	NO (see "Naming Conventions for DB2 under UNIX and PC Hosts" on page 458)
PRESERVE_TAB_NAMES=	NO (see "Naming Conventions for DB2 under UNIX and PC Hosts" on page 458)
QUERY_TIMEOUT=	0
READBUFF=	automatically calculated based on row length
READ_ISOLATION_LEVEL=	set by the user in the DB2Cli.ini file (see "Locking in the DB2 under UNIX and PC Hosts Interface" on page 456)
READ_LOCK_TYPE=	ROW
REREAD_EXPOSURE=	NO
SCHEMA=	your user ID
SPOOL=	YES
SQL_FUNCTIONS=	none
SQL_FUNCTIONS_COPY=	none
SQLGENERATION=	DBMS
STRINGDATES=	NO
UPDATE_ISOLATION_LEVEL=	CS (see "Locking in the DB2 under UNIX and PC Hosts Interface" on page 456)

Option	Default Value
UPDATE_LOCK_TYPE=	ROW
UTILCONN_TRANSIENT=	YES

DB2 under UNIX and PC Hosts LIBNAME Statement Example

In this example, the libref MyDBLib uses the DB2 engine and the NOPROMPT= option to connect to a DB2 database. PROC PRINT is used to display the contents of the DB2 table Customers.

```
libname mydblib db2
   noprompt="dsn=userdsn;uid=testuser;pwd=testpass;";

proc print data=mydblib.customers;
   where state='CA';
run;
```

Data Set Options for DB2 under UNIX and PC Hosts

All SAS/ACCESS data set options in this table are supported for DB2 under UNIX and PC Hosts. Default values are provided where applicable. For general information about this feature, see Data Set Options for Relational Databases on page 207.

Table 15.2 *SAS/ACCESS Data Set Options for DB2 under UNIX and PC Hosts*

Option	Default Value
BL_ALLOW_READ_ACCESS=	NO
BL_ALLOW_WRITE_ACCESS=	NO
BL_CODEPAGE=	the code page ID for the window
BL_COPY_LOCATION=	none
BL_CPU_PARALLELISM=	none
BL_DATA_BUFFER_SIZE=	none
BL_DATAFILE=	the current directory
BL_DELETE_DATAFILE=	YES
BL_DISK_PARALLELISM=	none
BL_EXCEPTION=	none

Option	Default Value
BL_INDEXING_MODE=	AUTOSELECT
BL_LOAD_REPLACE=	NO
BL_LOG=	the current directory
BL_METHOD=	none
BL_OPTIONS=	none
BL_PORT_MAX=	none
BL_PORT_MIN=	none
BL_RECOVERABLE=	NO
BL_REMOTE_FILE=	none
BL_SERVER_DATAFILE=	creates a data file in the current directory or with the default file specifications (same as for BL_DATAFILE=)
BL_WARNING_COUNT=	2147483646
BULKLOAD=	NO
CURSORTYPE=	LIBNAME option setting
DBCOMMIT=	LIBNAME option setting
DBCONDITION=	none
DBCREATE_TABLE_OPTS=	LIBNAME option setting
DBFORCE=	NO
DBGEN_NAME=	DBMS
DBINDEX=	LIBNAME option setting
DBKEY= on page 290	none
DBLABEL=	NO
DBMASTER=	none
DBMAX_TEXT=	1024
DBNULL=	_ALL_=YES
DBNULLKEYS=	LIBNAME option setting

Option	Default Value
DBPROMPT=	LIBNAME option setting
DBSASLABEL=	COMPAT
DBSASTYPE=	see “Data Types for DB2 under UNIX and PC Hosts ” on page 458
DBSLICE=	none
DBSLICEPARM=	THREADED_APPS,3
DBTYPE=	see “Data Types for DB2 under UNIX and PC Hosts ” on page 458
ERRLIMIT=	NO
FETCH_IDENTITY=	1
IGNORE_READ_ONLY_COLUMNS=	NO
IN=	LIBNAME option setting
INSERTBUFF=	LIBNAME option setting
NULLCHAR=	SAS
NULLCHARVAL=	a blank character
PRESERVE_COL_NAMES=	LIBNAME option setting
QUERY_TIMEOUT=	LIBNAME option setting
READ_ISOLATION_LEVEL=	LIBNAME option setting
READ_LOCK_TYPE=	LIBNAME option setting
READBUFF=	LIBNAME option setting
SASDATEFMT=	none
SCHEMA=	LIBNAME option setting
UPDATE_ISOLATION_LEVEL=	LIBNAME option setting
UPDATE_LOCK_TYPE=	LIBNAME option setting

SQL Pass-Through Facility Specifics for DB2 under UNIX and PC Hosts

Key Information

For general information about this feature, see "SQL Pass-Through Facility" on page 401. DB2 under UNIX and PC Hosts examples are available.

Here are the SQL pass-through facility specifics for the DB2 under UNIX and PC Hosts interface.

- The *dbms-name* is **DB2**.
- The CONNECT statement is required.
- You can connect to only one DB2 database at a time. However, you can use multiple CONNECT statements to connect to multiple DB2 data sources by using the *alias* argument to distinguish your connections.
- The *database-connection-arguments* for the CONNECT statement are identical to its LIBNAME connection options.
- These LIBNAME options are available with the CONNECT statement.
  - AUTOCOMMIT=
  - CURSOR_TYPE=
  - QUERY_TIMEOUT=
  - READBUFF=
  - READ_ISOLATION_LEVEL=

Examples

This example connects to the SAMPLE database and sends it two EXECUTE statements to process.

```
proc sql;
   connect to db2 (database=sample);
   execute (create view
               sasdemo.whotookorders as
            select ordernum, takenby,
                   firstname, lastname, phone
              from sasdemo.orders,
                   sasdemo.employees
              where sasdemo.orders.takenby=
                    sasdemo.employees.empid)
           by db2;
   execute (grant select on
            sasdemo.whotookorders to testuser)
           by db2;
   disconnect from db2;
quit;
```

This example connects to the SAMPLE database by using an alias (DB1) and performs a query, shown in italic type, on the SASDEMO.CUSTOMERS table.

```
proc sql;
   connect to db2 as db1 (database=sample);
   select *
      from connection to db1
         (select * from sasdemo.customers
             where customer like '1%');
   disconnect from db1;
quit;
```

Special Catalog Queries

SAS/ACCESS Interface to DB2 under UNIX and PC Hosts supports the following special queries. You can use the queries to call the ODBC-style catalog function application programming interfaces (APIs). Here is the general format of these queries:

DB2::SQLAPI "*parameter 1*","*parameter n*"

DB2::
: is required to distinguish special queries from regular queries.

SQLAPI
: is the specific API that is being called. Neither DB2:: nor SQLAPI are case sensitive.

"*parameter n*"
: is a quoted string that is delimited by commas.

Within the quoted string, two characters are universally recognized: the percent sign (%) and the underscore (_). The percent sign matches any sequence of zero or more characters, and the underscore represents any single character. To use either character as a literal value, you can use the backslash character (\) to escape the match characters. For example, this call to SQLTables usually matches table names such as mytest and my_test:

```
select * from connection to db2 (DB2::SQLTables "test","","my_test");
```

Use the escape character to search only for the my_test table:

```
select * from connection to db2 (DB2::SQLTables "test","","my\_test");
```

SAS/ACCESS Interface to DB2 under UNIX and PC Hosts supports these special queries:

DB2::SQLDataSources
: returns a list of database aliases that have been cataloged on the DB2 client.

DB2::SQLDBMSInfo
: returns information about the DBMS server and version. It returns one row with two columns that describe the DBMS name (such as DB2/NT) and version (such as 8.2).

Autopartitioning Scheme for DB2 under UNIX and PC Hosts

Overview

Autopartitioning for SAS/ACCESS Interface to DB2 for UNIX and PC Hosts is a modulo (MOD) function method. For general information about this feature, see "Autopartitioning Techniques in SAS/ACCESS" on page 63.

Autopartitioning Restrictions

SAS/ACCESS Interface to DB2 under UNIX and PC Hosts places additional restrictions on the columns that you can use for the partitioning column during the autopartitioning phase. Here is how columns are partitioned.

- INTEGER and SMALLINT columns are given preference.
- You can use other DB2 numeric columns for partitioning as long as the precision minus the scale of the column is between 0 and 10—namely, 0<(*precision-scale*)<10.

Nullable Columns

If you select a nullable column for autopartitioning, the **OR<*column-name*> IS NULL** SQL statement is appended at the end of the SQL code that is generated for the threaded Reads. This ensures that any possible NULL values are returned in the result set.

Using WHERE Clauses

Autopartitioning does not select a column to be the partitioning column if it appears in a SAS WHERE clause. For example, this DATA step cannot use a threaded Read to retrieve data because all numeric columns in the table (see the table definition in "Using DBSLICE=") are in the WHERE clause:

```
data work.locemp;
set trlib.MYEMPS;
where EMPNUM<=30 and ISTENURE=0 and
 SALARY<=35000 and NUMCLASS>2;
run;
```

Using DBSLICEPARM=

Although SAS/ACCESS Interface to DB2 under UNIX and PC Hosts defaults to three threads when you use autopartitioning, do not specify a maximum number of threads for the threaded Read in the DBSLICEPARM= LIBNAME option.

Using DBSLICE=

You might achieve the best possible performance when using threaded Reads by specifying the DBSLICE= data set option on page 299 for DB2 in your SAS operation. This is especially true if your DB2 data is evenly distributed across multiple partitions in a DB2 Enterprise Extended Edition (EEE) database system. When you create a DB2 table under the DB2 EEE model, you can specify the partitioning key that you want to use by appending the clause **PARTITIONING KEY(*column-name*)** to your CREATE TABLE statement. Here is how you can accomplish this by using the LIBNAME option, DBCREATE_TABLE_OPTS=, within the SAS environment.

```
/*points to a triple node server*/
libname trlib2 db2 user=db2user pw="db2pwd" datasrc=sample3c
DBCREATE_TABLE_OPTS='PARTITIONING KEY(EMPNUM);

proc datasets library=trlib;
  delete MYEMPS1;run;

data trlib.myemps(drop=morf whatstate
   DBTYPE=(HIREDATE="date" SALARY="numeric(8,2)"
   NUMCLASS="smallint" GENDER="char(1)" ISTENURE="numeric(1)" STATE="char(2)"
   EMPNUM="int NOT NULL Primary Key"));
format HIREDATE mmddyy10.;
do EMPNUM=1 to 100;
     morf=mod(EMPNUM,2)+1;
     if(morf eq 1) then
         GENDER='F';
     else
         GENDER='M';
     SALARY=(ranuni(0)*5000);
     HIREDATE=int(ranuni(13131)*3650);
     whatstate=int(EMPNUM/5);
     if(whatstate eq 1) then
         STATE='FL';
     if(whatstate eq 2) then
         STATE='GA';
     if(whatstate eq 3) then
         STATE='SC';
     if(whatstate eq 4) then
         STATE='VA';
     else
         state='NC';
     ISTENURE=mod(EMPNUM,2);
     NUMCLASS=int(EMPNUM/5)+2;
     output;
end;
run;
```

After the table MYEMPS is created on this three-node database, a third of the rows reside on each of the three nodes.

Optimization of the threaded Read against this partitioned table depends on the location of the DB2 partitions. If the DB2 partitions are on the same machine, you can use DBSLICE= with the DB2 NODENUMBER function in the WHERE clause:

```
proc print data=trlib2.MYEMPS(DBSLICE=("NODENUMBER(EMPNO)=0"
   "NODENUMBER(EMPNO)=1" "NODENUMBER(EMPNO)=2"));
run;
```

If the DB2 partitions reside on different physical machines, you can usually obtain the best results by using the DBSLICE= option with the SERVER= syntax in addition to the DB2 NODENUMBER function in the WHERE clause.

In the next example, DBSLICE= contains specific partitioning information for DB2. Also, Sample3a, Sample3b, and Sample3c are DB2 database aliases that point to individual DB2 EEE database nodes that exist on separate physical machines. For more information about the configuration of these nodes, see “Configuring DB2 EEE Nodes on Physically Partitioned Databases” on page 445.

```
proc print data=trlib2.MYEMPS(DBSLICE=(sample3a="NODENUMBER(EMPNO)=0"
   samble3b="NODENUMBER(EMPNO)=1" sample3c="NODENUMBER(EMPNO)=2"));
run;
```

NODENUMBER is not required to use threaded Reads for SAS/ACCESS Interface to DB2 under UNIX and PC Hosts. The methods and examples described in DBSLICE= on page 299 work well in cases where the table that you want to read is not stored in multiple partitions to DB2. These methods also give you full control over which column is used to execute the threaded Read. For example, if the STATE column in your employee table contains only a few distinct values, you can modify your DBSLICE= clause accordingly:

```
data work.locemp;
set trlib2.MYEMPS (DBSLICE=("STATE='GA'"
    "STATE='SC'" "STATE='VA'" "STATE='NC'"));
where EMPNUM<=30 and ISTENURE=0 and SALARY<=35000 and NUMCLASS>2;
run;
```

Configuring DB2 EEE Nodes on Physically Partitioned Databases

Assuming that the database SAMPLE is partitioned across three different machines, you can create a database alias for it at each node from the DB2 Command Line Processor by issuing these commands:

```
catalog tcpip node node1 remote <hostname> server 50000
catalog tcpip node node2 remote <hostname> server 50000
catalog tcpip node node3 remote <hostname> server 50000
catalog database sample as samplea at node node1
catalog database sample as sampleb at node node2
catalog database sample as samplec at node node3
```

This enables SAS/ACCESS Interface to DB2 to access the data for the SAMPLE table directly from each node. For more information about configuring DB2 EEE to use multiple physical partitions, see the *DB2 Administrator's Guide*.

Temporary Table Support for DB2 under UNIX and PC Hosts

Establishing a Temporary Table

For general information about this feature, see “Temporary Table Support for SAS/ACCESS ” on page 42.

To make full use of temporary tables, the CONNECTION=GLOBAL connection option is necessary. You can use this option to establish a single connection across SAS DATA step and procedure boundaries. The LIBNAME statement and the SQL pass-through facility can also share this connection. Because a temporary table only exists within a single connection, you must be able to share this single connection among all steps that reference the temporary table. The temporary table cannot be referenced from any other connection.

The type of temporary table that is used for this processing is created using the DECLARE TEMPORARY TABLE statement with the ON COMMIT PRESERVE clause. This type of temporary table lasts for the duration of the connection—unless it is explicitly dropped—and retains its rows of data beyond commit points.

DB2 places all global temporary tables in the SESSION schema. Therefore, to reference these temporary tables within SAS, you must explicitly provide the SESSION schema in pass-through SQL statements or use the SCHEMA= LIBNAME option with a value of SESSION.

Currently, the only supported way to create a temporary table is to use a PROC SQL pass-through statement. To use both the SQL pass-through facility and librefs to reference a temporary table, you need to specify a LIBNAME statement before the PROC SQL step. This enables the global connection to persist across SAS steps, even multiple PROC SQL steps, as shown in this example.

```
libname temp db2 database=sample user=myuser password=mypwd
        schema=SESSION connection=global;

proc sql;
   connect to db2 (datasrc=sample user=myuser pwd=mypwd connection=global);
   execute (declare global temporary table temptab1 like other.table
            on commit PRESERVE rows not logged) by db2;
quit;
```

At this point, you can refer to the temporary table by using the libref Temp or by using the CONNECTION=GLOBAL option with a PROC SQL step.

Terminating a Temporary Table

You can drop a temporary table at any time, or allow it to be implicitly dropped when the connection is terminated. Temporary tables do not persist beyond the scope of a single connection.

Examples

These examples assume that you already have the following data.

- a DeptInfo table on the DBMS that contains all of your department information
- a SAS data set with join criteria to retrieve certain rows from the DeptInfo table
- a SAS data set with updates to the DeptInfo table

Here are the librefs and temporary tables that are used.

```
libname saslib base 'SAS-library';
libname dept db2 datasrc=sample user=myuser pwd=mypwd connection=global;
libname temp db2 datasrc=sample user=myuser pwd=mypwd connection=global
        schema=SESSION;
/* Note that the temporary table has a schema of SESSION */

proc sql;
   connect to db2 (datasrc=sample user=myuser pwd=mypwd connection=global);
   execute (declare global temporary table
            temptab1 (dname char(20), deptno int)
            on commit PRESERVE rows not logged) by db2;
quit;
```

This example demonstrates how to take a heterogeneous join and use a temporary table to perform a homogeneous join on the DBMS (as opposed to reading the DBMS table into SAS to perform the join). Using the table created above, the SAS data is copied into the temporary table to perform the join.

```
proc sql;
   connect to db2 (datasrc=sample user=myuser pwd=mypwd connection=global);
   insert into temp.temptab1 select * from saslib.joindata;
   select * from dept.deptinfo info, temp.temptab1 tab
        where info.deptno = tab.deptno;
   /* remove the rows for the next example */
   execute (delete from session.temptab1) by db2;
   quit;
```

In this example, transaction processing on the DBMS occurs using a temporary table as opposed to using either DBKEY= or MULTI_DATASRC_OPT=IN_CLAUSE with a SAS data set as the transaction table.

```
connect to db2 (datasrc=sample user=myuser pwd=mypwd connection=global);
   insert into temp.temptab1 select * from saslib.transdat;
   execute (update deptinfo d set deptno =
           (select deptno from session.temptab1)
           where d.dname = (select dname from session.temptab1)) by db2;
quit;
```

Calling Stored Procedures in DB2 under UNIX and PC Hosts

Overview

A stored procedure is one or more SQL statements or supported third-generation languages (3GLs, such as C) statements that are compiled into a single procedure that exists in DB2. Stored procedures might contain static (hard-coded) SQL statements. Static SQL is optimized better for some DBMS operations. In a carefully managed

DBMS environment, programmers and database administrators can know the exact SQL to execute.

SAS usually generates SQL dynamically. However, database administrators can encode static SQL in a stored procedure and therefore restrict SAS users to a tightly controlled interface. When you use a stored procedure call, you must specify a schema.

SAS/ACCESS support for stored procedure includes passing input parameters, retrieving output parameters into SAS macro variables, and retrieving the result set into a SAS table. Although DB2 stored procedures can return multiple result sets, SAS/ACCESS Interface to DB2 under UNIX and PC Hosts can retrieve only a single result set.

You can call stored procedures only from PROC SQL.

Examples

Example 1: Specify a Basic Call

Use CALL statement syntax to call a stored procedure.

```
call "schema".stored_proc
```

The simplest way to call a stored procedure is to use the EXECUTE statement in PROC SQL. In this example, you execute STORED_PROC by using a CALL statement. SAS does not capture the result set.

```
proc sql;
connect to db2 (db=sample uid= pwd=);
execute (call "schema".stored_proc);
quit;
```

Example 2: Specify One Input Parameter That Returns a Result Set

You can also return a result set. In this example, MYPROC3 is executed using a CALL statement and returns a result set.

```
proc sql;
connect to db2 (db=sample uid= pwd=);
%let p1 = 2000;
select * from connection to db2 (call MYSCHEMA.MYPROC3(:p1));
```

Example 3: Specify Three Output Parameters

The CALL statement syntax supports passing of parameters. You can specify such input parameters as numeric constants, character constants, or a null value. You can also pass input parameters by using SAS macro variable references. To capture the value of an output parameter, a SAS macro variable reference is required. This example uses a constant (1), an input/output parameter (:INOUT), and an output parameters (:OUT). Not only is the result set returned to the SAS results table, the SAS macro variables INOUT and OUT capture the parameter outputs.

```
proc sql;
connect to db2 (db=sample uid= pwd=);
%let INOUT=2;
create table sasresults as select * from connection to db2
   (call "schema".stored_proc (1,:INOUT,:OUT));
quit;
```

Example 4: Specify Three Different Parameters

The CALL statement syntax supports passing of parameters. To capture the value of an output parameter, a SAS macro variable reference is required. This example uses three output parameters (:p1, :p2, :p3:) and displays the value of each.

```
proc sql;
connect to db2 (db=sample uid= pwd=);
execute (call MYSCHEMA.MYPROC(:p1,:p2,:p3)) by db2;
%put &p1 &p2 &p3 /* display values of output parameters */
```

Example 5: Pass a NULL Parameter

In these calls, NULL is passed as the parameter to the DB2 stored procedure.

- Null string literals in the call

  ```
  call proc('');
  call proc("")
  ```

- Literal period or literal NULL in the call

  ```
  call proc(.)
  call proc(NULL)
  ```

- SAS macro variable set to NULL string

  ```
  %let charparm=;
  call proc(:charparm)
  ```

- SAS macro variable set to period (SAS numeric value is missing)

  ```
  %let numparm=.;
  call proc(:numparm)
  ```

Only the literal period and the literal NULL work generically for both DB2 character parameters and DB2 numeric parameters. For example, a DB2 numeric parameter would reject "" and **`%let numparm=.;`** would not pass a DB2 NULL for a DB2 character parameter. As a literal, a period passes NULL for both numeric and character parameters. However, when it is in a SAS macro variable, it constitutes a NULL only for a DB2 numeric parameter.

You cannot pass NULL parameters by omitting the argument. For example, you cannot use this call to pass three NULL parameters.

```
call proc(,,)
```

You could use this call instead.

```
call proc(NULL,NULL,NULL)
```

Example 6: Specify a Schema

Use standard CALL statement syntax to execute a stored procedure that exists in another schema, as shown in this example.

```
proc sql;
   connect to db2 (db=sample uid= pwd=);
   execute (call OTHERSCHEMA.MYPROC1(:p1));
quit;
```

If the schema is in mixed case or lowercase, enclose the schema name in double quotation marks.

```
proc sql;
connect to db2 (db=sample uid= pwd=);
```

```
execute (call "OTHERSCHEMA".MYPROC1(:p1));
quit;
```

Example 7: Execute Remote Stored Procedures

If the stored procedure exists on a different DB2 instance, specify it with a valid three-part name.

```
select * from connection
proc sql;
connect to db2 (db=sample uid= pwd=);
to db2 (call MYSCHEMA.MYPROC1.prod5(:p1));
quit;
```

DBLOAD Procedure Specifics for DB2 under UNIX and PC Hosts

Key Information

For general information about this feature, see Chapter 2, “DBLOAD Procedure,” on page 901. DB2 under UNIX and PC Hosts examples are available.

SAS/ACCESS Interface to DB2 under UNIX and PC Hosts supports all DBLOAD procedure statements in batch mode. Here are the DBLOAD procedure specifics for the DB2 under UNIX and PC Hosts interface.

- DBMS= value is **DB2**.
- Here are the database description statements that PROC DBLOAD uses:

  IN= <'>*database-name*<'>;
  : specifies the name of the database in which you want to store the new DB2 table. The IN= statement is required and must immediately follow the PROC DBLOAD statement. The *database-name* is limited to eight characters. DATABASE= is an alias for the IN= statement.

    The database that you specify must already exist. If the database name contains the _, $, @, or # special character, you must enclose it in quotation marks. DB2 recommends against using special characters in database names, however.

  USER= <'>*user name*<'>;
  : lets you connect to a DB2 database with a user ID that is different from the default login ID.

    USER= is optional in SAS/ACCESS Interface to DB2 under UNIX and PC Hosts. If you specify USER=, you must also specify PASSWORD=. If USER= is omitted, your default user ID is used.

  PASSWORD= <'>*password*<'>;
  : specifies the password that is associated with your user ID.

    PASSWORD= is optional in SAS/ACCESS Interface to DB2 under UNIX and PC Hosts because users have default user IDs. If you specify USER=, however, you must specify PASSWORD=.

    If you do not wish to enter your DB2 password in uncoded text on this statement, see PROC PWENCODE in *Base SAS Procedures Guide* for a method to encode it.

- Here is the TABLE= statement:

  TABLE= <'><*schema-name.*>*table-name*<'>;
  : identifies the DB2 table or DB2 view that you want to use to create an access descriptor. The *table-name* is limited to 18 characters. If you use quotation marks, the name is case sensitive. The TABLE= statement is required.

    The *schema-name* is a person's name or group ID that is associated with the DB2 table. The schema name is limited to eight characters.

- Here is the NULLS statement.

  NULLS *variable-identifier-1* =Y|N|D < . . . *variable-identifier-n* =Y|N|D >;
  : lets you specify whether the DB2 columns that are associated with the listed SAS variables allow NULL values. By default, all columns accept NULL values.

    The NULLS statement accepts any one of these values.

  Y
  : specifies that the column accepts NULL values. This is the default.

  N
  : specifies that the column does not accept NULL values.

  D
  : specifies that the column is defined as NOT NULL WITH DEFAULT.

Examples

The following example creates a new DB2 table, SASDEMO.EXCHANGE, from the MYDBLIB.RATEOFEX data file. You must be granted the appropriate privileges in order to create new DB2 tables or views.

```
proc dbload dbms=db2 data=mydblib.rateofex;
   in='sample';
   user='testuser';
   password='testpass';
   table=sasdemo.exchange;
      rename fgnindol=fgnindollars
         4=dollarsinfgn;
   nulls updated=n fgnindollars=n
         dollarsinfgn=n country=n;
   load;
run;
```

The following example sends only a DB2 SQL GRANT statement to the SAMPLE database and does not create a new table. Therefore, the TABLE= and LOAD statements are omitted.

```
proc dbload dbms=db2;
   in='sample';
   sql grant select on sasdemo.exchange
      to testuser;
run;
```

Passing SAS Functions to DB2 under UNIX and PC Hosts

SAS/ACCESS Interface to DB2 under UNIX and PC Hosts passes the following SAS functions to DB2 for processing if the DBMS driver or client that you are using supports this function. Where the DB2 function name differs from the SAS function name, the DB2 name appears in parentheses. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

ABS	MIN
ARCOS (ACOS)	MINUTE
ARSIN (ASIN)	MOD
ATAN	MONTH
AVG	QTR (QUARTER)
BYTE (CHAR)	REPEAT
CEIL (CEILING)	SECOND
COMPRESS (REPLACE)	SIGN
COS	SIN
COSH	SINH
COUNT (COUNT_BIG)	SQRT
DAY (DAYOFMONTH)	STRIP (RTRIM, LTRIM)
EXP	SUBSTR (SUBSTRING)
FLOOR	SUM
HOUR	TAN
INDEX (LOCATE)	TANH
LENGTH	TRANWRD (REPLACE)
LOG	TRIMN (RTRIM)
LOG10	UPCASE (UCASE)
LOWCASE (LCASE)	WEEKDAY (DAYOFWEEK)
MAX	YEAR

SQL_FUNCTIONS=ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to DB2. Due to incompatibility in date and time functions between DB2 and SAS, DB2 might not process them correctly. Check your results to determine whether these functions are working as expected.

DATE (CURDATE)	TIME (CURTIME)
DATEPART (DATE)	TIMEPART
DATETIME (NOW)	TODAY (CURDATE)
SOUNDEX	

Passing Joins to DB2 under UNIX and PC Hosts

For a multiple-libref join to pass to DB2, all of these components of the LIBNAME statements must match exactly:

- user ID (USER=)
- password (PASSWORD=)
- update isolation level (UPDATE_ISOLATION_LEVEL=, if specified)
- read_isolation level (READ_ISOLATION_LEVEL=, if specified)
- qualifier (QUALIFIER=)
- data source (DATASRC=)
- prompt (PROMPT= must *not* be specified)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Bulk Loading for DB2 under UNIX and PC Hosts

Overview

Bulk loading is the fastest way to insert large numbers of rows into a DB2 table. Using this facility instead of regular SQL insert statements, you can insert rows two to ten times more rapidly. DB2 bulk-load examples on page 456 and tips for maximizing performance on page 455 are available. You must specify BULKLOAD= on page 274YES to use the bulk-load facility.

SAS/ACCESS Interface to DB2 under UNIX and PC Hosts offers LOAD, IMPORT, and CLI LOAD bulk-loading methods. The BL_REMOTE_FILE= and BL_METHOD= on page 253 data set options determine which method to use.

For more information about the differences between IMPORT, LOAD, and CLI LOAD, see the *DB2 Data Movement Utilities Guide and Reference.*

Using the LOAD Method

To use the LOAD method, you must have system administrator, database administrator, or load authority on the database and the INSERT privilege on the table to be loaded.

This method also requires that client and server machines can read and write files to a common location such as a mapped network drive or an NFS directory. To use this method, specify the BL_REMOTE_FILE= option.

Note: Because SAS/ACCESS Interface to DB2 uses the PC/IXF file format to transfer data to the DB2 LOAD utility, you cannot use this method to load data into partitioned databases.

Here are the bulk-load options available with the LOAD method. For details about these options, see "Overview" on page 207.

- BL_CODEPAGE=

- BL_DATAFILE=
- BL_DELETE_DATAFILE=
- BL_LOG= – The log file contains a summary of load information and error descriptions. On most platforms, the default filename is *BL_<table>_<unique-ID>*.log, where *table* specifies the table name and *unique-ID* specifies 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 this number.
- BL_OPTIONS=
- BL_REMOTE_FILE=
- BL_SERVER_DATAFILE=
- BL_WARNING_COUNT=

Using the IMPORT Method

The IMPORT method does not offer the same level of performance as the LOAD method. However, it is available to all users with INSERT privileges for the tables to be loaded. The IMPORT method does not require that the server and client have a common location in order to access the data file. If you do not specify BL_REMOTE_FILE=, the IMPORT method is automatically used.

Here are the bulk-loading options available with the IMPORT method. For detailed information about these options, see "Overview" on page 207.

- BL_CODEPAGE=
- BL_DATAFILE=
- BL_DELETE_DATAFILE=
- BL_LOG=
- BL_OPTIONS= on page 255

Using the CLI LOAD Method

The CLI LOAD method is an interface to the standard DB2 LOAD utility, which gives the added performance of using LOAD but without setting additional options for bulk load. This method requires the same privileges as the LOAD method, and is available only in DB2 Version 7 FixPak 4 and later clients and servers. If your client and server can support the CLI LOAD method, you can generally see the best performance by using it. The CLI LOAD method can also be used to load data into a partitioned DB2 database for client and database nodes that are DB2 Version 8.1 or later. To use this method, specify BL_METHOD=CLILOAD as a data set option. Here are the bulk-load options that are available with the CLI LOAD method:

- BL_ALLOW_READ_ACCESS=
- BL_ALLOW_WRITE_ACCESS=
- BL_COPY_LOCATION=
- BL_CPU_PARALLELISM=
- BL_DATA_BUFFER_SIZE=
- BL_DISK_PARALLELISM=

- BL_EXCEPTION
- BL_INDEXING_MODE=
- BL_LOAD_REPLACE=
- BL_LOG=
- BL_METHOD=
- BL_OPTIONS= on page 255
- BL_RECOVERABLE=
- BL_REMOTE_FILE=

Capturing Bulk-Load Statistics into Macro Variables

These bulk-loading macro variables capture how many rows are loaded, skipped, rejected, committed, and deleted before writing this information to the SAS log.

- SYSBL_ROWSCOMMITTED
- SYSBL_ROWSDELETED
- SYSBL_ROWSLOADED
- SYSBL_ROWSREJECTED
- SYSBL_ROWSSKIPPED

Maximizing Load Performance for DB2 under UNIX and PC Hosts

These tips can help you optimize LOAD performance when you are using the DB2 bulk-load facility.

- Specifying BL_REMOTE_FILE= causes the loader to use the DB2 LOAD utility, which is much faster than the IMPORT utility, but it requires database administrator authority.
- Performance might suffer if your setting for DBCOMMIT=is too low. Increase the default (which is 10000 when BULKLOAD=YES) for improved performance.
- Increasing the DB2 tuning parameters, such as Utility Heap and input-output characteristics, improves performance. These parameters are controlled by your database or server administrator.
- When using the IMPORT utility, specify BL_OPTIONS="COMPOUND=x"—where x is a number between 1 and 7 on Windows, and between 1 and 100 on UNIX. This causes the IMPORT utility to insert multiple rows for each execute instead of one row per execute.
- When using the LOAD utility on a multi-processor or multi-node DB2 server, specify BL_OPTIONS="ANYORDER" to improve performance. This might cause DB2 log entries to be out of order because it lets DB2 insert rows in a different order from how they appear in the loader data file.

Examples

The following example shows how to use a SAS data set, SASFLT.FLT98, to create and load a large DB2 table, FLIGHTS98. Because the code specifies BULKLOAD=YES and BL_REMOTE_FILE= is omitted, this load uses the DB2 IMPORT command.

```
libname sasflt 'SAS-library';
libname db2_air db2 user=louis using=fromage
        database='db2_flt' schema=statsdiv;

proc sql;
create table db2_air.flights98
       (bulkload=YES bl_options='compound=7 norowwarnings')
        as select * from sasflt.flt98;
quit;
```

The BL_OPTIONS= option passes DB2 file type modifiers to DB2. The **norowwarnings** modifier indicates that all row warnings about rejected rows are to be suppressed.

The following example shows how to append the SAS data set, SASFLT.FLT98 to a pre-existing DB2 table, ALLFLIGHTS. Because the code specifies BULKLOAD=YES and BL_REMOTE_FILE=, this load uses the DB2 LOAD command.

```
proc append base=db2_air.allflights
 (BULKLOAD=YES
  BL_REMOTE_FILE='/tmp/tmpflt'
  BL_LOG='/tmp/fltdata.log'
  BL_DATAFILE='/nfs/server/tmp/fltdata.ixf'
  BL_SERVER_DATAFILE='/tmp/fltdata.ixf')
data=sasflt.flt98;
run;
```

Here, BL_REMOTE_FILE= and BL_SERVER_DATAFILE= are paths relative to the server. BL_LOG= and BL_DATAFILE= are paths relative to the client.

The following example shows how to use the SAS data set SASFLT.ALLFLIGHTS to create and load a large DB2 table, ALLFLIGHTS. Because the code specifies BULKLOAD=YES and BL_METHOD=CLILOAD, this operation uses the DB2 CLI LOAD interface to the LOAD command.

```
data db2_air.allflights(BULKLOAD=YES BL_METHOD=CLILOAD);
set sasflt.allflights;
run;
```

Locking in the DB2 under UNIX and PC Hosts Interface

The following LIBNAME and data set options let you control how the DB2 under UNIX and PC Hosts interface handles locking. For general information about an option, see "LIBNAME Options for Relational Databases" on page 99. For additional information, see your DB2 documentation.

READ_LOCK_TYPE= ROW | TABLE

UPDATE_LOCK_TYPE= ROW | TABLE

READ_ISOLATION_LEVEL= RR | RS | CS | UR

The DB2 database manager supports the RR, RS, CS, and UR isolation levels that are defined in the following table. Regardless of the isolation level, the database manager places exclusive locks on every row that is inserted, updated, or deleted. All isolation levels therefore ensure that only this application process can change any given row during a unit of work: No other application process can change any rows until the unit of work is complete.

Table 15.3 *Isolation Levels for DB2 under UNIX and PC Hosts*

Isolation Level	Definition
RR (Repeatable Read)	no dirty Reads, no nonrepeatable Reads, no phantom Reads
RS (Read Stability)	no dirty Reads, no nonrepeatable Reads; does allow phantom Reads
CS (Cursor Stability)	no dirty Reads; does allow nonrepeatable Reads and phantom Reads
UR (Uncommitted Read)	allows dirty Reads, nonrepeatable Reads, and phantom Reads

Here is how the terms in the table are defined.

Dirty reads

A transaction that exhibits this phenomenon has very minimal isolation from concurrent transactions. In fact, it can see changes that those concurrent transactions made even before they commit them.

For example, suppose that transaction T1 performs an update on a row, transaction T2 then retrieves that row, and transaction T1 then terminates with rollback. Transaction T2 has then seen a row that no longer exists.

Nonrepeatable reads

If a transaction exhibits this phenomenon, it is possible that it might read a row once and, if it attempts to read that row again later in the course of the same transaction, another concurrent transaction might have changed or even deleted the row. Therefore, the read is not (necessarily) repeatable.

For example, suppose that transaction T1 retrieves a row, transaction T2 then updates that row, and transaction T1 then retrieves the same row again. Transaction T1 has now retrieved the same row twice but has seen two different values for it.

Phantom reads

When a transaction exhibits this phenomenon, a set of rows that it reads once might be a different set of rows if the transaction attempts to read them again.

For example, suppose that transaction T1 retrieves the set of all rows that satisfy some condition. Suppose that transaction T2 then inserts a new row that satisfies that same condition. If transaction T1 now repeats its retrieval request, it sees a row that did not previously exist (a “phantom”).

UPDATE_ISOLATION_LEVEL= CS | RS | RR

The DB2 database manager supports the CS, RS, and RR isolation levels defined in the preceding table. Uncommitted reads are not allowed with this option.

Naming Conventions for DB2 under UNIX and PC Hosts

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

The PRESERVE_TAB_NAMES= and PRESERVE_COL_NAMES= options determine how SAS/ACCESS Interface to DB2 under UNIX and PC Hosts handles case sensitivity, spaces, and special characters. (For information about these options, see "Overview: LIBNAME Statement for Relational Databases " on page 93.) DB2 is not case sensitive and all names default to uppercase.

DB2 objects include tables, views, columns, and indexes. They follow these naming conventions.

- A name can begin with a letter or one of these symbols: dollar sign ($), number or pound sign (#), or at symbol (@).
- A table name must be from 1 to 128 characters long. A column name must from 1 to 30 characters long.
- A name can contain the letters A to Z, any valid letter with a diacritic, numbers from 0 to 9, underscore (_), dollar sign ($), number or pound sign (#), or at symbol (@).
- Names are not case sensitive. For example, the table names **`CUSTOMER`** and **`Customer`** are the same, but object names are converted to uppercase when they are entered. If a name is enclosed in quotation marks, the name is case sensitive.
- A name cannot be a DB2- or an SQL-reserved word, such as WHERE or VIEW.
- A name cannot be the same as another DB2 object that has the same type.

Schema and database names have similar conventions, except that they are each limited to 30 and 8 characters respectively. For more information, see your DB2 SQL reference documentation.

Data Types for DB2 under UNIX and PC Hosts

Overview

Every column in a table has a name and a data type. The data type tells DB2 how much physical storage to set aside for the column and the form in which the data is stored. DB2 uses IBM SQL data types. This section includes information about DB2 data types, null and default values, and data conversions.

For more information about DB2 data types and to determine which data types are available for your version of DB2, see your DB2 SQL reference documentation.

Character Data

BLOB (binary large object)
: contains varying-length binary string data with a length of up to 2 gigabytes. It can hold structured data that user-defined types and functions can exploit. Similar to

FOR BIT DATA character strings, BLOB strings are not associated with a code page.

CLOB (character large object)
: contains varying-length character string data with a length of up to 2 gigabytes. It can store large single-byte character set (SBCS) or mixed (SBCS and multibyte character set, or MBCS) character-based data, such as documents written with a single character set. It therefore has an SBCS or mixed code page associated with it.

String Data

CHAR(*n*)
: specifies a fixed-length column for character string data. The maximum length is 254 characters.

VARCHAR(*n*)
: specifies a varying-length column for character string data. The maximum length of the string is 4000 characters. If the length is greater than 254, the column is a long-string column. SQL imposes some restrictions on referencing long-string columns. For more information about these restrictions, see your IBM documentation.

LONG VARCHAR
: specifies a varying-length column for character string data. The maximum length of a column of this type is 32700 characters. A LONG VARCHAR column cannot be used in certain functions, subselects, search conditions, and so on. For more information about these restrictions, see your IBM documentation.

GRAPHIC(*n*)
: specifies a fixed-length column for graphic string data. *n* specifies the number of double-byte characters and can range from 1 to 127. If *n* is not specified, the default length is 1.

VARGRAPHIC(*n*)
: specifies a varying-length column for graphic string data. *n* specifies the number of double-byte characters and can range from 1 to 2000.

LONG VARGRAPHIC
: specifies a varying-length column for graphic-string data. *n* specifies the number of double-byte characters and can range from 1 to 16350.

Numeric Data

BIGINT
: specifies a big integer. Values in a column of this type can range from –9223372036854775808 to +9223372036854775807. However, numbers that require decimal precision greater than 15 digits might be subject to rounding and conversion errors.

SMALLINT
: specifies a small integer. Values in a column of this type can range from –32768 to +32767.

INTEGER
: specifies a large integer. Values in a column of this type can range from –2147483648 to +2147483647.

FLOAT | DOUBLE | DOUBLE PRECISION
: specifies a floating-point number that is 64 bits long. Values in a column of this type can range from –1.79769E+308 to –2.225E−307 or +2.225E−307 to +1.79769E+308, or they can be 0. This data type is stored the same way that SAS stores its numeric data type. Therefore, numeric columns of this type require the least processing when SAS accesses them.

DECIMAL | DEC | NUMERIC | NUM
: specifies a mainframe-packed decimal number with an implicit decimal point. The precision and scale of the number determines the position of the decimal point. The numbers to the right of the decimal point are the scale, and the scale cannot be negative or greater than the precision. The maximum precision is 31 digits. Numbers that require decimal precision greater than 15 digits might be subject to rounding and conversion errors.

Date, Time, and Timestamp Data

SQL date and time data types are collectively called datetime values. The SQL data types for dates, times, and timestamps are listed here. Be aware that columns of these data types can contain data values that are out of range for SAS.

DATE
: specifies date values in various formats, as determined by the country code of the database. For example, the default format for the United States is *mm-dd-yyyy* and the European standard format is *dd.mm.yyyy*. The range is 01-01-0001 to 12-31-9999. A date always begins with a digit, is at least eight characters long, and is represented as a character string. For example, in the U.S. default format, January 25, 1991, would be formatted as 01-25-1991.

  The entry format can vary according to the edit codes that are associated with the field. For more information about edit codes, see your IBM documentation.

TIME
: specifies time values in a three-part format. The values range from 0 to 24 for hours (*hh*) and from 0 to 59 for minutes (*mm*) and seconds (*ss*). The default form for the United States is *hh:mm:ss*, and the IBM European standard format for time is *hh.mm[.ss]*. For example, in the U.S. default format 2:25 p.m. would be formatted as 14:25:00.

  The entry format can vary according to the edit codes that are associated with the field. For more information about edit codes, see your IBM documentation.

TIMESTAMP
: combines a date and time and adds an optional microsecond to make a seven-part value of the format *yyyy-mm-dd-hh.mm.ss[.nnnnnn].* For example, a timestamp for precisely 2:25 p.m. on January 25, 1991, would be 1991-01-25-14.25.00.000000. Values in a column of this type have the same ranges as described earlier for DATE and TIME.

For more information about SQL data types, datetime formats, and edit codes that are used in the United States and other countries, see your IBM documentation.

DB2 Null and Default Values

DB2 has a special value called NULL. A DB2 NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a DB2 NULL value, it interprets it as a SAS missing value.

You can define a column in a DB2 table so that it requires data. To do this in SQL, you specify a column as NOT NULL. NOT NULL tells SQL to only allow a row to be added to a table if there is a value for the field. For example, NOT NULL assigned to the field CUSTOMER in the table SASDEMO.CUSTOMER does not allow a row to be added unless there is a value for CUSTOMER. When creating a DB2 table with SAS/ACCESS, you can use the DBNULL= data set option to indicate whether NULL is a valid value for specified columns.

DB2 columns can also be defined as NOT NULL WITH DEFAULT. For more information about using the NOT NULL WITH DEFAULT value, see your DB2 SQL reference documentation.

Once you know whether a DB2 column enables NULLs or the host system supplies a default value for a column that is defined as NOT NULL WITH DEFAULT, you can write selection criteria and enter values to update a table. Unless a column is defined as NOT NULL or NOT NULL WITH DEFAULT, it allows NULL values.

For more information about how SAS handles NULL values, see "Potential Result Set Differences When Processing Null Data " on page 35.

To control how the DBMS handles SAS missing character values, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to DB2 assigns to SAS variables when using the LIBNAME statement on page 93 to read from a DB2 table. These default formats are based on DB2 column attributes.

Table 15.4 *LIBNAME Statement: Default SAS Formats for DB2 Data Types*

DB2 Data Type	SAS Data Type	Default SAS Format
BLOB	character	$HEX*n*.
CLOB	character	$*n*.
CHAR(*n*)* VARCHAR(*n*)* LONG VARCHAR	character	$*n*.
GRAPHIC(*n*)* VARGRAPHIC(*n*)* LONG VARGRAPHIC	character	$*n*.
INTEGER	numeric	11.
SMALLINT	numeric	6.
BIGINT	numeric	20.
DECIMAL	numeric	*m.n*
NUMERIC	numeric	*m.n*

DB2 Data Type	SAS Data Type	Default SAS Format
FLOAT	numeric	none
DOUBLE	numeric	none
TIME	numeric	TIME8.
DATE	numeric	DATE9.
TIMESTAMP	numeric	DATETIME*m.n*

* *n* in DB2 data types is equivalent to *w* in SAS formats.

This table shows the default DB2 data types that SAS/ACCESS assigns to SAS variable formats during output operations when you use the LIBNAME statement.

***Table 15.5** LIBNAME Statement: Default DB2 Data Types for SAS Variable Formats*

SAS Variable Format	DB2 Data Type
m.n	DECIMAL (*m,n*)*
other numerics	DOUBLE
$*n*.	VARCHAR(*n*)* (*n*<=4000) LONG VARCHAR(*n*)* (*n*>4000)
datetime formats	TIMESTAMP
date formats	DATE
time formats	TIME

* *n* in DB2 data types is equivalent to *w* in SAS formats.

DBLOAD Procedure Data Conversions

The following table shows the default DB2 data types that SAS/ACCESS assigns to SAS variable formats when you use the DBLOAD procedure on page 901.

***Table 15.6** PROC DBLOAD: Default DB2 Data Types for SAS Variable Formats*

SAS Variable Format	DB2 Data Type
$*w*.	CHAR(*n*)
w.	DECIMAL(*p*)
w.d	DECIMAL(*p,s*)
IB*w.d*, PIB*w.d*	INTEGER

SAS Variable Format	DB2 Data Type
all other numerics*	DOUBLE
datetime*w.d*	TIMESTAMP
date*w.*	DATE
time.**	TIME

* Includes all SAS numeric formats, such as BINARY8 and E10.0.

** Includes all SAS time formats, such as TOD*w,d* and HHMM*w,d*.

Chapter 16

SAS/ACCESS Interface to DB2 for z/OS

Introduction to SAS/ACCESS Interface to DB2 under z/OS

This section describes SAS/ACCESS Interface to DB2 under z/OS. For a list of SAS/ACCESS features that are available in this interface, see "SAS/ACCESS Interface to DB2 under z/OS: Supported Features" on page 79.

Note: z/OS is the successor to the OS/390 (formerly MVS) operating system. SAS 9.1 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.

LIBNAME Statement Specifics for DB2 under z/OS

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to DB2 under z/OS supports and includes an example. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing DB2 under z/OS interface.

LIBNAME *libref* **db2** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`db2`
specifies the SAS/ACCESS engine name for the DB2 under z/OS interface.

connection-options
provides connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. Here is how these options are defined.

USER=<'>user-name<'>
lets you connect to a DB2 database with a user ID that is different from the default ID. The value for this option cannot exceed 8 characters. USER= is optional. If you specify USER=, you must also specify PASSWORD=. If USER= is omitted, your default user ID for your operating environment is used. If you do not specify the SCHEMA= or AUTHID= LIBNAME option, the value of the USER= option (if present) is used as the default schema. Authentication options are not checked when the LIBNAME is issued. They are checked only when a statement involving the libref is run. However, for explicit pass-through, credentials are checked immediately.

PASSWORD=<'>password<'>
specifies the DB2 password that is associated with your DB2 user ID. PASSWORD= is optional. If you specify USER=, you must specify PASSWORD=.

LOCATION=*location*
: maps to the location in the SYSIBM.LOCATIONS catalog in the communication database. In SAS/ACCESS Interface to DB2 under z/OS, the location is converted to the first level of a three-level table name: *location.authid.table*. DB2 Distributed Data Facility (DDF) Communication Database (CDB) makes the connection implicitly to the remote DB2 subsystem when DB2 receives a three-level name in an SQL statement.

LOCATION= is optional. If you omit it, 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= on page 103 option.

SSID=*DB2-subsystem-id*
: specifies the DB2 subsystem ID to connect to at connection time. SSID= is optional. If you omit it, SAS connects to the DB2 subsystem that is specified in the DB2SSID= SAS system option. The DB2 subsystem ID is limited to four characters. For more information, see "Settings" on page 495.

SERVER=*DRDA-server*
: specifies the DRDA server to which you want to connect. SERVER= lets you access DRDA resources stored at remote locations. Check with your system administrator for system names. You can connect to only one server per LIBNAME statement.

SERVER= is optional. If you omit it, you access tables from your local DB2 database unless you have specified a value for the LOCATION= LIBNAME option on page 155.

Default: none.

For information about	See
accessing a database server on Linux, UNIX, or Windows using a libref	the REMOTE_DBTYPE= LIBNAME option on page 178
setting up DB2 z/OS so that SAS can connect to the DRDA server when the SERVER= option is used	installation instructions for this interface
configuring SAS to use the SERVER= option	

LIBNAME-options
: defines how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to DB2 under z/OS, with the applicable default values. For more detail about these options, see "LIBNAME Options for Relational Databases" on page 99.

Table 16.1 *SAS/ACCESS LIBNAME Options*

Option	Default Value
ACCESS=	none
AUTHDOMAIN=	none

Option	Default Value
AUTHID=	your user ID
CONNECTION=	SHAREDREAD
CONNECTION_GROUP=	none
DBCONINIT=	none
DBCONTERM=	none
DBCREATE_TABLE_OPTS=	none
DBGEN_NAME=	DBMS
DBLIBINIT=	none
DBLIBTERM=	none
DBMSTEMP=	NO
DBNULLKEYS=	YES
DBSASLABEL=	COMPAT
DBSLICEPARM=	THREADED_APPS,2
DEFER=	NO
DEGREE=	ANY
DIRECT_EXE=	none
DIRECT_SQL=	YES
IN=	none
LOCATION=	none
MULTI_DATASRC_OPT=	NONE
PRESERVE_COL_NAMES=	NO
PRESERVE_TAB_NAMES=	NO
READBUFF=	1
READ_ISOLATION_LEVEL=	DB2 z/OS determines the isolation level
READ_LOCK_TYPE=	none
REMOTE_DBTYPE=	ZOS

Option	Default Value
REREAD_EXPOSURE=	NO
SCHEMA=	your user ID
SPOOL=	YES
SQL_FUNCTIONS=	none
SQL_FUNCTIONS_COPY=	none
UPDATE_ISOLATION_LEVEL=	DB2 z/OS determines the isolation level
UPDATE_LOCK_TYPE=	none
UTILCONN_TRANSIENT=	YES

DB2 under z/OS LIBNAME Statement Example

In this example, the libref MYLIB uses the DB2 under z/OS interface to connect to the DB2 database that the SSID= option specifies, with a connection to the **testserver** remote server.

```
libname mylib db2 ssid=db2
   authid=testuser server=testserver;
proc print data=mylib.staff;
   where state='CA';
run;
```

Data Set Options for DB2 under z/OS

All SAS/ACCESS data set options in this table are supported for SAS/ACCESS Interface to DB2 under z/OS. Default values are provided where applicable. For general information about this feature, see Data Set Options for Relational Databases on page 207.

***Table 16.2** SAS/ACCESS Data Set Options for DB2 under z/OS*

Option	Default Value
AUTHID=	current LIBNAME option setting
BL_DBCURSOR=	none
BL_DB2DATACLAS=	none
BL_DB2DEVT_PERM=	SYSDA

Option	Default Value
BL_DB2DEVT_TEMP=	SYSDA
BL_DB2DISC=	a generated data set name
BL_DB2ERR=	a generated data set name
BL_DB2IN=	a generated data set name
BL_DB2LDCT1=	none
BL_DB2LDCT2=	none
BL_DB2LDCT3=	none
BK)DB2KDEXT-	GENRUN
BL_DB2MAP=	a generated data set name
BL_DB2MGMTCLAS=	none
BL_DB2PRINT=	a generated data set name
BL_DB2PRNLOG=	YES
BL_DB2REC=	a generated data set name
BL_DB2RECSP=	10
BL_DB2STRT=	NO
BL_DB2SPC_PERM=	10
BL_DB2SPC_TEMP=	10
BL_DB2STORCLAS=	none
BL_DB2TBLXST=	NO
BL_DB2UNITCOUNT=	none
BL_DB2UTID=	user ID and second level DSN qualifier
BULKLOAD=	NO
DBCOMMIT=	current LIBNAME option setting
DBCONDITION=	none
DBCREATE_TABLE_OPTS=	current LIBNAME option setting
DBFORCE=	NO

Option	Default Value
DBGEN_NAME=	DBMS
DBKEY=	none
DBLABEL=	NO
DBMASTER=	none
DBNULL=	YES
DBNULLKEYS=	current LIBNAME option setting
DBSASLABEL=	COMPAT
DBSASTYPE=	see “Data Types for DB2 under z/OS ” on page 503
DBSLICE=	none
DBSLICEPARM=	THREADED_APPS,2
DBTYPE=	none
DEGREE=	ANY
ERRLIMIT=	1
IN=	current LIBNAME option setting
LOCATION=	current LIBNAME option setting
NULLCHAR=	SAS
NULLCHARVAL=	a blank character
PRESERVE_COL_NAMES=	current LIBNAME option setting
READBUFF=	LIBNAME setting
READ_ISOLATION_LEVEL=	current LIBNAME option setting
READ_LOCK_TYPE=	current LIBNAME option setting
TRAP_151=	NO
UPDATE_ISOLATION_LEVEL=	current LIBNAME option setting
UPDATE_LOCK_TYPE=	current LIBNAME option setting

SQL Pass-Through Facility Specifics for DB2 under z/OS

Key Information

For general information about this feature, see "SQL Pass-Through Facility" on page 401. DB2 z/OS examples are available.

Here are the SQL pass-through facility specifics for the DB2 under z/OS interface:

- The *dbms-name* is **DB2**.
- The CONNECT statement is optional.
- The interface supports connections to multiple databases.
- Although you can specify in the CONNECT statement any *database-connection-arguments* from the LIBNAME on page 467, only SSID= on page 468 and SERVER= on page 468 are honored.

Examples

This example connects to DB2 and sends it two EXECUTE statements to process.

```
proc sql;
   connect to db2 (ssid=db2);
   execute (create view testid.whotookorders as
            select ordernum, takenby, firstname,
            lastname, phone
            from testid.orders, testid.employees
            where testid.orders.takenby=
                  testid.employees.empid)
           by db2;
   execute (grant select on testid.whotookorders
            to testuser) by db2;
   disconnect from db2;
quit;
```

This next example omits the optional CONNECT statement, uses the default DB2SSID= setting, and performs a query (shown in highlighting) on the Testid.Customers table.

```
proc sql;
   select * from connection to db2
     (select * from testid.customers where customer like '1%');
   disconnect from db2;
quit;
```

This example creates the Vlib.StockOrd SQL view that is based on the Testid.Orders table. Testid.Orders is an SQL/DS table that is accessed through DRDA.

```
libname vlib 'SAS-library'

proc sql;
  connect to db2 (server=testserver);
  create view vlib.stockord as
```

```
      select * from connection to db2
         (select ordernum, stocknum, shipto, dateorderd
             from testid.orders);
  disconnect from db2;
quit;
```

Autopartitioning Scheme for DB2 under z/OS

Overview

Autopartitioning for SAS/ACCESS Interface to DB2 under z/OS is a modulo (MOD) method. Threaded Reads for DB2 under z/OS involve a trade-off. A threaded Read with even distribution of rows across the threads substantially reduces elapsed time for your SAS step. So your job completes in less time. This is positive for job turnaround time, particularly if your job needs to complete within a constrained period of time. However, threaded Reads always increase the CPU time of your SAS job and the workload on DB2. If increasing CPU consumption or increasing DB2 workload for your job are unacceptable, you can turn threaded Reads off by specifying DBSLICEPARM=NONE. To turn off threaded Reads for all SAS jobs, set DBSLICEPARM=NONE in the SAS restricted options table.

For general information about this feature, see “Autopartitioning Techniques in SAS/ACCESS” on page 63.

Autopartitioning Restrictions

SAS/ACCESS Interface to DB2 under z/OS places additional restrictions on the columns that you can use for the partitioning column during the autopartitioning phase. Here are the column types that you can partition.

- INTEGER
- SMALLINT
- DECIMAL
- You must confine eligible DECIMAL columns to an integer range—specifically, DECIMAL columns with precision that is less than 10. For example, DECIMAL(5,0) and DECIMAL(9,2) are eligible.

Column Selection for MOD Partitioning

If multiple columns are eligible for partitioning, the engine queries the DB2 system tables for information about identity columns and simple indexes. Based on the information about the identity columns, simple indexes, column types, and column nullability, the partitioning column is selected in order by priority.

1. Identity column
2. Unique simple index: SHORT or INT, integral DECIMAL, and then nonintegral DECIMAL
3. Nonunique simple index: SHORT or INT (NOT NULL), integral DECIMAL (NOT NULL), and then nonintegral DECIMAL (NOT NULL)

4. Nonunique simple index: SHORT or INT (nullable), integral DECIMAL (nullable), and then nonintegral DECIMAL (nullable)
5. SHORT or INT (NOT NULL), integral DECIMAL (NOT NULL), and then nonintegral DECIMAL (NOT NULL)
6. SHORT or INT (nullable), integral DECIMAL (nullable), and then nonintegral DECIMAL (nullable)

If a nullable column is selected for autopartitioning, the SQL statement **`OR<column-name>IS NULL`** is appended at the end of the SQL code that is generated for one read thread. This ensures that any possible NULL values are returned in the result set.

How WHERE Clauses Restrict Autopartitioning

Autopartitioning does not select a column to be the partitioning column if it appears in a SAS WHERE clause. For example, the following DATA step cannot use a threaded Read to retrieve the data because all numeric columns in the table (see the table definition in "Using DBSLICE= " on page 475) are in the WHERE clause.

```
data work.locemp;
   set trlib.MYEMPS;
   where EMPNUM<=30 and ISTENURE=0 and
         SALARY<=35000 and NUMCLASS>2;
run;
```

Using DBSLICEPARM=

SAS/ACCESS Interface to DB2 under z/OS defaults to two threads when you use autopartitioning.

Using DBSLICE=

You can achieve the best possible performance when using threaded Reads by specifying the DBSLICE= data set option on page 299 for DB2 in your SAS operation.

Temporary Table Support for DB2 under z/OS

Establishing a Temporary Table

For general information about this feature, see "Temporary Table Support for SAS/ACCESS " on page 42.

To make full use of temporary tables, the CONNECTION=GLOBAL connection option is necessary. You can use this option to establish a single connection across SAS DATA step and procedure boundaries. The connection can also be shared between the LIBNAME statement and the SQL pass-through facility. Because a temporary table only exists within a single connection, you must be able to share this single connection among all steps that reference the temporary table. The temporary table cannot be referenced from any other connection.

The type of temporary table that is used for this processing is created using the DECLARE TEMPORARY TABLE statement with the ON COMMIT PRESERVE clause. This type of temporary table lasts for the duration of the connection—unless it is explicitly dropped—and retains its rows of data beyond commit points.

To create a temporary table, use a PROC SQL pass-through statement. To use both the SQL pass-through facility and librefs to reference a temporary table, you need to specify DBMSTEMP=YES in a LIBNAME statement that persists beyond the PROC SQL step. The global connection then persists across SAS DATA steps and even multiple PROC SQL steps, as shown in this example.

```
libname temp db2 connection=global;

proc sql;
   connect to db2 (connection=global);
   exec (declare global temporary table temptab1
         like other.table on commit PRESERVE rows) by db2;
quit;
```

At this point, you can refer to the temporary table by using the Temp libref or the CONNECTION=GLOBAL option with a PROC SQL step.

Terminating a Temporary Table

You can drop a temporary table at any time, or allow it to be implicitly dropped when the connection is terminated. Temporary tables do not persist beyond the scope of a singe connection.

Examples

These examples are based on these assumptions:

- There is a DeptInfo table on the DBMS that has all of your department information.
- You have a SAS data set with join criteria that you want to use to retrieve certain rows from the DeptInfo table.
- You have another SAS data set with updates to the DeptInfo table.

Here are the librefs and temporary tables that are used.

```
libname saslib base 'my.sas.library';
libname dept db2 connection=global schema=dschema;
libname temp db2 connection=global schema=SESSION;
/* Note that temporary table has a schema of SESSION */

proc sql;
   connect to db2 (connection=global);
   exec (declare global temporary table temptab1
        (dname char(20), deptno int)
         on commit PRESERVE rows) by db2;
quit;
```

To perform a homogeneous join on the DBMS, the next example shows how to use a heterogeneous join with a temporary table. It does this instead of reading the DBMS table into SAS to perform the join. Using the previously created table, SAS data is copied into the temporary table to perform the join.

```
proc append base=temp.temptab1 data=saslib.joindata;
run;
proc sql;
   connect to db2 (connection=global);
```

```
   select * from dept.deptinfo info, temp.temptab1 tab
        where info.deptno = tab.deptno;
   /* remove the rows for the next example */
   exec(delete from session.temptab1) by db2;
quit;
```

In this example, transaction processing on the DBMS occurs using a temporary table. It does this instead of using either DBKEY= or MULTI_DATASRC_OPT=IN_CLAUSE with a SAS data set as the transaction table.

```
proc append base=temp.temptab1 data=saslib.transdat;
run;

proc sql;
   connect to db2 (connection=global);
   exec(update dschema.deptinfo d set deptno = (select deptn from
        session.temptab1)
        where d.dname = (select dname from session.temptab1)) by db2;
quit;
```

Calling Stored Procedures in DB2 under z/OS

Overview

A stored procedure is one or more SQL statements or supported third-generation languages (3GLs, such as C) statements that are compiled into a single procedure that exists in DB2. Stored procedures might contain static (hard-coded) SQL statements. Static SQL is optimized better for some DBMS operations. In a carefully managed DBMS environment, programmers and database administrators can know the exact SQL to execute.

SAS usually generates SQL dynamically. However, the database administrator can encode static SQL in a stored procedure and therefore restrict SAS users to a tightly controlled interface. When you use a stored procedure call, you must specify a schema.

SAS/ACCESS support for stored procedure includes passing input parameters, retrieving output parameters into SAS macro variables, and retrieving the result set into a SAS table. Although DB2 stored procedures can return multiple result sets, SAS/ACCESS Interface to DB2 under z/OS can retrieve only a single result set.

You can call stored procedures only from PROC SQL.

Examples

Example 1: Specify a Basic Call

Use CALL statement syntax to call a stored procedure.

```
call "schema".stored_proc
```

The simplest way to call a stored procedure is to use the EXECUTE statement in PROC SQL. In this example, you execute STORED_PROC by using a CALL statement. SAS does not capture the result set.

```
proc sql;
connect to db2;
```

```
execute (call "schema".stored_proc);
quit;
```

Example 2: Specify One Input Parameter That Returns a Result Set

You can also return the result set to a SAS table. In this example, STORED_PROC is executed using a CALL statement. The result is returned to a SAS table, SasResults.

```
proc sql;
connect to db2;
create table sasresults as select * from connection to db2 (call "schema".stored_proc);
quit;
```

Example 3: Specify Three Output Parameters

The CALL statement syntax supports passing of parameters. You can specify such input parameters as numeric constants, character constants, or a null value. You can also pass input parameters by using SAS macro variable references. To capture the value of an output parameter, a SAS macro variable reference is required. This example uses a constant (1), an input/output parameter (:INOUT), and an output parameters (:OUT). Not only is the result set returned to the SAS results table, the SAS macro variables INOUT and OUT capture the parameter outputs.

```
proc sql;
connect to db2;
%let INOUT=2;
create table sasresults as select * from connection to db2
   (call "schema".stored_proc (1,:INOUT,:OUT));
quit;
```

Example 4: Pass a NULL Parameter

In these calls, NULL is passed as the parameter to the DB2 stored procedure.

- Null string literals in the call

  ```
  call proc('');
  call proc("")
  ```

- Literal period or literal NULL in the call

  ```
  call proc(.)
  call proc(NULL)
  ```

- SAS macro variable set to NULL string

  ```
  %let charparm=;
  call proc(:charparm)
  ```

- SAS macro variable set to period (SAS numeric value is missing)

  ```
  %let numparm=.;
  call proc(:numparm)
  ```

Only the literal period and the literal NULL work generically for both DB2 character parameters and DB2 numeric parameters. For example, a DB2 numeric parameter would reject "" and **`%let numparm=.;`** would not pass a DB2 NULL for a DB2 character parameter. As a literal, a period passes NULL for both numeric and character parameters. However, when it is in a SAS macro variable, it constitutes a NULL only for a DB2 numeric parameter.

You cannot pass NULL parameters by omitting the argument. For example, you cannot use this call to pass three NULL parameters.

```
call proc(,,)
```

You could use this call instead.

```
call proc(NULL,NULL,NULL)
```

Example 5: Specify a Schema

Use standard CALL statement syntax to execute a stored procedure that exists in another schema, as shown in this example.

```
proc sql;
   connect to db2;
   execute (call otherschema.stored_proc);
quit;
```

If the schema is in mixed case or lowercase, enclose the schema name in double quotation marks.

```
proc sql;
   connect to db2;
   execute (call "lowschema".stored_proc);
quit;
```

Example 6: Execute Remote Stored Procedures

If the stored procedure exists on a different DB2 instance, specify it with a valid three-part name.

```
proc sql;
   connect to db2;
   create table sasresults as select * from connection to db2
          (call otherdb2.procschema.prod5 (1, NULL));
quit;
```

ACCESS Procedure Specifics for DB2 under z/OS

Key Information

See the ACCESS procedure on page 881 for general information about this feature. DB2 under z/OS examples on page 480 are available.

SAS/ACCESS Interface to DB2 under z/OS supports all ACCESS procedure statements on page 882 in interactive line, noninteractive, and batch modes.

Here are the ACCESS procedure specifics for the DB2 under z/OS interface.

- The DBMS= value is **db2.**
- The *database-connection-arguments* are SSID= on page 468, SERVER= on page 468, and LOCATION=.

  LOCATION=*location*
  : lets you further qualify where a table is located. In the DB2 z/OS engine, 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 when DB2 receives a three-level table name in an SQL statement.

LOCATION= is optional. If you omit it, SAS accesses the data from the local DB2 database.

- Here is the TABLE= statement:

  TABLE= <*authorization-id.*>*table-name*
  : identifies the DB2 table or DB2 view that you want to use to create an access descriptor. The *table-name* is limited to 18 characters. The TABLE= statement is required.

    The *authorization-id* is a user ID or group ID that is associated with the DB2 table. The authorization ID is limited to eight characters. If you omit the authorization ID, DB2 uses your TSO (or z/OS) user ID. In batch mode, however, you must specify an authorization ID, otherwise an error message is generated.

Examples

This example creates an access descriptor and a view descriptor that are based on DB2 data.

```
options linesize=80;
libname adlib 'SAS-library';
libname vlib 'SAS-library';

proc access dbms=db2;

   /* create access descriptor  */
   create adlib.customr.access;
   table=testid.customers;
   ssid=db2;
   assign=yes;
   rename customer=custnum;
   format firstorder date7.;
   list all;

   /* create vlib.usacust view */
   create vlib.usacust.view;
   select customer state zipcode name
          firstorder;
   subset where customer like '1%';
run;
```

This next example uses the SERVER= statement to access the SQL/DS table Testid.Orders from a remote location. Access and view descriptors are then created based on the table.

```
libname adlib 'SAS-library';
libname vlib 'SAS-library';

proc access dbms=db2;
  create adlib.customr.access;
  table=testid.orders;
  server=testserver;
  assign=yes;
  list all;

 create vlib.allord.view;
```

```
  select ordernum stocknum shipto dateorderd;

    subset where stocknum = 1279;
run;
```

DBLOAD Procedure Specifics for DB2 under z/OS

Key Information

See the DBLOAD procedure on page 901 for general information about this feature. DB2 z/OS examples on page 481 are available.

SAS/ACCESS Interface to DB2 under z/OS supports all DBLOAD procedure statements on page 902 in interactive line, noninteractive, and batch modes.

Here are the DBLOAD procedure specifics for SAS/ACCESS Interface to DB2 under z/OS.

- The DBMS= value is **`DB2.`**
- The *database-connection-arguments* are SSID= on page 468 and SERVER= on page 468.
- Here is the NULLS= statement:

  NULLS *variable-identifier-1* =Y|N|D < . . . *variable-identifier-n* =Y|N|D >

  lets you specify whether DB2 columns that are associated with the listed SAS variables allow NULL values. By default, all columns accept NULL values.

  The NULLS statement accepts any one of these values.

  - Y: Specifies that the column accepts NULL values. This is the default.
  - N: Specifies that the column does not accept NULL values.
  - D: Specifies that the column is defined as NOT NULL WITH DEFAULT.

  See "DB2 Null and Default Values" on page 505 for information about NULL values that is specific to DB2.

- Here is the TABLE= statement:

  TABLE= <*authorization-id.*>*table-name*;

  identifies the DB2 table or DB2 view that you want to use to create an access descriptor. The *table-name* is limited to 18 characters. The TABLE= statement is required.

  The *authorization-id* is a user ID or group ID that is associated with the DB2 table. The authorization ID is limited to eight characters. If you omit the authorization ID, DB2 uses your TSO (or z/OS) user ID. However, in batch mode, you must specify an authorization ID or an error message is generated.

Examples

This example creates a new DB2 table, Testid.Invoice, from the Dlib.Invoice data file. The AmtBilled column and the fifth column in the table (AmountInUS) are renamed. You must have the appropriate privileges before you can create new DB2 tables.

```
libname adlib 'SAS-library';
libname dlib 'SAS-library';

proc dbload dbms=db2 data=dlib.invoice;
   ssid=db2;
   table=testid.invoice;
   accdesc=adlib.invoice;
   rename amtbilled=amountbilled
          5=amountindollars;
   nulls invoicenum=n amtbilled=n;
   load;
run;
```

For example, you can create a SAS data set, Work.Schedule, that includes the names and work hours of your employees. You can use the SERVER= command to create the DB2 table, Testid.Schedule, and load it with the schedule data on the DRDA resource, TestServer, as shown in this example.

```
libname adlib 'SAS-library';

proc dbload dbms=db2 data=work.schedule;
  in sample;
  server=testserver;
  accdesc=adlib.schedule;
  table=testid.schedule;
  list all;
  load;
run;
```

The DB2EXT Procedure

Overview

The DB2EXT procedure creates SAS data sets from DB2 under z/OS data. PROC DB2EXT runs interactively, noninteractively, and in batch mode. The generated data sets are not password protected. However, you can edit the saved code to add password protection.

PROC DB2EXT ensures that all SAS names that are generated from DB2 column values are unique. A numeric value is appended to the end of a duplicate name. If necessary, the procedure truncates the name when appending the numeric value.

Syntax

Here is the syntax for the DB2EXT procedure.

PROC DB2EXT <*options*>;

FMT *column-number-1*='*SAS-format-name-1*'
<... *column-number-n*='*SAS-format-name-n*'>;

RENAME *column-number–1*='*SAS-name–1*'
<... *column-number–n*='*SAS-name–n*'>;

SELECT *DB2–SQL-statement*;

EXIT;

PROC DB2EXT Statement Options

IN=*SAS-data-set*
: specifies a mapping data set that contains information such as DB2 names, SAS variable names, and formats for input to PROC DB2EXT. This option is available for use only with previously created mapping data sets. You cannot create new mapping data sets with DB2EXT.

OUT=*SAS-data-set* | *libref.SAS-data-set*
: specifies the name of the SAS data set that is created. If you omit OUT=, the data set is named "work.DATAn", where n is a number that is sequentially updated. The data set is not saved when your SAS session ends. If a file with the name that you specify in the OUT= option already exists, it is overwritten. However, you receive a warning that this is going to happen.

SSID=*subsystem-name*
: specifies the name of the DB2 subsystem that you want to access. If you omit SSID=, the subsystem name defaults to DB2. The subsystem name defaults to the subsystem that is defined in the DB2SSID= option. It defaults to DB2 only if neither the SSID= option nor the DB2SSID= option are specified.

FMT Statement

FMT *column-number-1*='*SAS-format-name-1*'
<... *column-number-n*='*SAS-format-name-n*'>;

The FMT statement assigns a SAS output format to the DB2 column that is specified by *column-number*. The *column-number* is determined by the order in which you list the columns in your SELECT statement. If you use SELECT *, the *column-number* is determined by the order of the columns in the database. You must enclose the format name in single quotation marks. You can specify multiple column formats in a single FMT statement.

RENAME Statement

RENAME *column-number-1*='*SAS-name-1*'
<... *column-number-n*='*SAS-name-n*'>;

The RENAME statement assigns the *SAS-name* to the DB2 column that is specified by *column-number*. The *column-number* is determined by the order in which you list the columns in your SELECT statement. If you use SELECT *, the *column-number* is determined by the order of the columns in the database.

You can rename multiple columns in a single RENAME statement.

SELECT Statement

SELECT *DB2-SQL-statement*;

The *DB2-SQL-statement* defines the DB2 data that you want to include in the SAS data set. You can specify table names, column names, and data subsets in your SELECT statement. For example, this statement selects all columns from the Employee table and includes only employees whose salary is greater than $40,000.

```
select * from employee where salary > 40000;
```

EXIT Statement

EXIT;

The EXIT statement terminates the procedure without further processing.

Examples

This code creates a SAS data set named MyLib.NoFmt that includes three columns from the DB2 table EmplInfo. The RENAME statement changes the name of the third column that is listed in the SELECT statement (from **firstname** in the DB2 table to **fname** in the SAS data set.

```
/* specify the SAS library where the SAS data set is to be saved */

libname mylib 'userid.xxx';

proc db2ext ssid=db25 out=mylib.nofmt;
   select employee, lastname, firstname from sasdemo.emplinfo;
   rename 3=fname;
run;
```

This code uses a mapping file to specify which data to include in the SAS data set and how to format that data.

```
/* specify the SAS library where the SAS data set is to be saved */
libname mylib 'userid.xxx';

/* specify the SAS library that contains the mapping data set */
libname inlib 'userid.maps';

proc db2ext in=inlib.mapping out=mylib.mapout ssid=db25;
run;
```

The DB2UTIL Procedure

Overview

You can use the DB2UTIL procedure to insert, update, or delete rows in a DB2 table using data from a SAS data set. You can choose one of two methods of processing: creating an SQL output file or executing directly. PROC DB2UTIL runs interactively, noninteractively, or in batch mode.

Support for the DB2UTIL procedure provides compatibility with SAS 5 version of SAS/ACCESS Interface to DB2 under z/OS. It is not added to other SAS/ACCESS DBMS interfaces, and enhancement of this procedure for future releases of SAS/ACCESS are not guaranteed. It is recommended that you write new applications by using LIBNAME features.

The DB2UTIL procedure uses the data in an input SAS data set, along with your mapping specifications, to generate SQL statements that modify the DB2 table. The DB2UTIL procedure can perform these functions.

DELETE
: deletes rows from the DB2 table according to the search condition that you specify.

INSERT
: builds rows for the DB2 table from the SAS observations, according to the map that you specify, and inserts the rows.

UPDATE
: sets new column values in your DB2 table by using the SAS variable values that are indicated in your map.

When you execute the DB2UTIL procedure, you specify an input SAS data set, an output DB2 table, and how to modify the data. To generate data, you must also supply instructions for mapping the input SAS variable values to the appropriate DB2 columns.

In each execution, the procedure can generate and execute SQL statements to perform one type of modification only. However, you can also supply your own SQL statements (except the SQL SELECT statement) to perform various modifications against your DB2 tables, and the procedure executes them.

For more information about the types of modifications that are available and how to use them, see "Modifying DB2 Data" on page 487. For an example of how to use this procedure, see the PROC DB2UTIL example on page 488.

DB2UTIL Syntax

The PROC DB2UTIL statement calls the DB2UTIL procedure on page 484. These statements are used with PROC DB2UTIL.

PROC DB2UTIL <*options*>;

MAPTO *SAS-name-1=DB2-name-1* <*…SAS-name-n=DB2-name-n*>;

RESET ALL|*SAS-name*| COLS;

SQL *SQL-statement*;

UPDATE;

WHERE *SQL-WHERE-clause*;

ERRLIMIT=*error-limit*;

EXIT;

DB2UTIL Options

DATA=*SAS-data-set* | <*libref.*>*SAS-data-set*
: specifies the name of the SAS data set that contains the data with which you want to update the DB2 table. DATA= is required unless you specify an SQL file with the SQLIN= option.

TABLE=*DB2-tablename*
: specifies the name of the DB2 table that you want to update. TABLE= is required unless you specify an SQL file with the SQLIN= option.

FUNCTION= D | I | U | DELETE | INSERT | UPDATE
: specifies the type of modification to perform on the DB2 table by using the SAS data set as input. See "Modifying DB2 Data" on page 487 for a detailed description of this option. FUNCTION= is required unless you specify an SQL file with the SQLIN= option.

COMMIT=*number*
: specifies the maximum number of SQL statements to execute before issuing an SQL COMMIT statement to establish a synchpoint. The default is 3.

ERROR=*fileref*|*fileref.member*
: specifies an external file where error information is logged. When DB2 issues an error return code, the procedure writes all relevant information, including the SQL statement that is involved, to this external file. If you omit the ERROR= statement, the procedure writes the error information to the SAS log.

LIMIT=*number*
: specifies the maximum number of SQL statements to issue in an execution of the procedure. The default value is 5000. If you specify LIMIT=0, no limit is set. The procedure processes the entire data set regardless of its size.

SQLIN=*fileref* | *fileref.member*
: specifies an intermediate SQL output file that is created by a prior execution of PROC DB2UTIL by using the SQLOUT= option. The file that is specified by SQLIN= contains SQL statements to update a DB2 table. If you specify an SQLIN= file, the procedure reads the SQL statements and executes them in line mode. When you specify an SQLIN= file, DATA=, TABLE=, and SQLOUT= are ignored.

SQLOUT=*fileref* | *fileref.member*
: specifies an external file where the generated SQL statements are to be written. This file is either a z/OS sequential data set or a member of a z/OS partitioned data set. Use this option to update or delete data. When you specify the SQLOUT= option, the procedure edits your specifications, generates the SQL statements to perform the update, and writes them to the external file for later execution. When they are input to the later run for execution, the procedure passes them to DB2.

SSID=*subsystem-name*
: specifies the name of the DB2 subsystem that you want to access. If you omit DB2SSID=, the subsystem name defaults to DB2. See “Settings” on page 495 for more information.

DB2UTIL Statements

MAPTO Statement

MAPTO *SAS-name-1=DB2-name-1*<… SAS-name-n=DB2-name-n>;

The MAPTO statement maps the SAS variable name to the DB2 column name. You can specify as many values in one MAPTO statement as you want.

RESET Statement

RESET ALL | *SAS-name* | COLS;

Use the RESET statement to erase the editing that was done to SAS variables or DB2 columns. The RESET statement can perform one or more of these actions:

ALL
: resets all previously entered map and column names to default values for the procedure.

SAS-name
: resets the map entry for that SAS variable.

COLS
: resets the altered column values.

SQL Statement

SQL *SQL-statement*;

The SQL statement specifies an SQL statement that you want the procedure to execute dynamically. The procedure rejects SQL SELECT statements.

UPDATE Statement

UPDATE;

The UPDATE statement causes the table to be updated by using the mapping specifications that you supply. If you do not specify an input or an output mapping data set or an SQL output file, the table is updated by default.

If you have specified an output mapping data set in the SQLOUT= option, PROC DB2UTIL creates the mapping data set and ends the procedure. However, if you specify UPDATE, the procedure creates the mapping data set and updates the DB2 table.

WHERE Statement

WHERE *SQL-WHERE-clause*;

The WHERE statement specifies the SQL WHERE clause that you want to use to update the DB2 table. This statement is combined with the SQL statement generated from your mapping specifications. Any SAS variable names in the WHERE clause are substituted at that time, as shown in this example.

```
where db2col = %sasvar;
```

ERRLIMIT Statement

ERRLIMIT=*error-limit*;

The ERRLIMIT statement specifies the number of DB2 errors that are permitted before the procedure terminates.

EXIT Statement

EXIT;

The EXIT statement exits from the procedure without further processing. No output data is written, and no SQL statements are issued.

Modifying DB2 Data

Overview

The DB2UTIL procedure generates SQL statements by using data from an input SAS data set. However, the SAS data set plays a different role for each type of modification that is available through PROC DB2UTIL. These sections show how you use each type and how each type uses the SAS data set to make a change in the DB2 table.

Inserting Data

You can insert observations from a SAS data set into a DB2 table as rows in the table. To use this insert function, name the SAS data set and the DB2 table in the PROC DB2UTIL statement. You can then use the MAPTO statement to map values from SAS variables to columns in the DB2 table. If you do not want to insert the values for all variables in the SAS data set into the DB2 table, map only the variables that you want to insert. However, you must map all DB2 columns to a SAS column.

Updating Data

You can change the values in DB2 table columns by replacing them with values from a SAS data set. You can change a column value to another value for every row in the table, or you can change column values only when certain criteria are met. For example, you can change the value of the NUM DB2 column to 10 for every row in the table. You can also change the value of the NUM DB2 column to the value in the NUMBER SAS variable if the DB2 column name value and the SAS data set variable name match.

You specify the name of the SAS data set and the DB2 table to be updated when you execute PROC DB2UTIL. You can specify that only certain variables be updated by naming only those variables in your mapping specifications.

You can use the WHERE clause to specify that only the rows on the DB2 table that meet certain criteria are updated. For example, you can use the WHERE clause to specify that only the rows with a certain range of values are updated. Or you can specify that rows to be updated when a certain column value in the row matches a certain SAS variable value in the SAS data set. In this case, you could have a SAS data set with several observations in it. For each observation in the data set, the DB2UTIL procedure updates the values for all rows in the DB2 table that have a matching value. The the procedure then goes on to the next observation in the SAS data set and continues to update values in DB2 columns in rows that meet the comparison criteria.

Deleting Data

You can remove rows from a DB2 table when a certain condition is met. You can delete rows from the table when a DB2 column value in the table matches a SAS variable value in the SAS data set. Name the DB2 table from which you want to delete rows and the SAS data set that contains the target deletion values in the PROC DB2UTIL statement. You can then use the WHERE statement to specify the DB2 column name and the SAS variable whose values must match before the deletion is performed. To delete values that are based on criteria other than values in SAS data variables (for example, deleting every row with a department number of 600), you can use an SQL DELETE statement.

PROC DB2UTIL Example

This example uses the UPDATE function in PROC DB2UTIL to update a list of telephone extensions from a SAS data set. The master list of extensions is in the DB2 table Testid.Employees and is updated from the SAS data set Trans. First, create the SAS data set.

```
options db2dbug;

data trans;
   empno=321783;
   ext='3999';
   output;
   empno=320001;
   ext='4321';
   output;
   empno=212916;
   ext='1300';
   output;
run;
```

Next, specify the data set in PROC DB2UTIL.

```
proc db2util data=trans table=testid.employees function=u;
   mapto ext=phone;
```

```
    where empid=%empno;
    update;
run;
```

The row that includes EMPID=320001 is not found in the Testid.Employees table and is therefore not updated. You can ignore the warning in the SAS log.

Maximizing DB2 under z/OS Performance

Assessing When to Tune Performance

Among the factors that affect DB2 performance are the size of the table that is being accessed and the form of the SQL SELECT statement. If the table that is being accessed is larger than 10,000 rows (or 1,000 pages), you should evaluate all SAS programs that access the table directly. When you evaluate the programs, consider these questions.

- Does the program need all columns that the SELECT statement retrieves?
- Do the WHERE clause criteria retrieve only those rows that are needed for subsequent analysis?
- Is the data going to be used by more than one procedure in one SAS session? If so, consider extracting the data into a SAS data file for SAS procedures to use instead of allowing the data to be accessed directly by each procedure.
- Do the rows need to be in a particular order? If so, can an indexed column be used to order them? If there is no index column, is DB2 doing the sort?
- Do the WHERE clause criteria allow DB2 to use the available indexes efficiently?
- What type of locks does DB2 need to acquire?
- Are the joins being passed to DB2?
- Can your DB2 system use parallel processing to access the data more quickly?

In addition, the DB2 Resource Limit Facility limits execution time of dynamic SQL statements. If the time limit is exceeded, the dynamic statement is terminated and the SQL code -905 is returned. This list describes several situations in which the RLF could stop a user from consuming large quantities of CPU time.

- An extensive join of DB2 tables with the SAS SQL procedure.
- An extensive search by the FSEDIT, FSVIEW, or FSBROWSE procedures or an SCL application.
- Any extensive extraction of data from DB2.
- An extensive select.
- An extensive load into a DB2 table. In this case, you can break up the load by lowering the commit frequency, or you can use the bulk-load facility on page 496 through SAS/ACCESS Interface to DB2 under z/OS.

Methods for Improving Performance

You can do several things in your SAS application to improve DB2 engine performance.

- Set the SAS system option DB2DBUG. This option prints to the SAS log the dynamic SQL that the DB2 engine generated and all other SQL that the DB2 engine executed. You can then verify that all WHERE clauses, PROC SQL joins, and ORDER BY clauses are being passed to DB2. This option is for debugging purposes and should not be set once the SAS application is used in production. The NODB2DBUG option disables this behavior.

- Verify that all SAS procedures and DATA steps that read DB2 data share connections where possible. You can do this by using one libref to reference all SAS applications that read DB2 data and by accepting the default value of SHAREDREAD for the CONNECTION= option.
- If your DB2 subsystem supports parallel processing, you can assign a value to the CURRENT DEGREE special register. Setting this register might enable your SQL query to use parallel operations. You can set the special register by using the LIBNAME options DBCONINIT= or DBLIBINIT= with the SET statement as shown in this example:

```
libname mydb2 db2 dbconinit="SET CURRENT DEGREE='ANY'";
```

- Use the view descriptor WHERE clause or the DBCONDITION= option to pass WHERE clauses to DB2. You can also use these methods to pass sort operations to DB2 with the ORDER BY clause instead of performing a sort within SAS.
- If you are using a SAS application or an SCL application that reads the DB2 data twice, let the DB2 engine spool the DB2 data. This happens by default because the default value for the SPOOL= option is YES.

  The spool file is read both when the application rereads the DB2 data and when the application scrolls forward or backward through the data. If you do not use spooling but need to scroll backward through the DB2 table, the DB2 engine must start reading from the beginning of the data and down to the row to which you want to scroll back.
- Use the SQL procedure to pass joins to DB2 instead of using the MATCH MERGE capability (that is, merging with a BY statement) of the DATA step.
- Use the DBKEY= option when you are doing SAS processing that involves the KEY= option. When you use the DBKEY= option, the DB2 engine generates a WHERE clause that uses parameter markers. During the execution of the application, the values for the key are substituted into the parameter markers in the WHERE clause.

  If you do not use the DBKEY= option, the entire table is retrieved into SAS, and the join is performed in SAS.
- Consider using stored procedures when they can improve performance in client-server applications by reducing network traffic. You can execute a stored procedure by using the DBCONINIT= or DBLIBINIT= LIBNAME options.
- Use the READBUFF= LIBNAME option on page 174 to retrieve records in blocks instead of one at a time.

Optimizing Your Connections

Since SAS 7, the DB2 engine supports more than one connection to DB2 per SAS session. This is an improvement over SAS 6 in a number of ways, especially in a server environment. One advantage is being able to separate tasks that fetch rows from a cursor from tasks that must issue commits. This separation eliminates having to resynchronize the cursor, prepare the statement, and fetch rows until you are positioned back on the row that you were on. It also enables tasks that must issue commits to eliminate locking contention to do this sooner because they are not delayed until after cursors are closed to prevent having to resynchronize. In general, tables that are opened for input fetch from cursors do not issue commits, although update openings might, and output openings do, issue commits.

You can control how the DB2 engine uses connections by using the CONNECTION= on page 113 option in the LIBNAME statement. At one extreme is CONNECTION=UNIQUE, which causes each table access, whether it is for input, update, or output, to create and use its own connection. Conversely, CONNECTION=SHARED means that only one connection is made, and that input, update, and output accesses all share that connection.

The default value for the CONNECTION= option is CONNECTION=SHAREDREAD, which means that tables opened for input share one connection. Update and output openings obtain their own connections. CONNECTION=SHAREDREAD allows for the best separation between tasks that fetch from cursors and tasks that must issue commits, eliminating the resynchronizing of cursors.

The values GLOBAL and GLOBALREAD perform similarly to SHARED and SHAREDREAD. The difference is that you can share the given connection across any of the librefs that you specify as GLOBAL or GLOBALREAD.

Although the default value of CONNECTION=SHAREDREAD is usually optimal, at times another value might be better. If you must use multiple librefs, you might want to set them each as GLOBALREAD. In this case, you have one connection for all of your input openings, regardless of which libref you use, as opposed to one connection per libref for input openings. In a single-user environment (as opposed to a server session), you might know that you do not have multiple openings occurring at the same time. In this case, you might want to use SHARED—or GLOBAL for multiple librefs. By using such a setting, you eliminate the overhead of creating separate connections for input, update, and output transactions. If you have only one opening at a time, you eliminate the problem of resynchronizing input cursors if a commit occurs.

Another reason for using SHARED or GLOBAL is the case of opening a table for output while opening another table within the same database for input. This can result in a -911 deadlock situation unless both opens occur in the same connection.

As explained in “DB2 under z/OS Information for the Database Administrator” on page 511, the first connection to DB2 is made from the main SAS task. Subsequent connections are made from corresponding subtasks, which the DB2 engine attaches; DB2 allows only one connection per task. Due to the system overhead of intertask communication, the connection established from the main SAS task is a faster connection in terms of CPU time. Because this is true, you can expect better performance (less CPU time) if you use the first connection for these operations when you read or write large numbers of rows. If you read-only rows, SHAREDREAD or GLOBALREAD can share the first connection. However, if you are both reading and writing rows (input and output opens), you can use CONNECTION=UNIQUE to make each opening use the first connection. UNIQUE causes each opening to have its own connection. If you have only one opening at a time and some are input while others are output (for large amounts of data), the performance benefit of using the main SAS task connection far outweighs the overhead of establishing a new connection for each opening.

The utility connection is another type of connection that the DB2 engine uses, which the use does not control. This connection is a separate connection that can access the system catalog and issue commits to release locks. Utility procedures such as DATASETS and CONTENTS can cause this connection to be created, although other actions necessitate it as well. There is one connection of this type per libref, but it is not created until it is needed. If you have critical steps that must use the main SAS task connection for performance reasons, refrain from using the DEFER=YES option in the LIBNAME statement. It is possible that the utility connection can be established from that task, causing the connection that you use for your opening to be from a slower subtask.

In summary, no one value works best for the CONNECTION= option in all possible situations. You might need to try different values and arrange your SAS programs in different ways to obtain the best performance possible.

Passing SAS Functions to DB2 under z/OS

SAS/ACCESS Interface to DB2 under z/OS passes the following SAS functions to DB2 for processing if the DBMS driver or client that you are using supports this function. Where the DB2 function name differs from the SAS function name, the DB2 name appears in parentheses. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

ABS	MIN
ARCOS (ACOS)	MINUTE
ARSIN (ASIN)	MOD
ATAN	QTR (QUARTER)
AVG	REPEAT
CEIL	RIGHT (RTRIM)
COS	SECOND
COSH	SIGN
COUNT	SIN
DTEXTDAY	SINH
DTEXTMONTH	SQRT
DTEXTWEEKDAY	STRIP
DTEXTYEAR	SUBSTR
EXP	SUM
FLOOR	TAN
HOUR	TANH
INDEX (LOCATE)	TRANWRD (REPLACE)
LEFT (LTRIM)	TRIMN (RTRIM)
LOWCASE (LCASE)	TRUNC
LOG	UPCASE (UCASE)
LOG10	WEEKDAY (DAYOFWEEK)
MAX	

SQL_FUNCTIONS= on page 185ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to DB2. Due to incompatibility in date and time functions between DB2 and SAS, DB2 might not process them correctly. Check your results to determine whether these functions are working as expected.

DATEPART (DATE)	TODAY (CURRENT DATE)
LENGTH	TRANSLATE
TIMEPART (TIME)	

These functions are not passed to the DBMS in DB2 V5. None of them existed in DB2 prior to DB2 V6. Also, there is no way to determine what location you are connected to and which functions are supported when you connect using DRDA on page 512.

The following functions *are* passed to the DBMS in DB2 V5, V6, and later. They are not passed to the DBMS when you connect using DRDA.

YEAR
MONTH
DAY

Passing Joins to DB2 under z/OS

With these exceptions, multiple libref joins are passed to DB2 z/OS.

- If you specify the SERVER= option for one libref, you must also specify it for the others, and its value must be the same for all librefs.
- If you specify the DIRECT_SQL= option for one or multiple librefs, you must not set it to NO, NONE, or NOGENSQL.

For completeness, the portable code checks these options, regardless of the engine:

- DBCONINIT=
- DBCONTERM=
- DBLIBINIT=
- DBLIBTERM=
- DIRECT_EXE=
- DIRECT_SQL=
- PRESERVE_COL_NAMES=
- PRESERVE_TAB_NAMES=

For more information about when and how SAS/ACCESS passes joins to the DBMS, see “Passing Joins to the DBMS” on page 47.

SAS System Options, Settings, and Macros for DB2 under z/OS

System Options

You can use these SAS system options when you start a SAS session that accesses DB2 under z/OS.

DB2DBUG | NODB2DBUG
: used to debug SAS code. When you submit a SAS statement that accesses DB2 data, DB2DBUG displays any DB2 SQL queries (generated by SAS) that are processed by DB2. The queries are written to the SAS log. NODB2DBUG is the default.

For example, if you submit a PROC PRINT statement that references a DB2 table, the DB2 SQL query is displayed in the SAS log. SAS/ACCESS Interface to DB2 under z/OS generates this query.

```
libname mylib db2 ssid=db2;

proc print data=mylib.staff;
run;

proc sql;
select * from mylib.staff
   order by idnum;
quit;
```

DB2 statements that appear in the SAS log are prepared and described to determine whether the DB2 table exists and can be accessed.

DB2DECPT=*decimal-value*
: specifies the setting of the DB2 DECPOINT= option. The *decpoint-value* argument can be a period (.) or a comma (,). The default is a period (.).

  DB2DECPT= is valid as part of the configuration file when you start SAS.

DB2IN= '*database-name.tablespace-name*' | 'DATABASE *database-name*'
: lets you specify the database and table space in which you want to create a new table. The DB2IN= 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 and table space.

  database.tablespace specifies the names of the database and table space.

  'DATABASE *database-name*' specifies only the database name. Enclose the entire specification in single quotation marks.

  You can override the DB2IN= system option with the IN= LIBNAME or data set option.

DB2PLAN=*plan-name*
: specifies the name of the plan that is used when connecting (or binding) SAS to DB2. SAS provides and supports this plan, which can be adapted for each user's site. The value for DB2PLAN= can be changed at any time during a SAS session, so that different plans can be used for different SAS steps. However, if you use more than one plan during a single SAS session, you must understand how and when SAS/ACCESS Interface to DB2 under z/OS makes the connections. If one plan is in effect and you specify a new plan, the new plan does not affect the existing DB2 connections.

DB2RRS | NODB2RRS
: specifies the attachment facility to be used for a SAS session when connecting to DB2. This option is an invocation-only option.

  Specify NODB2RRS, the default, to use the Call Attachment Facility (CAF). Specify DB2RRS to use the Recoverable Resource Manager Services Attachment Facility (RRSAF). For details about using RRSAF, see "How the Interface to DB2 Works" on page 511.

DB2RRSMP | NODB2RRSMP
: specifies that the multiphase SRRCMIT commit and SRRBACKrollback calls are used instead of the COMMIT and ROLLBACK SQL statements. This option is ignored unless DB2RRS is specified. This option is available only at invocation.

  Specify NODB2RRSMP, the default, when DB2 is the only Resource Manager for your application. Specify DB2RRSMP when your application has other resource managers, which requires the use of the multiphase calls. Using the multiphase calls when DB2 is your only resource manager can have performance implications. Using COMMIT and ROLLBACK when you have more than one resource manager can result in an error, depending on the release of DB2.

DB2SSID=*subsystem-name*
: specifies the DB2 subsystem name. The *subsystem-name* argument is one to four characters that consist of letters, numbers, or national characters (#, $, or @); the first character must be a letter. The default value is DB2. For more information, see "Settings" on page 495.

  DB2SSID= is valid in the OPTIONS statement, as part of the configuration file, and when you start SAS.

  You can override the DB2SSID= system option with the SSID= connection option on page 467.

DB2UPD=Y | N
: specifies whether the user has privileges through SAS/ACCESS Interface to DB2 under z/OS to update DB2 tables. This option applies only to the user's Update privileges through the interface and not necessarily to the user's privileges while using DB2 directly. Altering the setting of DB2UPD= has no effect on your DBMS privileges, which have been set with the GRANT statement. The default is Y (Yes).

  DB2UPD= is valid in the OPTIONS statement, as part of the configuration file, and when you start SAS. This option does not affect the SQL pass-through facility, PROC DBLOAD, or the SAS 5 compatibility procedures.

Settings

To connect to DB2, you must specify a valid DB2 subsystem name in one of these ways.

- the DB2SSID= system option. SAS/ACCESS Interface to DB2 under z/OS uses this value if no DB2 subsystem is specified.
- the SSID= option in the PROC ACCESS statement
- the SSID= statement of PROC DBLOAD
- the SSID= option in the PROC SQL CONNECT statement, which is part of the SQL pass-through facility
- the SSID= connection option in the LIBNAME statement

If a site does not specify a valid DB2 subsystem when it accesses DB2, this message is generated:

```
ERROR: Cannot connect to DB2 subsystem XXXX,
  rc=12, reason code = 00F30006.  See the
  Call Attachment Facility documentation
  for an explanation.
```

XXXX is the name of the subsystem to which SAS tried to connect. To find the correct value for your DB2 subsystem ID, contact your database administrator.

Macros

Use the automatic SYSDBRC macro variable to capture DB2 return codes when using the DB2 engine. The macro variable is set to the last DB2 return code that was encountered only when execution takes place through SAS/ACCESS Interface to DB2 under z/OS. If you reference SYSDBRC before engine processing takes place, you receive this message:

```
WARNING: Apparent symbolic reference SYSDBRC not resolved.
```

Use SYSDBRC for conditional post-processing. Below is an example of how to abend a job. The table DB2TEST is dropped from DB2 after the view descriptor is created, resulting in a -204 code.

```
data test;
x=1;
y=2;
proc dbload dbms=db2 data=test;
table=db2test;
   in 'database test';
load;
run;

proc access dbms=db2;
create work.temp.access;
table=user1.db2test;
create work.temp.view;
select all;
run;
proc sql;
execute(drop table db2test)by db2;
quit;

proc print data=temp;
run;

data _null_;
if "&sysdbrc" not in ('0','100') then
 do;
   put 'The DB2 Return Code is: ' "&sysdbrc";
   abort abend;
 end;
run;
```

Because the abend prevents the log from being captured, you can capture the SAS log by using the SAS system option, ALTLOG.

Bulk Loading for DB2 under z/OS

Overview

By default, the DB2 under z/OS interface loads data into tables by preparing an SQL INSERT statement, executing the INSERT statement for each row, and issuing a COMMIT statement. You must specify BULKLOAD=YES to start the DB2 LOAD utility. You can then bulk-load rows of data as a single unit, which can significantly enhance performance. For smaller tables, the extra overhead of the bulk-loading process might slow performance. For larger tables, the speed of the bulk-loading process outweighs the overhead costs. DB2 under z/OS bulk-loading examples on page 499 are available.

When you use bulk load, see the SYSPRINT output for information about the load. If you run the LOAD utility and it fails, ignore the messages in the SAS log because they might be inaccurate. However, if errors existed before you ran the LOAD utility, error messages in the SAS log might be valid.

SAS/ACCESS Interface to DB2 under z/OS provides bulk loading through DSNUTILS, an IBM stored procedure that start the DB2 LOAD utility. DSNUTILS is included in DB2 Version 6 and later, and it is available for DB2 Version 5 in a maintenance release. Because the LOAD utility is complex, familiarize yourself with it before you use it through SAS/ACCESS. Also check with your database administrator to determine whether this utility is available.

Data Set Options for Bulk Loading

Below are the DB2 under z/OSbulk-load data set options. All begin with BL_ for bulk load. To use the bulk-load facility, you must specify BULKLOAD=YES or all bulk-load options are ignored. (The DB2 under z/OS interface alias for BULKLOAD= is DB2LDUTIL=.)

- BL_DB2CURSOR=
- BL_DB2DATACLAS=
- BL_DB2DEVT_PERM=
- BL_DB2DEVT_TEMP=
- BL_DB2DISC=
- BL_DB2ERR=
- BL_DB2IN=
- BL_DB2LDCT1=
- BL_DB2LDCT2=
- BL_DB2LDCT3=
- BL_DB2LDEXT=
- BL_DB2MAP=
- BL_DB2MGMTCLAS=
- BL_DB2PRINT=
- BL_DB2PRNLOG=
- BL_DB2REC=
- BL_DB2RECSP=
- BL_DB2RSTRT=
- BL_DB2SPC_PERM=
- BL_DB2SPC_TEMP=
- BL_DB2STORCLAS=
- BL_DB2TBLXST=
- BL_DB2UNITCOUNT=
- BL_DB2UTID=
- BULKLOAD=

File Allocation and Naming for Bulk Loading

When you use bulk loading, these files (data sets) are allocated.

- The DB2 DSNUTILS procedure allocates these as new and catalogs the SysDisc, SysMap, and SysErr files unless BL_DB2LDEXT=USERUN. If BL_DB2LDEXT=USERUN, data sets are allocated as old and are kept.
- The DB2 interface engine allocates as new and catalogs the files SysIn and SysRec when the execution method specifies to generate them.
- The DB2 interface engine allocates as new and catalogs the file SysPrint when the execution method specifies to run the utility.

All allocations of these data sets are reversed by the end of the step. If errors occur before SysRec is generated, any of these data sets that were allocated as new and cataloged are deleted as part of cleanup because they would be empty.

The interface engine uses these options when it allocates nonexisting SYS data set names.

- DSNUTILS uses BL_DB2DEVT_PERM= and BL_DB2SPC_PERM= for SysDisc, SysMap, and SysErr.
- The DB2 interface engine uses BL_DB2DEVT_PERM= for SysIn, SysRec, and SysPrint.
- SysRec uses BL_DB2RECSPC=. BL_DB2RECSPC= is necessary because the engine cannot determine how much space the SysRec requires—it depends on the volume of data being loaded into the table.
- DSNUTILs uses BL_DB2DEVT_TEMP= and BL_DB2SPC_TEMP= to allocate the other data set names that the LOAD utility requires.

This table shows how SysIn and SysRec are allocated based on the values of BL_DB2LDEXT= and BL_DB2IN=, and BL_DB2REC=.

Table 16.3 *SysIn and SysRec Allocation*

BL_DB2LDEXT=	BL_DB2IN=/ BL_DB2REC=	Data set name	DISPOSITION
GENRUN	not specified	generated	NEW, CATALOG, DELETE
GENRUN	specified	specified	NEW, CATALOG, DELETE
GENONLY	not specified	generated	NEW, CATALOG, DELETE
GENONLY	specified	specified	NEW, CATALOG, DELETE
USERUN	not specified	ERROR	
USERUN	specified	specified	OLD, KEEP, KEEP

When SAS/ACCESS Interface to DB2 under z/OS uses existing files, you must specify the filenames. When the interface generates the files, it creates them with names that you provide or with unique names that it generates. Engine-generated filenames use system-generated data set names with the format *SYSyyddd.Thhmmss.RA000.jobname.name.Hgg*, where

SYSyyddd
: is replaced by the user ID. The user ID that is used to prequalify these generated data set names is determined the same as within the rest of SAS, except when running in a server environment, where the authenticated ID of the client is used.

name
: is replaced by the given SYS ddname of the data set.

For example, if you do not specify any data set names and run GENRUN under TSO, you obtain a set of files allocated with names such as

```
USERID.T125547.RA000.USERID.DB2DISC.H01
USERID.T125547.RA000.USERID.DB2ERR.H01
USERID.T125547.RA000.USERID.DB2IN.H01
USERID.T125547.RA000.USERID.DB2MAP.H01
USERID.T125547.RA000.USERID.DB2PRINT.H01
USERID.T125547.RA000.USERID.DB2REC.H01
```

This naming convention produces unique names, even within a sysplex (within one second per user ID per system). It therefore makes it easy to associate all information for each utility execution and separate it from other executions.

Bulk-load files are removed at the end of the load process to save space. They are not removed if the utility fails to allow for the load process to be restarted.

Examples

Use these LIBNAME statements for all examples.

```
libname db2lib db2;
libname shlib db2 connection=shared;
```

Create a table.

```
data db2lib.table1 (bulkload=yes);
   x=1;
   name='Tom';
run;
```

Append Table1 to itself.

```
data shlib.table1
   (bulkload=yes bl_db2tblxst=yes bl_db2ldct1='RESUME YES');
   set shlib.table1;
run;
```

Replace Table1 with itself.

```
data shlib.table1
   (bulkload=yes bl_db2tblxst=yes bd_db2ldct1='REPLACE');
   set shlib.table1;
run;
```

Load DB2 tables directly from other objects.

```
data db2lib.emp (bulkload=yes);
   bl_db2ldct1='replace log no nocopypend'
   bl_db2cursor='select * from dsn8710.emp');
   set db2lib.emp (obs=0);
run;
```

You can also use this option in a PROC SQL statement to load DB2 tables directly from other objects, as shown below.

```
options sastrace=',,,d';
libname db2lib db2 authid=dsn8710;
libname mylib db2;

proc datasets library=mylib;
  delete emp;run;

proc sql;
     connect to db2;
     create table mylib.emp
            (BULKLOAD=YES
             BL_DB2LDCT1='REPLACE LOG NO NOCOPYPEND'
             BL_DB2CURSOR='SELECT FIRSTNAME, LASTNAME, WORKDEPT,
                                  HIREDATE, JOB, SALARY, BONUS, COMM
                           FROM DSN8710.EMP')
     as select firstname, lastname, workdept,
               hiredate, job, salary, bonus, comm
        from db2lib.emp (obs=0);
quit;
```

Here is another similar example.

```
options sastrace=',,,d';
libname db2lib db2 authid=dsn8710;
libname mylib db2;

proc datasets library=mylib;
  delete emp;run;

proc sql;
     connect to db2;
     create table mylib.emp
            (BULKLOAD=YES
             BL_DB2LDCT1='REPLACE LOG NO NOCOPYPEND'
             BL_DB2CURSOR='SELECT FIRSTNAME, LASTNAME, WORKDEPT,
                                  HIREDATE, JOB, SALARY, BONUS, COMM
                           FROM DSN8710.EMP'
             BL_DB2LDCT3='RUNSTATS TABLESPACE DSNDB04.TEMPTTABL
                          TABLE(ALL) INDEX(ALL) REPORT YES')
     as select firstname, lastname, workdept,
               hiredate, job, salary, bonus, comm
        from db2lib.emp (obs=0);
quit;
```

Generate control and data files, create the table, but do not run the utility to load it.

```
data shlib.table2 (bulkload=yes
   bl_db2ldext=genonly bl_db2in='userid.sysin' bl_db2rec='userid.sysrec');
   set shlib.table1;
run;
```

Use the control and data files that you generated in the preceding example load the table. The OBS=1 data set option on the input file prevents the DATA step from reading the whole file. Because the data is really in SysRec, you need only the input file to satisfy the engine.

```
data db2lib.table2 (bulkload=yes bl_db2tblxst=yes
     bl_db2ldext=userun bl_db2in='userid.sysin' bl_db2rec='userid.sysrec');
   set db2lib.table1 (obs=1);
run;
```

A more efficient approach than the previous example is to eliminate going to DB2 to read even one observation from the input table. This also means that the DATA step processes only one observation, without any input I/O. Note that the one variable V is not on the table. Any variables listed here (there is no need for more than one), are irrelevant because the table already exists; they are not used.

```
data db2lib.table2 (bulkload=yes bl_db2tblxst=yes
     bl_db2ldext=userun bl_db2in='userid.sysin' bl_db2rec='userid.sysrec');
   v=0;
run;
```

Generate control and data files, but do not create the table or run the utility. Setting BL_DB2TBLXST=YES when the table does not exist prevents you from creating the table; this only makes sense because you are not going to load any data into the table at this time.

```
data db2lib.table3 (bulkload=yes bl_db2tblxst=yes
     bl_db2ldext=genonly bl_db2in='userid.sysin' bl_db2rec='userid.sysrec');
   set db2lib.table1;
run;
```

Use the control and data files that you generated in the preceding example to load the table. The OBS=1 data set option on the input file prevents the DATA step from reading the whole file. In this case, you must specify the input file because it contains the column definitions that are necessary to create the table.

```
data shlib.table3 (bulkload=yes bl_db2ldext=userun
     bl_db2in='userid.sysin' bl_db2rec='userid.sysrec');
   set shlib.table1 (obs=1);
run;
```

If you know the column names, a more efficient approach than the previous example is to eliminate going to DB2 to obtain the column definitions. In this case, the variable names and data types must match, because they are used to create the table. However, the values specified for the variables are not included on the table, because all data to load comes from the existing SysRec.

```
data db2lib.table3 (bulkload=yes bl_db2ldext=userun
     bl_db2in='userid.sysin' bl_db2rec='userid.sysrec');
   x=0;
   name='???';
run;
```

You can use other applications that do output processing.

```
data work.a;
   x=1;
run;

proc sql;
   create db2lib.table4 (bulkload=yes) as select * from a;
quit;
```

Locking in the DB2 under z/OS Interface

The following LIBNAME and data set options let you control how the DB2 under z/OS interface handles locking. For general information about an option, see "LIBNAME Statement Syntax for Relational Databases" on page 95. For additional information, see your DB2 documentation.

READ_LOCK_TYPE=TABLE

UPDATE_LOCK_TYPE=TABLE

READ_ISOLATION_LEVEL= CS | UR | RR | "RR KEEP UPDATE LOCKS" | RS | "RS KEEP UPDATE LOCKS"

Here are the valid values for this option. DB2 determines the default isolation level.

Table 16.4 *Isolation Levels for DB2 under z/OS*

Value	Isolation Level
CS	Cursor stability
UR	Uncommitted read
RR	Repeatable read
RR KEEP UPDATE LOCKS*	Repeatable read keep update locks
RS	Read stability
RS KEEP UPDATE LOCKS*	Read stability keep update locks

* When specifying a value that consists of multiple words, enclose the entire string in quotation marks.

UPDATE_ISOLATION_LEVEL= CS | UR | RR | "RR KEEP UPDATE LOCKS" | RS | "RS KEEP UPDATE LOCKS"

The valid values for this option are described in the preceding table. The default isolation level is determined by DB2.

Naming Conventions for DB2 under z/OS

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= LIBNAME options determine how SAS/ACCESS Interface to DB2 under z/OS handles case sensitivity, spaces, and special characters. The default for both of these options is NO. Although DB2 is case-sensitive, it converts table and column names to uppercase by default. To preserve the case of the table and column names that you send to DB2, enclose them in quotation marks. For information about these options, see "Overview: LIBNAME Statement for Relational Databases " on page 93.

DB2 objects include tables, views, columns, and indexes. They follow these naming conventions.

- These objects must have names of the following length in characters: column (1–30), index (1–18), table (1–18), view (1–18), alias (1–18), synonym (1–18), or correlation (1–128). However, SAS limits table names to 32 bytes. This limitation prevents database table objects that are defined through a DATA step—for example, to have names that are longer than 32.

  These objects must have names from 1–8 characters long: authorization ID, referential constraint, database, table space, storage group, package, or plan.

  A location name can be 1–16 characters long.
- A name must begin with a letter. If the name is in quotation marks, it can start with and contain any character. Depending on how your string delimiter is set, quoted strings can contain quotation marks such as “O'Malley”.
- A name can contain the letters A–Z, numbers from 0–9, number or pound sign (#), dollar sign ($), or at symbol (@).
- Names are not case sensitive. For example, **`CUSTOMER`** and **`Customer`** are the same. However, if the name of the object is in quotation marks, it is case sensitive.
- A name cannot be a reserved word in DB2.
- A name cannot be the same as another DB2 object. For example, each column name within the same table must be unique.

Data Types for DB2 under z/OS

Overview

Every column in a table has a name and a data type. The data type tells DB2 how much physical storage to set aside for the column and the form in which the data is stored. This section includes information about DB2 data types, NULL and default values, and data conversions.

For more information about DB2 data types, see your DB2 SQL reference documentation.

SAS/ACCESS Interface to DB2 under z/OS supports all DB2 data types.

String Data

CHAR(*n*)
: specifies a fixed-length column of length *n* for character string data. The maximum for *n* is 255.

VARCHAR(*n*)
: specifies a varying-length column for character string data. *n* specifies the maximum length of the string. If *n* is greater than 255, the column is a long string column. DB2 imposes some restrictions on referencing long string columns.

LONG VARCHAR
: specifies a varying-length column for character string data. DB2 determines the maximum length of this column. A column that is defined as LONG VARCHAR is always a long string column and is therefore subject to referencing restrictions.

GRAPHIC(*n*), VARGRAPHIC(*n*), LONG VARGRAPHIC
: specifies graphic strings and is comparable to the types for character strings. However, *n* specifies the number of double-byte characters, so the maximum value for *n* is 127. If *n* is greater than 127, the column is a long string column and is subject to referencing restrictions.

Numeric Data

SMALLINT
: specifies a small integer. Values in a column of this type can range from –32,768 to +32,767.

INTEGER | INT
: specifies a large integer. Values in a column of this type can range from –2,147,483,648 to +2,147,483,647.

REAL | FLOAT(*n*)
: specifies a single-precision, floating-point number. If *n* is omitted or if *n* is greater than 21, the column is double-precision. Values in a column of this type can range from approximately –7.2E+75 through 7.2E+75.

FLOAT(*n*) | DOUBLE PRECISION | FLOAT | DOUBLE
: specifies a double-precision, floating-point number. *n* can range from 22 through 53. If *n* is omitted, 53 is the default. Values in a column of this type can range from approximately –7.2E+75 through 7.2E+75.

DECIMAL(*p,s*) | DEC(*p,s*)
: specifies a packed-decimal number. *p* is the total number of digits (precision) and *s* is the number of digits to the right of the decimal point (scale). The maximum precision is 31 digits. The range of *s* is $0 \le s \le p$.

  If *s* is omitted, 0 is assigned and *p* might also be omitted. Omitting both *s* and *p* results in the default DEC(5,0). The maximum range of *p* is $1 - 10^{31}$ to $10^{31} - 1$.

Even though the DB2 numeric columns have these distinct data types, the DB2 engine accesses, inserts, and loads all numerics as FLOATs.

Date, Time, and Timestamp Data

DB2 date and time data types are similar to SAS date and time values in that they are stored internally as numeric values and are displayed in a site-chosen format. The DB2 data types for dates, times, and timestamps are listed here. Note that columns of these data types might contain data values that are out of range for SAS, which handles dates from 1582 A.D. through 20,000 A.D.

DATE
: specifies date values in the format YYYY-MM-DD. For example, January 25, 1989, is input as 1989-01-25. Values in a column of this type can range from 0001-01-01 through 9999-12-31.

TIME
: specifies time values in the format HH.MM.SS. For example, 2:25 p.m. is input as 14.25.00. Values in a column of this type can range from 00.00.00 through 24.00.00.

TIMESTAMP
: combines a date and time and adds a microsecond to make a seven-part value of the format YYYY-MM-DD-HH.MM.SS.MMMMMM. For example, a timestamp for precisely 2:25 p.m. on January 25, 1989, is 1989-01-25-14.25.00.000000. Values in a

column of this type can range from 0001-01-01-00.00.00.000000 through 9999-12-31-24.00.00.000000.

DB2 Null and Default Values

DB2 has a special value that is called NULL. A DB2 NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a DB2 NULL value, it interprets it as a SAS missing value.

DB2 columns can be defined so that they do not allow NULL data. For example, NOT NULL would indicate that DB2 does not allow a row to be added to the TestID.Customers table unless there is a value for CUSTOMER. When creating a DB2 table with SAS/ACCESS, you can use the DBNULL= on page 294 data set option to indicate whether NULL is a valid value for specified columns.

You can also define DB2 columns as NOT NULL WITH DEFAULT. The following table lists default values that DB2 assigns to columns that you define as NOT NULL WITH DEFAULT. An example of such a column is STATE in Testid.Customers. If a column is omitted from a view descriptor, default values are assigned to the column. However, if a column is specified in a view descriptor and it has no values, no default values are assigned.

Table 16.5 *Default Values That DB2 Assigns for Columns Defined as NOT NULL WITH DEFAULT*

DB2 Column Type	DB2 Default*
CHAR(*n*) | GRAPHIC(*n*)	blanks, unless the NULLCHARVAL= option is specified
VARCHAR | LONG VARCHAR | VARGRAPHIC | LONG VARGRAPHIC	empty string
SMALLINT | INT | FLOAT | DECIMAL | REAL	0
DATE	current date, derived from the system clock
TIME	current time, derived from the system clock
TIMESTAMP	current timestamp, derived from the system clock

* The default values that are listed in this table pertain to values that DB2 assigns.

Knowing whether a DB2 column allows NULL values or whether DB2 supplies a default value can assist you in writing selection criteria and in entering values to update a table. Unless a column is defined as NOT NULL or NOT NULL WITH DEFAULT, the column allows NULL values.

For more information about how SAS handles NULL values, see “Potential Result Set Differences When Processing Null Data ” on page 35.

To control how the DBMS handles SAS missing character values, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to DB2 assigns to SAS variables when using the LIBNAME statement on page 93 to read from a DB2 table. These default formats are based on DB2 column attributes.

***Table 16.6** LIBNAME Statement: Default SAS Formats for DB2 Data Types*

DB2 Column Type	Default SAS Format
CHAR(*n*) VARCHAR(*n*) LONG VARCHAR(*n*)	$*n*
GRAPHIC(*n*) VARGRAPHIC(*n*) LONG VARGRAPHIC	$*n*.(*n*<=127) $127. (*n*>127)
INTEGER	*m.n*
SMALLINT	*m.n*
DECIMAL(*m,n*)	*m.n*
FLOAT	none
NUMERIC(*m,n*)	*m.n*
DATE	DATE9.
TIME	TIME8.
DATETIME	DATETIME30.6

This table shows the default DB2 data types that SAS/ACCESS assigns to SAS variable formats during output operations.

***Table 16.7** LIBNAME Statement: Default DB2 Data Types for SAS Variable Formats*

SAS Variable Format	DB2 Data Type
$*w*. $CHAR*w*. $VARYING*w*. $HEX*w*.	CHARACTER(*w*) for 1–255 VARCHAR(*w*) for >255
any date format	DATE
any time format	TIME

SAS Variable Format	DB2 Data Type
any datetime format	TIMESTAMP
all other numeric formats	FLOAT

ACCESS Procedure Data Conversions

The following table shows the default SAS variable formats that SAS/ACCESS assigns to DB2 data types when you use the ACCESS procedure on page 881.

Table 16.8 *ACCESS Procedure: Default SAS Formats for DB2 Data Types*

DB2 Column Type	Default SAS Format
CHAR(*n*)	$*n*. (*n*<=199)
VARCHAR(*n*)	$*n*. $200. (*n*>200)
LONG VARCHAR	$*n*.
GRAPHIC(*n*) VARGRAPHIC(*n*) LONG VARGRAPHIC	$*n*.(*n*<=127) $127. (*n*>127)
INTEGER	11.0
SMALLINT	6.0
DECIMAL(*m,n*)	*m*+2.*s* for example, DEC(6,4) = 8.4
REAL	E12.6
DOUBLE PRECISION	E12.6
FLOAT(*n*)	E12.6
FLOAT	E12.6
NUMERIC(*m,n*)	*m.n*
DATE	DATE7.
TIME	TIME8.
DATETIME	DATETIME30.6

You can use the YEARCUTOFF= option to make your DATE7. dates comply with Year 2000 standards. For more information about this SAS system option, see *SAS System Options: Reference*.

DBLOAD Procedure Data Conversions

The following table shows the default DB2 data types that SAS/ACCESS assigns to SAS variable formats when you use the DBLOAD procedure on page 901.

Table 16.9 *DBLOAD Procedure: Default DB2 Data Types for SAS Variable Formats*

SAS Variable Format	DB2 Data Type
$*w*. $CHAR*w*. $VARYING*w*. $HEX*w*.	CHARACTER
any date format	DATE
any time format	TIME
any datetime format	TIMESTAMP
w.d IB, IBR, PIB, PIBR all other numeric formats	DECIMAL(*w.d*) INTEGER FLOAT

Understanding DB2 under z/OS Client/Server Authorization

Libref Connections

When you use the DB2 interface, you can enable each client to control its own connections using its own authority—instead of sharing connections with other clients—by using the DB2 Recoverable Resource Manager Services Attachment Facility (RRSAF). See "DB2 Attachment Facilities (CAF and RRSAF) " on page 513 for information about this facility.

When you use SAS/ACCESS Interface to DB2 under z/OS with RRSAF, the authorization mechanism works differently than it does in Base SAS:

- In Base SAS, the SAS server *always* validates the client's authority before allowing the client to access a resource.
- In SAS/ACCESS Interface to DB2 under z/OS (with RRSAF), DB2 checks the authorization identifier that is carried by the connection from the SAS server. In most situations, this is the client's authorization identifier. In one situation, however, this is the SAS server's authorization identifier. A client can access a resource by

using the *server's* authorization identifier only if the client uses a libref that was predefined in the server session.

In this next example, a user assigns the libref SRVPRELIB in the SRV1 server session. In the client session, a user then issues a LIBNAME statement that makes a logical assignment using the libref MYPRELIB, and the user specifies the LIBNAME option SERVER=srv1. The client can then access resources by using the server's authority for the connection.

1. In the server session

```
libname srvprelib db2 ssid=db25;
proc server id=srv1;
run;
```

2. In the client session

```
libname myprelib server=srv1 slibref=srvprelib;
proc print data=myprelib.db2table;
run;
```

In this example, because the client specifies a regular libref, MYDBLIB, the client has its own authority for the connections.

1. In the server session

```
libname myprelib db2 ssid=db25;
proc server id=srv1;
run;
```

2. In the client session

```
libname mydblib server=srv1 roptions='ssid=db25' rengine=db2;
proc print data=mydblib.db2table;
run;
```

In this table, SAS/SHARE clients use LIBNAME statements to access SAS libraries and DB2 data through the server. In this description, a *logical* LIBNAME statement is a statement that associates a libref with another libref that was previously assigned.

Table 16.10 *Librefs and Their Authorization Implications*

Client Session	

libname local v8 'SAS.library' disp=old;
 	
libname dblocal db2 connection=unique;
	These statements execute in the client session. these are local assignments. The authority ID is the ID of the client.

libname remote 'SAS.library' server=serv1 rengine=v8 roptions='disp=old'; libname dbremote server=serv1 rengine=db2 roptions='connection=unique';	These statements execute in the server session on behalf of the client. Libref Remote is a Base SAS engine remote assignment. Libref DbRemote is a DB2 engine remote assignment. In both cases, the authority ID is the ID of the client.
Server Session (id=serv1)	

libname predef v8 'SAS.library' disp=old; libname dbpredef db2 connection=unique;	Because librefs PreDef and DbPreDef are defined in the server session, they can be referenced only by a client using a logical LIBNAME statement. There is no authority ID because clients cannot access these librefs directly.
Logical Assignments - Client Session	
libname alias (local); libname dbalias (dblocal);	These statements create aliases ALIAS and DBALIAS for librefs Local and DbLocal, which were assigned in the client session above. The authority ID is the ID of the client.
libname logic server=serv1 slibref=predef; libname dblogic server=serv1 slibref=dbpredef;	These statements refer to librefs PreDef and DbPreDef, which were assigned in the server session above. Libref Logic is a Base SAS engine logical assignment of remote libref PreDef. The authority ID for libref Logic is the ID of the client. Libref DbLogic is a DB2 engine logical assignment of remote libref DbPreDef. The authority ID for libref DbLogic is the ID of the server.

For the Base SAS engine Remote and Logic librefs, it is the client's authority that is verified. (This is true for all Base SAS engine assignments.) DbRemote and DbLogic DB2 engine librefs refer to the same resources. However, it is the client's authority that is verified for DbRemote, whereas it is the server's authority that is verified for DbLogic. When using the DB2 interface, you can determine whether to use the client's authority or the server's authority to access DB2 data.

Non-Libref Connections

When you make connections using the SQL pass-through facility or view descriptors, the connections to the database are not based on a DB2 engine libref. A connection that is created in the server, by using these features from a client, always has the authority of the client, because there is no server-established connection to reference.

This example uses the SAS/SHARE Remote SQL pass-through facility. The client has its own authority for the connections.

1. In the server session:

   ```
   proc server id=srv1;
   run;
   ```

2. In the client session

   ```
   proc sql;
      connect to remote (server=srv1 dbms=db2 dbmsarg=(ssid=db25));
      select * from connection to remote
         (select * from db2table);
      disconnect from remote;
   quit;
   ```

This example uses a previously created view descriptor. The client has its own authority for the connections. The PreLib libref PreLib that was previously assigned and the client-assigned libref MyLib have no relevant difference. These are Base SAS engine librefs and not DB2 engine librefs.

1. In the server session

```
libname prelib V8 'SAS.library';
proc server id=srv1;
run;
```

2. In the client session

```
libname prelib server=srv1;
proc print data=prelib.accview;
run;
```

3. In the client session

```
libname mylib 'SAS.library2' server=srv1 rengine=v8;
proc print data=mylib.accview;
run;
```

Known Issues with RRSAF Support

SAS/SHARE can use various communication access methods to communicate with clients. You can specify these through the COMAMID and COMAUX1 system options.

When you use XMS (Cross Memory Services) as an access method, DB2 also uses XMS in the same address space. Predefining DB2 server librefs before starting PROC SERVER can result in errors due to the loss of the XMS Authorization Index, because both SAS and DB2 are acquiring and releasing it. When using XMS as an access method, use only client-assigned librefs on the server.

This problem does not occur when you use the TCP/IP access method. So if you use TCP/IP instead of XMS, you can use both client-assigned (client authority) and server-preassigned (server authority) librefs. You can also use either access method if your connection is not based on a libref (client authority).

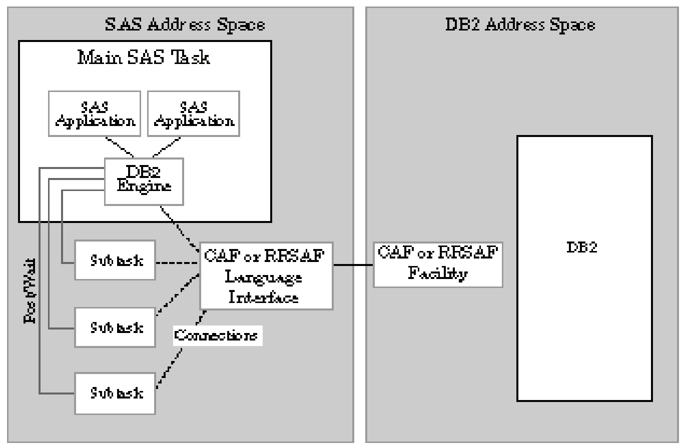
DB2 under z/OS Information for the Database Administrator

How the Interface to DB2 Works

SAS/ACCESS Interface to DB2 under z/OS uses either the Call Attachment Facility (CAF) or the Recoverable Resource Management Services Attachment Facility (RRSAF) to communicate with the local DB2 subsystem. Both attachment facilities enable programs to connect to DB2 and to use DB2 for SQL statements and commands. SAS/ACCESS Interface to DB2 under z/OS uses the attachment facilities to establish and control its connections to the local DB2 subsystem. DB2 allows only one connection for each task control block (TCB), or task. SAS and SAS executables run under one TCB, or task.

The DB2 LIBNAME statement lets SAS users connect to DB2 more than once. Because the CAF and RRSAF allow only one connection per TCB, SAS/ACCESS Interface to DB2 under z/OS attaches a subtask for each subsequent connection that is initiated. It uses the ATTACH, DETACH, POST, and WAIT assembler macros to create and communicate with the subtasks. It does not limit the number of connections or subtasks that a single SAS user can initiate. This image illustrates how the DB2 engine works.

Display 16.1 *Design of the DB2 Engine*



How and When Connections Are Made

SAS/ACCESS Interface to DB2 under z/OS always makes an explicit connection to the local DB2 subsystem (SSID). When a connection executes successfully, a thread to DB2 is established. For each thread's or task's connection, DB2 establishes authorization identifiers (AUTHIDs).

The DB2 interface determines when to make a connection to DB2 based on the type of open mode (Read, Update, or output) that a SAS application requests for the DB2 tables. Here is the default behavior.

- SAS/ACCESS Interface to DB2 under z/OS shares the connection for all openings in Read mode for each DB2 LIBNAME statement
- SAS/ACCESS Interface to DB2 under z/OS acquires a separate connection to DB2 for every opening in update or output mode.

You can change this default behavior by using the CONNECTION= on page 113 option.

Several SAS applications require SAS/ACCESS Interface to DB2 under z/OS to query the DB2 system catalogs. When this type of query is required, the DB2 interface acquires a separate connection to DB2 to avoid contention with other applications that are accessing the DB2 system catalogs. See "Accessing DB2 System Catalogs" on page 514 for more information.

The DEFER= LIBNAME option on page 141 also controls when a connection is established. UTILCONN_TRANSIENT= on page 202 also allows control of the utility connection—namely, whether it must stay open.

DDF Communication Database

DB2 Distributed Data Facility (DDF) Communication Database (CDB) enables DB2 z/OS applications to access data on other systems. Database administrators are responsible for customizing CDB. SAS/ACCESS Interface to DB2 under z/OS supports both types of DDF: system-directed access (private protocol) and Distributed Relational Database Architecture.

System-directed access enables one DB2 z/OS subsystem to execute SQL statements on another DB2 z/OS subsystem. System-directed access uses a private protocol only for DB2. It is known as a private protocol because you can use only it between DB2 databases. IBM recommends that users use DRDA. Although SAS/ACCESS Interface to DB2 under z/OS cannot explicitly request a connection, it can instead perform an implicit connection when SAS initiates a distributed request. To initiate an implicit connection, you must specify the LOCATION= on page 155 option. When you specify this option, the three-level table name (*location.authid.table*) is used in the SQL statement that SAS/ACCESS Interface to DB2 under z/OS generates. When the SQL statement that contains the three-level table name is executed, an implicit connection is made to the remote DB2 subsystem. The primary authorization ID of the initiating process must be authorized to connect to the remote location.

Distributed Relational Database Architecture (DRDA) is a set of protocols that lets a user access distributed data. This lets SAS/ACCESS Interface to DB2 under z/OS access multiple remote tables at various locations. The tables can be distributed among multiple platforms, and both like and unlike platforms can communicate with one another. In a DRDA environment, DB2 acts as the client, server, or both.

To connect to a DRDA remote server or location, SAS/ACCESS Interface to DB2 under z/OS uses an explicit connection. To establish an explicit connection, SAS/ACCESS Interface to DB2 under z/OS first connects to the local DB2 subsystem through an attachment facility on page 513 (CAF or RRSAF). It then issues an SQL CONNECT statement to connect from the local DB2 subsystem to the remote DRDA server before it accesses data. To initiate a connection to a DRDA remote server, you must specify the SERVER= connection option on page 467. When you specify this option, SAS uses a separate connection for each remote DRDA location.

DB2 Attachment Facilities (CAF and RRSAF)

By default, SAS/ACCESS Interface to DB2 under z/OS uses the Call Attachment Facility (CAF) to make its connections to DB2. SAS supports multiple CAF connections for a SAS session. Therefore, for a SAS server, all clients can have their own connections to DB2; multiple clients no longer have to share one connection. However, because CAF does not support sign-on, each connection that the SAS server makes to DB2 has the z/OS authorization identifier of the server. It does not have the authorization identifier of the client for which the connection is made.

If you specify the DB2ERRS on page 493 system option, SAS/ACCESS Interface to DB2 under z/OS engine uses the Recoverable Resource Manager Services Attachment Facility (RRSAF). Only one attachment facility can be used at a time, so the DB2RRS or NODB2RRS system option can be specified only when a SAS session is started. SAS supports multiple RRSAF connections for a SAS session. RRSAF is a new feature in DB2 Version 5, Release 1, and its support in SAS/ACCESS Interface to DB2 under z/OS was new in SAS 8.

The RRSAF is intended for use by SAS servers, such as the ones that SAS/SHARE software use. RRSAF supports the ability to associate a z/OS authorization identifier with each connection at sign on. This authorization identifier is not the same as the authorization ID that you specify in the AUTHID= on page 103 data set or LIBNAME option. DB2 uses the RRSAF-supported authorization identifier to validate a given connection's authorization to use both DB2 and system resources when those connections are made using the System Authorization Facility and other security products such as RACF. This authorization identifier is basically the user ID with which you are logged on to z/OS.

With RRSAF, the SAS server makes the connections for each client and the connections have the client z/OS authorization identifier associated with them. This is true only for clients that the SAS server authenticated, which occurred when the client specified a user ID and password. Servers authenticate their clients when the clients provide their user IDs and passwords. Generally, this is the default way that servers are run. If a client connects to a SAS server without providing a user ID and password, the identifier associated with its connections is that of the server (as with CAF) and not the client's identifier.

Other than specifying DB2RRS at SAS start-up, you do not need to do anything else to use RSSAF. the DB2 interface automatically signs on each connection that it makes to DB2 with the identifier of either the authenticated client or of the SAS server for non-authenticated clients. Authenticated clients have the same authorities to DB2 as they have when they run their own SAS session from their own ID and access DB2.

Accessing DB2 System Catalogs

For many types of SAS procedures, the DB2 interface must access DB2 system catalogs for information. This information is limited to a list of all tables for a specific authorization identifier. The interface generates this SQL query to obtain information from system catalogs:

```
SELECT NAME FROM SYSIBM.SYSTABLES
  WHERE (CREATOR = 'authid');
```

Unless you specify the AUTHID= on page 103 option, the authorization ID is the z/OS user ID that is associated with the job step.

The SAS procedures or applications that request the list of DB2 tables includes, but is not limited to, PROC DATASETS and PROC CONTENTS, or any application that needs a member list. If the SAS user does not have the necessary authorization to read the DB2 system catalogs, the procedure or application fails.

Because querying the DB2 system catalogs can cause some locking contentions, SAS/ACCESS Interface to DB2 under z/OS initiates a separate connection for the query to the DB2 system catalogs. After the query completes, a COMMIT WORK command is executed.

Under certain circumstances, you can access a catalog file by overriding the default value for the "DB2CATALOG= System Option" on page 378.

Chapter 17

SAS/ACCESS Interface to Greenplum

Introduction to SAS/ACCESS Interface to Greenplum

This section describes SAS/ACCESS Interface to Greenplum. For a list of SAS/ACCESS features that are available for this interface, see "SAS/ACCESS Interface to Greenplum: Supported Features" on page 80.

LIBNAME Statement Specifics for Greenplum

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to Greenplum supports and includes examples. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing Greenplum.

LIBNAME *libref* **greenplm** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`greenplm`
specifies the SAS/ACCESS engine name for the Greenplum interface.

connection-options
provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. When you use the LIBNAME statement, you can connect to the Greenplum database in two ways. Specify *only one* of these methods for each connection because they are mutually exclusive.

- SERVER=, DATABASE=, PORT=, USER=, PASSWORD=
- DSN=, USER=, PASSWORD=

Here is how these options are defined.

SERVER=<'>*server-name*<'>
specifies the Greenplum server name or the IP address of the server host. If the server name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

DATABASE=<'>*database-name*<'>
specifies the Greenplum database that contains the tables and views that you want to access. If the database name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

Alias: DB=

PORT=*port*
: specifies the port number that is used to connect to the specified Greenplum database. If you do not specify a port, the default is 5432.

USER=<'>*Greenplum user-name*<'>
: specifies the Greenplum user name (also called the user ID) that is used to connect to the database. If the user name contains spaces or nonalphanumeric characters, use quotation marks.

PASSWORD=<'>*Greenplum password*<'>
: specifies the password that is associated with your Greenplum user ID. If the password contains spaces or nonalphabetic characters, you must enclose it in quotation marks. You can also specify PASSWORD= with the PWD=, PASS=, and PW= aliases.

DSN=<'>*Greenplum data-source*<'>
: specifies the configured Greenplum ODBC data source to which you want to connect. It is recommended that you use this option only if you have existing Greenplum ODBC data sources configured on your client. This method requires additional setup—either through the ODBC Administrator control panel on Windows platforms, or through the odbc.ini file or a similarly named configuration file on UNIX platforms. It is recommended that you use this connection method only if you have existing, functioning data sources that have been defined.

LIBNAME -options
: define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to Greenplum with the applicable default values. For more detail about these options, see "LIBNAME Options for Relational Databases" on page 99.

Table 17.1 *SAS/ACCESS LIBNAME Options for Greenplum*

Option	Default Value
ACCESS=	none
AUTHDOMAIN=	none
AUTOCOMMIT= on page 103	operation-specific
CONNECTION=	SHAREDREAD
CONNECTION_GROUP=	none
DBCOMMIT=	1000 when inserting rows; 0 when updating rows
DBCONINIT=	none
DBCONTERM=	none
DBCREATE_TABLE_OPTS=	none
DBGEN_NAME=	DBMS

Option	Default Value
DBINDEX=	YES
DBLIBINIT=	none
DBLIBTERM=	none
DBMAX_TEXT=	1024
DBMSTEMP=	none
DBNULLKEYS=	none
DBPROMPT=	none
DBSASLABEL=	COMPAT
DEFER= on page 141	none
DELETE_MULT_ROWS=	NO
DIRECT_EXE=	none
DIRECT_SQL=	YES
IGNORE_READ_ONLY_COLUMNS=	none
INSERTBUFF=	automatically calculated based on row length
MULTI_DATASRC_OPT=	none
PRESERVE_COL_NAMES=	see “Naming Conventions for Greenplum” on page 530
PRESERVE_TAB_NAMES=	see “Naming Conventions for Greenplum” on page 530
QUERY_TIMEOUT=	0
QUOTE_CHAR=	none
READBUFF=	automatically calculated based on row length
REREAD_EXPOSURE=	none
SCHEMA=	none
SPOOL=	none
SQL_FUNCTIONS=	none

Option	Default Value
SQL_FUNCTIONS_COPY=	none
SQLGENERATION=	DBMS
STRINGDATES=	none
TRACE=	none
TRACEFILE=	none
UPDATE_MULT_ROWS=	NO
UTILCONN_TRANSIENT=	none

LIBNAME Statement Examples

In this example, SERVER=, DATABASE=, PORT=, USER=, and PASSWORD= are the connection options.

```
libname mydblib greenplm server=gplum04 db=customers port=5432
    user=gpusr1 password=gppwd1;

proc print data=mydblib.customers;
   where state='CA';
run;
```

In the next example, DSN=, USER=, and PASSWORD= are the connection options. The Greenplum data source is configured in the ODBC Administrator Control Panel on Windows platforms. It is also configured in the odbc.ini file or a similarly named configuration file on UNIX platforms.

```
libname mydblib greenplm DSN=gplumSalesDiv user=gpusr1 password=gppwd1;

proc print data=mydblib.customers;
   where state='CA';
```

Data Set Options for Greenplum

All SAS/ACCESS data set options in this table are supported for Greenplum. Default values are provided where applicable. For details about this feature, see Data Set Options for Relational Databases on page 207.

Table 17.2 *SAS/ACCESS Data Set Options for Greenplum*

Option	Default Value
BL_DATAFILE=	none

Option	Default Value
BL_DELTE_DATAFILE=	none
BL_DELIMITER=	|
BL_ENCODING=	DEFAULT
BL_ESCAPE=	\
BL_EXCEPTION=	none
BL_EXECUTE_CMD=	none
BL_EXECUTE_LOCATION=	none
BL_EXTERNAL_WEB=	
BL_FORCE_NOT_NULL=	none
BL_FORMAT= on page 246	TEXT
BL_HEADER=	NO
BL_HOST=	127.0.0.1
BL_NULL=	'\N' [TEXT mode], unquoted empty value [CSV mode]
BL_PORT=	8080
BL_PROTOCOL=	'gpfdist'
BL_QUOTE=	" (double quotation mark)
BL_REJECT_LIMIT=	none
BL_REJECT_TYPE=	ROWS
BULKLOAD=	none
DBCOMMIT=	LIBNAME option setting
DBCONDITION=	none
DBCREATE_TABLE_OPTS=	LIBNAME option setting
DBFORCE=	none
DBGEN_NAME=	DBMS
DBINDEX=	LIBNAME option setting
DBKEY=	none

Option	Default Value
DBLABEL=	none
DBMASTER=	none
DBMAX_TEXT=	1024
DBNULL=	none
DBNULLKEYS=	LIBNAME option setting
DBPROMPT=	LIBNAME option setting
DBSASTYPE=	see "Data Types for Greenplum " on page 530
DBTYPE=	see "Data Types for Greenplum " on page 530
DISTRIBUTED_BY=	DISTRIBUTED_RANDOMLY
ERRLIMIT=	1
IGNORE_READ_ONLY_COLUMNS=	none
INSERTBUFF=	LIBNAME option setting
NULLCHAR=	SAS
NULLCHARVAL=	a blank character
PRESERVE_COL_NAMES=	LIBNAME option setting
QUERY_TIMEOUT=	LIBNAME option setting
READBUFF=	LIBNAME option setting
SASDATEFMT=	none
SCHEMA=	LIBNAME option setting

SQL Pass-Through Facility Specifics for Greenplum

Key Information

For general information about this feature, see "About SQL Procedure Interactions " on page 401. Greenplum examples on page 522 are available.

Here are the SQL pass-through facility specifics for the Greenplum interface.

- The *dbms-name* is **GREENPLM**.
- The CONNECT statement is required.
- PROC SQL supports multiple connections to Greenplum. If you use multiple simultaneous connections, you must use the *alias* argument to identify the different connections. If you do not specify an alias, the default **GREENPLM** alias is used.
- The CONNECT statement *database-connection-arguments* are identical to its LIBNAME connection options.

CONNECT Statement Example

This example uses the DBCON alias to connect to the **greenplum04** Greenplum server database and execute a query. The connection alias is optional.

```
proc sql;
   connect to greenplm as dbcon
   (server=greenplum04 db=sample port=5432 user=gpusr1 password=gppwd1);
select * from connection to dbcon
   (select * from customers where customer like '1%');
quit;
```

Special Catalog Queries

SAS/ACCESS Interface to Greenplum supports the following special queries. You can use the queries to call functions in ODBC-style function application programming interfaces (APIs). Here is the general format of the special queries:

Greenplum::SQLAPI *'parameter–1'*, *'parameter-n'*

Greenplum::
: is required to distinguish special queries from regular queries. Greenplum:: is not case sensitive.

SQLAPI
: is the specific API that is being called. SQLAPI is not case sensitive.

'parameter n'
: is a quoted string that is delimited by commas.

Within the quoted string, two characters are universally recognized: the percent sign (%) and the underscore (_). The percent sign matches any sequence of zero or more characters, and the underscore represents any single character. To use either character as a literal value, you can use the backslash character (\) to escape the match characters. For example, this call to SQLTables usually matches table names such as myatest and my_test:

```
select * from connection to greenplm
  (Greenplum::SQLTables "test","","my_test");
```

Use the escape character to search only for the my_test table:

```
select * from connection to greenplm
  (Greenplum::SQLTables "test","","my\_test");
```

SAS/ACCESS Interface to Greenplum supports these special queries.

Greenplum::SQLTables <*'Catalog'*, *'Schema'*, *'Table-name'*, *'Type'*>
: returns a list of all tables that match the specified arguments. If you do not specify any arguments, all accessible table names and information are returned.

Greenplum::SQLColumns <*'Catalog'*, *'Schema'*, *'Table-name'*, *'Column-name'*>
: returns a list of all tables that match the specified arguments. If you do not specify any arguments, all accessible table names and information are returned.

Greenplum::SQLColumns <*'Catalog'*, *'Schema'*, *'Table-name'*, *'Column-name'*>
: returns a list of all columns that match the specified arguments. If you do not specify any argument, all accessible column names and information are returned.

Greenplum::SQLPrimaryKeys <*'Catalog'*, *'Schema'*, *'Table-name''Type'* >
: returns a list of all columns that compose the primary key that matches the specified table. A primary key can be composed of one or more columns. If you do not specify any table name, this special query fails.

Greenplum::SQLStatistics <*'Catalog'*, *'Schema'*, *'Table-name'*>
: returns a list of the statistics for the specified table name, with options of SQL_INDEX_ALL and SQL_ENSURE set in the SQLStatistics API call. If you do not specify any table name argument, this special query fails.

Greenplum::SQLGetTypeInfo
: returns information about the data types that the Greenplum *n*Cluster database supports.

Autopartitioning Scheme for Greenplum

Overview

Autopartitioning for SAS/ACCESS Interface to Greenplum is a modulo (MOD) function method. For general information about this feature, see “Autopartitioning Techniques in SAS/ACCESS” on page 63.

Autopartitioning Restrictions

SAS/ACCESS Interface to Greenplum places additional restrictions on the columns that you can use for the partitioning column during the autopartitioning phase. Here is how columns are partitioned.

- INTEGER and SMALLINT columns are given preference.
- You can use other numeric columns for partitioning if the precision minus the scale of the column is greater than 0 but less than 10—namely, 0<(*precision-scale*)<10.

Nullable Columns

If you select a nullable column for autopartitioning, the `OR<column-name>IS NULL` SQL statement is appended at the end of the SQL code that is generated for the threaded Read. This ensures that any possible NULL values are returned in the result set.

Using WHERE Clauses

Autopartitioning does not select a column to be the partitioning column if it appears in a SAS WHERE clause. For example, this DATA step cannot use a threaded Read to retrieve the data because all numeric columns in the table are in the WHERE clause:

```
data work.locemp;
set trlib.MYEMPS;
where EMPNUM<=30 and ISTENURE=0 and
 SALARY<=35000 and NUMCLASS>2;
run;
```

Using DBSLICEPARM=

Although SAS/ACCESS Interface to Greenplum defaults to three threads when you use autopartitioning, do not specify a maximum number of threads for the threaded Read in DBSLICEPARM= LIBNAME option on page 139.

Using DBSLICE=

You might achieve the best possible performance when using threaded Reads by specifying the DBSLICE= data set option on page 299 for Greenplum in your SAS operation. This is especially true if your Greenplum data is evenly distributed across multiple partitions in a Greenplum database system.

When you create a Greenplum table using the Greenplum database partition model, you can specify the partitioning key that you want to use by appending the **PARTITION BY<*column-name*>** clause to your CREATE TABLE statement. Here is how you can accomplish this by using the DBCREATE_TABLE_OPTS=LIBNAME option within the SAS environment.

```
/* Points to a triple-node server. */
libname mylib sasiogpl user=myuser pw=mypwd db=greenplum;
DBCREATE_TABLE_OPTS='PARTITION BY(EMPNUM);

proc datasets library=mylib;
  delete MYEMPS1;run;

data mylib.myemps(drop=morf whatstate
   DBTYPE=(HIREDATE="date" SALARY="numeric(8,2)"
   NUMCLASS="smallint" GENDER="char(1)" ISTENURE="numeric(1)" STATE="char(2)"
   EMPNUM="int NOT NULL Primary Key"));
format HIREDATE mmddyy10.;
do EMPNUM=1 to 100;
     morf=mod(EMPNUM,2)+1;
     if(morf eq 1) then
         GENDER='F';
     else
         GENDER='M';
     SALARY=(ranuni(0)*5000);
     HIREDATE=int(ranuni(13131)*3650);
     whatstate=int(EMPNUM/5);
     if(whatstate eq 1) then
         STATE='FL';
```

```
     if(whatstate eq 2) then
         STATE='GA';
     if(whatstate eq 3) then
         STATE='SC';
     if(whatstate eq 4) then
         STATE='VA';
     else
         state='NC';
     ISTENURE=mod(EMPNUM,2);
     NUMCLASS=int(EMPNUM/5)+2;
     output;
end;
run;
```

After the MYEMPS table is created on this three-node database, a third of the rows reside on each of the three nodes.

Using DBSLICE= works well when the table that you want to read is not stored in multiple partitions. It gives you flexibility in column selection. For example, if you know that the STATE column in your employee table contains only a few distinct values, you can modify your DBSLICE= option accordingly.

```
data work.locemp;
set mylib.MYEMPS (DBSLICE=("STATE='GA'"
    "STATE='SC'" "STATE='VA'" "STATE='NC'"));
where EMPNUM<=30 and ISTENURE=0 and SALARY<=35000 and NUMCLASS>2;
run;
```

Passing SAS Functions to Greenplum

SAS/ACCESS Interface to Greenplum passes the following SAS functions to Greenplum for processing. Where the Greenplum function name differs from the SAS function name, the Greenplum name appears in parentheses. For more information, see “Passing Functions to the DBMS Using PROC SQL” on page 46.

ABS	MIN
ARCOS (ACOS)	MINUTE (DATEPART)
ARSIN (ASIN)	MOD
ATAN	MONTH (DATEPART)
ATAN2	QTR (DATEPART)
AVG	REPEAT
BYTE (CHR)	SECOND (DATEPART)
CEIL	SIGN
COS	SIN
COUNT	SQRT
DAY (DATEPART)	STRIP (BTRIM)
EXP	SUBSTR (SUBSTRING)
FLOOR	SUM
HOUR (DATEPART)	TAN
INDEX (STRPOS)	TRANWRD (REPLACE)
LENGTH	TRIMN (RTRIM)

LOG (LN)	UPCASE (UPPER)
LOG10 (LOG)	WEEKDAY (DATEPART)
LOWCASE (LOWER)	YEAR (DATEPART)
MAX	

SQL_FUNCTIONS=ALL enables for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to Greenplum. Due to incompatibility in date and time functions between Greenplum and SAS, Greenplum might not process them correctly. Check your results to determine whether these functions are working as expected. See"SQL_FUNCTIONS= LIBNAME Option" on page 185.

COMPRESS (REPLACE)	SOUNDEX
DATE (NOW)	TIME (current_time)
DATEPART (CONVERT)	TIMEPART (TIME)
DATETIME (NOW)	TODAY (NOW)

Passing Joins to Greenplum

For a multiple libref join to pass to Greenplum, all of these components of the LIBNAME statements must match exactly.

- user ID (USER=)
- password (PASSWORD=)
- host(HOST=)
- server (SERVER=)
- database (DATABASE=)
- port (PORT=)
- data source (DSN=, if specified)
- SQL functions (SQL_FUNCTIONS=)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Bulk Loading for Greenplum

Overview

Bulk loading provides high-performance access to external data sources. Multiple Greenplum instances read data in parallel, which enhances performance.

Bulk loading enables you to insert large data sets into Greenplum tables in the shortest span of time. You can also use bulk loading to execute high-performance SQL queries against external data sources, without first loading those data sources into a Greenplum database. These fast SQL queries enable you to optimize the extraction, transformation, and loading tasks that are common in data warehousing.

Two types of external data sources, external tables and Web tables, have different access methods. External tables contain static data that can be scanned multiple times. The data does not change during queries. Web tables provide access to dynamic data sources as if those sources were regular database tables. Web tables cannot be scanned multiple times. The data can change during the course of a query.

The following sections show you how to access external tables and Web tables using the bulk-loading facility.

Using Protocols to Access External Tables

Use these protocols to access (static) external tables.

gpfdist://
: To use the gpfdist:// protocol, install and configure the gpfdist (Greenplum file distribution) program on the host that stores the external tables see "Configuring the File Server" on page 527 . The gpfdist utility serves external tables in parallel to the primary Greenplum database segments. The gpfdist:// protocol is advantageous because it ensures that all Greenplum database segments are used during the loading of external tables.

  To specify files to gpfdist, use the BL_DATAFILE= data set option. Specify file paths that are relative to the directory from which gpfdist is serving files (the directory where you executed gpfdist).

  The gpfdist utility is part of the loader package for the platform where SAS is running. You can also download it from the Greenplum Web site: www.greenplum.com.

file://
: To use the file:// protocol, external tables must reside on a segment host in a location that Greenplum superusers (gpadmin) can access. The segment host name must match the host name, as specified in the gp_configuration system catalog table. In other words, the external tables that you want to load must reside on a host that is part of the set of servers that comprise the database configuration. The file:// protocol is advantageous because it does not require configuration.

Configuring the File Server

Follow these steps to configure the gpfdist file server.

1. Download and install gpfdist from www.greenplum.com.
2. Define and load a new environment variable called GPLOAD_HOME.
3. Set the value of the variable to the directory that contains the external tables that you want to load.

   The directory path must be relative to the directory in which you execute gpfdist, and it must exist before gpfdist tries to access it.

   - For Windows, open **My Computer**, select the **Advanced** tab, and click the **Environment Variables** button.
   - For UNIX, enter this command or add it to your profile:

     ```
     export GPLOAD_HOME=directory
     ```
4. Start gpfdist as shown in these examples.
   - For Windows:

```
C:> gpfdist -d %GPLOAD_HOME% -p 8081 -l %GPLOAD_HOME%\gpfdist.log
```

- For UNIX:

```
$ gpfdist -d $GPLOAD_HOME -p 8081 -l $GPLOAD_HOME/gpfdist.log &
```

You can run multiple instances of gpfdist on the same host as long each instance has a unique port and directory.

If you do not set GPLOAD_HOME, the value of the BL_DATAFILE= data set option specifies the directory that contains the external tables to be loaded. If BL_DATAFILE is not specified, the current directory is assumed to contain the external tables.

Stopping gpfdist

In Windows, to stop an instance of gpfdist, use the Task Manager or close the Command Window that you used to start that instance of gpfdist.

Follow these steps In UNIX to stop an instance of gpfdist.

1. Find the process ID:

```
$ ps ax | grep gpfdist (Linux)
$ ps -ef | grep gpfdist (Solaris)
```

2. Kill the process. Here is an example:

```
$ kill 3456
```

Troubleshooting gpfdist

Run this command to test connectivity between an instance of gpfdist and a Greenplum database segment.

```
$ wget http://gpfdist_hostname:port/filename
```

Using the file:// Protocol

You can use the file:// protocol to identify external files for bulk loading with no additional configuration required. However, using the GPLOAD_HOME environment variable is highly recommended. If you do not specify GPLOAD_HOME, the BL_DATAFILE data set option specifies the source directory. The default source directory is the current directory if you do not set BL_DATAFILE=. The Greenplum server must have access to the source directory.

Accessing Dynamic Data in Web Tables

Use these data set options to access Web tables:

- BL_EXECUTE_CMD=
- BL_LOCATION=

Data Set Options for Bulk Loading

Here are the Greenplum bulk-load data set options. For detailed information about these options, see Data Set Options for Relational Databases on page 207.

- BL_DATAFILE=
- BL_CLIENT_DATAFILE=
- BL_DELETE_DATAFILE=
- BL_DELIMITER=
- BL_ENCODING=
- BL_ESCAPE=
- BL_EXCEPTION=
- BL_EXECUTE_CMD=
- BL_EXECUTE_LOCATION=
- BL_EXTERNAL_WEB=
- BL_FORCE_NOT_NULL=
- BL_FORMAT=
- BL_HEADER=
- "BL_HOST= Data Set Option"BL_HOST=
- BL_NULL=
- BL_PORT=
- BL_PROTOCOL=
- BL_QUOTE=
- BL_REJECT_LIMIT=
- BL_REJECT_TYPE=
- BL_USE_PIPE=
- BULKLOAD=

Examples

This first example shows how you can use a SAS data set, SASFLT.FLT98, to create and load a Greenplum table, FLIGHTS98.

```
libname sasflt 'SAS-data-library';
libname mydblib greenplm host=iqsvr1 server=iqsrv1_users
    db=users user=iqusr1 password=iqpwd1;

proc sql;
create table net_air.flights98
     (BULKLOAD=YES
      BL_DATAFILE='c:\temp\greenplum\data.dat'
      BL_USE_PIPE=NO
      BL_DELETE_DATAFILE=yes
      BL_HOST='192.168.x.x'
      BL_PORT=8081)
    as select * from sasflt.flt98;
quit;
```

This next example shows how you can append the SAS data set, SASFLT.FLT98, to the existing Greenplum table ALLFLIGHTS. The BL_USE_PIPE=NO option forces

SAS/ACCESS Interface to Greenplum to write data to a flat file, as specified in the BL_DATAFILE= option. Rather than deleting the data file, BL_DELETE_DATAFILE=NO causes the engine to leave it after the load has completed.

```
proc append base=new_air.flights98
    (BULKLOAD=YES
     BL_DATAFILE='c:\temp\greenplum\data.dat'
     BL_USE_PIPE=NO
     BL_DELETE_DATAFILE=yes
     BL_HOST='192.168.x.x'
     BL_PORT=8081)
    data=sasflt.flt98;
run;
```

Naming Conventions for Greenplum

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

Since SAS 7, most SAS names can be up to 32 characters long. SAS/ACCESS Interface to Greenplum supports table names and column names that contain up to 32 characters. If DBMS column names are longer than 32 characters, they are truncated to 32 characters. If truncating a column name results in identical names, SAS generates a unique name by replacing the last character with a number. DBMS table names must be 32 characters or less because SAS does not truncate a longer name. If you already have a table name that is greater than 32 characters, it is recommended that you create a table view.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options determine how SAS/ACCESS Interface to Greenplum handles case sensitivity. (For information about these options, see "Overview: LIBNAME Statement for Relational Databases " on page 93.) Greenplum is not case sensitive, so all names default to lowercase.

Greenplum objects include tables, views, and columns. They follow these naming conventions.

- A name can contain as many as 128 characters.
- The first character in a name can be a letter or @, _, or #.
- A name cannot be a Greenplum reserved word, such as WHERE or VIEW.
- A name must be unique within each type of each object.

For more information, see the *Greenplum Database Administrator Guide*.

Data Types for Greenplum

Overview

Every column in a table has a name and a data type. The data type tells Greenplum how much physical storage to set aside for the column and the form in which the data is stored. This section includes information about Greenplum data types, null and default values, and data conversions.

For more information about Greenplum data types and to determine which data types are available for your version of Greenplum, see the *Greenplum Database Administrator Guide*.

SAS/ACCESS Interface to Greenplum does not directly support any data types that are not listed below. Any columns using these types are read into SAS as character strings.

String Data

CHAR(n)
: specifies a fixed-length column for character string data. The maximum length is 32,767 characters. If the length is greater than 254, the column is a long-string column. SQL imposes some restrictions on referencing long-string columns. For more information about these restrictions, see the *Greenplum Database Administrator Guide*.

VARCHAR(n)
: specifies a varying-length column for character string data. The maximum length is 32,767 characters. If the length is greater than 254, the column is a long-string column. SQL imposes some restrictions on referencing long-string columns. For more information about these restrictions, see the *Greenplum Database Administrator Guide*.

LONG VARCHAR(n)
: specifies a varying-length column for character string data. The maximum size is limited by the maximum size of the database file. To determine the maximum size of your database, see the *Greenplum Database Administrator Guide*.

Numeric Data

BIGINT
: specifies a big integer. Values in a column of this type can range from –9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 .

SMALLINT
: specifies a small integer. Values in a column of this type can range from –32768 to +32767.

INTEGER
: specifies a large integer. Values in a column of this type can range from –2147483648 to +2147483647.

BIT
: specifies a Boolean type. Values in a column of this type can be either 0 or 1. Inserting any nonzero value into a BIT column stores a 1 in the column.

DOUBLE | DOUBLE PRECISION
: specifies a floating-point number that is 64 bits long. Values in a column of this type can range from -1.79769E+308 to –2.225E-307 or +2.225E-307 to +1.79769E+308, or they can be 0. This data type is stored the same way that SAS stores its numeric data type. Therefore, numeric columns of this type require the least processing when SAS accesses them.

REAL
: specifies a floating-point number that is 32 bits long. Values in a column of this type can range from approximately -3.4E38 to –1.17E-38 and +1.17E-38 to +3.4E38.

FLOAT
specifies a floating-point number. If you do not supply the precision, the FLOAT data type is the same as the REAL data type. If you supply the precision, the FLOAT data type is the same as the REAL or DOUBLE data type, depending on the value of the precision. The cutoff between REAL and DOUBLE is platform-dependent. It is the number of bits that are used in the mantissa of the single-precision floating-point number on the platform.

DECIMAL | DEC | NUMERIC
specifies a fixed-point decimal number. The precision and scale of the number determines the position of the decimal point. The numbers to the right of the decimal point are the scale, and the scale cannot be negative or greater than the precision. The maximum precision is 126 digits.

Date, Time, and Timestamp Data

SQL date and time data types are collectively called datetime values. The SQL data types for dates, times, and timestamps are listed here. Be aware that columns of these data types can contain data values that are out of range for SAS.

CAUTION:
The following data types can contain data values that are out of range for SAS.

DATE
specifies date values. The range is 01-01-0001 to 12-31-9999. The default format is *YYYY-MM-DD*. An example is 1961-06-13.

TIME
specifies time values in hours, minutes, and seconds to six decimal positions: hh:mm:ss[.nnnnnn]. The range is 00:00:00.000000 to 23:59:59.999999. Due to the ODBC-style interface that SAS/ACCESS Interface to Greenplum uses to communicate with the server, fractional seconds are lost in the data transfer from server to client.

TIMESTAMP
combines a date and time in the default format of *yyyy-mm-dd hh:mm:ss[.nnnnnn]*. For example, a timestamp for precisely 2:25 p.m. on January 25, 1991, would be 1991-01-25-14.25.00.000000. Values in a column of this type have the same ranges and limitations as described for DATE and TIME.

Greenplum Null Values

Greenplum has a special value called NULL. A Greenplum NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a Greenplum NULL value, it interprets it as a SAS missing value. When loading SAS tables from Greenplum sources, SAS/ACCESS stores Greenplum NULL values as SAS missing values.

In Greenplum tables, NULL values are valid in all columns by default. There are two methods to define a column in a Greenplum table so that it requires data:

- Using SQL, you specify a column as NOT NULL. This tells SQL to allow only a row to be added to a table if a value exists for the field. Rows that contain NULL values in that column are not added to the table.
- Another approach is to assert NOT NULL DEFAULT. For more information, see the *Greenplum Database Administrator Guide*.

When creating Greenplum tables with SAS/ACCESS, you can use the DBNULL= data set option to specify the treatment of NULL values. For more information about how SAS handles NULL values, see "Potential Result Set Differences When Processing Null Data " on page 35.

Once you know whether a Greenplum column enables NULLs or the host system supplies a default value for a column that is defined as NOT NULL WITH DEFAULT, you can write selection criteria and enter values to update a table. Unless a column is defined as NOT NULL or NOT NULL DEFAULT, it enables NULL values.

To control how the DBMS handles SAS missing character values, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

The following table shows the default formats that SAS/ACCESS Interface to Greenplum assigns to SAS variables when using the LIBNAME statement on page 93.

These default formats are based on Greenplum column attributes.

Table 17.3 *LIBNAME Statement: Default SAS Formats for Greenplum Data Types*

Greenplum Data Type	SAS Data Type	Default SAS Format
CHAR(*n*)*	character	$*n.*
VARCHAR(*n*)*	character	$*n.*
INTEGER	numeric	11.
SMALLINT	numeric	6.
TINYINT	numeric	4.
BIT	numeric	1.
BIGINT	numeric	20.
DECIMAL(*p,s*)	numeric	*m.n*
NUMERIC(*p,s*)	numeric	*m.n*
REAL	numeric	none
DOUBLE	numeric	none
TIME	numeric	TIME8.
DATE	numeric	DATE9.
TIMESTAMP	numeric	DATETIME*25.6*

* *n* in Greenplum data types is equivalent to *w* in SAS formats.

The next table shows the default Greenplum data types that SAS/ACCESS assigns to SAS variable formats during output operations when you use the LIBNAME statement.

Table 17.4 *LIBNAME Statement: Default Greenplum Data Types for SAS Variable Formats*

SAS Variable Format	Greenplum Data Type
m.n	DECIMAL(*p*,*s*)
other numerics	DOUBLE
$*n*.	VARCHAR(*n*)*
datetime formats	TIMESTAMP
date formats	DATE
time formats	TIME

* *n* in Greenplum data types is equivalent to *w* in SAS formats.

Chapter 18
SAS/ACCESS Interface to Hadoop

Introduction to SAS/ACCESS Interface to Hadoop

This section describes SAS/ACCESS Interface to Hadoop. For a list of SAS/ACCESS features that are available in this interface, see "SAS/ACCESS Interface to Hadoop: Supported Features" on page 80.

SAS interfaces to Hadoop include SAS/ACCESS Interface to Hadoop (this document), the Hadoop procedure (see *Base SAS Procedures Guide*), and the FILENAME statement for the Hadoop access method (see *SAS Statements: Reference*).

LIBNAME Statement Specifics for Hadoop

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to Hadoop supports and includes examples. For general information about this feature, see LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing Hadoop.

LIBNAME *libref* **hadoop** *<connection-options> <LIBNAME-options>*

Security Limitations

SAS/ACCESS uses the Hadoop Hive Server to access Hadoop data. SAS/ACCESS creates a JDBC connection and passes the user ID and password that you specify to Hive. Through Hive 8, JDBC ignores these credentials, instead associating Hive permissions with the UNIX user ID that started the Hive Service.

SAS/ACCESS creates new Hive tables using the Hadoop Distributed File System (HDFS) Streaming API. The API requires a valid user ID and password. When you create or append to a table, the user ID and password that you provide in the LIBNAME statement is passed to this streaming method. The user ID must be valid on the Hadoop cluster and needs Write access to the Hadoop /tmp and the Hive warehouse directories. Therefore, the Hadoop system administrator should ensure Write access to the HDFS /tmp and Hive warehouse directories for all user IDs that are permitted to create or append to Hive tables with SAS/ACCESS.

To optimize large data reads, SAS/ACCESS can use the API instead of JDBC to read Hive result sets. Similar requirements apply: a valid Hadoop user ID and password, and Write access to Hadoop /tmp.

Future Hive editions will address JDBC security limitations. While USER= and PASSWORD= are not yet fully enabled, use both USER= and PASSWORD= on LIBNAME and CONNECT statements to prepare for future Hive releases.

Arguments

libref
specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

hadoop
specifies the SAS/ACCESS engine name for the Hadoop interface.

connection-options
provide connection information and control how SAS manages the timing and concurrence of the connection to Hadoop. Here is how these options are defined.

USER=<'>*user-name*<'>
lets you connect to a Hadoop database with a user ID that is different from the default ID. USER= is required for creating tables but is ignored for reading them.

PASSWORD=<'>*password*<'>
specifies the Hadoop password that is associated with your user ID. If it contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. If you do not want to enter your Hadoop password in uncoded text on this statement, see PROC PWENCODE in the *Base SAS Procedures Guide* for a method to encode it.

Alias: PASS=, PWD=, PW=

SERVER=<'>*server-name*<'>
specifies the Hadoop server name that runs the Hive Server. If the server name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

Alias: HOST=

PORT=*port*
specifies the port number that is used to connect to the specified Hive Server.

Default: 10000

SCHEMA=*Hive-schema*
specifies the Hive schema.

Default: SCHEMA=`default`

Alias: DATABASE=, DB=

CONFIG=*config-file*
specifies the Hadoop configuration file for SAS/ACCESS to use. URL components in the configuration file should match the values in the SERVER=, PORT=, and BL_PORT= LIBNAME options. Specify CONFIG= only if you are connecting to a Hadoop system that is not based on HDFS, such as the MapR Hadoop Distribution.

LIBNAME-options
define how SAS processes DBMS objects. The following table describes the LIBNAME options for SAS/ACCESS Interface to Hadoop, with the applicable default values. For details about these options, see "LIBNAME Options for Relational Databases" on page 99.

***Table 18.1** SAS/ACCESS LIBNAME Options for Hadoop*

Option	Default Value
ACCESS=	none
AUTHDOMAIN=	none
BL_DELIMITER=	\001 (Ctrl-A)
BL_PORT=	8020 (required only if Hadoop HDFS streaming is running on a nondefault port)
BULKLOAD=	none
DBCREATE_TABLE_EXTERNAL=	NO
DBCREATE_TABLE_OPTS=	none
DBGEN_NAME=	DBMS
DBSASLABEL= on page 297	COMPAT
DIRECT_SQL=	YES
SPOOL=	YES
SQL_FUNCTIONS=	none

Hadoop LIBNAME Statement Examples

This example uses the default Hive port and schema.

```
libname hdp hadoop server=hxpduped
   user=hadoop_usr password=hadoop_pwd;
```

This example explicitly specifies the default Hive port and schema.

```
libname hdp hadoop server=hxpduped port=10000 schema=default
   user=hadoop_usr password=hadoop_pwd;
```

Data Set Options for Hadoop

All SAS/ACCESS data set options in this table are supported for Hadoop. Default values are provided where applicable. For general information about this feature, see Data Set Options for Relational Databases on page 207.

Table 18.2 *SAS/ACCESS Data Set Options for Hadoop*

Option	Default Value
DBCONDITION=	none
DBCREATE_TABLE_EXTERNAL= data set option	NO
DBCREATE_TABLE_LOCATION=	
DBCREATE_TABLE_OPTS=	LIBNAME option setting
DBGEN_NAME=	DBMS
DBLABEL=	NO
DBMASTER=	none
DBSASLABEL=	COMPAT
DBSASTYPE=	see "Data Types for Hadoop" on page 544
DBTYPE=	see "Data Types for Hadoop" on page 544

SQL Pass-Through Facility Specifics for Hadoop

Key Information

For general information about this feature, see "About SQL Procedure Interactions " on page 401. Hadoop examples on page 539 are available.

Here are the SQL pass-through facility specifics for the Hadoop interface.

- The *dbms-name* is **HADOOP**.
- The CONNECT statement is required.
- PROC SQL supports multiple connections to Hadoop. If you use multiple simultaneous connections, you must use the *alias* argument to identify the different connections. If you do not specify an alias, the default **HADOOP** alias is used.
- The *database-connection-arguments* for the CONNECT statement are identical to its LIBNAME connection options on page 537.

CONNECT Statement Examples

This example uses the default Hive port and schema.

```
proc sql;
   connect to hadoop (user="myuser" pw="mypwd" server=hxpduped);
```

This example explicitly specifies the default Hive port and schema.

```
proc sql;
   connect to hadoop (user="myuser" pw="mypwd"
      server=hxpduped port=10000 schema=default);
```

Passing SAS Functions to Hadoop

SAS/ACCESS Interface to Hadoop passes the following SAS functions to Hadoop for processing. Where the Hadoop function name differs from the SAS function name, the Hadoop name appears in parentheses. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

ABS	MAX
ARCOS (ACOS)	MIN
ARSIN (ASIN)	MINUTE
ATAN	MONTH
AVG	SECOND
CEIL	SIN
COALESCE	SQRT
COS	STD (STDDEV_SAMP)
COUNT	STRIP (TRIM)
DAY	SUBSTR
EXP	SUM
FLOOR	TAN
HOUR	TRANSTRN (REGEXP_REPLACE)
INDEX (LOCATE)	TRIMN (RTRIM)
LENGTH	UPCASE (UPPER)
LOG (LN)	VAR (VAR_SAMP)
LOG10	YEAR
LOWCASE (LOWER)	

SQL_FUNCTIONS=ALL on page 185 allows for SAS functions that have slightly different behavior from corresponding Hadoop functions. Due to incompatibility in date and time functions between Hadoop and SAS, Hadoop might not process them identically. Check your results to determine whether these functions are working as expected. See"SQL_FUNCTIONS= LIBNAME Option" on page 185.

COMPRESS (REGEXP_REPLACE)	DATETIME (FROM_UNIXTIME(UNIX_TIMESTAMP))
DATE (TO_DATE(UNIX_TIMESTAMP))	REPEAT
DATEPART (TO_DATE(UNIX_TIMESTAMP))	TODAY (TO_DATE(UNIX_TIMESTAMP)

Passing Joins to Hadoop

The SAS/ACCESS engine does not pass LIBNAME-referenced cross-schema joins to Hadoop. To pass a multiple-libref join to Hadoop, the schemas for each LIBNAME statement must be identical. You can use the SQL pass-through facility on page 539 to pass a cross-schema join to Hadoop.

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Bulk Loading for Hadoop

Loading

SAS/ACCESS Interface to Hadoop has no differentiation between a bulk load and a standard load process. Although BULKLOAD=YES syntax is supported, it does not change the underlying load process.

Here is how bulk loading works in the Hadoop interface.

1. SAS issues a CREATE TABLE command to the Hive server. The command contains all table metadata (column definitions) and the table properties that are specific to SAS that refine Hive metadata to handle maximum string lengths and date/time formats.

2. SAS initiates an HDFS streaming connection to upload table data to the HDFS /tmp directory. The resulting file is a UTF-8 text file that uses CTRL-A ('\001') as a field delimiter and a newline ('\n') character as a record separator. These are the defaults that SAS uses for Hive.

3. SAS issues a LOAD DATA command to the Hive server to move the data file from the HDFS /tmp directory to the appropriate Hive warehouse location. The data file is now considered to be an internal table.

Hives considers a table to be a collection of files in a directory that bears the table name. The CREATE TABLE command creates this directory either directly in the Hive warehouse or in a subdirectory, if a nondefault schema is used. The LOAD DATA command moves the table to the correct location.

When PROC APPEND is used, the CREATE TABLE command is skipped because the base table already exists. The data to be appended is streamed to the HDFS /tmp directory, and the LOAD DATA command moves the file as an additional file into the Hive warehouse directory. After that, multiple files exist for a given table, using unique names. During read, Hive concatenates all files belonging to a table to render the table as a whole.

Note: These functions are not currently supported.

- loading Hbase tables
- loading tables that use nontext format and custom serializers and deserializers (*serdes*)

External Tables

External tables are stored in HDFS but outside of the directories set aside for Hive. Reading an external table that was created manually is transparent to SAS because Hive handles the physical location of the file. However, Hadoop bulk loading always creates internal tables within the Hive warehouse directory structure. Follow these manual steps to create an external table.

1. Issue the appropriate CREATE TABLE command, using either the Hive command interface or by using explicit SQL in SAS.
2. Create the table file in HDFS. One way to do this is by using the Hadoop Access Method.
3. Issue the LOAD DATA command by using the LOCAL keyword, which indicates that the table file is local to HDFS or external to Hive.

For details about the Hadoop Access Method, see the *Base SAS Procedures Guide*.

File Formats

SAS can read Hive tables that are formatted in any way that is compatible with Hive. However, Hive tables that SAS creates through bulk loading always use these characteristics:

- They are stored as text files. Numeric data is expressed as strings of formatted ASCII characters.
- The field delimiter is a CTRL-A (\001), which is the Hive default.
- The record delimiter is a newline (\n) character, which also the Hive default.
- Null (empty or missing) values for strings are represented as \N (backslash with a capital N)—not to be confused with \n, which is only a single newline character.
- The Hive default SerDe (serializer/deserializer) for text files is used (LazySimpleSerDe).

File Locations

Internal Hive tables are stored in the Hive warehouse. By default, this is in the HDFS file system under /user/hive/warehouse. Creating a table creates a directory with the table name, and all files in that directory are considered to be part of the table. HDFS also splits and replicates the data among the Hadoop member nodes, but that is transparent to operations and commands that are described in this document. To manually check these files, log on to the Hadoop namenode and use the **`hadoop fs`** command to navigate the HDFS file system. Here are some examples.

Command	Description
`hadoop fs -lsr /user/hive/warehouse`	Lists all Hive files
`hadoop fs -ls /user/hive/warehouse/myschema.db`	Lists all Hive table directories in **`myschema`**

Command	Description
hadoop fs -ls /user/hive/warehouse/mytable	Lists all Hive table files that belong to the **mytable** table in the default schema
hadoop fs -cat /user/hive/warehouse/mytable/<filename>	Displays the actual contents of the file
hadoop fs -ls /user/hive/warehouse	Lists all Hive table directories in the default schema

Examples

This example creates and loads the FLIGHTS98 Hadoop table from the SASFLT.FLT98 SAS data set.

```
libname sasflt 'SAS-library';
libname hdp_air hadoop user=louis pwd=louispwd server='hdpcluster' schema=statsdiv;

proc sql;
create table hdp_air.flights98
        as select * from sasflt.flt98;
quit;
```

This example also creates and loads the ALLFLIGHTS Hadoop table from the SASFLT.ALLFLIGHTS SAS data set.

```
data hdp_air.allflights;
set sasflt.allflights;
run;
```

In this example, the SASFLT.FLT98 SAS data set is appended to the existing ALLFLIGHTS Hadoop table.

```
proc append base=hdp_air.allflights
data=sasflt.flt98;
run;
```

Naming Conventions for Hive

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

SAS and Hadoop objects include tables, views, table references, columns, and indexes. They follow these naming conventions.

- A SAS name must be from 1 to 32 characters long. When Hive column names and table names are 32 characters or less, SAS handles them seamlessly. When SAS reads Hive column names that are longer than 32 characters, a generated SAS variable name is truncated to 32 characters. Hive table names should be 32 characters or less because SAS cannot truncate a table reference. If you already have a table name that is greater than 32 characters, create a Hive table view or use the explicit SQL feature of PROC SQL to access the table.

- If truncating would result in identical names, SAS generates a unique name.
- Even when it is enclosed in single or double quotation marks, a Hive name does not retain case sensitivity. Hive table and column names can contain uppercase letters A through Z (A-Z), lowercase letters A through Z (a-z), numbers from 0 to 9, and the underscore (_). Hive converts uppercase characters to lowercase. Therefore, such SAS table references as MYTAB and mytab are synonyms—referring to the same table.
- A name can begin with a letter or an underscore but not a number.
- A name cannot be a Hadoop reserved word. If a name generates a Hadoop error, try to append a number or underscore in an appropriate place. For example, if *shipped* results in an error, try *shipped1* or *ship_date*.

Although the PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options are supported for SAS/ACCESS Interface to Hadoop, you should not need to use them.

Data Types for Hadoop

Overview

Hive is a data warehouse that supplies metadata about data that is stored in Hadoop files. Hive includes a data dictionary and an accompanying SQL-like interface called HiveQL or Hive SQL. HiveQL implements data definition language (DDL) and data manipulation language (DML) statements similar to many relational database management systems. Hive tables are defined with a CREATE TABLE statement, so every column in a table has a name and a data type. The data type indicates the format in which the data is stored. Hive exposes data that is stored in HDFS and other file systems through the data types that are described in this section. These are accepted column types in a Hive CREATE TABLE statement. This section includes information about Hive data types and data conversion between Hive and SAS.

For more information about Hive, Hadoop, and data types, see these online documents at `https://cwiki.apache.org/confluence/display/Hive`.

- *Hive Getting Started*
- *Hive Language Manual*
- *Hive Language Manual UDF*
- *Hive Tutorial*

Simple Hive Data Types

STRING
: specifies variable-length character data. The maximum length is 2 gigabytes.

TINYINT
: specifies a signed one-byte integer.

SMALLINT
: specifies a signed two-byte integer.

INT
: specifies a signed four-byte integer.

BIGINT
specifies a signed eight-byte integer.

FLOAT
specifies a signed four-byte, floating-point number.

DOUBLE
specifies a signed eight-byte, floating-point number.

BOOLEAN
specifies a textual true or false value.

Complex Hive Data Types

ARRAY
specifies an array of integers (indexable lists).

MAP
specifies an associated array from strings to string (key-value pairs).

STRUCT
specifies how to access elements within the type—namely, by using the dot (.) notation.

Hive Date, Time, and Timestamp Data

Hive does not define data types for dates, times, or timestamps. By convention, these are typically stored in ANSI format in Hive STRING columns. For example, the last day of this millennium is stored as the string *'2999-12-31'*. SAS/ACCESS assumes ANSI format for dates, times, and timestamps that are stored in Hadoop. Therefore, if you use the DBSASTYPE option to indicate that a Hive STRING column contains a date, SAS/ACCESS expects an ANSI date that it converts to SAS date format. For output, SAS DATE, TIME, and DATETIME formats are converted to ANSI format and are stored in Hive STRING columns.

SAS/ACCESS does not currently support conversion of Hadoop binary UNIX timestamps to SAS DATETIME format.

SAS Data Types

SAS has two fundamental data types, character and numeric. SAS character variables (columns) are of a fixed length with a maximum of 32,767 characters. SAS numeric variables are signed eight-byte, floating-point numbers. When SAS numerics are used in conjunction with SAS formats, they can represent a number of data types, including DATE, TIME, and DATETIME. For more detailed information about SAS data types, see *SAS Language Reference: Concepts*.

Data Conversion from Hive to SAS

This table shows the default SAS formats that are assigned to SAS variables that are created when SAS/ACCESS reads Hive table columns.

Table 18.3 *Hive to SAS: Default SAS Formats for Hive Data Types*

Hive Data Type	SAS Data Type	Default SAS Format
STRING	character	$32767.
TINYINT	numeric	4.
SMALLINT	numeric	6.
INT	numeric	11.
BIGINT	numeric	20.
FLOAT	numeric	none
DOUBLE	numeric	none
BOOLEAN	numeric	1.

Issues When Converting Data from Hive to SAS

Below are some potential conversion issues.

- Hive STRING columns that contain ANSI date, time, or timestamp values do not automatically convert respectively to SAS DATE, TIME, or DATETIME types.
- STRING: Depending on the length of Hadoop STRING data, the SAS character $32767. format might be unnecessarily large for short STRING columns or can truncate Hadoop STRING columns that contain more than 32767 characters.
- BIGINT: Converting Hadoop BIGINT to a SAS numeric can result in loss of precision because the internal SAS eight-byte, floating-point format accurately preserves only 15 digits of precision. A BIGINT preserves up to 19.

Work-arounds are based on how you access data.

- explicitly using pass-through SQL, see “Address Issues When Converting Data from Hive to SAS for Pass-Through SQL” on page 552.
- using the LIBNAME statement, see “Address Issues When Converting Data from Hive to SAS with Table Properties” on page 549.

SAS Table Properties for Hive and Hadoop

Although HiveQL supplies critical metadata for Hadoop files, in some cases more metadata is beneficial. Fortunately, HiveQL CREATE TABLE and ALTER TABLE statements provide an extensible feature called table properties. For more information, see the Hive Language Manual at **`https://cwiki.apache.org/confluence/display/Hive`**.) SAS/ACCESS uses table properties to describe and interpret the contents of Hive STRING columns.

Here is an example of a new Hive table that is created with SAS/ACCESS. A SAS variable (column) has an associated SAS DATETIME format, so SAS/ACCESS creates a DATETIME table property for the Hive column.

```
libname hdp hadoop server=dbihadoop user=hadoop_usr pwd= hadoop_pwd;
data hdp.datetime_tableproperty_sample;
format dt_stamp datetime25.6;
dt_stamp=datetime();
run;
```

This code creates a new Hive table, DATETIME_TABLEPROPERTY_SAMPLE, by generating this HiveQL:

```
CREATE TABLE `DATETIME_TABLEPROPERTY_SAMPLE` (`dt_stamp` STRING)
   ROW FORMAT DELIMITED FIELDS TERMINATED BY '\001'
   STORED AS TEXTFILE TBLPROPERTIES ('SASFMT:dt_stamp'='DATETIME(25.6)')
```

SAS stores **dt_stamp** as a Hive ANSI STRING, as in this example:

```
2012-02-23 09:51:37.218
```

Based on the SAS DATETIME25.6 format, SAS/ACCESS also generates the Hive table property that describes STRING column **dt_stamp** as DATETIME(25.6).

When SAS/ACCESS reads this Hive table, the SASFMT table property indicates STRING column **dt_stamp** contains an ANSI timestamp. SAS/ACCESS automatically converts and formats it a SAS DATETIME25.6 variable, as in this example:

```
data;
set hdp.datetime_tableproperty_sample;
put dt_stamp=;
run;
```

```
dt_stamp=23FEB2012:09:51:37.218000
NOTE: There were 1 observations read from the data set
HDP.DATETIME_TABLEPROPERTY_SAMPLE.
```

When SAS/ACCESS creates a new Hive table, it generates table properties for SAS variables with character, date, datetime, and time formats—all of which produce Hive STRING columns. See the generated table properties in "Data Conversion from SAS to Hive" on page 547

Data Conversion from SAS to Hive

This table shows the Hive data types and table properties that are assigned when SAS/ACCESS creates a Hive table.

SAS sets table properties only when creating a new Hive table. It does not create or alter table properties when appending to an existing Hive table.

Table 18.4 *Hive Data Types and Table Properties That Are Created for SAS Data Type and Format Combinations*

SAS Data Type	SAS Format	Hive Data Type	Hive Table Property
character	$n.	STRING	CHAR(n)
numeric	DATETIMEw.p	STRING	DATETIME(w.p)

SAS Data Type	SAS Format	Hive Data Type	Hive Table Property
numeric	DATEw.	STRING	DATE(w.0)
numeric	TIMEw.p.	STRING	TIME(w.p)
numeric	1. to 4.	SMALLINT	none
numeric	5. to 9.	INT	none
numeric	other numeric formats	DOUBLE	none

Leverage Table Properties for Existing Hive Tables

SAS/ACCESS generates SAS table properties only when creating a new Hive table. Many or perhaps all of your Hive tables are created by other means. For example, your Hadoop administrator might create Hive table definitions by submitting DDL scripts to the Hive CLI. SAS and SAS users can benefit by adding SAS table properties to existing Hive table definitions. Here is an example, where a Hive table has already been defined.

```
CREATE EXTERNAL TABLE weblogs (extract_date STRING,
   extract_type INT, webdata STRING) ROW FORMAT DELIMITED FIELDS
   TERMINATED BY ',' STORED AS TEXTFILE LOCATION '/user/hadoop/web_data'
```

Based on this table definition, here is how SAS interprets the columns.

```
libname hdp sasiohdp server=dbihadoop user=hadoop_usr pwd= hadoop_pwd;
data sheetmetal_sales; set hdp.weblogs(obs=1);
put extract_date= extract_type=;
put webdata=;
run;
```

```
extract_date=2012-02-21 extract_type=1
webdata=http://www.sas.com/industry/oilgas
NOTE: There were 1 observations read from the data set HDP.WEBLOGS.
```

```
proc contents data=hdp.weblogs; run;
```

```
          Alphabetic List of Variables and Attributes
#  Variable       Type  Len    Format   Informat   Label
1  extract_date   Char  32767  $32767.  $32767.    extract_date
2  extract_type   Num       8  11.      11.        extract_type
3  webdata        Char  32767  $32767.  $32767.    webdata
```

Notice that Hive describes the **`extract_date`** column to SAS as a 32767 length STRING. It also describes the **`webdata`** column as a 32767 length STRING. So SAS/ACCESS enters both of these columns as character data and uses $32767. to format them. The result is an overly wide SAS data set with an observation (row) width of 64 kilobytes that also does not format **`extract_date`** to a SAS DATE.

SAS issues a warning message for this situation, which includes the maximum column length that was in the result set. In the example, the maximum length read for the extract_date STRING column is 10 bytes. The maximum length read for the webdata STRING column was 320 bytes.

```
WARNING: SAS/ACCESS assigned these columns a length of 32767. If
resulting SAS character variables remain this length, SAS
performance is impacted. See SAS/ACCESS documentation for details.
Columns followed by the maximum length observed were:
extract_date:10, webdata:320
```

The example below assumes that the length of the **webdata** STRING in Hive never exceeds 1000 characters. A Hadoop user ID with the appropriate authority can issue Hive ALTER TABLE statements to add SAS table properties to the Hive table definition.

```
ALTER TABLE weblogs SET TBLPROPERTIES ('SASFMT:extract_date'='DATE(9.0)')
ALTER TABLE weblogs SET TBLPROPERTIES ('SASFMT:webdata'='CHAR(1000)')
```

SAS/ACCESS honors the added properties, and here is the result.

```
libname hdp sasiohdp server=dbihadoop user=hadoop_usr pwd= hadoop_pwd;
data sheetmetal_sales; set hdp.weblogs(obs=1);
put extract_date= extract_type=;
put webdata=;
run;
```

```
extract_date=21FEB2012 extract_type=1
webdata=http://www.sas.com/industry/oilgas
NOTE: There were 1 observations read from the data set HDP.WEBLOGS.
```

```
proc contents data=hdp.weblogs; run;
```

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Format	Informat	Label
1	extract_date	Num	8	DATE9.	DATE9.	extract_date
2	extract_type	Num	8	11.	11.	extract_type
3	webdata	Char	1000	$1000	$1000.	webdata

The resulting SAS data set that is created from the Hive table has a much smaller observation width, which helps SAS save disk space and reduce CPU consumption. It also automatically converts and formats **extract_date** to SAS standard DATE9. format.

Adding SAS properties to existing Hive tables does not impact table use by software that is not SAS. You can also issue ALTER TABLE and other DDL statements using SAS/ACCESS explicit SQL. (See SQL Pass-Through Facility Specifics for Hadoop on page 539). Issuing such DDL as an ALTER TABLE statement can be restricted to only the Hadoop administrator.

Address Issues When Converting Data from Hive to SAS with Table Properties

Some issues currently exist when reading Hadoop data into SAS. (See "Issues When Converting Data from Hive to SAS" on page 546.) For example, Hive STRING columns default to the $32767. SAS character format without a defined SASFMT table property or a SAS override option such as DBSASTYPE=.

Here is how you can address specific conversion issues.

STRING issues

To automatically convert Hive STRING columns that contain ANSI date, timestamp, or time values to suitable SAS formats, you can use the following ALTER TABLE statements. In the statements are these sample Hive columns: **d** contains ANSI date, **ts** contains ANSI timestamp, and **t** contains ANSI time.

```
ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:d'='DATE(9.0)')
ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:ts'='DATETIME(25.6)')
ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:t'='TIME(15.6)')
```

Instead, you could use these statements to create SAS character variables of optimal length that contain the identical ANSI representation as those that are stored in Hive:

```
ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:d'='CHAR(9)')
ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:ts'='CHAR(25)')
ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:t'='CHAR(15)')
```

You can use the following statement for other Hive STRING columns where the maximum length is less than 32767. Here, the **string_col** column has a maximum length of 100.

```
ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:string_col'='CHAR(100)')
```

However, if you anticipate that the **string_col** column in Hive might grow to a maximum length of 200 in the future, you could instead use this statement to set the table property.

```
ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:string_col'='CHAR(200)')
```

Hive STRING columns longer than 32767 characters are truncated when they are read into SAS. Here is how the warning for this data loss is flagged in the SAS log:

```
WARNING: Column 'string_col' was truncated 1 times.
Observation (row) number 2 was the first observation truncated.
```

BIGINT issues

Converting a Hadoop BIGINT column to a SAS numeric column can cause a loss of precision. A SAS numeric column can accurately preserve only 15 digits of precision, whereas a BIGINT column can preserve up to 19 significant digits of precision. You can address this issue by applying a CHAR(19) table property format. SAS then automatically reads a Hive BIGINT column into a SAS character string with $19. format, which preserves all BIGINT digits in character format. Here is an example, using the **bgint** BIGINT column.

```
ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:bgint'='CHAR(19)')
```

Keep this important consideration in mind, however: For Hive tables that SAS/ACCESS creates, you might not need to issue ALTER TABLE statements. See "Data Conversion from SAS to Hive" on page 547 for table properties that SAS/ACCESS automatically generates when it creates a Hive table.

CAUTION:

Do not create multiple table properties for a single Hive column. Unpredictable data conversion can result.

Alternatives to Table Properties for Issues with Data Conversion from Hive to SAS

For various reasons, it might be impractical or undesirable to issue ALTER TABLE statements to create SAS table properties. In such cases, you can instead use these data set options.

DBSASTYPE=

Use DBSASTYPE= in your SAS code to cause data conversion from Hive to SAS that is identical to automatic conversion with table properties. The pairs below are

SAS DATA steps with identical behavior. The first of each pair uses a SASFMT table property, the second one uses no table property, and DBSASTYPE= is added to achieve the same functionality. (For details, see the DBSASTYPE= data set option on page 298.) Here is the SAS LIBNAME statement for all of these SAS DATA steps.

```
libname hdp sasiohdp server=dbihadoop user=hadoop_usr pwd= hadoop_pwd;

ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:d'='DATE(9.0)')
data work.local_sample; set hdp.sample_table( keep=d ); run;

[---no table property for column 'd'---]
data work.local_sample;
set hdp.sample_table( keep=d dbsastype=(d='DATE(9.0)') ); run;

ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:ts'='DATETIME(25.6)')
data work.local_sample; set hdp.sample_table( keep=ts ); run;

[---no table property for column 'ts'---]
data work.local_sample;
set hdp.sample_table( keep=ts dbsastype=(ts='DATETIME(25.6)') ); run;


ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:t'='TIME(15.6)')
data work.local_sample; set hdp.sample_table( keep=t ); run;

[---no table property for column 't'---]
data work.local_sample;
set hdp.sample_table( keep=t dbsastype=(t='TIME(15.6)') ); run;


ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:string_col'='CHAR(200)')
data work.local_sample; set hdp.sample_table( keep=string_col ); run;

[---no table property for column 'string_col'---]
data work.local_sample;
set hdp.sample_table( keep=string_col dbsastype=(string_col='CHAR(200)') );
run;
```

COMPRESS=YES

If you can use neither table properties nor DBSASTYPE= to reduce the default 32767 SAS character length for Hive STRING columns, consider using this data set option. Although SAS character variables still use the $32767. format, using SAS data set compression, you can realize significant savings in terms of disk space and CPU consumption. Here is an example.

```
data work.local_sample( COMPRESS=YES ); set hdp.sample_table; run;
```

Apply COMPRESS=YES to the SAS data set reference, not to the Hadoop table reference. Also consider using this value even if you reduce default SAS character 32767 lengths with table properties or DBSASTYPE=, but also when many strings in the Hadoop data are much smaller than the reduced SAS character variable length. For example, you could use COMPRESS=YES if you had the table property shown below, but most values of **address_1** are less than 100 characters.

```
ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:address_l'='CHAR(1000)')
```

For more information about this data set option, see *SAS Data Set Options: Reference*.

Address Issues When Converting Data from Hive to SAS for Pass-Through SQL

Neither table properties nor DBSASTYPE= address data conversion issues from Hive to SAS if you use pass-through SQL to read Hive data. For pass-through SQL, you might need to explicitly convert and format each Hive column as you want it to be represented in SAS. You can see this if you use SAS to create a table with SAS table properties that are generated for all but the BIGINT column. Here is the table that SAS creates.

```
libname hdp sasiohdp server=dbihadoop user=hadoop_usr pwd=hadoop_pwd;
data hdp.passthrough_ex( dbtype=(bgint="BIGINT") );
bgint='1234567890123456789';
format ts datetime25.6; ts=datetime();
format d date9.; d=today();
format t time10.; t=time();
format string_col $20.; string_col='hello';
run;
```

SAS issues this HiveQL when creating the table.

```
CREATE TABLE `PASSTHROUGH_EX` (`bgint` BIGINT,`ts` STRING,
   `d` STRING,`t` STRING,`string_col` STRING) ROW FORMAT DELIMITED FIELDS
   TERMINATED BY '\001' STORED AS TEXTFILE TBLPROPERTIES
('SASFMT:ts'='DATETIME(25.6)','SASFMT:d'='DATE(9.0)',
   'SASFMT:t'='TIME(10.0)','SASFMT:string_col'='CHAR(20)')
```

Next, an ALTER TABLE statement is issued to add a table property for **`BIGINT column bgint`**.

```
ALTER TABLE passthrough_ex SET TBLPROPERTIES ('SASFMT:bgint'='CHAR(19)')
```

A LIBNAME-based table that is read to SAS honors the table properties.

```
data work.local; set hdp.passthrough_ex;
run;
data _null_; set work.local;
put bgint=; put ts=; put d=; put t=; put string_col=;
run;
```

```
   bgint=1234567890123456789
   ts=25FEB2012:02:00:55.141000
   d=25FEB2012
   t=2:00:55
   string_col=hello
```

This pass-through SQL step converts and formats each column identically to the LIBNAME-based step that applied the table properties.

```
proc sql; connect to sasiohdp(server=dbihadoop user=hadoop_usr pwd=hadoop_pwd);
create table work.local as select
bgint length 19 format $19. informat $19.,
input(ts, IS8601DT26.) as ts format datetime25.6 informat datetime25.6,
input(d, yymmdd10.) as d format date9. informat date9.,
input(t, IS8601TM15.) as t format time15.6 informat time15.6,
string_col length  20 format $20. informat $20.
from connection to sasiohdp( select cast(bgint as STRING)
   as bgint,ts,d,t,string_col from passthrough_ex );
quit;
data _null_; set work.local;
```

```
put bgint=; put ts=; put d=; put t=; put string_col=;
run;
```

```
bgint=1234567890123456789
ts=25FEB2012:02:00:55.141000
d=25FEB2012
t=2:00:55.141000
string_col=hello
```

Hadoop Null Values

Hadoop has a special value called NULL. A Hadoop NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a Hadoop NULL value, it interprets it as a SAS missing value. For more information about how SAS handles NULL values, see "Potential Result Set Differences When Processing Null Data " on page 35.

Sample Code for Hadoop

Code Snippets

The code snippets in this section resemble those for most other SAS/ACCESS interfaces.

This snippet shows a list of available Hive tables.

```
proc datasets lib=hdp; quit;
```

Here is the metadata for the mytab Hive table.

```
proc contents data=hdp.mytab; quit;
```

This snippet extracts mytab data into SAS.

```
data work.a;
set hdp.mytab;
run;
```

This extracts a subset of the mytab rows and columns into SAS. Subsetting the rows (with a WHERE statement, for example) can help avoid extracting too much data into SAS.

```
data work.a;
set hdp.mytab (keep=col1 col2);
where col2=10;
run;
```

Use DBSASTYPE= to Load Hadoop Data into SAS

This example uses the DBSASTYPE= data set option to load Hadoop textual dates, timestamps, and times into the corresponding SAS DATE, DATETIME, and TIME formats. The first step reads in a SAS character string to display the data and make clear what occurs in successive steps.

```
data; set hdp.testHiveDate; put dt; run;
```

```
2011-10-17
2009-07-30 12:58:59
11:30:01
```

```
data; set hdp.testHiveDate(dbsastype=(dt='date')); put dt; run;
```

```
17OCT2011
30JUL2009
.
```

```
data; set hdp.testHiveDate(dbsastype=(dt='datetime')); put dt; run;
```

```
17OCT2011:00:00:00
30JUL2009:12:58:59
.
```

```
data; set hdp.testHiveDate(dbsastype=(dt='time')); put dt; run;
```

```
.
12:58:59
11:30:01
```

This code uses SAS SQL to access a Hadoop table.

```
proc sql;
create table work.a as select * from hdp.newtab;
quit;
```

SAS data is then loaded into Hadoop.

```
data hdp.newtab2;
set work.a;
run;
```

Use implicit pass-through SQL to extract only 10 rows from the newtab table and load the work SAS data set with the results.

```
proc sql;
connect to hadoop (server=hxpduped);
create table work.a as
   select * from connection to hadoop (select * from newtab limit 10);
```

Chapter 19

SAS/ACCESS Interface to HP Neoview

Introduction to SAS/ACCESS Interface to HP Neoview

This section describes SAS/ACCESS Interface to HP Neoview. For a list of SAS/ACCESS features that are available in this interface, see “SAS/ACCESS Interface to HP Neoview: Supported Features” on page 81.

LIBNAME Statement Specifics for HP Neoview

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to HP Neoview supports and includes examples. For details about this feature, see the LIBNAME Statement on page 93.

Here is the LIBNAME statement syntax for accessing HP Neoview.

LIBNAME *libref* **neoview** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`neoview`
specifies the SAS/ACCESS engine name for the HP Neoview interface.

connection-options
provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. When you use the LIBNAME statement on UNIX or Microsoft Windows, you can connect to HP Neoview Database Connectivity Service (NDCS) by connecting a client to a data source. Specify *only one* of the following methods for each connection because each is mutually exclusive.

- SERVER=, SCHEMA=, PORT=, USER=, PASSWORD=
- DSN=, USER=, PORT=

Here is how these options are defined.

SERVER=<'>*server-name*<'>
specifies the server name or IP address of the HP Neoview server to which you want to connect. This server accesses the database that contains the tables and views that you want to access. If the server name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

SCHEMA=<'>*schema-name*<'>
specifies the name of a schema. When you use it with SERVER= or PORT=, it is passed directly as a connection option to the database. When you use it with

DSN=, it qualifies SQL statements as a LIBNAME option on page 180. You can also use it as a data set option on page 344.

PORT=*port*

specifies the port number that is used to connect to the specified HP Neoview server. If you do not specify a port, the default is 18650.

USER=<'>*Neoview-user-name*<'>

specifies the HP Neoview user name (also called the user ID) that you use to connect to your database. If the user name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

PASSWORD=<'>*Neoview-password*<'>

specifies the password that is associated with your HP Neoview user name. If the password contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. You can also specify PASSWORD= with the PWD=, PASS=, and PW= aliases.

DSN=<'>*Neoview-data-source*<'>

specifies the configured HP Neoview ODBC data source to which you want to connect. Use this option if you have existing HP Neoview ODBC data sources that are configured on your client. This method requires additional setup—either through the ODBC Administrator control panel on Windows platforms or through the MXODSN file or a similarly named configuration file on UNIX platforms. So it is recommended that you use this connection method only if you have existing, functioning data sources that have been defined.

LIBNAME-options

define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to HP Neoview, with the applicable default values. For more detail about these options, see "LIBNAME Statement Syntax for Relational Databases" on page 95.

Table 19.1 *SAS/ACCESS LIBNAME Options for HP Neoview*

Option	Default Value
ACCESS= on page 99	none
AUTHDOMAIN= on page 102	none
AUTOCOMMIT= on page 103	operation-specific
BL_NUM_ROW_SEPS= on page 106	1
BULKEXTRACT= on page 107	NO
CONNECTION= on page 113	UNIQUE
CONNECTION_GROUP= on page 118	none
CONNECTION_TIMEOUT= on page 119	0
DBCOMMIT= on page 124	1000 when inserting rows; 0 when updating rows

Option	Default Value
DBCONINIT= on page 125	none
DBCONTERM= on page 126	none
DBCREATE_TABLE_OPTS= on page 128	none
DBGEN_NAME= on page 128	DBMS
DBINDEX= on page 129	YES
DBLIBINIT= on page 130	none
DBLIBTERM= on page 131	none
DBMAX_TEXT= on page 133	1024
DBMSTEMP= on page 133	NO
DBNULLKEYS= on page 135	YES
DBPROMPT= on page 136	NO
DBSLICEPARM= on page 139	THREADED_APPS,3
DEFER= on page 141	NO
DELETE_MULT_ROWS= on page 143	NO
DIRECT_EXE= on page 144	none
DIRECT_SQL= on page 145	YES
IGNORE_READ_ONLY_COLUMNS= on page 150	NO
INSERTBUFF= on page 153	automatically calculated based on row length
MULTI_DATASRC_OPT= on page 160	none
PRESERVE_COL_NAMES= on page 167	see “Naming Conventions for HP Neoview” on page 570
PRESERVE_TAB_NAMES= on page 168	see “Naming Conventions for HP Neoview” on page 570
QUALIFIER= on page 170	none
QUERY_TIMEOUT= on page 172	0
QUOTE_CHAR= on page 173	none

Option	Default Value
READBUFF= on page 174	automatically calculated based on row length
REREAD_EXPOSURE= on page 179	NO
SCHEMA= on page 180	none
SPOOL= on page 184	YES
SQL_FUNCTIONS= on page 185	none
SQL_FUNCTIONS_COPY= on page 188	none
STRINGDATES= on page 190	NO
TRACE= on page 195	NO
TRACEFILE= on page 196	none
UPDATE_MULT_ROWS= on page 199	
UTILCONN_TRANSIENT= on page 202	NO

HP Neoview LIBNAME Statement Examples

In this example, SERVER=, DATABASE=, USER=, and PASSWORD= are connection options.

```
libname mydblib neoview server=ndcs1 schema=USR user=neo1 password=neopwd1;
```

In the next example, DSN=, USER=, and PASSWORD= are connection options.

```
libname mydblib neoview DSN=TDM_Default_DataSource user=neo1 password=neopwd1;
```

Data Set Options for HP Neoview

All SAS/ACCESS data set options in this table are supported for HP Neoview. Default values are provided where applicable. For general information about this feature, see Data Set Options for Relational Databases on page 207.

Table 19.2 *SAS/ACCESS Data Set Options for HP Neoview*

Option	Default Value
BL_BADDATA_FILE= on page 210	When BL_USE_PIPE=NO, creates a file in the current directory or with the default file specifications.

Option	Default Value
BL_DATAFILE= on page 216	When BL_USE_PIPE=NO, creates a file in the current directory or with the default file specifications.
BL_DELETE_DATAFILE= on page 233	YES (only when BL_USE_PIPE=NO)
BL_DELIMITER= on page 235		(the pipe symbol)
BL_DISCARDS= on page 239	1000
BL_ERRORS= on page 240	1000
BL_FAILEDDATA= on page 245	creates a data file in the current directory or with the default file specifications
BL_HOSTNAME= on page 247	none
BL_NUM_ROW_SEPS= on page 254	1
BL_PORT= on page 258	none
BL_RETRIES= on page 264	3
BL_ROWSETSIZE= on page 265	none
BL_STREAMS= on page 267	4 for extracts, no default for loads
BL_SYNCHRONOUS= on page 269	YES
BL_SYSTEM= on page 269	none
BL_TENACITY= on page 270	15
BL_TRIGGER= on page 270	YES
BL_TRUNCATE= on page 271	NO
BL_USE_PIPE= on page 271	YES
BULKEXTRACT= on page 274	LIBNAME option setting
BULKLOAD= on page 274	NO
DBCOMMIT= on page 280	LIBNAME option setting
DBCONDITION= on page 281	none
DBCREATE_TABLE= on page 285	LIBNAME option setting
DBFORCE= on page 286	NO
DBGEN_NAME= on page 287	DBMS

Option	Default Value
DBINDEX= on page 288	LIBNAME option setting
DBKEY= on page 290	none
DBLABEL= on page 291	NO
DBMASTER= on page 293	none
DBMAX_TEXT= on page 293	1024
DBNULL= on page 294	YES
DBNULLKEYS= on page 295	LIBNAME option setting
DBPROMPT= on page 296	LIBNAME option setting
DBSASLABEL= on page 137	COMPAT
DBSASTYPE= on page 298	see "Data Types for HP Neoview " on page 571
DBSLICE= on page 299	none
DBSLICEPARM= on page 301	THREADED_APPS,3
DBTYPE= on page 303	see "Data Types for HP Neoview " on page 571
ERRLIMIT= on page 308	1
IGNORE_READ_ONLY_COLUMNS= on page 311	NO
INSERTBUFF= on page 314	LIBNAME option setting
NULLCHAR= on page 329	SAS
NULLCHARVAL= on page 330	a blank character
PRESERVE_COL_NAMES= on page 336	LIBNAME option setting
QUALIFIER= on page 338	LIBNAME option setting
QUERY_TIMEOUT= on page 339	LIBNAME option setting
READBIFF on page 339	LIBNAME option setting
SASDATEFMT= on page 342	none
SCHEMA= on page 344	LIBNAME option setting

SQL Pass-Through Facility Specifics for HP Neoview

Key Information

For general information about this feature, see "About SQL Procedure Interactions " on page 401. HP Neoview examples are available.

Here are the SQL pass-through facility specifics for the HP Neoview interface.

- The *dbms-name* is **NEOVIEW**.
- The CONNECT statement is required.
- PROC SQL supports multiple connections to HP Neoview. If you use multiple simultaneous connections, you must use the *alias* argument to identify the different connections. If you do not specify an alias, the default **neoview** alias is used.
- The CONNECT statement *database-connection-arguments* are identical to its LIBNAME connection options on page 556.
- You can use the SCHEMA= option only with the SERVER= and PORT= connection options. It is not valid with DSN= in a pass-through connection.

CONNECT Statement Example

This example, uses the DBCON alias to connection to the **ndcs1** HP Neoview server and execute a query. The connection alias is optional.

```
proc sql;
   connect to neoview as dbcon
   (server=ndcs1 schema=TEST user=neo1 password=neopwd1);
select * from connection to dbcon
   (select * from customers where customer like '1%');
quit;
```

Special Catalog Queries

SAS/ACCESS Interface to HP Neoview supports the following special queries. You can use the queries to call the ODBC-style catalog function application programming interfaces (APIs). Here is the general format of the special queries:

Neoview::SQLAPI "*parameter 1*","*parameter n*"

Neoview::
: is required to distinguish special queries from regular queries.

SQLAPI
: is the specific API that is being called. Neither Neoview:: nor SQLAPI are case sensitive.

"*parameter n*"
: is a quoted string that is delimited by commas.

Within the quoted string, two characters are universally recognized: the percent sign (%) and the underscore (_). The percent sign matches any sequence of zero or more characters, and the underscore represents any single character. To use either character as a literal value, you can use the backslash character (\) to escape the match characters. For example, this call to SQLTables usually matches table names such as mytest and my_test:

```
select * from connection to neoview (NEOVIEW::SQLTables ","my_test");
```

Use the escape character to search only for the my_test table:

```
select * from connection to neoview (NEOVIEW::SQLTables ","my\_test");
```

SAS/ACCESS Interface to HP Neoview supports these special queries:

Neoview::SQLTables <*"Catalog"*, *"Schema"*, *"Table-name"*, "Type">
: returns a list of all tables that match the specified arguments. If you do not specify any arguments, all accessible table names and information are returned.

Neoview::SQLColumns <*"Catalog"*, *"Schema"*, *"Table-name"*, *"Column-name"*>
: returns a list of all columns that match the specified arguments. If you do not specify any argument, all accessible column names and information are returned.

Neoview::SQLPrimaryKeys <*"Catalog"*, *"Schema"*, *"Table-name"*>
: returns a list of all columns that compose the primary key that matches the specified table. A primary key can be composed of one or more columns. If you do not specify any table name, this special query fails.

Neoview::SQLSpecialColumns <*"Identifier-type"*, *"Catalog-name"*, *"Schema-name"*, *"Table-name"*, "*Scope"*, *"Nullable"*>
: returns a list of the optimal set of columns that uniquely identify a row in the specified table.

Neoview::SQLStatistics <*"Catalog"*, *"Schema"*, *"Table-name"*>
: returns a list of the statistics for the specified table name, with options of SQL_INDEX_ALL and SQL_ENSURE set in the SQLStatistics API call. If you do not specify any table name argument, this special query fails.

Neoview::SQLGetTypeInfo
: returns information about the data types that the HP Neoview server supports.

Autopartitioning Scheme for HP Neoview

Overview

Autopartitioning for SAS/ACCESS Interface to HP Neoview is a modulo (MOD) function method. For general information about this feature, see "Autopartitioning Techniques in SAS/ACCESS" on page 63.

Autopartitioning Restrictions

SAS/ACCESS Interface to HP Neoview places additional restrictions on the columns that you can use for the partitioning column during the autopartitioning phase. Here is how columns are partitioned.

- BIGINT, INTEGER, SMALLINT, and SMALLINT columns are given preference.

- You can use DECIMAL, DOUBLE, FLOAT, NUMERIC, or REAL columns for partitioning if the precision minus the scale of the column is greater than 0 but less than 19—namely, 0<(*precision-scale*)<19.

Nullable Columns

If you select a nullable column for autopartitioning, the **`OR<column-name>IS NULL`** SQL statement is appended at the end of the SQL code that is generated for the threaded Read. This ensures that any possible NULL values are returned in the result set.

Using WHERE Clauses

Autopartitioning does not select a column to be the partitioning column if it appears in a SAS WHERE clause. For example, this DATA step cannot use a threaded Read to retrieve the data because all numeric columns in the table are in the WHERE clause:

```
data work.locemp;
  set neolib.MYEMPS;
  where EMPNUM<=30 and ISTENURE=0 and
        SALARY<=35000 and NUMCLASS>2;
run;
```

Using DBSLICEPARM=

Although SAS/ACCESS Interface to HP Neoview defaults to three threads when you use autopartitioning, do not specify a maximum number of threads for the threaded Read in the DBSLICEPARM= LIBNAME option on page 139.

Using DBSLICE=

You might achieve the best possible performance when using threaded Reads by specifying the DBSLICE= data set option on page 299 for HP Neoview in your SAS operation. This is especially true if you defined an index on one column in the table. SAS/ACCESS Interface to HP Neoview selects only the first integer-type column in the table. This column might not be the same column that is being used as the partitioning key. If so, you can specify the partition column using DBSLICE=, as shown in this example.

```
proc print data=neolib.MYEMPS(DBSLICE=("EMPNUM BETWEEN 1 AND 33"
"EMPNUM BETWEEN 34 AND 66" "EMPNUM BETWEEN 67 AND 100"));
run;
```

Using DBSLICE= also gives you flexibility in column selection. For example, if you know that the STATE column in your employee table contains only a few distinct values, you can customize your DBSLICE= clause accordingly.

```
datawork.locemp;
set neolib2.MYEMP(DBSLICE=("STATE='FL'" "STATE='GA'"
  "STATE='SC'" "STATE='VA'" "STATE='NC'"));
where EMPNUM<=30 and ISTENURE=0 and SALARY<=35000 and NUMCLASS>2;
run;
```

Temporary Table Support for HP Neoview

General Information

See "Temporary Table Support for SAS/ACCESS " on page 42 for general information about this feature.

Establishing a Temporary Table

To make full use of temporary tables, the CONNECTION=GLOBAL connection option is necessary. This option lets you use a single connection across SAS DATA steps and SAS procedure boundaries. This connection can also be shared between LIBNAME statements and the SQL pass-through facility. Because a temporary table exists only within a single connection, you need to be able to share this single connection among all steps that reference the temporary table. The temporary table cannot be referenced from any other connection.

You can currently use only a PROC SQL statement to create a temporary table. To use both the SQL pass-through facility and librefs to reference a temporary table, you must specify a LIBNAME statement before the PROC SQL step. A global connection then persists across SAS steps and even across multiple PROC SQL steps. Here is an example:

```
proc sql;
   connect to neoview (dsn=NDCS1_DataSource
       user=myuser password=mypwd connection=global);
   execute (create volatile table temptab1
       as select * from permtable ) by neoview;
quit;
```

At this point, you can refer to the temporary table by using either the Temp libref or the CONNECTION=GLOBAL option with a PROC SQL step.

Terminating a Temporary Table

You can drop a temporary table at any time or allow it to be implicitly dropped when the connection is terminated. Temporary tables do not persist beyond the scope of a single connection.

Examples

These assumptions apply to the examples in this section.

- The DeptInfo table already exists on the DBMS that contains all your department information.
- One SAS data set contains join criteria that you want to use to extract specific rows from the DeptInfo table.
- The other SAS data set contains updates to the DeptInfo table.

These examples use these librefs and temporary tables.

```
libname saslib base 'SAS-library';
libname dept neoview dsn=Users_DataSource
    user=myuser pwd=mypwd connection=global;

proc sql;
   connect to neoview (dsn=Users_DataSource
      user=myuser pwd=mypwd connection=global);
   execute (create volatile table temptab1
      (dname char(20), deptno int)) by neoview;
quit;
```

This first example shows how to use a heterogeneous join with a temporary table. It performs a homogeneous join on the DBMS instead of reading the DBMS table into SAS to perform the join. By using the table that was created previously, you can copy SAS data into the temporary table to perform the join.

```
proc sql;
   connect to neoview (dsn=Users_DataSource
      user=myuser pwd=mypwd connection=global);
   insert into dept.temptab1 select * from saslib.joindata;
   select * from dept.deptinfo info, dept.temptab1 tab
        where info.deptno = tab.deptno;
   /* remove the rows for the next example */
   execute (delete from temptab1) by neoview;
   quit;
```

In this example, transaction processing on the DBMS occurs by using a temporary table instead of either DBKEY= or MULTI_DATASRC_OPT=IN_CLAUSE with a SAS data set as the transaction table.

```
proc sql;
connect to neoview (dsn=Users_DataSource
   user=myuser pwd=mypwd connection=global);
   insert into dept.temptab1 select * from saslib.transdat;
   execute (update deptinfo d set dname = (select dname from temptab1)
            where d.deptno = (select deptno from temptab1)) by neoview;
quit;
```

Passing SAS Functions to HP Neoview

SAS/ACCESS Interface to HP Neoview passes the following SAS functions to HP Neoview for processing. Where the HP Neoview function name differs from the SAS function name, the HP Neoview name appears in parentheses. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

ABS	MAX
ARCOS (ACOS)	MIN
ARSIN (ASIN)	MINUTE
ATAN	MOD
ATAN2	MONTH
AVG	QTR
BYTE (CHAR)	REPEAT
CEIL(CEILING)	SECOND

COALESCE	SIGN
COMPRESS (REPLACE)	SIN
COS	SINH
COSH	SQRT
COUNT	STRIP (TRIM)
DAY	SUBSTR
EXP	SUM
FLOOR	TAN
HOUR	TANH
INDEX (LOCATE)	TRANWRD (REPLACE)
LEFT(LTRIM)	TRIMN (RTRIM)
LOG	UPCASE (UPPER)
LOG10	WEEKDAY (DAYOFWEEK)
LOWCASE (LOWER)	YEAR

SQL_FUNCTIONS=ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. (See SQL_FUNCTIONS= on page 185.) Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to HP Neoview. Due to incompatibility in date and time functions between HP Neoview and SAS, HP Neoview might not process them correctly. Check your results to determine whether these functions are working as expected.

DATE (CURRENT_DATE)	ROUND
DATEPART (CAST)	TIME (CURRENT_TIME)
DATETIME (CURRENT_TIMESTAMP)	TIMEPART (CAST)
LENGTH	TODAY (CURRENT_DATE)

Passing Joins to HP Neoview

For a multiple libref join to pass to HP Neoview, all of these components of the LIBNAME statements must match exactly:

- user ID (USER=)
- password (PASSWORD=)
- server (SERVER=)
- port (PORT=)
- data source (DSN=, if specified)
- SQL functions (SQL_FUNCTIONS=)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see “Passing Joins to the DBMS” on page 47.

Bulk Loading and Extracting for HP Neoview

Loading

Overview

Bulk loading is the fastest way to insert large numbers of rows into an HP Neoview table. To use the bulk-load facility, specify BULKLOAD= on page 274YES. The bulk-load facility uses the HP Neoview Transporter with an HP Neoview control file to move data from the client to HP Neoview.

Here are the HP Neoview bulk-load data set options. For detailed information about these options, see Data Set Options for Relational Databases on page 568.

- BL_BADDATA_FILE=
- BL_DATAFILE=
- BL_DELETE_DATAFILE=
- BL_DELIMITER=
- BL_DISCARDS=
- BL_ERRORS=
- BL_FAILEDDATA=
- BL_HOSTNAME=
- BL_NUM_ROW_SEPS=
- BL_PORT=
- BL_RETRIES=
- BL_ROWSETSIZE=
- BL_SYNCHRONOUS= on page 269
- BL_TENACITY=
- BL_TRIGGER=
- BL_TRUNCATE=
- BL_USE_PIPE=
- BULKLOAD=

Examples

This first example shows how you can use a SAS data set, SASFLT.FLT98, to create and load a large HP Neoview table, FLIGHTS98:

```
libname sasflt 'SAS-library';
libname net_air neoview DSN=air2 user=louis
        pwd=fromage schema=FLIGHTS;

proc sql;
create table net_air.flights98
       (bulkload=YES bl_system=FLT0101)
```

```
        as select * from sasflt.flt98;
quit;
```

This next example shows how you can append the SAS data set, SASFLT.FLT98, to the existing HP Neoview table, ALLFLIGHTS. The BL_USE_PIPE=NO option forces SAS/ACCESS Interface to HP Neoview to write data to a flat file, as specified in the BL_DATAFILE= option. Rather than deleting the data file, BL_DELETE_DATAFILE=NO causes the engine to leave it after the load has completed.

```
proc append base=net_air.allflights
 (BULKLOAD=YES
  BL_DATAFILE='/tmp/fltdata.dat'
  BL_USE_PIPE=NO
  BL_DELETE_DATAFILE=NO)
  BL_SYSTEM=FLT0101
data=sasflt.flt98;
run;
```

Extracting

Overview

Bulk extracting is the fastest way to retrieve large numbers of rows from an HP Neoview table. To use the bulk-extract facility, specify BULKEXTRACT=YES. (See BULKEXTRACT= on page 274.) The bulk extract facility uses the HP Neoview Transporter with an HP Neoview control file to move data from the client to HP Neoview into SAS.

Here are the HP Neoview bulk-extract data set options:

- BL_BADDATA_FILE=
- BL_DATAFILE=
- BL_DELETE_DATAFILE=
- BL_DELIMITER=
- BL_FAILEDDATA=
- BL_SYSTEM=
- BL_TRUNCATE=
- BL_USE_PIPE=
- BULKEXTRACT=

Examples

This first example shows how you can read the large HP Neoview table, FLIGHTS98, to create and populate a SAS data set, SASFLT.FLT98:

```
libname sasflt 'SAS-library';
libname net_air neoview DSN=air2 user=louis
        pwd=fromage schema=FLIGHTS;

proc sql;
create table sasflt.flt98
        as select * from net_air.flights98
       (bulkextract=YES bl_system=FLT0101);
quit;
```

This next example shows how you can append the contents of the HP Neoview table, ALLFLIGHTS, to an existing SAS data set, SASFLT.FLT98. The BL_USE_PIPE=NO option forces SAS/ACCESS Interface to HP Neoview to read data from a flat file, as specified in the BL_DATAFILE= option. Rather than deleting the data file, BL_DELETE_DATAFILE=NO causes the engine to leave it after the extract has completed.

```
proc append base=sasflt.flt98
data=net_air.allflights
 (BULKEXTRACT=YES
  BL_DATAFILE='/tmp/fltdata.dat'
  BL_USE_PIPE=NO
  BL_DELETE_DATAFILE=NO);
BL_SYSTEM=FLT0101
run;
```

Naming Conventions for HP Neoview

For general information about this feature, see “SAS Names and Support for DBMS Names” on page 13.

Since SAS 7, most SAS names can be up to 32 characters long. SAS/ACCESS Interface to HP Neoview supports table names and column names that contain up to 32 characters. If DBMS column names are longer than 32 characters, they are truncated to 32 characters. If truncating a column name would result in identical names, SAS generates a unique name by replacing the last character with a number. DBMS table names must be 32 characters or less because SAS does not truncate a longer name. If you already have a table name that is greater than 32 characters, it is recommended that you create a table view.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options determine how SAS/ACCESS Interface to HP Neoview handles case sensitivity. (For information about these options, see “Overview: LIBNAME Statement for Relational Databases ” on page 93.) HP Neoview is not case sensitive by default, and all names default to uppercase.

HP Neoview objects include tables, views, and columns. Follow these naming conventions:

- A name must be from 1 to 128 characters long.
- A name must begin with a letter (A through Z, or a through z). However, if the name appears within double quotation marks, it can start with any character.
- A name cannot begin with an underscore (_). Leading underscores are reserved for system objects.
- Names are not case sensitive. For example, **`CUSTOMER`** and **`Customer`** are the same, but object names are converted to uppercase when they are stored in the HP Neoview database. However, if you enclose a name in quotation marks, it is case sensitive.
- A name cannot be a reserved word in HP Neoview, such as WHERE or VIEW.
- A name cannot be the same as another HP Neoview object that has the same type.

For more information, see your *HP Neoview SQL Reference Manual*.

Data Types for HP Neoview

Overview

Every column in a table has a name and a data type. The data type tells HP Neoview how much physical storage to set aside for the column and the form in which the data is stored. This section includes information about HP Neoview data types, null and default values, and data conversions.

For more information about HP Neoview data types and to determine which data types are available for your version of HP Neoview, see your *HP Neoview SQL Reference Manual*.

SAS/ACCESS Interface to HP Neoview does not directly support HP Neoview INTERVAL types. Any columns using these types are read into SAS as character strings.

String Data

CHAR(*n*)
: specifies a fixed-length column for character string data. The maximum length is 32,708 characters.

VARCHAR(*n*)
: specifies a varying-length column for character string data. The maximum length is 32,708 characters.

Numeric Data

LARGEINT
: specifies a big integer. Values in a column of this type can range from –9223372036854775808 to +9223372036854775807.

SMALLINT
: specifies a small integer. Values in a column of this type can range from –32768 through +32767.

INTEGER
: specifies a large integer. Values in a column of this type can range from –2147483648 through +2147483647.

DOUBLE
: specifies a floating-point number that is 64 bits long. Values in a column of this type can range from –1.79769E+308 to –2.225E-307 or +2.225E-307 to +1.79769E+308, or they can be 0. This data type is stored the same way that SAS stores its numeric data type. Therefore, numeric columns of this type require the least processing when SAS accesses them.

FLOAT
: specifies an approximate numeric column. The column stores floating-point numbers and designates from 1 through 52 bits of precision. Values in a column of this type can range from +/–2.2250738585072014e-308 to +/–1.7976931348623157e+308 stored in 8 bytes.

REAL
specifies a floating-point number that is 32 bits long. Values in a column of this type can range from approximately –3.4E38 to –1.17E-38 and +1.17E-38 to +3.4E38.

DECIMAL | DEC | NUMERIC
specifies a fixed-point decimal number. The precision and scale of the number determines the position of the decimal point. The numbers to the right of the decimal point are the scale, and the scale cannot be negative or greater than the precision. The maximum precision is 38 digits.

Date, Time, and Timestamp Data

SQL date and time data types are collectively called datetime values. The SQL data types for dates, times, and timestamps are listed here. Be aware that columns of these data types can contain data values that are out of range for SAS.

DATE
specifies date values. The range is 01-01-0001 to 12-31-9999. The default format *YYYY-MM-DD* (for example, 1961–06–13). HP Neoview supports many other formats for entering date data. For more information, see your *HP Neoview SQL Reference Manual*.

TIME
specifies time values in hours, minutes, and seconds to six decimal positions: hh:mm:ss[.nnnnnn]. The range is 00:00:00.000000 to 23:59:59.999999. However, because the HP Neoview engine uses the ODBC-style interface to communicate with the HP Neoview server, fractional seconds are lost in the transfer of data from server to client.

TIMESTAMP
combines a date and time in the default format of *yyyy-mm-dd hh:mm:ss[.nnnnnn]*. For example, a timestamp for precisely 2:25 p.m. on January 25, 1991, would be 1991-01-25-14.25.00.000000. Values in a column of this type have the same ranges as described for DATE and TIME.

HP Neoview Null Values

HP Neoview has a special value called NULL. An HP Neoview NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads an HP Neoview NULL value, it interprets it as a SAS missing value.

You can define a column in an HP Neoview table so that it requires data. To do this in SQL, specify a column as NOT NULL. This tells SQL to allow a row to be added to a table only if a value exists for the field. For example, NOT NULL assigned to the CUSTOMER field in the SASDEMO.CUSTOMER table does not allow a row to be added unless there is a value for CUSTOMER. When creating an HP Neoview table with SAS/ACCESS, you can use the DBNULL= on page 294 data set option to indicate whether NULL is a valid value for specified columns.

For more information about how SAS handles NULL values, see “Potential Result Set Differences When Processing Null Data ” on page 35.

To control how the DBMS handles SAS missing character values, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to HP Neoview assigns to SAS variables when using the LIBNAME statement on page 93 to read from an HP Neoview table. These default formats are based on HP Neoview column attributes.

Table 19.3 *LIBNAME Statement: Default SAS Formats for HP Neoview Data Types*

HP Neoview Data Type	SAS Data Type	Default SAS Format
CHAR(*n*)	character	$*n*.
VARCHAR(*n*)	character	$*n*.
LONGVARCHAR(*n*)	character	$*n*.
DECIMAL(*p,s*)	numeric	*m.n*
NUMERIC(*p,s*)	numeric	p,s
SMALLINT	numeric	6.
INTEGER	numeric	11.
REAL	numeric	none
FLOAT(*p*)	numeric	*p*
DOUBLE	numeric	none
LARGEINT	numeric	20.
DATE	numeric	DATE9.
TIME	numeric	TIME8.
TIMESTAMP	numeric	DATETIME*25.6*

The following table shows the default HP Neoview data types that SAS/ACCESS assigns to SAS variable formats during output operations when you use the LIBNAME statement.

Table 19.4 *LIBNAME Statement: Default HP Neoview Data Types for SAS Variable Formats*

SAS Variable Format	HP Neoview Data Type
m.n	DECIMAL (*m,n*)*
other numerics	DOUBLE

SAS Variable Format	HP Neoview Data Type
$*n*.	VARCHAR(*n*)*
datetime formats	TIMESTAMP
date formats	DATE
time formats	TIME

* *n* in HP Neoview data types is equivalent to *w* in SAS formats.

Chapter 20

SAS/ACCESS Interface to Informix

Introduction to SAS/ACCESS Interface to Informix

Overview

This section describes SAS/ACCESS Interface to Informix. See “SAS/ACCESS Interface to Informix: Supported Features” on page 82 for a list of SAS/ACCESS features that are available in this interface. For background information about Informix, see “Overview: Informix Servers ” on page 591.

Default Environment

When you access Informix tables by using SAS/ACCESS Interface to Informix, the default Informix read isolation level is set for committed reads, and SAS spooling is on. Committed reads enable you to read rows unless another user or process is updating the rows. Reading in this manner does not lock the rows. SAS spooling guarantees that you obtain identical data each time you re-read a row because SAS buffers the rows after you read them the first time. This default environment is suitable for most users. If this default environment is unsuitable for your needs, see “Locking in the Informix Interface” on page 586.

To see the SQL statements that SAS issues to the Informix server, include the SASTRACE= option in your code:

```
option sastrace=',,,d';
```

If you use quotation marks in your Informix SQL statements, set your DELIMIDENT= environment variable to DELIMIDENT=YES or Informix might reject your statements. Because some SAS options that preserve case generate SQL statements that contain quotation marks, you should set DELIMIDENT=YES in your environment.

LIBNAME Statement Specifics for Informix

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to Informix supports and includes an example. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing Informix.

LIBNAME *libref* **informix** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
: specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`informix`
: specifies the SAS/ACCESS engine name for the Informix interface.

connection-options
: provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. Here is how these options are defined.

USER=<'>*Informix-user-name*<'>
: specifies the Informix user name that you use to connect to the database that contains the tables and views that you want to access. If you omit the USER= option, your operating environment account name is used, if applicable to your operating environment.

USING=<'>*Informix-password*<'>
: specifies the password that is associated with the Informix user. If you omit the password, Informix uses the password in the **`/etc/password`** file.

USING= can also be specified with the PASSWORD= and PWD= aliases.

SERVER=<'>*ODBC-data-source*<'>
: specifies the ODBC data source to which you want to connect. An error occurs if the SERVER= option is not set. For UNIX platforms, you must configure the data source by modifying the odbc.ini file. See your ODBC driver documentation for details.

For the SAS/ACCESS 9 Interface to Informix, the Informix ODBC Driver API is used to connect to Informix, and connection options have changed accordingly. The DATABASE= option from the SAS 8 version of SAS/ACCESS was removed. If you need to specify a database, set it in the odbc.ini file. For SERVER= options, instead of specifying the server name, as in SAS 8, specify an ODBC data source name. You can also use a user ID and password with SERVER=.

DBDATASRC=<'>*database-data-source*<'>
: environment variable that lets you set a default data source. This value is used if you do not specify a SERVER= connection option.

LIBNAME-options
: define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to Informix, with the applicable default values. For more detail about these options, see "LIBNAME Options for Relational Databases" on page 99.

Table 20.1 *SAS/ACCESS LIBNAME Options for Informix*

Option	Default Value
ACCESS=	none
AUTHDOMAIN=	none

Option	Default Value
AUTOCOMMIT=	YES
CONNECTION=	SHAREDREAD
CONNECTION_GROUP=	none
DBCOMMIT=	1000 when inserting rows; 0 when updating rows
DBCONINIT=	none
DBCONTERM=	none
DBCREATE_TABLE_OPTS=	none
DBGEN_NAME=	DBMS
DBINDEX= on page 129	NO
DBLIBINIT=	none
DBLIBTERM=	none
DBNULLKEYS=	NO
DBPROMPT=	NO
DBSASLABEL=	COMPAT
DBSLICEPARM=	THREADED_APPS,2 or 3
DEFER=	NO
DIRECT_EXE=	none
DIRECT_SQL=	YES
LOCKTABLE=	no locking
LOCKTIME=	none
LOCKWAIT=	not set
MULTI_DATASRC_OPT=	NONE
PRESERVE_COL_NAMES=	NO
PRESERVE_TAB_NAMES=	NO
READ_ISOLATION_LEVEL=	COMMITTED READ (see "Locking in the Informix Interface" on page 586)

Option	Default Value
REREAD_EXPOSURE=	NO
SCHEMA=	your user name
SPOOL=	YES
SQL_FUNCTIONS=	none
UTILCONN_TRANSIENT=	NO

Informix LIBNAME Statement Example

In this example, the libref MYDBLIB uses the Informix interface to connect to an Informix database:

```
libname mydblib informix user=testuser using=testpass server=testdsn;
```

In this example USER=, USING=, and SERVER= are connection options.

Data Set Options for Informix

All SAS/ACCESS data set options in this table are supported for Informix. Default values are provided where applicable. For general information about this feature, see “Overview” on page 207.

Table 20.2 *SAS/ACCESS Data Set Options for Informix*

Option	Default Value
DBCOMMIT=	LIBNAME option setting
DBCONDITION=	none
DBCREATE_TABLE_OPTS=	LIBNAME option setting
DBFORCE=	NO
DBGEN_NAME=	DBMS
DBINDEX=	LIBNAME option setting
DBKEY=	none
DBLABEL=	NO
DBMASTER=	none
DBNULL=	_ALL_=YES

Option	Default Value
DBNULLKEYS=	LIBNAME option setting
DBSASLABEL=	COMPAT
DBSASTYPE=	see “Data Types for Informix ” on page 588
DBSLICE=	none
DBSLICEPARM=	THREADED_APPS,2 or 3
DBSLICEPARM=	see “Data Types for Informix ” on page 588
ERRLIMIT=	1
LOCKTABLE=	LIBNAME option setting
NULLCHAR=	SAS
NULLCHARVAL=	a blank character
PRESERVE_COL_NAMES=	LIBNAME option setting
SASDATEFMT=	DATETIME
SCHEMA=	LIBNAME option setting

SQL Pass-Through Facility Specifics for Informix

Key Information

For general information about this feature, see “About SQL Procedure Interactions ” on page 401. Informix examples are available.

Here are the SQL pass-through facility specifics for the Informix interface.

- The *dbms-name* is **informix**.
- The CONNECT statement is optional when you are connecting to an Informix database if the DBDATASRC environment variable has been set. When you omit a CONNECT statement, an implicit connection is performed when the first EXECUTE statement or CONNECTION TO component is passed to the DBMS.
- You can connect to only one Informix database at a time. However, you can specify multiple CONNECT statements if they all connect to the same Informix database. If you use multiple connections, you must use an *alias* to identify the different connections. If you omit an alias, **informix** is automatically used.
- The CONNECT statement *database-connection-arguments* are identical to its connection-options. (See “connection-options” on page 577.)

- If you use quotation marks in your Informix pass-through statements, your DELIMIDENT= environment variable must be set to DELIMIDENT=YES, or your statements are rejected by Informix.

Stored Procedures and the SQL Pass-Through Facility

The SQL pass-through facility recognizes two types of stored procedures in Informix that perform only database functions. The methods for executing the two types of stored procedures are different.

- Procedures that return no values to the calling application:

  Stored procedures that do not return values can be executed directly by using the Informix SQL EXECUTE statement. Stored procedure execution is initiated with the Informix EXECUTE PROCEDURE statement. The following example executes the stored procedure **`make_table.`** The stored procedure has no input parameters and returns no values.

  ```
  execute (execute procedure make_table())
     by informix;
  ```

- Procedures that return values to the calling application:

  Stored procedures that return values must be executed by using the PROC SQL SELECT statement with a CONNECTION TO component. This example executes the stored procedure **`read_address`**, which has one parameter, **`"Putnum"`**.

  The values that **`read_address`** returns serve as the contents of a virtual table for the PROC SQL SELECT statement.

  ```
  select * from connection to informix
          (execute procedure read_address ("Putnum"));
  ```

  For example, when you try to execute a stored procedure that returns values from a PROC SQL EXECUTE statement, you receive this error message:

  ```
  execute (execute procedure read_address
     ("Putnum")) by informix;

  ERROR: Informix EXECUTE Error: Procedure
     (read_address) returns too many values.
  ```

Command Restrictions for the SQL Pass-Through Facility

Informix SQL contains extensions to the ANSI-89 standards. Some of these extensions, such as LOAD FROM and UNLOAD TO, are restricted from use by any applications other than the Informix DB-Access product. Specifying these extensions in the PROC SQL EXECUTE statement generates this error:

```
-201
A syntax error has occurred
```

Examples

This example connects to Informix by using data source **`testdsn`**:

```
proc sql;
   connect to informix
   (user=SCOTT password=TIGER server=testdsn);
```

You can use the DBDATASRC environment variable to set the default data source.

This next example grants UPDATE and INSERT authority to user **gomez** on the Informix ORDERS table. Because the CONNECT statement is omitted, an implicit connection is made. The connection uses a default value of **informix** as the connection alias and default values for the SERVER= argument.

```
proc sql;
   execute (grant update, insert on ORDERS to gomez) by informix;
quit;
```

This example connects to Informix and drops (removes) the table TempData from the database. The alias Temp5 that is specified in the CONNECT statement is used in the EXECUTE statement's BY clause.

```
proc sql;
   connect to informix as temp5
   (server=testdsn);
   execute (drop table tempdata) by temp5;
   disconnect from temp5;
quit;
```

This example sends an SQL query, shown with highlighting, to the database for processing. The results from the SQL query serve as a virtual table for the PROC SQL FROM clause. In this example DBCON is a connection alias.

```
proc sql;
connect to informix as dbcon
   (user=testuser using=testpass
    server=testdsn);

select *
   from connection to dbcon
      (select empid, lastname, firstname,
      hiredate, salary
          from employees
          where hiredate>='31JAN88');

disconnect from dbcon;
quit;
```

This next example gives the previous query a name and stores it as the PROC SQL view Samples.Hires88. The CREATE VIEW statement appears in highlighting.

```
libname samples 'SAS-library';

proc sql;
connect to informix as mycon
   (user=testuser using=testpass
     server=testdsn);

create view samples.hires88 as
  select *
    from connection to mycon
      (select empid, lastname, firstname,
       hiredate, salary from employees
```

```
      where hiredate>='31JAN88');

disconnect from mycon;
quit;
```

This example connects to Informix and executes the stored procedure **testproc**. The **select *** clause displays the results from the stored procedure.

```
proc sql;
   connect to informix as mydb
      (server=testdsn);
   select * from connection to mydb
      (execute procedure testproc('123456'));
   disconnect from mydb;
quit;
```

Autopartitioning Scheme for Informix

Overview

Autopartitioning for SAS/ACCESS Interface to Informix is a modulo (MOD) function method. For general information about this feature, see "Autopartitioning Techniques in SAS/ACCESS" on page 63.

Autopartitioning Restrictions

SAS/ACCESS Interface to Informix places additional restrictions on the columns that you can use for the partitioning column during the autopartitioning phase. Here is how columns are partitioned.

- INTEGER
- SMALLINT
- BIT
- TINYINT
- You can also use DECIMALS with 0-scale columns as the partitioning column.
- Nullable columns are the least preferable.

Using WHERE Clauses

Autopartitioning does not select a column to be the partitioning column if it appears in a SAS WHERE clause. For example, the following DATA step cannot use a threaded Read to retrieve the data because all numeric columns in the table are in the WHERE clause:

```
data work.locemp;
set trlib.MYEMPS;
where EMPNUM<=30 and ISTENURE=0 and
 SALARY<=35000 and NUMCLASS>2;
run;
```

Using DBSLICEPARM=

Although SAS/ACCESS Interface to Informix defaults to three threads when you use autopartitioning, do not specify a maximum number of threads in DBSLICEPARM= LIBNAME option on page 139 to use for the threaded Read.

This example shows how to use of DBSLICEPARM= with the maximum number of threads set to five:

```
libname x informix user=dbitest using=dbigrp1 server=odbc15;
proc print data=x.dept(dbsliceparm=(ALL,5));
run;
```

Using DBSLICE=

You can achieve the best possible performance when using threaded Reads by specifying the DBSLICE= data set option on page 299 for Informix in your SAS operation. This example shows how to use it.

```
libname x informix user=dbitest using=dbigrp1 server=odbc15;
data xottest;
set x.invoice(dbslice=("amtbilled<10000000" "amtbilled>=10000000"));
run;
```

Temporary Table Support for Informix

Overview

For general information about this feature, see "Temporary Table Support for SAS/ACCESS " on page 42.

Establishing a Temporary Table

To establish the DBMS connection to support the creation and use of temporary tables, issue a LIBNAME statement with the connection options CONNECTION_GROUP=*connection-group* and CONNECTION=GLOBAL. This LIBNAME statement is required even if you connect to the database using the pass-through facility CONNECT statement, because it establishes a connection group.

For every new PROC SQL step or LIBNAME statement, you must reissue a CONNECT statement. In the CONNECT statement, set the CONNECTION_GROUP= option to the same value so that the connection can be reused.

Terminating a Temporary Table

To terminate a temporary table, disassociate the libref by issuing this statement:

```
libname libref clear;
```

Example

In this pass-through example, joins are pushed to Informix:

```
libname x informix user=tester using=xxxxx server=dsn_name
                   connection=global connection_group=mygroup;

proc sql;
   connect to informix (user=tester using=xxxxx server=dsn_name
                        connection=global connection_group=mygroup);
   execute (select * from t1 where (id >100)
            into scratch scr1 ) by informix;
   create table count2 as select * from connection to informix
          (select count(*) as totrec from scr1);
quit;

proc print data=count2;
run;

proc sql;
   connect to informix (user=tester using=xxxxx server=dsn_name
                        connection=global connection_group=mygroup);
   execute(select t2.fname, t2.lname, scr1.dept from t2, scr1 where
          (t2.id = scr1.id) into scratch t3 ) by informix;
quit;

libname x clear;  /* connection closed, temp table closed */
```

Passing SAS Functions to Informix

SAS/ACCESS Interface to Informix passes the following SAS functions to Informix for processing if the DBMS driver or client that you are using supports this function. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

ABS	LOG
ARCOS	LOG10
ARSIN	MAX
ATAN	MDY
ATAN2	MIN
AVG	MINUTE
COS	MONTH
COUNT	SECOND
DATE	SIN
DAY	SQRT
DTEXTDAY	STRIP
DTEXTMONTH	SUM
DTEXTYEAR	TAN
DTEXTWEEKDAY	TODAY

EXP	WEEKDAY
HOUR	YEAR
INT	

SQL_FUNCTIONS= on page 185ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to Informix. Due to incompatibility in date and time functions between Informix and SAS, the Informix server might not process them correctly. Check your results to determine whether these functions are working as expected.

DATEPART TIMEPART

Passing Joins to Informix

For a multiple libref join to pass to Informix, all of these components of the LIBNAME statements must match exactly:

user ID ()

password ()

server ()

Due to an Informix database limitation, the maximum number of tables that you can specify to perform a join is 22. An error message appears if you specify more than 22.

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Locking in the Informix Interface

In most cases, SAS spooling is on by default for the Informix interface and provides the data consistency that you need.

To control how the Informix interface handles locking, you can use the "READ_ISOLATION_LEVEL= LIBNAME Option" on page 175. Here are the valid values.

COMMITTED_READ
: retrieves only committed rows. No locks are acquired, and rows can be locked exclusively for update by other users or processes. This is the default setting.

REPEATABLE_READ
: gives you a shared lock on every row that is selected during the transaction. Other users or processes can also acquire a shared lock, but no other process can modify any row that is selected by your transaction. If you repeat the query during the transaction, you re-read the same information. The shared locks are released only when the transaction commits or rolls back. Another process cannot update or delete a row that is accessed by using a repeatable read.

DIRTY_READ
: retrieves committed and uncommitted rows that might include phantom rows, which are rows that are created or modified by another user or process that might

subsequently be rolled back. This type of read is most appropriate for tables that are not frequently updated.

CURSOR_STABILITY
: gives you a shared lock on the selected row. Another user or process can acquire a shared lock on the same row, but no process can acquire an exclusive lock to modify data in the row. When you retrieve another row or close the cursor, the shared lock is released.

If you set READ_ISOLATION_LEVEL= to REPEATABLE_READ or CURSOR_STABILITY, it is recommended that you assign a separate libref and clear that libref when you finish working with the tables. This technique minimizes the negative performance impact on other users that occurs when you lock the tables. To clear the libref, include this code:

```
libname libref clear;
```

For current Informix releases, READ_ISOLATION_LEVEL= is valid only when transaction logging is enabled. If transaction logging is not enabled, an error is generated when you use this option. Also, locks placed when READ_ISOLATION_LEVEL= REPEATABLE READ or CURSOR_STABILITY are *not freed* until the libref is cleared.

To see the SQL locking statements that SAS issues to the Informix server, include in your code the "SASTRACE= System Option" on page 385.

```
option sastrace=',,,d';
```

For more details about Informix locking, see your Informix documentation.

Naming Conventions for Informix

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= LIBNAME options determine how SAS/ACCESS Interface to Informix handles case sensitivity, spaces, and special characters. For information about these options, see "Overview: LIBNAME Statement for Relational Databases " on page 93.

Informix objects include tables and columns. They follow these naming conventions.

- Although table and column names must be from 1 to 32 characters, the limitation on some Informix servers might be lower.
- Table and column names must begin with a letter or an underscore (_) that is followed by letters, numbers, or underscores. Special characters are not supported. However, if you enclose a name in quotation marks and PRESERVE_TAB_NAMES=YES (when applicable), it can begin with any character.

Because several problems were found in the Informix ODBC driver that result from using uppercase or mixed case, Informix encourages users to use lowercase for table and column names. Informix currently has no schedule for fixing these known problems.

Data Types for Informix

Overview

Every column in a table has a name and a data type. The data type tells Informix how much physical storage to set aside for the column and the form in which the data is stored. This section includes information about Informix data types, null values, and data conversions.

Character Data

CHAR(*n*), NCHAR(*n*)
: contains character string data from 1 to 32,767 characters in length and can include tabs and spaces.

VARCHAR(*m,n*), NVARCHAR(*m,n*)
: contains character string data from 1 to 255 characters in length.

TEXT
: contains unlimited text data, depending on memory capacity.

BYTE
: contains binary data of variable length.

Numeric Data

DECIMAL, MONEY, NUMERIC
: contains numeric data with definable scale and precision. The amount of storage that is allocated depends on the size of the number.

FLOAT, DOUBLE PRECISION
: contains double-precision numeric data up to 8 bytes.

INTEGER
: contains an integer up to 32 bits (from -2^{31} to $2^{31}-1$).

REAL, SMALLFLOAT
: contains single-precision, floating-point numbers up to 4 bytes.

SERIAL
: stores sequential integers up to 32 bits.

SMALLINT
: contains integers up to 2 bytes.

INT8
: contains an integer up to 64 bits ($-2^{(63-1)}$ to $2^{(63-1)}$).

SERIAL8
: contains sequential integers up to 64 bits.

When the length value of INT8 or SERIAL8 is greater than 15, the last few digits currently do not display correctly due to a display limitation.

Date, Time, and Interval Data

DATE
: contains a calendar date in the form of a signed integer value.

DATETIME
: contains a calendar date and time of day stored in 2 to 11 bytes, depending on precision.

  When the DATETIME column is in an uncommon format (for example, DATETIME MINUTE TO MINUTE or DATETIME SECOND TO SECOND), the date and time values might not be displayed correctly.

INTERVAL
: contains a span of time stored in 2 to 12 bytes, depending on precision.

Informix Null Values

Informix has a special value that is called NULL. An Informix NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads an Informix NULL value, it interprets it as a SAS missing value.

If you do not indicate a default value for an Informix column, the default value is NULL. You can specify the keywords NOT NULL after the data type of the column when you create an Informix table to prevent NULL values from being stored in the column. When creating an Informix table with SAS/ACCESS, you can use the DBNULL= on page 294 data set option to indicate whether NULL is a valid value for specified columns.

For more information about how SAS handles NULL values, see “Potential Result Set Differences When Processing Null Data ” on page 35.

To control how the DBMS handles SAS missing character values, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to Informix assigns to SAS variables when using the LIBNAME statement to read from an Infomix table. These default formats are based on Infomix column attributes. To override these default data types, use the DBTYPE= on page 303 data set option on a specific data set.

***Table 20.3** LIBNAME Statement: Default SAS Formats for Informix Data Types*

Informix Column Type	Default SAS Format
CHAR(*n*)	$*n*
DATE	DATE9.
DATETIME***	DATETIME24.5
DECIMAL	*m*+2.*n*
DOUBLE PRECISION	none

Informix Column Type	Default SAS Format
FLOAT	none
INTEGER	none
INT8**	none
INTERVAL	$*n*
MONEY	none
NCHAR(*n*)	$*n* NLS support required
NUMERIC	none
NVARCHAR(*m,n*)*	$*m* NLS support required
REAL	none
SERIAL	none
SERIAL8**	none
SMALLFLOAT	none
SMALLINT	none
TEXT*	$*n*
VARCHAR(*m,n*)*	$*m*

* Only supported by Informix online databases.

** The precision of an INT8 or SERIAL8 is 15 digits.

*** If the Informix field qualifier specifies either HOUR, MINUTE, SECOND, or FRACTION as the largest unit, the value is converted to a SAS TIME value. All other values (such as YEAR , MONTH , or DAY) are converted to a SAS DATETIME value.

The following table shows the default Informix data types that SAS/ACCESS applies to SAS variable formats during output operations when you use the LIBNAME statement.

Table 20.4 *LIBNAME Statement: Default Informix Data Types for SAS Variable Formats*

SAS Variable Format	Informix Data Type
$*w*.	CHAR(w).
w. with SAS format name of NULL	DOUBLE
w.d with SAS format name of NULL	DOUBLE
all other numerics	DOUBLE

SAS Variable Format	Informix Data Type
datetime*w.d*	DATETIME YEAR TO FRACTION(5)
date*w.*	DATE
time.	DATETIME HOUR TO SECOND

SQL Pass-Through Facility Data Conversions

The SQL pass-through facility on page 403 uses the same default conversion formats as the LIBNAME statement. For conversion tables, see "LIBNAME Statement Data Conversions" on page 589.

Overview: Informix Servers

Informix Database Servers

There are two types of Informix database servers, the Informix OnLline and Informix SE servers. *Informix OnLine database servers* can support many users and provide tools that ensure high availability, high reliability, and that support critical applications. *Informix SE database servers* are designed to manage relatively small databases that individuals use privately or that a small number of users share.

Using the DBDATASRC Environment Variable

The SQL pass-through facility supports the DBDATASRC environment variable, which is an extension to the Informix environment variable. If you set DBDATASRC, you can omit the CONNECT statement. The value of DBDATASRC is used instead of the SERVER= argument in the CONNECT statement. The syntax for setting DBDATASRC is like the syntax of the SERVER= argument:

Bourne shell:

```
export DBDATABASE='testdsn'
```

C shell:

```
setenv DBDATASRC testdsn
```

If you set DBDATASRC, you can issue a PROC SQL SELECT or EXECUTE statement without first connecting to Informix with the CONNECT statement.

If you omit the CONNECT statement, an implicit connection is performed when the SELECT or EXECUTE statement is passed to Informix.

If you create an SQL view without an explicit CONNECT statement, the view can dynamically connect to different databases, depending on the value of the DBDATASRC environment variable.

Using Fully Qualified Table Names

Informix supports a connection to only one database. If you have data that spans multiple databases, you must use fully qualified table names to work within the Informix single-connection constraints.

In this example, the tables Tab1 and Tab2 reside in different databases, MyDB1 and MyDB2, respectively.

```
proc sql;
   connect to informix
   (server=testdsn);

   create view tab1v as
      select * from connection
      to informix
         (select * from mydb1.tab1);

   create view tab2v as
      select * from connection
      to informix
         (select * from mydb2.tab2);
quit;

data getboth;
   merge tab1v tab2v;
   by common;
run;
```

Because the tables reside in separate databases, you cannot connect to each database with a PROC SQL CONNECT statement and then retrieve the data in a single step. Using the fully qualified table name (that is, *database.table*) enables you to use any Informix database in the CONNECT statement and access Informix tables in the *same or different* databases in a single SAS procedure or DATA step.

Chapter 21

SAS/ACCESS Interface to Microsoft SQL Server

Introduction to SAS/ACCESS Interface to Microsoft SQL Server

This section describes SAS/ACCESS Interface to Microsoft SQL Server. For a list of SAS/ACCESS features that are available for this interface, see "SAS/ACCESS Interface to Microsoft SQL Server: Supported Features" on page 83.

SAS/ACCESS Interface to Microsoft SQL Server has been tested and certified against Data Direct Technologies Connect ODBC and Data Direct SequeLink ODBC products.

LIBNAME Statement Specifics for Microsoft SQL Server

Overview

This section describes the LIBNAME statement as supported in SAS/ACCESS Interface to Microsoft SQL Server and includes examples. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing Microsoft SQL Server.

LIBNAME *libref* **sqlsvr** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`sqlsvr`
specifies the SAS/ACCESS engine name for the Microsoft SQL Server interface.

connection-options
provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. When you use the LIBNAME statement, you can connect to Microsoft SQL Server in many different ways. Specify *only one* of these methods for each connection because they are mutually exclusive.

- USER=, PASSWORD=, and DATASRC=
- COMPLETE=
- NOPROMPT=
- PROMPT=
- REQUIRED=

Here is how these options are defined.

USER=<'>*user-name*<'>
lets you connect to Microsoft SQL Server with a user ID that is different from the default ID. USER= is optional. UID= is an alias for this option.

PASSWORD=<'>*password*<'>
specifies the Microsoft SQL Server password that is associated with your user ID. PASSWORD= is optional. PWD= is an alias for this option.

DATASRC=<'>*SQL-Server-data-source*<'>
specifies the Microsoft SQL Server data source to which you want to connect. For UNIX platforms, data sources must be configured by modifying the .ODBC.ini file. DSN= is an alias for this option that indicates that the connection is attempted using the ODBC SQLConnect API, which requires a data source name. You can also use a user ID and password with DSN=. This API is guaranteed to be present in all drivers.

Alias: DATABASE=, DB=, DSN=

COMPLETE=<'>*SQL-Server-connection-options*<'>
: specifies connection options for your data source or database. Separate multiple options with a semicolon. When connection succeeds, the complete connection string is returned in the SYSDBMSG macro variable. If you do not specify enough correct connection options, you are prompted with a dialog box that displays the values from the COMPLETE= connection string. You can edit any field before you connect to the data source. See your driver documentation for more details.

NOPROMPT=<'>*SQL-Server-connection-options*<'>
: specifies connection options for your data source or database. Separate multiple options with a semicolon. If you do not specify enough correct connection options, an error is returned. No dialog box is displayed to help you with the connection string.

PROMPT=<'> *SQL-Server-connection-options*<'>
: specifies connection options for your data source or database. Separate multiple options with a semicolon. When connection succeeds, the complete connection string is returned in the SYSDBMSG macro variable. PROMPT= does not immediately try to connect to the DBMS. Instead, it displays a dialog box that contains the values that you entered in the PROMPT= connection string. You can edit values or enter additional values in any field before you connect to the data source.

REQUIRED=<'>*SQL-Server-connection-options*<'>
: specifies connection options for your data source or database. Separate multiple options with a semicolon. When connection succeeds, the complete connection string is returned in the SYSDBMSG macro variable. If you do not specify enough correct connection options, a dialog box prompts you for the connection options. REQUIRED= lets you modify only required fields in the dialog box.

These Microsoft SQL Server connection options are not supported on UNIX.

- BULKCOPY=
- COMPLETE=
- PROMPT=
- REQUIRED=

LIBNAME-options
: define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to Microsoft SQL Server, with the applicable default values. For more detail about these options, see "LIBNAME Options for Relational Databases" on page 99.

Table 21.1 *SAS/ACCESS LIBNAME Options for Microsoft SQL Server*

Option	Default Value
ACCESS= on page 99	none
AUTHDOMAIN= on page 102	none
AUTOCOMMIT= on page 103	varies with transaction type
BL_LOG= on page 105	none

Option	Default Value
CONNECTION= on page 113	data-source specific
CONNECTION_GROUP= on page 118	none
CURSOR_TYPE= on page 119	DYNAMIC
DBCOMMIT= on page 124	1000 when inserting rows; 0 when updating rows
DBCONINIT= on page 125	none
DBCONTERM= on page 126	none
DBCREATE_TABLE_OPTS= on page 128	none
DBGEN_NAME= on page 128	DBMS
DBINDEX= on page 129	YES
DBLIBINIT= on page 130	none
DBLIBTERM= on page 131	none
DBMAX_TEXT= on page 133	1024
DBMSTEMP= on page 133	NO
DBNULLKEYS= on page 135	YES
DBPROMPT= on page 136	NO
DBSLICEPARM= on page 139	THREADED_APPS,2 or 3
DEFER= on page 141	NO
DELETE_MUTL_ROWS= on page 143	NO
DIRECT_EXE= on page 144	none
DIRECT_SQL= on page 145	YES
IGNORE_READ_ONLY_COLUMNS= on page 150	NO
INSERT_SQL= on page 152	YES
INSERTBUFF= on page 153	1
KEYSET_SIZE= on page 155	0
MUTLI_DATASRC_OPT= on page 160	NONE

Option	Default Value
PRESERVE_COL_NAMES= on page 167	see "Naming Conventions for Microsoft SQL Server " on page 605
PRESERVE_TAB_NAMES= on page 168	see "Naming Conventions for Microsoft SQL Server " on page 605
QUALIFIER= on page 170	none
QUERY_TIMEOUT= on page 172	0
QUOTE_CHAR= on page 173	none
READBUFF= on page 174	0
READ_ISOLATION_LEVEL= on page 175	RC (see "Locking in the Microsoft SQL Server Interface" on page 603)
READ_LOCK_TYPE= on page 176	ROW
REREAD_EXPOSURE= on page 179	NO
SCHEMA= on page 180	none
SPOOL= on page 184	YES
STRINGDATES= on page 190	NO
TRACE= on page 195	NO
TRACEFILE= on page 196	none
UPDATE_ISOLATION_LEVEL= on page 196	RC (see "Locking in the Microsoft SQL Server Interface" on page 603)
UDPATE_LOCK_TYPE= on page 197	ROW
UPDATE_MULT_ROWS= on page 199	NO
UPDATE_SQL= on page 200	driver-specific
USE_ODBC_CL= on page 202	NO
UTILCONN_TRANSIENT= on page 202	NO

Microsoft SQL Server LIBNAME Statement Examples

In following example, USER= and PASSWORD= are connection options.

```
libname mydblib sqlsvr user=testuser password=testpass;
```

In the following example, the libref MYDBLIB connects to a Microsoft SQL Server database using the NOPROMPT= option.

```
libname mydblib sqlsvr
   noprompt="uid=testuser;
   pwd=testpass;
   dsn=sqlservr;"
   stringdates=yes;

proc print data=mydblib.customers;
   where state='CA';
run;
```

Data Set Options for Microsoft SQL Server

All SAS/ACCESS data set options in this table are supported for Microsoft SQL Server. Default values are provided where applicable. For general information about this feature, see “Overview” on page 207.

Table 21.2 *SAS/ACCESS Data Set Options for Microsoft SQL Server*

Option	Default Value
CURSOR_TYPE= on page 278	LIBNAME option setting
DBCOMMIT= on page 280	LIBNAME option setting
DBCONDITION= on page 281	none
DBCREATE_TABLE_OPTS= on page 285	LIBNAME option setting
DBFORCE= on page 286	NO
DBGEN_NAME= on page 287	DBMS
DBINDEX= on page 288	LIBNAME option setting
DBKEY= on page 290	none
DBLABEL= on page 291	NO
DBMASTER= on page 293	none
DBMAX_TEXT= on page 293	1024
DBNULL= on page 294	YES
DBNULLKEYS= on page 295	LIBNAME option setting
DBPROMPT= on page 296	LIBNAME option setting
DBSASLABEL= on page 297	COMPAT

Option	Default Value
DBSASTYPE= on page 137	see "Data Types for Microsoft SQL Server " on page 605
DBSLICE= on page 299	none
DBSLICEPARM= on page 301	THREADED_APPS,2 or 3
DBTYPE= on page 303	see "Data Types for Microsoft SQL Server " on page 605
ERRLIMIT= on page 308	1
IGNORE_READ_ONLY_COLUMNS= on page 311	NO
INSERT_SQL= on page 313	LIBNAME option setting
INSERTBUFF= on page 314	LIBNAME option setting
KEYSET_SIZE= on page 315	LIBNAME option setting
NULLCHAR= on page 329	SAS
NULLCHARVAL= on page 330	a blank character
PRESERVE_COL_NAMES= on page 336	LIBNAME option setting
QUALIFIER= on page 338	LIBNAME option setting
QUERY_TIMEOUT= on page 339	LIBNAME option setting
READBUFF= on page 339	LIBNAME option setting
READ_ISOLATION_LEVEL= on page 340	LIBNAME option setting
READ_LOCK_TYPE= on page 341	LIBNAME option setting
SASDATEFMT= on page 342	none
SCHEMA= on page 344	LIBNAME option setting
UPDATE_ISOLATION_LEVEL= on page 371	LIBNAME option setting
UPDATE_LOCK_TYPE= on page 371	LIBNAME option setting
UPDATE_SQL= on page 373	LIBNAME option setting

SQL Pass-Through Facility Specifics for Microsoft SQL Server

Key Information

For general information about this feature, see "Autopartitioning Techniques in SAS/ACCESS" on page 63. Microsoft SQL Server examples are available.

Here are the SQL pass-through facility specifics for the Microsoft SQL Server interface under UNIX hosts.

- The *dbms-name* is **SQLSVR**.
- The CONNECT statement is required.
- PROC SQL supports multiple connections to Microsoft SQL Server. If you use multiple simultaneous connections, you must use the *alias* argument to identify the different connections. If you do not specify an alias, the default alias is used. The functionality of multiple connections to the same Microsoft SQL Server data source might be limited by the particular data source driver.
- The CONNECT statement *database-connection-arguments* are identical to its LIBNAME statement connection options.
- These LIBNAME options are available with the CONNECT statement:
  - AUTOCOMMIT=
  - CURSOR_TYPE=
  - KEYSET_SIZE=
  - QUERY_TIMEOUT=
  - READBUFF=
  - READ_ISOLATION_LEVEL=
  - TRACE=
  - TRACEFILE=
  - USE_ODBC_CL=
- The *DBMS-SQL-query* argument can be a DBMS-specific SQL EXECUTE statement that executes a DBMS stored procedure. However, if the stored procedure contains more than one query, only the first query is processed.

CONNECT Statement Examples

These examples connect to a data source that is configured under the data source name **User's Data** using the alias USER1. The first example uses the connection method that is guaranteed to be present at the lowest level of conformance. Note that DATASRC= names can contain quotation marks and spaces.

```
proc sql;
   connect to sqlsvr as user1
   (datasrc="User's Data" user=testuser password=testpass);
```

This example uses the connection method that represents a more advanced level of Microsoft SQL Server ODBC conformance. It uses the input dialog box that is provided by the driver. The DSN= and UID= arguments are within the connection string. The SQL pass-through facility therefore does not parse them but instead passes them to the ODBC driver manager.

```
proc sql;
   connect to SQLSVR as user1
   (required = "dsn=User's Data; uid=testuser");
```

In this example, you can select any data source that is configured on your machine. The example uses the connection method that represents a more advanced level of Microsoft SQL Server ODBC conformance, Level 1. When connection succeeds, the connection string is returned in the SQLXMSG and SYSDBMSG macro variables. It can then be stored if you use this method to configure a connection for later use.

```
proc sql;
   connect to SQLSVR (required);
```

This example prompts you to specify the information that is required to make a connection to the DBMS. You are prompted to supply the data source name, user ID, and password in the dialog boxes that are displayed.

```
proc sql;
   connect to SQLSVR (prompt);
```

Connection to Component Examples

This example sends Microsoft SQL Server 6.5 (configured under the data source name "SQL Server") an SQL query for processing. The results from the query serve as a virtual table for the PROC SQL FROM clause. In this example MYDB is the connection alias.

```
proc sql;
   connect to SQLSVR as mydb
      (datasrc="SQL Server" user=testuser password=testpass);
   select * from connection to mydb
      (select CUSTOMER, NAME, COUNTRY
          from CUSTOMERS
          where COUNTRY <> 'USA');
quit;
```

This next example returns a list of the columns in the CUSTOMERS table.

```
proc sql;
   connect to SQLSVR as mydb
      (datasrc = "SQL Server" user=testuser password=testpass);
   select * from connection to mydb
      (ODBC::SQLColumns (, , "CUSTOMERS"));
quit;
```

DBLOAD Procedure Specifics for Microsoft SQL Server

Overview

For general information about this feature, see the DBLOAD Procedure on page 901. Microsoft SQL Server examples on page 662 are available.

The Microsoft SQL Server under UNIX hosts interface supports all DBLOAD procedure statements on page 902 (except ACCDESC=) in batch mode. Here are SAS/ACCESS Interface to Microsoft SQL Server specifics for the DBLOAD procedure.

- The DBLOAD step DBMS= value is **SQLSVR**.
- Here are the database description statements that PROC DBLOAD uses:

  DSN= <'>*database-name*<'>;
  : specifies the name of the database in which you want to store the new Microsoft SQL Server table. The *database-name* is limited to eight characters.

    The database that you specify must already exist. If the database name contains the _, $, @, or # special character, you must enclose it in quotation marks. The Microsoft SQL Server standard recommends against using special characters in database names, however

  USER= <'>*user name*<'>;
  : enables you to connect to a Microsoft SQL Server database with a user ID that is different from the default ID.

    USER= is optional in the Microsoft SQL Server interface. If you specify USER=, you must also specify PASSWORD=. If USER= is omitted, your default user ID is used.

  PASSWORD=<'>*password*<'>;
  : specifies the Microsoft SQL Server password that is associated with your user ID.

    PASSWORD= is optional in the Microsoft SQL Server interface because users have default user IDs. If you specify USER=, you must specify PASSWORD=. If you do not wish to enter your SQL Server password in clear text on this statement, see PROC PWENCODE in the *Base SAS Procedures Guide* for a method to encode it.

Examples

This example creates a new Microsoft SQL Server table, TESTUSER.EXCHANGE, from the DLIB.RATEOFEX data file. You must be granted the appropriate privileges in order to create new Microsoft SQL Server tables or views.

```
proc dbload dbms=SQLSVR data=dlib.rateofex;
   dsn=sample;
   user='testuser';
   password='testpass';
   table=exchange;
   rename fgnindol=fgnindollars
```

```
         4=dollarsinfgn;
   nulls updated=n fgnindollars=n
         dollarsinfgn=n country=n;
   load;
run;
```

This example only sends a Microsoft SQL Server SQL GRANT statement to the SAMPLE database and does not create a new table. Therefore, the TABLE= and LOAD statements are omitted.

```
proc dbload dbms=SQLSVR;
   user='testuser';
   password='testpass';
   dsn=sample;
   sql grant select on testuser.exchange
      to dbitest;
run;
```

Passing SAS Functions to Microsoft SQL Server

SAS/ACCESS Interface to Microsoft SQL Server passes the following SAS functions to the data source for processing if the DBMS server supports this function. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

ABS	LOWCASE
ARCOS	MAX
ARSIN	MIN
ATAN	SIGN
AVGCEIL	SIN
COS	SQRT
EXP	TAN
FLOOR	UPCASE
LOG	SUM
LOG10	COUNT

Locking in the Microsoft SQL Server Interface

The following LIBNAME and data set options let you control how the Microsoft SQL Server interface handles locking. For general information about an option, see "LIBNAME Options for Relational Databases" on page 99.

READ_LOCK_TYPE= ROW | TABLE | NOLOCK

UPDATE_LOCK_TYPE= ROW | TABLE | NOLOCK

READ_ISOLATION_LEVEL= S | RR | RC | RU | V
The Microsoft SQL Server ODBC driver manager supports the S, RR, RC, RU, and V isolation levels, as defined in this table.

Table 21.3 *Isolation Levels for Microsoft SQL Server*

Isolation Level	Definition
S (serializable)	Does not allow dirty Reads, nonrepeatable Reads, or phantom Reads.
RR (repeatable read)	Does not allow dirty Reads or nonrepeatable Reads; does allow phantom Reads.
RC (read committed)	Does not allow dirty Reads or nonrepeatable Reads; does allow phantom Reads.
RU (read uncommitted)	Allows dirty Reads, nonrepeatable Reads, and phantom Reads.
V (versioning)	Does not allow dirty Reads, nonrepeatable Reads, or phantom Reads. These transactions are serializable but higher concurrency is possible than with the serializable isolation level. Typically, a nonlocking protocol is used.

Here is how the terms in the table are defined.

Dirty read

A transaction that exhibits this phenomenon has very minimal isolation from concurrent transactions. In fact, the transaction can see changes that are made by those concurrent transactions even before they commit.

For example, if transaction T1 performs an update on a row, transaction T2 then retrieves that row, and transaction T1 then terminates with rollback. Transaction T2 has then seen a row that no longer exists.

Nonrepeatable read

If a transaction exhibits this phenomenon, it is possible that it might read a row once and, if it attempts to read that row again later in the course of the same transaction, the row might have been changed or even deleted by another concurrent transaction. Therefore, the read is not necessarily repeatable.

For example, if transaction T1 retrieves a row, transaction T2 updates that row, and transaction T1 then retrieves the same row again. Transaction T1 has now retrieved the same row twice but has seen two different values for it.

Phantom reads

When a transaction exhibits this phenomenon, a set of rows that it reads once might be a different set of rows if the transaction attempts to read them again.

For example, transaction T1 retrieves the set of all rows that satisfy some condition. If transaction T2 inserts a new row that satisfies that same condition and transaction T1 repeats its retrieval request, it sees a row that did not previously exist, a *phantom*.

UPDATE_ISOLATION_LEVEL= S | RR | RC | V

The Microsoft SQL Server ODBC driver manager supports the S, RR, RC, and V isolation levels that are defined in the preceding table.

Naming Conventions for Microsoft SQL Server

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= LIBNAME options determine how SAS/ACCESS Interface to Microsoft SQL Server handles case sensitivity, spaces, and special characters. (For information about these options, see "Overview: LIBNAME Statement for Relational Databases " on page 93.) The default value for both of these options is YES for Microsoft Access, Microsoft Excel, and Microsoft SQL Server; NO for all others. For additional information about these options, see "Overview: LIBNAME Statement for Relational Databases " on page 93.

Microsoft SQL Server supports table names and column names that contain up to 32 characters. If DBMS column names are longer than 32 characters, SAS truncates them to 32 characters. If truncating a column name would result in identical names, SAS generates a unique name by replacing the last character with a number. DBMS table names must be 32 characters or less because SAS does *not* truncate a longer name. If you already have a table name that is greater than 32 characters, it is recommended that you create a table view.

Data Types for Microsoft SQL Server

Overview

This section includes information about Microsoft SQL Server null and default values on page 605 and data conversions on page 606.

Every column in a table has a name and a data type. The data type tells the Microsoft SQL Server how much physical storage to set aside for the column and the form in which the data is stored.

Microsoft SQL Server Null Values

Microsoft SQL Server has a special value called NULL. A Microsoft SQL Server NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a Microsoft SQL Server NULL value, it interprets it as a SAS missing value.

Microsoft SQL Server columns can be defined as NOT NULL so that they require data—they cannot contain NULL values. When a column is defined as NOT NULL, the DBMS does not add a row to the table unless the row has a value for that column. When creating a DBMS table with SAS/ACCESS, you can use the DBNULL= on page 294 data set option to indicate whether NULL is a valid value for specified columns.

For more information about how SAS handles NULL values, see "Potential Result Set Differences When Processing Null Data " on page 35.

To control how SAS missing character values are handled, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

The following table shows all data types and default SAS formats that SAS/ACCESS Interface to Microsoft SQL Server supports.

Table 21.4 *Microsoft SQL Server Data Types and Default SAS Formats*

Microsoft SQL Server Data Type	Default SAS Format
SQL_CHAR	$*n*
SQL_VARCHAR	$*n*
SQL_LONGVARCHAR	$*n*
SQL_BINARY	$*n.**
SQL_VARBINARY	$*n.**
SQL_LONGVARBINARY	$*n.**
SQL_DECIMAL	*m* or *m.n* or none if *m* and *n* are not specified
SQL_NUMERIC	*m* or *m.n* or none if *m* and *n* are not specified
SQL_INTEGER	11.
SQL_SMALLINT	6.
SQL_TINYINT	4.
SQL_BIT	1.
SQL_REAL	none
SQL_FLOAT	none
SQL_DOUBLE	none
SQL_BIGINT	20.
SQL_DATE	DATE9.
SQL_TIME	TIME8. Microsoft SQL Server cannot support fractions of seconds for time values
SQL_TIMESTAMP	DATETIME*m.n* where *m* and *n* depend on precision

* Because the Microsoft SQL Server driver does the conversion, this field is displayed as if the $HEX*n*. format were applied.

The following table shows the default data types that the Microsoft SQL Server interface uses when creating tables.

Table 21.5 *Default Microsoft SQL Server Output Data Types*

SAS Variable Format	Default Microsoft SQL Server Data Type
m.n	SQL_DOUBLE or SQL_NUMERIC using *m.n* if the DBMS allows it
$*n.*	SQL_VARCHAR using *n*
datetime formats	SQL_TIMESTAMP
date formats	SQL_DATE
time formats	SQL_TIME

The Microsoft SQL Server interface allows non-default data types to be specified with the DBTYPE= on page 303 data set option.

Chapter 22

SAS/ACCESS Interface to MySQL

Introduction to SAS/ACCESS Interface to MySQL

This section describes SAS/ACCESS Interface to MySQL. For a list of SAS/ACCESS features that are available in this interface, see "SAS/ACCESS Interface to MySQL: Supported Features" on page 83.

LIBNAME Statement Specifics for MySQL

Overview

This section describes the LIBNAME statements that SAS/ACCESS Interface to MySQL supports and includes examples. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing MySQL.

LIBNAME *libref* **mysql** <*connection-options* ><*LIBNAME-options*>;

Arguments

libref
: specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables.

`mysql`
: specifies the SAS/ACCESS engine name for MySQL interface.

connection-options
: provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. Here is how these options are defined.

USER=<'>*user name*<'>
: specifies the MySQL user login ID. If this argument is not specified, the current user is assumed. If the user name contains spaces or nonalphanumeric characters, you must enclose the user name in quotation marks.

PASSWORD=<'>*password*<'>
: specifies the MySQL password that is associated with the MySQL login ID. If the password contains spaces or nonalphanumeric characters, you must enclose the password in quotation marks.

DATABASE=<'>*database*<'>
: specifies the MySQL database to which you want to connect. If the database name contains spaces or nonalphanumeric characters, you must enclose the database name in quotation marks.

SERVER=<'>*server*<'>
: specifies the server name or IP address of the MySQL server. If the server name contains spaces or nonalphanumeric characters, you must enclose the server name in quotation marks.

PORT=*port*
: specifies the port used to connect to the specified MySQL server. If you do not specify a value, 3306 is used.

LIBNAME-options
: define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to MySQL, with the applicable default values. For more detail about these options, see "LIBNAME Options for Relational Databases" on page 99.

Table 22.1 SAS/ACCESS LIBNAME Options for MySQL

Option	Default Value
ACCESS= on page 99	none
AUTHDOMAIN= on page 102	none
AUTOCOMMIT= on page 103	YES
CONNECTION= on page 113	SHAREDREAD
CONNECTION_GROUP= on page 118	none
DBCOMMIT= on page 124	1000 when inserting rows; 0 when updating, deleting, or appending rows to an existing table
DBCONINIT= on page 125	none
DBCONTERM= on page 126	none
DBCREATE_TABLE_OPTS= on page 128	none
DBGEN_NAME= on page 128	DBMS
DBINDEX= on page 129	NO
DBLIBINIT= on page 130	none
DBLIBTERM= on page 131	none
DBMAX_TEXT= on page 133	1024
DBMSTEMP= on page 133	NO
DBPROMPT= on page 136	NO
DBSASLABEL= on page 137	COMPAT
DEFER= on page 141	NO
DIRECT_EXE= on page 144	none
DIRECT_SQL= on page 145	YES
ESCAPE_BACKSLASH= on page 148	
INSERTBUFF= on page 153	0
MULTI_DATASRC_OPT= on page 160	none
PRESERVE_COL_NAMES= on page 167	YES

Option	Default Value
PRESERVE_TAB_NAMES= on page 168	YES
QUALIFIER= on page 170	none
REREAD_EXPOSURE= on page 179	NO
RESULTS= on page 179	MEMORY
SPOOL= on page 184	YES
SQL_FUNCTIONS= on page 185	none
SQL_FUNCTIONS_COPY= on page 188	none
UTILCONN_TRANSIENT= on page 202	NO

MySQL LIBNAME Statement Examples

In the following example, the libref MYSQLLIB uses SAS/ACCESS Interface to MySQL to connect to a MySQL database. The SAS/ACCESS connection options are USER=, PASSWORD=, DATABASE=, SERVER=, and PORT=.

```
libname mysqllib mysql user=testuser password=testpass database=mysqldb
server=mysqlserv port=9876;

proc print data=mysqllib.employees;
   where dept='CSR010';
run;
```

Data Set Options for MySQL

All SAS/ACCESS data set options in this table are supported for MySQL. Default values are provided where applicable. For general information about this feature, see "Overview" on page 207.

Table 22.2 *Data Set Options for MySQL*

Option	Default Value
AUTOCOMMIT= on page 208	the current LIBNAME option setting
DBCOMMIT= on page 280	the current LIBNAME option setting
DBCONDITION= on page 281	none
DBCREATE_TABLE_OPTS= on page 285	the current LIBNAME option setting

Option	Default Value
DBGEN_NAME= on page 287	DBMS
DBINDEX= on page 288	the current LIBNAME option setting
DBKEY= on page 290	none
DBLABEL= on page 291	NO
DBMASTER= on page 293	none
DBMAX_TEXT= on page 293	1024
DBNULL= on page 294	YES
DBPROMPT= on page 296	the current LIBNAME option setting
DBSASLABEL= on page 297	COMPAT
DBSASTYPE= on page 298	see "Data Types for MySQL " on page 618
DBTYPE= on page 303	see "LIBNAME Statement Data Conversions" on page 620
ESCAPE_BACKSLASH= on page 309	NO
INSERTBUFF= on page 314	0
NULLCHAR= on page 329	SAS
NULLCHARVAL= on page 330	a blank character
PRESERVE_COL_NAMES= on page 336	current LIBNAME option setting
QUALIFIER= on page 338	the current LIBNAME option setting
SASDATEFORMAT= on page 342	DATETIME20.0
UPDATE_ISOLATION_LEVEL= on page 371	the current LIBNAME option setting

SQL Pass-Through Facility Specifics for MySQL

Key Information

For general information about this feature, see "SQL Pass-Through Facility" on page 401. MySQL examples are available.

Here are the SQL pass-through facility specifics for MySQL.

- The *dbms-name* is **mysql**.
- If you call MySQL stored procedures that return multiple result sets, SAS returns only the last result set.
- Here are the *database-connection-arguments* for the CONNECT statement.

  USER=<'>*MySQL-login-ID*<'>
  : specifies an optional MySQL login ID. If USER= is not specified, the current user is assumed. If you specify USER=, you also must specify PASSWORD=.

  PASSWORD=<'>*MySQL-password*<'>
  : specifies the MySQL password that is associated with the MySQL login ID. If you specify PASSWORD=, you also must specify USER=.

  DATABASE=<'>*database-name*<'>
  : specifies the MySQL database.

  SERVER=<'>*server-name*<'>
  : specifies the name or IP address of the MySQL server to which to connect. If *server-name* is omitted or set to **localhost**, a connection to the local host is established.

  PORT=*port*
  : specifies the port on the server that is used for the TCP/IP connection.

  Note: Due to a current limitation in the MySQL client library, you cannot run MySQL stored procedures when SAS is running on AIX.

Examples

This example uses the alias DBCON for the DBMS connection (the connection alias is optional):

```
proc sql;
   connect to mysql as dbcon
       (user=testuser password=testpass server=mysqlserv
        database=mysqldb port=9876);
quit;
```

This example connects to MySQL and sends it two EXECUTE statements to process:

```
proc sql;
   connect to mysql (user=testuser password=testpass server=mysqlserv
        database=mysqldb port=9876);
   execute (create table whotookorders as
      select ordernum, takenby,
             firstname, lastname, phone
         from orders, employees
         where orders.takenby=employees.empid)
      by mysql;
   execute (grant select on whotookorders
           to testuser) by mysql;
   disconnect from mysql;
quit;
```

This example performs a query, shown in highlighted text, on the MySQL table CUSTOMERS:

```
proc sql;
connect to mysql (user=testuser password=testpass server=mysqlserv
```

```
        database=mysqldb port=9876);
select *
   from connection to mysql
     (select * from customers
      where customer like '1%');
    disconnect from mysql;
quit;
```

Autocommit and Table Types

MySQL supports several table types, two of which are InnoDB (the default) and MyISAM. A single database can contain tables of different types. The behavior of a table is determined by its table type. For example, by definition, a table created of MyISAM type does not support transactions. Consequently, all DML statements (updates, deletes, inserts) are automatically committed. If you need transactional support, specify a table type of InnoDB in the DBCREATE_TABLE_OPTS= on page 128 LIBNAME option. This table type allows for updates, deletes, and inserts to be rolled back if an error occurs; or updates, deletes, and inserts to be committed if the SAS DATA step or procedure completes successfully.

By default, the MYSQL LIBNAME engine sets **AUTOCOMMIT=YES** regardless of the table type. If you are using tables of the type InnoDB, set the LIBNAME option AUTOCOMMIT=NO on page 103 to improve performance. To control how often COMMITS are executed, set the DBCOMMIT= on page 124 option.

Note: The DBCOMMIT option can affect SAS/ACCESS performance. Experiment with a value that best fits your table size and performance needs before using it for production jobs. Transactional tables require significantly more memory and disk space requirements.

Understanding MySQL Update and Delete Rules

To avoid data integrity problems when updating or deleting data, you need a primary key defined on your table. See the MySQL documentation for more information about table types and transactions.

The following example uses AUTOCOMMIT=NO and DBTYPE to create the primary key, and DBCREATE_TABLE_OPTS to determine the MySQL table type.

```
libname invty mysql user=dbitest server=d6687 database=test autocommit=no
reread_exposure=no;

proc sql;
drop table invty.STOCK23;
quit;

/* Create DBMS table with primary key and of type INNODB*/
data invty.STOCK23(drop=PARTNO DBTYPE=(RECDATE="date not null,
     primary key(RECDATE)") DBCREATE_TABLE_OPTS="type = innodb");
  input PARTNO $ DESCX $ INSTOCK @17
        RECDATE date7. @25 PRICE;
  format RECDATE date7.;
```

```
  datalines;
K89R seal   34   27jul95 245.00
M447 sander 98   20jun95 45.88
LK43 filter 121 19may96 10.99
MN21 brace 43   10aug96 27.87
BC85 clamp 80   16aug96 9.55
KJ66 cutter 6   20mar96 24.50
UYN7 rod  211   18jun96 19.77
JD03 switch 383 09jan97 13.99
BV1I timer 26   03jan97 34.50
;
```

The next examples show how you can update the table now that STOCK23 has a primary key:

```
proc sql;
update invty.STOCK23 set price=price*1.1 where INSTOCK > 50;
quit;
```

Passing SAS Functions to MySQL

SQL_FUNCTIONS=ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. (See SQL_FUNCTIONS= on page 185.) Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to MySQL. Due to incompatibility in date and time functions between MySQL and SAS, MySQL might not process them correctly. Check your results to determine whether these functions are working as expected.

Where the MySQL function name differs from the SAS function name, the MySQL name appears in parentheses. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

ABS	LOG2
ARCOS (ACOS)	LOG10
ARSIN (ASIN)	LOWCASE (LCASE)
ATAN	MAX
AVG	MIN
BYTE (CHAR)	MINUTE
CEIL (CEILING)	MOD
COALESCE	MONTH
COMPRESS (REPLACE)	QTR (QUARTER)
COS	REPEAT
COT	ROUND
COUNT	SECOND
DATE (CURDATE)	SIGN
DATEPART	SIN
DATETIME (NOW)	SOUNDEX
DAY (DAYOFMONTH)	SQRT
DTEXTDAY	STRIP (TRIM)

DTEXTMONTH	SUBSTR (SUBSTRING)
DTEXTWEEKDAY	TAN
DTEXTYEAR	TIME (CURTIME())
EXP	TIMEPART
FLOOR	TODAY (CURDATE())
HOUR	TRIMN (RTRIM)
INDEX (LOCATE)	UPCASE (UCASE)
LENGTH	WEEKDAY (DAYOFWEEK)
LOG	YEAR

Passing Joins to MySQL

For a multiple libref join to pass to MySQL, all of these components of the LIBNAME statements must match exactly:

- user (USER=)
- password (PASSWORD=)
- database DATABASE=)
- server (SERVER=)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Naming Conventions for MySQL

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

MySQL database identifiers that you can name include databases, tables, and columns. They follow these naming conventions.

- Aliases must be from 1 to 255 characters long. All other identifier names must be from 1 to 64 characters long.
- Database names can use any character that is allowed in a directory name except for a period, a backward slash (\), or a forward slash (/).
- By default, MySQL encloses column names and table names in quotation marks.
- Table names can use any character that is allowed in a filename except for a period or a forward slash.
- Table names must be 32 characters or less because SAS does not truncate a longer name. If you already have a table name that is greater than 32 characters, it is recommended that you create a table view.
- Column names and alias names allow all characters.
- Embedded spaces and other special characters are not permitted unless you enclose the name in quotation marks.
- Embedded quotation marks are not permitted.

- Case sensitivity is set when a server is installed. By default, the names of database objects are case sensitive on UNIX and not case sensitive on Windows. For example, the names **`CUSTOMER`** and **`Customer`** are different on a case-sensitive server.
- A name cannot be a reserved word in MySQL unless you enclose the name in quotation marks. See the MySQL documentation for more information about reserved words.
- Database names must be unique. For each user within a database, names of database objects must be unique across all users. For example, if a database contains a department table that User A created, no other user can create a department table in the same database.

  MySQL does not recognize the notion of schema, so tables are automatically visible to all users with the appropriate privileges. Column names and index names must be unique within a table.

For detailed information about naming conventions, see your MySQL documentation.

Data Types for MySQL

Overview

Every column in a table has a name and a data type. The data type tells MySQL how much physical storage to set aside for the column and the form in which the data is stored. This section includes information about MySQL data types and data conversions.

Character Data

BLOB (binary large object)
: contains binary data of variable length up to 64 kilobytes. When you enter variables of this type into columns, you must insert them as character strings.

CHAR *(n)*
: contains fixed-length character string data with a length of *n*, where *n* must be at least 1 and cannot exceed 255 characters.

ENUM (*“value1”*, *“value2”*, “*value3”*,…)
: contains a character value that can be chosen from the list of allowed values. You can specify up to 65535 ENUM values. If the column contains a string not specified in the value list, the column value is set to “0”.

LONGBLOB
: contains binary data of variable length up to 4 gigabytes. Variables entered into columns of this type must be inserted as character strings. Available memory considerations might limit the size of a LONGBLOB data type.

LONGTEXT
: contains text data of variable length up to 4 gigabytes. Available memory considerations might limit the size of a LONGTEXT data type.

MEDIUMBLOB
: contains binary data of variable length up to 16 megabytes. Variables entered into columns of this type must be inserted as character strings.

MEDIUMTEXT
: contains text data of variable length up to 16 megabytes.

SET (*"value1"*, *"value2*", *"value3"*,…)
contains zero or more character values that must be chosen from the list of allowed values. You can specify up to 64 SET values.

TEXT
contains text data of variable length up to 64 kilobytes.

TINYBLOB
contains binary data of variable length up to 256 bytes. Variables entered into columns of this type must be inserted as character strings.

TINYTEXT
contains text data of variable length up to 256 bytes.

VARCHAR (*n*)
contains character string data with a length of *n*, where *n* is a value from 1 to 255.

Numeric Data

BIGINT (*n*)
specifies an integer value, where *n* indicates the display width for the data. You might experience problems with MySQL if the data column contains values that are larger than the value of *n*. Values for BIGINT can range from –9223372036854775808 to 9223372036854775808.

DECIMAL (*length*, *decimals*)
specifies a fixed-point decimal number, where *length* is the total number of digits (precision), and *decimals* is the number of digits to the right of the decimal point (scale).

DOUBLE (*length*, *decimals*)
specifies a double-precision decimal number, where *length* is the total number of digits (precision), and *decimals* is the number of digits to the right of the decimal point (scale). Values can range from approximately –1.8E308 to –2.2E-308 and 2.2E-308 to 1.8E308 (if UNSIGNED is specified).

FLOAT (*length*, *decimals*)
specifies a floating-point decimal number, where *length* is the total number of digits (precision) and *decimals* is the number of digits to the right of the decimal point (scale). Values can range from approximately –3.4E38 to –1.17E-38 and 1.17E-38 to 3.4E38 (if UNSIGNED is specified).

INT (*n*)
specifies an integer value, where *n* indicates the display width for the data. You might experience problems with MySQL if the data column contains values that are larger than the value of *n*. Values for INT can range from –2147483648 to 2147483647.

MEDIUMINT (*n*)
specifies an integer value, where *n* indicates the display width for the data. You might experience problems with MySQL if the data column contains values that are larger than the value of *n*. Values for MEDIUMINT can range from –8388608 to 8388607.

SMALLINT (*n*)
specifies an integer value, where *n* indicates the display width for the data. You might experience problems with MySQL if the data column contains values that are larger than the value of *n*. Values for SMALLINT can range from –32768 to 32767.

TINYINT (*n*)
: specifies an integer value, where *n* indicates the display width for the data. You might experience problems with MySQL if the data column contains values that are larger than the value of *n*. Values for TINYINT can range from –128 to 127.

Date, Time, and Timestamp Data

DATE
: contains date values. Valid dates are from January 1, 1000, to December 31, 9999. The default format is YYYY-MM-DD. An example is 1961-06-13.

DATETIME
: contains date and time values. Valid values are from 00:00:00 on January 1, 1000, to 23:59:59 on December 31, 9999. The default format is YYYY-MM-DD HH:MM:SS (for example, 1992-09-20 18:20:27).

TIME
: contains time values. Valid times are –838 hours, 59 minutes, 59 seconds to 838 hours, 59 minutes, 59 seconds. The default format is HH:MM:SS (for example, 12:17:23).

TIMESTAMP
: contains date and time values used to mark data operations. Valid values are from 00:00:00 on January 1, 1970, to 2037. The default format is YYYY-MM-DD HH:MM:SS (for example, 1995–08–09 15:12:27).

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to MySQL assigns to SAS variables when using the LIBNAME statement on page 93 to read from a MySQL table. These default formats are based on MySQL column attributes.

Table 22.3 *LIBNAME Statement: Default SAS Formats for MySQL Data Types*

MySQL Column Type	SAS Data Type	Default SAS Format
CHAR(*n*)	character	$*n*.
VARCHAR(*n*)	character	$*n*.
TINYTEXT	character	$*n*.
TEXT	character	$*n*. (where *n* is the value of the DBMAX_TEXT= on page 293 option)
MEDIUMTEXT	character	$*n*. (where *n* is the value of the DBMAX_TEXT= on page 293 option)
LONGTEXT	character	$*n*. (where *n* is the value of the DBMAX_TEXT= on page 293 option)
TINYBLOB	character	$*n*. (where *n* is the value of the DBMAX_TEXT= on page 293 option)

MySQL Column Type	SAS Data Type	Default SAS Format
BLOB	character	$*n*. (where *n* is the value of the DBMAX_TEXT= on page 293 option)
MEDIUMBLOB	character	$*n*. (where *n* is the value of the DBMAX_TEXT= on page 293 option)
LONGBLOB	character	$*n*. (where *n* is the value of the DBMAX_TEXT= on page 293 option)
ENUM	character	$*n*.
SET	character	$*n*.
TINYINT	numeric	4.0
SMALLINT	numeric	6.0
MEDIUMINT	numeric	8.0
INT	numeric	11.0
BIGINT	numeric	20.
DECIMAL	numeric	*m.n*
FLOAT	numeric	
DOUBLE	numeric	
DATE	numeric	DATE
TIME	numeric	TIME
DATETIME	numeric	DATETIME
TIMESTAMP	numeric	DATETIME

This table shows the default MySQL data types that SAS/ACCESS assigns to SAS variable formats during output operations when you use the LIBNAME statement.

Table 22.4 *LIBNAME Statement: Default MySQL Data Types for SAS Variable Formats*

SAS Variable Format	MySQL Data Type
*m.n**	DECIMAL ([*m*-1],*n*)**
n (where *n* <= 2)	TINYINT
n (where *n* <= 4)	SMALLINT
n (where *n* <=6)	MEDIUMINT

SAS Variable Format	MySQL Data Type
n (where *n* <= 17)	BIGINT
other numerics	DOUBLE
$*n* (where *n* <= 255)	VARCHAR(*n*)
$*n* (where *n* > 255)	TEXT
datetime formats	TIMESTAMP
date formats	DATE
time formats	TIME

* *n* in MySQL data types is equivalent to *w* in SAS formats.

** DECIMAL types are created as (m-1, n). SAS includes space to write the value, the decimal point, and a minus sign (if necessary) in its calculation for precision. These must be removed when converting to MySQL.

Case Sensitivity for MySQL

In MySQL, databases and tables correspond to directories and files within those directories. Consequently, the case sensitivity of the underlying operating system determines the case sensitivity of database and table names. This means database and table names are not case sensitive in Windows, and case sensitive in most varieties of UNIX.

In SAS, names can be entered in either uppercase or lowercase. MySQL recommends that you adopt a consistent convention of either all uppercase or all lowercase table names, especially on UNIX hosts. This can be easily implemented by starting your server with *-O lower_case_table_names=1*. Please see the MySQL documentation for more details.

If your server is on a case-sensitive platform, and you choose to allow case sensitivity, be aware that when you reference MYSQL objects through the SAS/ACCESS interface, objects are case sensitive and require no quotation marks. Also, in the SQL pass-through facility, all MySQL object names are case sensitive. Names are passed to MySQL exactly as they are entered.

For more information about case sensitivity and MySQL names, see "Naming Conventions for MySQL" on page 617.

Chapter 23
SAS/ACCESS Interface to Netezza

Introduction to SAS/ACCESS Interface to Netezza

This section describes SAS/ACCESS Interface to Netezza. For a list of SAS/ACCESS features that are available in this interface, see “SAS/ACCESS Interface to Netezza: Supported Features” on page 84.

LIBNAME Statement Specifics for Netezza

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to Netezza supports and includes examples. For general information about this feature, see LIBNAME Option for Relational Databases on page 93.

Here is the LIBNAME statement syntax for accessing Netezza.

LIBNAME *libref* **netezza** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
: specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`netezza`
: specifies the SAS/ACCESS engine name for the Netezza interface.

connection-options
: provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. When you use the LIBNAME statement, you can connect to the Netezza Performance Server in two ways. Specify *only one* of these methods for each connection because they are mutually exclusive.

- SERVER=, DATABASE=, PORT=, USER=, PASSWORD=, READ_ONLY=
- USER=, DSN=, PASSWORD=

Here is how these options are defined.

SERVER=<'>*server-name*<'>
: specifies the server name or IP address of the Netezza Performance Server to which you want to connect. This server accesses the database that contains the tables and views that you want to access. If the server name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

DATABASE=<'>*database-name*<'>
: specifies the name of the database on the Netezza Performance Server that contains the tables and views that you want to access. If the database name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

Alias: DB=

PORT=*port*
: specifies the port number that is used to connect to the specified Netezza Performance Server. If you do not specify a port, the default is 5480.

USER=<'>*Netezza-user-name*<'>
: specifies the Netezza user name (also called the user ID) that you use to connect to your database. If the user name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

PASSWORD=<'>*Netezza-password*<'>
: specifies the password that is associated with your Netezza user name. If the password contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. You can also specify PASSWORD= with the PWD=, PASS=, and PW= aliases.

READ_ONLY=YES | NO
: specifies whether to connect to the Netezza database in Read-Only mode (YES) or read-write (NO) mode. If you do not specify anything for READ_ONLY=, the default of NO is used. You can also specify READ_ONLY with the READONLY= alias.

DSN=<'>*Netezza-data-source*<'>
: specifies the configured Netezza ODBC data source to which you want to connect. Use this option if you have existing Netezza ODBC data sources that are configured on your client. This method requires additional setup—either through the ODBC Administrator control panel on Windows platforms, or through the odbc.ini file or a similarly named configuration file on UNIX platforms. So it is recommended that you use this connection method only if you have existing, functioning data sources that have been defined.

LIBNAME-options
: define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to Netezza, with the applicable default values. For details about these options, see “LIBNAME Options for Relational Databases” on page 99.

***Table 23.1** SAS/ACCESS LIBNAME Options for Netezza*

Option	Default Value
ACCESS= on page 99	none
AUTHDOMAIN= on page 102	none
AUTOCOMMIT= on page 103	operation-specific
BULKUNLOAD= on page 108	NO
CHAR_AS_NCHAR= on page 112	NO
CONNECTION= on page 113	UNIQUE
CONNECTION_GROUP= on page 118	none
DBCOMMIT= on page 124	1000 when inserting rows; 0 when updating rows
DBCONINIT= on page 125	none
DBCONTERM= on page 126	none
DBCREATE_TABLE_OPTS= on page 128	none
DBGEN_NAME= on page 128	DBMS

Option	Default Value
DBINDEX= on page 129	YES
DBLIBINIT= on page 130	none
DBLIBTERM= on page 131	none
DBMAX_TEXT= on page 133	1024
DBMSTEMP= on page 133	NO
DBNULLKEYS= on page 135	YES
DBPROMPT= on page 136	NO
DBSASLABEL= on page 297	COMPAT
DEFER= on page 141	NO
DELETE_MULT_ROWS= on page 143	NO
DIRECT_EXE= on page 144	none
DIRECT_SQL= on page 145	YES
IGNORE_READ_ONLY_COLUMNS= on page 150	NO
INSERTBUFF= on page 153	automatically calculated based on row length
LOGIN_TIMEOUT= on page 159	0
MULTI_DATASRC_OPT= on page 160	none
PRESERVE_COL_NAMES= on page 167	see “Naming Conventions for Netezza” on page 636
PRESERVE_TAB_NAMES= on page 168	see “Naming Conventions for Netezza” on page 636
QUALIFIER= on page 170	none
QUERY_TIMEOUT= on page 172	0
QUOTE_CHAR= on page 173	none
READBUFF= on page 174	automatically calculated based on row length
REREAD_EXPOSURE= on page 179	NO
SCHEMA= on page 180	none

Option	Default Value
SPOOL= on page 184	YES
SQL_FUNCTIONS= on page 185	none
SQL_FUNCTIONS_COPY= on page 188	none
STRINGDATES= on page 190	NO
TRACE= on page 195	NO
TRACEFILE= on page 196	none
UPDATE_MULT_ROWS= on page 199	NO
USE_ODBC_CL= on page 201	NO
UTILCONN_TRANSIENT= on page 202	NO

Netezza LIBNAME Statement Examples

In this example, SERVER=, DATABASE=, USER=, and PASSWORD= are connection options.

```
libname mydblib netezza server=npssrv1 database=test
   user=netusr1 password=netpwd1;

proc print data=mydblib.customers;
   where state='CA';
run;
```

In the next example, DSN=, USER=, and PASSWORD= are connection options. The NZSQL data source is configured in the ODBC Administrator Control Panel on Windows platforms or in the odbc.ini file or a similarly named configuration file on UNIX platforms.

```
libname mydblib netezza dsn=NZSQL
   user=netusr1 password=netpwd1;

proc print data=mydblib.customers;
   where state='CA';
run;
```

Data Set Options for Netezza

All SAS/ACCESS data set options in this table are supported for Netezza. Default values are provided where applicable. For general information about this feature, see Data Set Options for Relational Databases on page 207.

***Table 23.2** SAS/ACCESS Data Set Options for Netezza*

Option	Default Value
BL_DATAFILE= on page 216	When BL_USE_PIPE=NO, creates a file in the current directory or with the default file specifications.
BL_DELETE_DATAFILE= on page 233	YES (only when BL_USE_PIPE=NO)
BL_DELIMITER= on page 235	| (the pipe symbol)
BL_OPTIONS= on page 255	none
BL_USE_PIPE= on page 271	YES
BULKLOAD= on page 274	NO
BULKUNLOAD= on page 275	LIBNAME option setting
DBCOMMIT= on page 280	LIBNAME option setting
DBCONDITION= on page 281	none
DBCREATE_TABLE_OPTS= on page 285	LIBNAME option setting
DBFORCE= on page 286	NO
DBGEN_NAME= on page 287	DBMS
DBINDEX= on page 288	LIBNAME option setting
DBKEY= on page 290	none
DBLABEL= on page 291	NO
DBMASTER= on page 293	none
DBMAX_TEXT= on page 293	1024
DBNULL= on page 294	YES
DBNULLKEYS= on page 295	LIBNAME option setting
DBPROMPT=< on page 296	LIBNAME option setting
DBSASTYPE= on page 298	see "Data Types for Netezza " on page 636
DBTYPE= on page 303	see "Data Types for Netezza " on page 636
DISTRIBUTE_ON= on page 306	none
ERRLIMIT= on page 308	1

Option	Default Value
IGNORE_READ_ONLY_COLUMNS= on page 311	NO
INSERTBUFF= on page 314	LIBNAME option setting
NULLCHAR= on page 329	SAS
NULLCHARVAL= on page 330	a blank character
PRESERVE_COL_NAMES= on page 336	LIBNAME option setting
QUALIFIER= on page 338	LIBNAME option setting
QUERY_TIMEOUT= on page 339	LIBNAME option setting
READBUFF= on page 339	LIBNAME option setting
SASDATEFMT= on page 342	none
SCHEMA= on page 344	LIBNAME option setting

SQL Pass-Through Facility Specifics for Netezza

Key Information

For general information about this feature, see "About SQL Procedure Interactions " on page 401. Netezza examples on page 629 are available.

Here are the SQL pass-through facility specifics for the Netezza interface.

- The *dbms-name* is **NETEZZA**.
- The CONNECT statement is required.
- PROC SQL supports multiple connections to Netezza. If you use multiple simultaneous connections, you must use the *alias* argument to identify the different connections. If you do not specify an alias, the default **netezza** alias is used.
- The CONNECT statement *database-connection-arguments* are identical to its LIBNAME connection-options on page 624.

CONNECT Statement Examples

This example uses the DBCON alias to connection to the **mynpssrv** Netezza Performance Server and execute a query. The connection alias is optional.

```
proc sql;
   connect to netezza as dbcon
   (server=mynpssrv database=test user=myuser password=mypwd);
select * from connection to dbcon
```

```
   (select * from customers where customer like '1%');
quit;
```

Special Catalog Queries

SAS/ACCESS Interface to Netezza supports the following special queries. You can the queries use to call the ODBC-style catalog function application programming interfaces (APIs). Here is the general format of the special queries:

Netezza::SQLAPI "*parameter 1*","*parameter n*"

Netezza::
: is required to distinguish special queries from regular queries.

SQLAPI
: is the specific API that is being called. Neither Netezza:: nor SQLAPI are case sensitive.

"*parameter n*"
: is a quoted string that is delimited by commas.

Within the quoted string, two characters are universally recognized: the percent sign (%) and the underscore (_). The percent sign matches any sequence of zero or more characters, and the underscore represents any single character. To use either character as a literal value, you can use the backslash character (\) to escape the match characters. For example, this call to SQLTables usually matches table names such as mytest and my_test:

```
select * from connection to netezza (NETEZZA::SQLTables "test","","my_test");
```

Use the escape character to search only for the my_test table:

```
select * from connection to netezza (NETEZZA::SQLTables "test","","my\_test");
```

SAS/ACCESS Interface to Netezza supports these special queries:

Netezza::SQLTables <*"Catalog"*, *"Schema"*, *"Table-name"*, *"Type"*>
: returns a list of all tables that match the specified arguments. If you do not specify any arguments, all accessible table names and information are returned.

Netezza::SQLColumns <*"Catalog"*, *"Schema"*, *"Table-name"*, *"Column-name"*>
: returns a list of all columns that match the specified arguments. If you do not specify any argument, all accessible column names and information are returned.

Netezza::SQLPrimaryKeys <*"Catalog"*, *"Schema"*, *"Table-name"*>
: returns a list of all columns that compose the primary key that matches the specified table. A primary key can be composed of one or more columns. If you do not specify any table name, this special query fails.

Netezza::SQLSpecialColumns <*"Identifier-type"*, *"Catalog-name"*, *"Schema-name"*, *"Table-name"*, *"Scope"*, *"Nullable"*>
: returns a list of the optimal set of columns that uniquely identify a row in the specified table.

Netezza::SQLStatistics <*"Catalog"*, *"Schema"*, *"Table-name"*>
: returns a list of the statistics for the specified table name, with options of SQL_INDEX_ALL and SQL_ENSURE set in the SQLStatistics API call. If you do not specify any table name argument, this special query fails.

Netezza::SQLGetTypeInfo
: returns information about the data types that the Netezza Performance Server supports.

Temporary Table Support for Netezza

General Information

See "Temporary Table Support for SAS/ACCESS " on page 42 for general information about this feature.

Establishing a Temporary Table

To make full use of temporary tables, the CONNECTION=GLOBAL connection option is necessary. This option lets you use a single connection across SAS DATA steps and SAS procedure boundaries. This connection can also be shared between LIBNAME statements and the SQL pass-through facility. Because a temporary table exists only within a single connection, you need to be able to share this single connection among all steps that reference the temporary table. The temporary table cannot be referenced from any other connection.

You can currently use only a PROC SQL statement to create a temporary table. To use both the SQL pass-through facility and librefs to reference a temporary table, you must specify a LIBNAME statement before the PROC SQL step so that global connection persists across SAS steps and even across multiple PROC SQL steps. Here is an example.

```
proc sql;
   connect to netezza (server=nps1 database=test
       user=myuser password=mypwd connection=global);
   execute (create temporary table temptab1
       as select * from permtable ) by netezza;
quit;
```

At this point, you can refer to the temporary table by using either the Temp libref or the CONNECTION=GLOBAL option with a PROC SQL step.

Terminating a Temporary Table

You can drop a temporary table at any time or allow it to be implicitly dropped when the connection is terminated. Temporary tables do not persist beyond the scope of a single connection.

Examples

These assumptions apply to the examples in this section.

- The DeptInfo table already exists on the DBMS that contains all of your department information.
- One SAS data set contains join criteria that you want to use to extract specific rows from the DeptInfo table.
- Another SAS data set contains updates to the DeptInfo table.

Examples use these librefs and temporary tables.

```
libname saslib base 'SAS-library';
libname dept netezza server=nps1 database=test
    user=myuser pwd=mypwd connection=global;
libname temp netezza server=nps1 database=test
    user=myuser pwd=mypwd connection=global;

proc sql;
   connect to netezza (server=nps1 database=test
        user=myuser pwd=mypwd connection=global);
   execute (create temporary table temptab1 (dname char(20),
        deptno int)) by netezza;
quit;
```

This example shows how to use a heterogeneous join with a temporary table to perform a homogeneous join on the DBMS, instead of reading the DBMS table into SAS to perform the join. By using the table that was created previously, you can copy SAS data into the temporary table to perform the join.

```
proc sql;
   connect to netezza (server=nps1 database=test
        user=myuser pwd=mypwd connection=global);
   insert into temp.temptab1 select * from saslib.joindata;
   select * from dept.deptinfo info, temp.temptab1 tab
        where info.deptno = tab.deptno;
   /* remove the rows for the next example */
   execute (delete from temptab1) by netezza;
   quit;
```

In this example, transaction processing on the DBMS occurs by using a temporary table instead of using either DBKEY= or MULTI_DATASRC_OPT=IN_CLAUSE with a SAS data set as the transaction table.

```
proc sql;
connect to netezza (server=nps1 database=test
   user=myuser pwd=mypwd connection=global);
   insert into temp.temptab1 select * from saslib.transdat;
   execute (update deptinfo d set dname = (select dname from temptab1)
            where d.deptno = (select deptno from temptab1)) by netezza;
quit;
```

Passing SAS Functions to Netezza

SAS/ACCESS Interface to Netezza passes the following SAS functions to Netezza for processing. Where the Netezza function name differs from the SAS function name, the Netezza name appears in parentheses. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

ABS	LOG10 (log)
ARCOS (ACOS)	LOWCASE (lower)
ARSIN (ASIN)	MAX
ATAN	MIN
ATAN2	MINUTE (date_part)
AVG	MOD
BAND (int4and)	MONTH (date_part)

BNOT (int4not)	QTR (date_part)
BLSHIFT (int4shl)	REPEAT
BRSHIFT (int4shr)	SECOND (date_part)
BOR (int4or)	SIGN
BXOR (int4xor)	SIN
BYTE (chr)	SOUNDEX
CEIL	SQRT
COALESCE	STRIP (btrim)
COMPRESS (translate)	SUBSTR
COS	SUM
COUNT	TAN
DAY (date_part)	TRANWRD (translate)
EXP	TRIMN (rtrim)
FLOOR	UPCASE (upper)
HOUR (date_part)	WEEK[<SAS date val>, V'] (date part)
INDEX (position)	YEAR (date_part)
LOG (ln)	

SQL_FUNCTIONS= on page 185ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to Netezza. Due to incompatibility in date and time functions between Netezza and SAS, Netezza might not process them correctly. Check your results to determine whether these functions are working as expected.

DATE (current_date)	TIMEPART (cast)
DATEPART (cast)	TODAY (current_date)
DATETIME (now)	TRANSLATE
LENGTH	WEEK[<SAS date val>] (date part)
ROUND	WEEK[<SAS date val>, 'U'] (date part)
TIME (current_time)	WEEK[<SAS date val>, 'W'] (date part)

Passing Joins to Netezza

For a multiple libref join to pass to Netezza, all of these components of the LIBNAME statements must match exactly:

- user ID (USER=)
- password (PASSWORD=)
- server (SERVER=)
- port (PORT=)
- data source (DSN=, if specified)
- SQL functions (SQL_FUNCTIONS=)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see “Passing Joins to the DBMS” on page 47.

Bulk Loading and Unloading for Netezza

Loading

Overview

Bulk loading is the fastest way to insert large numbers of rows into a Netezza table. To use the bulk-load facility, specify BULKLOAD= on page 274YES. The bulk-load facility uses the Netezza Remote External Table interface to move data from the client to the Netezza Performance Server.

Here are the Netezza bulk-load data set options. For detailed information about these options, see Data Set Options for Relational Databases on page 207.

- BL_DATAFILE=
- BL_DELETE_DATAFILE=
- BL_DELIMITER=
- BL_OPTIONS= on page 255
- BL_USE_PIPE=
- BULKLOAD=

Examples

This first example shows how you can use a SAS data set, SASFLT.FLT98, to create and load a large Netezza table, FLIGHTS98:

```
libname sasflt 'SAS-library';
libname net_air netezza user=louis pwd=fromage
        server=air2 database=flights;

proc sql;
create table net_air.flights98
       (bulkload=YES bl_options='logdir "c:\temp\netlogs"')
        as select * from sasflt.flt98;
quit;
```

You can use BL_OPTIONS= to pass specific Netezza options to the bulk-loading process. The logdir option specifies the directory for the nzbad and nzlog files to be generated during the load.

This next example shows how you can append the SAS data set, SASFLT.FLT98, to the existing Netezza table, ALLFLIGHTS. The BL_USE_PIPE=NO option forces SAS/ACCESS Interface to Netezza to write data to a flat file, as specified in the BL_DATAFILE= option. Rather than deleting the data file, BL_DELETE_DATAFILE=NO causes the engine to leave it after the load has completed.

```
proc append base=net_air.allflights
 (BULKLOAD=YES
  BL_DATAFILE='/tmp/fltdata.dat'
  BL_USE_PIPE=NO
  BL_DELETE_DATAFILE=NO)
```

```
data=sasflt.flt98;
run;
```

Unloading

Overview

Bulk unloading is the fastest way to insert large numbers of rows from a Netezza table. To use the bulk-unload facility, specify BULKUNLOAD=YES. (See BULKUNLOAD= on page 275.) The bulk-unload facility uses the Netezza Remote External Table interface to move data from the client to the Netezza Performance Server into SAS.

Here are the Netezza bulk-unload data set options:

BL_DATAFILE=

BL_DELETE_DATAFILE=

BL_DELIMITER=

BL_OPTIONS=

BL_USE_PIPE=

BULKLOAD=

Examples

This first example shows how you can read the large Netezza table, FLIGHTS98, to create and populate a SAS data set, SASFLT.FLT98:

```
libname sasflt 'SAS-library';
libname net_air netezza user=louis pwd=fromage
        server=air2 database=flights;

proc sql;
create table sasflt.flt98
        as select * from net_air.flights98
       (bulkunload=YES bl_options='logdir "c:\temp\netlogs"');
quit;
```

You can use BL_OPTIONS= to pass specific Netezza options to the unload process. The logdir option specifies the directory for the nzbad and nzlog files to be generated during the unload.

This next example shows how you can append the contents of the Netezza table, ALLFLIGHTS, to an existing SAS data set, SASFLT.FLT98. The BL_USE_PIPE=NO option forces SAS/ACCESS Interface to Netezza to read data from a flat file, as specified in the BL_DATAFILE= option. Rather than deleting the data file, BL_DELETE_DATAFILE=NO causes the engine to leave it after the unload has completed.

```
proc append base=sasflt.flt98
data=net_air.allflights
 (BULKUNLOAD=YES
  BL_DATAFILE='/tmp/fltdata.dat'
  BL_USE_PIPE=NO
  BL_DELETE_DATAFILE=NO);
run;
```

Naming Conventions for Netezza

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

Since SAS 7, most SAS names can be up to 32 characters long. SAS/ACCESS Interface to Netezza supports table names and column names that contain up to 32 characters. If DBMS column names are longer than 32 characters, they are truncated to 32 characters. If truncating a column name would result in identical names, SAS generates a unique name by replacing the last character with a number. DBMS table names must be 32 characters or less because SAS does not truncate a longer name. If you already have a table name that is greater than 32 characters, it is recommended that you create a table view.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options determine how SAS/ACCESS Interface to Netezza handles case sensitivity. (For information about these options, see "Overview: LIBNAME Statement for Relational Databases " on page 93.) Netezza is not case sensitive, and all names default to lowercase.

Netezza objects include tables, views, and columns. Follow these naming conventions:

- A name must be from 1 to 128 characters long.
- A name must begin with a letter (A through Z), diacritic marks, or non-Latin characters (200-377 octal).
- A name cannot begin with an underscore (_). Leading underscores are reserved for system objects.
- Names are not case sensitive. For example, **`CUSTOMER`** and **`Customer`** are the same, but object names are converted to lowercase when they are stored in the Netezza database. However, if you enclose a name in quotation marks, it is case sensitive.
- A name cannot be a Netezza reserved word, such as WHERE or VIEW.
- A name cannot be the same as another Netezza object that has the same type.

For more information, see your *Netezza Database User's Guide*.

Data Types for Netezza

Overview

Every column in a table has a name and a data type. The data type tells Netezza how much physical storage to set aside for the column and the form in which the data is stored. This section includes information about Netezza data types, null and default values, and data conversions.

For more information about Netezza data types and to determine which data types are available for your version of Netezza, see your *Netezza Database User's Guide*.

SAS/ACCESS Interface to Netezza does not directly support TIMETZ or INTERVAL types. Any columns using these types are read into SAS as character strings.

String Data

CHAR(*n*), NCHAR(*n*)
: specifies a fixed-length column for character string data. The maximum length is 32,768 characters. NCHAR data is stored as UTF-8 in the Netezza database.

VARCHAR(*n*), NVARCHAR(*n*)
: specifies a varying-length column for character string data. The maximum length is 32,768 characters. NVARCHAR data is stored as UTF-8 in the Netezza database.

Numeric Data

BIGINT
: specifies a big integer. Values in a column of this type can range from –9223372036854775808 to +9223372036854775807.

SMALLINT
: specifies a small integer. Values in a column of this type can range from –32768 through +32767.

INTEGER
: specifies a large integer. Values in a column of this type can range from –2147483648 through +2147483647.

BYTEINT
: specifies a tiny integer. Values in a column of this type can range from –128 to +127.

DOUBLE | DOUBLE PRECISION
: specifies a floating-point number that is 64 bits long. Values in a column of this type can range from –1.79769E+308 to –2.225E-307 or +2.225E-307 to +1.79769E+308, or they can be 0. This data type is stored the same way that SAS stores its numeric data type. Therefore, numeric columns of this type require the least processing when SAS accesses them.

REAL
: specifies a floating-point number that is 32 bits long. Values in a column of this type can range from approximately –3.4E38 to –1.17E-38 and +1.17E-38 to +3.4E38.

DECIMAL | DEC | NUMERIC | NUM
: specifies a fixed-point decimal number. The precision and scale of the number determines the position of the decimal point. The numbers to the right of the decimal point are the scale, and the scale cannot be negative or greater than the precision. The maximum precision is 38 digits.

Date, Time, and Timestamp Data

SQL date and time data types are collectively called datetime values. The SQL data types for dates, times, and timestamps are listed here. Be aware that columns of these data types can contain data values that are out of range for SAS.

DATE
: specifies date values. The range is 01-01-0001 to 12-31-9999. The default format *YYYY-MM-DD* (for example, 1961-06-13). Netezza supports many other formats for entering date data. For more information, see your *Netezza Database User's Guide*.

TIME
: specifies time values in hours, minutes, and seconds to six decimal positions: hh:mm:ss[.nnnnnn]. The range is 00:00:00.000000 to 23:59:59.999999. However, due to the ODBC-style interface that SAS/ACCESS Interface to Netezza uses to communicate with the Netezza Performance Server, any fractional seconds are lost in the transfer of data from server to client.

TIMESTAMP
: combines a date and time in the default format of *yyyy-mm-dd hh:mm:ss[.nnnnnn]*. For example, a timestamp for precisely 2:25 p.m. on January 25, 1991, would be 1991-01-25-14.25.00.000000. Values in a column of this type have the same ranges as described for DATE and TIME.

Netezza Null Values

Netezza has a special value called NULL. A Netezza NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a Netezza NULL value, it interprets it as a SAS missing value.

You can define a column in a Netezza table so that it requires data. To do this in SQL, you specify a column as NOT NULL, which tells SQL to allow only a row to be added to a table if a value exists for the field. For example, NOT NULL assigned to the CUSTOMER field in the SASDEMO.CUSTOMER table does not allow a row to be added unless there is a value for CUSTOMER. When creating a Netezza table with SAS/ACCESS, you can use the DBNULL= on page 294 data set option to indicate whether NULL is a valid value for specified columns.

You can also define Netezza columns as NOT NULL DEFAULT. For more information about using the NOT NULL DEFAULT value, see your *Netezza Database User's Guide*.

Once you know whether a Netezza column enables NULLs or the host system supplies a default value for a column that is defined as NOT NULL WITH DEFAULT, you can write selection criteria and enter values to update a table. Unless a column is defined as NOT NULL or NOT NULL DEFAULT, it allows NULL values.

For more information about how SAS handles NULL values, see "Potential Result Set Differences When Processing Null Data " on page 35.

To control how the DBMS handles SAS missing character values, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to Netezza assigns to SAS variables when using the LIBNAME statement on page 93 to read from a Netezza table. These default formats are based on Netezza column attributes.

Table 23.3 *LIBNAME Statement: Default SAS Formats for Netezza Data Types*

Netezza Data Type	SAS Data Type	Default SAS Format
CHAR(*n*)*	character	$*n*.
VARCHAR(*n*)*	character	$*n*.
INTEGER	numeric	11.

Netezza Data Type	SAS Data Type	Default SAS Format
SMALLINT BYTEINT	numeric numeric	6. 4.
BIGINT	numeric	20.
DECIMAL(*p,s*)	numeric	*m.n*
NUMERIC(*p,s*)	numeric	*m.n*
REAL	numeric	none
DOUBLE	numeric	none
TIME	numeric	TIME8.
DATE	numeric	DATE9.
TIMESTAMP	numeric	DATETIME*25.6*

* *n* in Netezza data types is equivalent to *w* in SAS formats.

The following table shows the default Netezza data types that SAS/ACCESS assigns to SAS variable formats during output operations when you use the LIBNAME statement.

Table 23.4 *LIBNAME Statement: Default Netezza Data Types for SAS Variable Formats*

SAS Variable Format	Netezza Data Type
m.n	DECIMAL(*p,s*)
other numerics	DOUBLE
$*n.*	VARCHAR(*n*)*
datetime formats	TIMESTAMP
date formats	DATE
time formats	TIME

* *n* in Netezza data types is equivalent to *w* in SAS formats.

Chapter 24
SAS/ACCESS Interface to ODBC

Introduction to SAS/ACCESS Interface to ODBC

Overview

This section describes SAS/ACCESS Interface to ODBC. For a list of SAS/ACCESS features that are available in this interface, see "SAS/ACCESS Interface to ODBC: Supported Features" on page 85.

ODBC Concepts

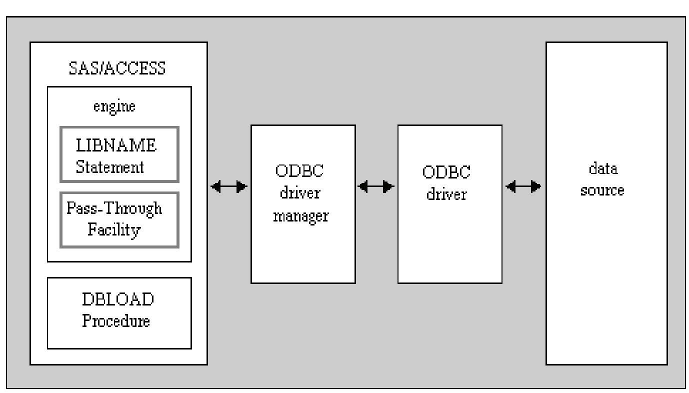
Overview

Open database connectivity (ODBC) standards provide a common interface to a variety of data sources, including dBASE, Microsoft Access, Microsoft SQL Server, Oracle, and Paradox. The goal of ODBC is to enable access to data from any application, regardless of which DBMS handles the data. ODBC accomplishes this by inserting a middle layer—consisting of an ODBC driver manager and an ODBC driver—between an application and the target DBMS. The purpose of this layer is to translate application data queries into commands that the DBMS understands. Specifically, ODBC standards define application programming interfaces (APIs) that enable applications such as SAS software to access a database. For all of this to work, both the application and the DBMS must be ODBC-compliant. This means that the application must be able to issue ODBC commands, and the DBMS must be able to respond to these.

Here are the basic components and features of ODBC.

Three components provide ODBC functionality: the client interface, the ODBC driver manager, and the ODBC driver for the data source with which you want to work, as shown below.

Figure 24.1 *The ODBC Interface to SAS*



For PC and UNIX environments, SAS provides SAS/ACCESS Interface to ODBC as the client interface. Consisting of the ODBC driver manager and the ODBC driver, the client setup with which SAS/ACCESS Interface to ODBC works is quite different between the two platforms.

ODBC on a PC Platform

On the PC side, the Microsoft ODBC Data Source Administrator is the ODBC driver manager. You can open the ODBC Data Source Administrator from the Windows control panel. Through a series of dialog boxes, you can create an ODBC data source name (DSN). You can select a particular ODBC driver for the database with which you want to work from the list of available drivers. You can then provide specific connection information for the database that the specific driver can access.

USER DSN
: specific to an individual user. It is available only to the user who creates it.

SYSTEM DSN
: not specific to an individual user. Anyone with permission to access the data source can use it.

FILE DSN
: not specific to an individual user. It can be shared among users even though it is created locally. Because this DSN is file-based, it contains all information that is required to connect to a data source.

You can create multiple DSNs in this way and then reference them in your PC-based SAS/ACCESS Interface to ODBC code.

When you use the ODBC Data Source Administrator on the PC to create your ODBC data sources, the ODBC drivers for the particular databases from which you want to enable access to data are often in the list of available drivers, especially those for the more common databases. If the ODBC driver that you want is not listed, you must work to obtain one.

ODBC on a UNIX Platform

ODBC on UNIX works a bit differently. The ODBC driver manager and ODBC drivers on the PC are available by default, so you need only plug them in. Because these components are not generally available on UNIX, you must instead work with third-party vendors to obtain them.

When you submit SAS/ACCESS Interface to ODBC code, SAS looks first for an ODBC driver manager. It checks the directories that are listed in such environment variables settings as LD_LIBRARY_PATH, LIBPATH, or SHLIB_PATH, depending on your UNIX platform. It uses the first ODBC driver manager that it finds.

The ODBC driver manager then checks .INI files—either a stand-alone ODBC.INI file, or a combination of ODBC.INI and ODBCINST.INI files—for the DSNs that you specified in your code. To make sure that the intended .INI files are referenced, you can use such environment variables settings as ODBCINI or ODBCSYSINI, depending on how your .INI files are set up. You can set up global .INI files for all your users, or you can set up .INI files for single users or groups of users. This is similar to using the ODBC Data Source Administrator to create either SYSTEM or USER DSNs for PC platforms. One or more .INI files include a section for each DSN, and each section includes specific connection information for each data source from which you ultimately want to enable access to data. Some ODBC driver vendors provide tools with which you can build one or more of your .INI files. However, editing a sample generic .INI file that is provided with the ODBC driver is often done manually.

Most database vendors (such as Sybase, Oracle, or DB2) include ODBC drivers for UNIX platforms. To use SAS/ACCESS Interface to ODBC, pair an ODBC driver manager that is based in UNIX with your ODBC driver that is also based in UNIX.. Freeware ODBC driver managers for UNIX such as unixODBC are generally available for download. Another alternative is to obtain the required ODBC client components for UNIX platforms from third-party vendors who market both ODBC drivers for various databases and an ODBC driver manager that works with these drivers. To use SAS/ACCESS Interface to ODBC, you can select any ODBC client solution that you want as long as it is ODBC-compliant.

ODBC for PC and UNIX Platforms

These concepts are common across both PC and UNIX platforms.

- ODBC uses SQL syntax for queries and statement execution, or for statements that are executed as commands. However, all databases that support ODBC are not necessarily SQL databases. For example, many databases do not have system tables. Also, the term *table* can describe a variety of items–including a file, a part of a file, a group of files, a typical SQL table, generated data, or any potential source of data. This is an important distinction. All ODBC data sources respond to a base set of SQL statements such as SELECT, INSERT, UPDATE, DELETE, CREATE, and DROP in their simplest forms. However, some databases do not support other statements and more complex forms of SQL statements.
- The ODBC standard allows for various levels of conformance that is generally categorized as low, medium, and high. As previously mentioned, the level of SQL syntax that is supported varies. Also, some driver might not support many programming interfaces. SAS/ACCESS Interface to ODBC works with API calls that conform to the lowest level of ODBC compliance, Level 1. However, it does use some Level 2 API calls if they are available.

  SAS programmers or end users must make sure that their particular ODBC driver supports the SQL syntax to be used. If the driver supports a higher level of API conformance, some advanced features are available through the PROC SQL CONNECT statement and special queries that SAS/ACCESS Interface to ODBC supports. For more information, see “Special Catalog Queries” on page 655.
- The ODBC manager and drivers return standard operation states and custom text for any warnings or errors. The state variables and their associated text are available through the SAS SYSDBRC and SYSDBMSG macro variables.

Key Considerations When Using ODBC Drivers and SAS on UNIX

SAS/ACCESS Interface to ODBC on UNIX allows SAS customers to surface data from a wide variety of external data sources. Many customers using SAS on UNIX have had success using SAS/ACCESS Interface to ODBC with their ODBC client setups. These setups consist of an ODBC driver manager and ODBC drivers for the specific data sources to which customers need access. Critical to this success are the quality and completeness of third-party ODBC client components on UNIX that customers have chosen to use.

To maximize your chances of success, your ODBC driver must comply with the ODBC 3.5 (or later) specification. It must also but support the call sequences that SAS/ACCESS Interface to ODBC sends to the driver. Specifically, your ODBC driver manager and ODBC driver must support these ODBC calls:

SQLAllocConnect	SQLFreeStmt
SQLAllocEnv	SQLGetConnectAttr
SQLAllocHandle	SQLGetConnectOption
SQLAllocStmt	SQLGetCursorName
SQLBindCol	SQLGetDiagRec
SQLBindParameter	SQLGetFunctions
SQLBulkOperations	SQLGetInfo
SQLCancel	SQLGetStmtAttr

SQLColAttribute	SQLGetStmtOption
SQLColumnPrivileges	SQLGetTypeInfo
SQLColumns	SQLMoreResults
SQLConnect	SQLNumResultCols
SQLDataSources	SQLPrepare
SQLDescribeCol	SQLPrepareW
SQLDescribeColW	SQLPrimaryKeys
SQLDisconnect	SQLProcedureColumns
SQLDriverConnect	SQLProcedures
SQLEndTran	SQLRowCount
SQLExecDirect	SQLSetConnectAttr
SQLExecDirectW	SQLSetConnectOption
SQLExecute	SQLSetEnvAttr
SQLExtendedFetch	SQLSetPos
SQLFetch	SQLSetStmtAttr
SQLFetchScroll	SQLSetStmtOption
SQLForeignKeys	SQLSpecialColumns
SQLFreeConnect	SQLStatistics
SQLFreeEnv	SQLTablePrivileges
SQLFreeHandle	SQLTables

SAS/ACCESS Interface to ODBC sends a sequence of ODBC calls to the ODBC driver that you have chosen. The types and sequence in which these calls are made are compliant with the ODBC specification. If your ODBC driver fails to return the correct result or fails to work with SAS/ACCESS Interface to ODBC, here are your options.

- Make sure that you are running the current versions of your ODBC client components.
- Try to connect using a query tool that is not SAS. Most third-party ODBC driver and driver manager sources include such a query tool with their offerings. However, keep in mind that in some cases you might be able to connect using a query tool (that is not SAS) but not with SAS/ACCESS Interface to ODBC. This is because SAS calls might make a wider range of ODBC calls than an ODBC query tool that is not SAS would make.
- SAS Technical Support offers additional tools that can help you identify the root of your ODBC-related client problems and subsequently debug them.
- You can address some ODBC client issues by using certain SAS/ACCESS Interface to ODBC options or alternative engines to SAS/ACCESS Interface to ODBC.
- Once you have determined that your ODBC client issues are not related to SAS, you need to report your debugging results to your ODBC client providers. If you received your ODBC client components from a commercial ODBC driver vendor, you can work through that vendor’s technical support. If you use freeware or open-source ODBC client components—where formal technical support is not always available—your only recourse might be to communicate with the freeware user community.

SAS has not validated all ODBC drivers on the market and therefore makes no claims of certification or support.

LIBNAME Statement Specifics for ODBC

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to ODBC supports and includes examples. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing ODBC.

LIBNAME *libref* **odbc** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`odbc`
specifies the SAS/ACCESS engine name for the ODBC interface.

connection-options
provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. When you use the LIBNAME statement, you can connect to ODBC in many different ways. Specify *only one* of these methods for each connection because they are mutually exclusive.

- USER=, PASSWORD=, DATASRC=
- COMPLETE=
- NOPROMPT=
- PROMPT=
- READBUFF=
- REQUIRED=

Here is how these options are defined.

USER=<'>*user-name*<'>
lets you connect to an ODBC database with a user ID that is different from the default ID. USER= is optional. UID= is an alias for this option.

PASSWORD=<'>*password*<'>
specifies the ODBC password that is associated with your user ID. PASSWORD= is optional. PWD is an alias for this option. If it contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. If you do not want to enter your DB2 password in uncoded text on this statement, see PROC PWENCODE in the *Base SAS Procedures Guide* for a method to encode it.

DATASRC=<'>*ODBC-data-source*<'>
specifies the ODBC data source to which you want to connect. For PC platforms, data sources must be configured by using the ODBC icon in the Windows Control Panel. For UNIX platforms, data sources must be configured by

modifying the .odbc.ini file. DSN= is an alias for this option that indicates that the connection is attempted using the ODBC SQLConnect API, which requires a data source name. You can also use a user ID and password with DSN=. If you want to use an ODBC file DSN, instead of supplying DATASRC=<'>*ODBC-data-source*<'>, use the PROMPT= or NOPROMPT= option, followed by "*filedsn*=(*name-of-your-file-dsn*);". For example:

```
libname mydblib odbc noprompt="filedsn=d:\share\msafiledsn.dsn;";
```

Alias: DATABASE=, DB=, DSN=

COMPLETE=<'>*ODBC-connection-options*<'>
: specifies connection options for your data source or database. Separate multiple options with a semicolon. When connection succeeds, the complete connection string is returned in the SYSDBMSG macro variable. If you do not specify enough correct connection options, you are prompted with a dialog box that displays the values from the COMPLETE= connection string. You can edit any field before you connect to the data source. This option is not supported on UNIX platforms. See your ODBC driver documentation for more details.

NOPROMPT=<'>*ODBC-connection-options*<'>
: specifies connection options for your data source or database. Separate multiple options with a semicolon. If you do not specify enough correct connection options, an error is returned. No dialog box is displayed to help you complete the connection string.

PROMPT=<'>*ODBC-connection-information*<'>
: specifies connection options for your data source or database. Separate multiple options with a semicolon. When connection succeeds, the complete connection string is returned in the SYSDBMSG macro variable. PROMPT= does not immediately try to connect to the DBMS. Instead, it displays a dialog box that contains the values that you entered in the PROMPT= connection string. You can edit values or enter additional values in any field before you connect to the data source. This option is not supported on UNIX platforms.

READBUFF= *number-of-rows*
: Use this argument to improve the performance of most queries to ODBC. By setting the value of the READBUFF= argument in your SAS programs, you can find the optimal number of rows for a specified query on a specified table. The default buffer size is one row per fetch. The maximum is 32,767 rows per fetch, although a practical limit for most applications is less and depends on the available memory.

REQUIRED=<'>*ODBC-connection-options*<'>
: specifies connection options for your data source or database. Separate multiple options with a semicolon. When connection succeeds, the complete connection string is returned in the SYSDBMSG macro variable. If you do not specify enough correct connection options, a dialog box prompts you for the connection options. REQUIRED= lets you modify only required fields in the dialog box. This option is not supported on UNIX platforms.

See your ODBC driver documentation for a list of the ODBC connection options that your ODBC driver supports.

These ODBC connection options are not supported on UNIX.

- BULKCOPY=
- COMPLETE=
- PROMPT=

- REQUIRED=

LIBNAME-options

define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to ODBC, with the applicable default values. For more detail about these options, see "LIBNAME Options for Relational Databases" on page 99.

Table 24.1 *SAS/ACCESS LIBNAME Options for ODBC*

Option	Default Value
ACCESS= on page 99	none
AUTHDOMAIN= on page 102	none
AUTOCOMMIT= on page 103	data-source specific
BL_LOG= on page 105	none
BL_OPTIONS= on page 106	none
BULKLOAD= on page 108	NO
CONNECTION= on page 113	data-source specific
CONNECTION_GROUP= on page 118	none
CURSOR_TYPE= on page 119	FORWARD_ONLY
DBCOMMIT= on page 124	1000 when inserting rows; 0 when updating rows
DBCONINIT= on page 125	none
DBCONTERM= on page 126	none
DBCREATE_TABLE_OPTS= on page 128	none
DBGEN_NAME= on page 128	DBMS
DBINDEX= on page 129	YES
DBLIBINIT= on page 130	none
DBLIBTERM= on page 131	none
DBMAX_TEXT= on page 133	1024
DBMSTEMP= on page 133	NO
DBNULLKEYS= on page 135	YES

Option	Default Value
DBPROMPT= on page 136	NO
DBSLICEPARM= on page 139	THREADED_APPS,2 or 3
DEFER= on page 141	NO
DELETE_MULT_ROWS= on page 143	NO
DIRECT_EXE= on page 144	none
DIRECT_SQL= on page 145	YES
IGNORE_READ_ONLY_COLUMNS= on page 150	NO
INSERT_SQL= on page 152	data-source specific
INSERTBUFF= on page 153	1
KEYSET_SIZE= on page 155	0
LOGIN_TIMEOUT= on page 159	0
MULTI_DATASRC_OPT= on page 160	NONE
PRESERVE_COL_NAMES= on page 167	see "Naming Conventions for ODBC" on page 668
PRESERVE_TAB_NAMES = on page 168	see "Naming Conventions for ODBC" on page 668
QUALIFIER= on page 170	none
QUERY_TIMEOUT= on page 172	0
QUOTE_CHAR= on page 173	none
READ_ISOLATION_LEVEL= on page 175	RC (see "Locking in the ODBC Interface" on page 666)
READ_LOCK_TYPE= on page 176	ROW
READBUFF= on page 174	0
REREAD_EXPOSURE= on page 179	NO
SCHEMA= on page 180	none
SPOOL= on page 184	YES
SQL_FUNCTIONS= on page 185	none

Option	Default Value
SQL_FUNCTIONS_COPY= on page 188	none
STRINGDATES= on page 190	NO
TRACE= on page 195	NO
TRACEFILE= on page 196	none
UPDATE_ISOLATION_LEVEL= on page 196	RC (see "Locking in the ODBC Interface" on page 666)
UPDATE_LOCK_TYPE= on page 197	ROW
UPDATE_MULT_ROWS= on page 199	NO
UPDATE_SQL= on page 200	driver-specific
USE_ODBC_CL= on page 201	NO
UTILCONN_TRANSIENT= on page 202	NO

ODBC LIBNAME Statement Examples

In this example, USER=, PASSWORD=, and DATASRC= are connection options.

```
libname mydblib odbc user=testuser password=testpass datasrc=mydatasource;
```

In this next example, the libref MYLIB uses the ODBC engine to connect to an Oracle database. The connection options are USER=, PASSWORD=, and DATASRC=.

```
libname mydblib odbc datasrc=orasrvr1 user=testuser  password=testpass;

proc print data=mydblib.customers;
   where state='CA';
run;
```

In the next example, the libref MYDBLIB uses the ODBC engine to connect to a Microsoft SQL Server database. The connection option is NOPROMPT=.

```
libname mydblib odbc
   noprompt="uid=testuser;pwd=testpass;dsn=sqlservr;"
   stringdates=yes;

proc print data=mydblib.customers;
   where state='CA';
run;
```

Data Set Options for ODBC

All SAS/ACCESS data set options in this table are supported for ODBC. Default values are provided where applicable. For general information about this feature, see Data Set Options for Relational Databases on page 207.

Table 24.2 *SAS/ACCESS Data Set Options for ODBC*

Option	Default Value
BULKLOAD= on page 274	LIBNAME option setting
CURSOR_TYPE= on page 278	LIBNAME option setting
DBCOMMIT= on page 280	LIBNAME option setting
DBCONDITION= on page 281	none
DBCREATE_TABLE_OPTS= on page 285	LIBNAME option setting
DBFORCE= on page 286	NO
DBGEN_NAME= on page 287	DBMS
DBINDEX= on page 288	LIBNAME option setting
DBKEY= on page 290	none
DBLABEL= on page 291	NO
DBMASTER= on page 293	none
DBMAX_TEXT= on page 293	1024
DBNULL= on page 294	YES
DBNULLKEYS= on page 295	LIBNAME option setting
DBPROMPT= on page 296	LIBNAME option setting
DBSASLABEL= on page 137	COMPAT
DBSASTYPE= on page 298	see "Data Types for ODBC " on page 668
DBSLICE= on page 299	none
DBSLICEPARM= on page 301	THREADED_APPS,2 or 3
DBTYPE= on page 303	see "Data Types for ODBC " on page 668

Option	Default Value
ERRLIMIT= on page 308	1
IGNORE_READ_ONLY_COLUMNS= on page 311	NO
INSERT_SQL= on page 313	LIBNAME option setting
INSERTBUFF= on page 314	LIBNAME option setting
KEYSET_SIZE= on page 315	LIBNAME option setting
NULLCHAR= on page 329	SAS
NULLCHARVAL= on page 330	a blank character
PRESERVE_COL_NAMES= on page 336	LIBNAME option setting
QUALIFIER= on page 338	LIBNAME option setting
QUERY_TIMEOUT= on page 339	LIBNAME option setting
READ_ISOLATION_LEVEL= on page 340	LIBNAME option setting
READ_LOCK_TYPE= on page 341	LIBNAME option setting
READBUFF= on page 339	LIBNAME option setting
SASDATEFMT= on page 342	none
SCHEMA= on page 344	LIBNAME option setting
UPDATE_ISOLATION_LEVEL= on page 371	LIBNAME option setting
UPDATE_LOCK_TYPE= on page 371	LIBNAME option setting
UPDATE_SQL= on page 373	LIBNAME option setting

SQL Pass-Through Facility Specifics for ODBC

Key Information

For general information about this feature, see "SQL Pass-Through Facility" on page 401. ODBC examples are available.

Here are the SQL pass-through facility specifics for the ODBC interface.

- The *dbms-name* is **ODBC**.
- The CONNECT statement is required.

- PROC SQL supports multiple connections to ODBC. If you use multiple simultaneous connections, you must use the *alias* argument to identify the different connections. If you do not specify an alias, the default **odbc** alias is used. The functionality of multiple connections to the same ODBC data source might be limited by the particular data source driver.
- The CONNECT statement *database-connection-arguments* are identical to its LIBNAME connection-options on page 646. Not all ODBC drivers support all of these arguments. See your driver documentation for more information.
- On some DBMSs, the *DBMS-SQL-query* argument can be a DBMS-specific SQL EXECUTE statement that executes a DBMS stored procedure. However, if the stored procedure contains more than one query, only the first query is processed.
- These options are available with the CONNECT statement. For information, see the LIBNAME statement on page 93 section.
  - AUTOCOMMIT= on page 103
  - CURSOR_TYPE= on page 119
  - KEYSET_SIZE= on page 155
  - QUERY_TIMEOUT= on page 172
  - READBUFF= on page 174
  - READ_ISOLATION_LEVEL= on page 175
  - TRACE= on page 195
  - TRACEFILE= on page 196
  - USE_ODBC_CL= on page 201
  - UTILCONN_TRANSIENT= on page 202

CONNECT Statement Examples

These examples use ODBC to connect to a data source that is configured under the data source name **User's Data** using the alias USER1. The first example uses the connection method that is guaranteed to be present at the lowest level of ODBC conformance. DATASRC= names can contain quotation marks and spaces.

```
proc sql;
   connect to ODBC as user1
   (datasrc="User's Data" user=testuser password=testpass);
```

This example uses the connection method that represents a more advanced level of ODBC conformance. It uses the input dialog box that is provided by the driver. The DATASRC= and USER= arguments are within the connection string. The SQL pass-through facility therefore does not parse them but instead passes them to the ODBC manager.

```
proc sql;
   connect to odbc as user1
   (required = "dsn=User's Data;uid=testuser");
```

This example enables you to select any data source that is configured on your machine. The example uses the connection method that represents a more advanced level of ODBC conformance, Level 1. When connection succeeds, the connection string is returned in the SQLXMSG and SYSDBMSG macro variables and can be stored if this method is used to configure a connection for later use.

```
proc sql;
   connect to odbc (required);
```

This next example prompts you to specify the information that is required to make a connection to the DBMS. You are prompted to supply the data source name, user ID, and password in the dialog boxes that are displayed.

```
proc sql;
   connect to odbc (prompt);
```

Connection to Component Examples

This example sends an Oracle SQL query (presented in highlighted text) to the Oracle database for processing. The results from the query serve as a virtual table for the PROC SQL FROM clause. In this example MYCON is a connection alias.

```
proc sql;
connect to odbc as mycon
   (datasrc=ora7 user=testuser password=testpass);

select *
   from connection to mycon
      (select empid, lastname, firstname,
      hiredate, salary
      from sasdemo.employees
         where hiredate>='31.12.1988');

disconnect from mycon;
quit;
```

This next example gives the previous query a name and stores it as the SQL view Samples.Hires88. The CREATE VIEW statement appears highlighted.

```
libname samples 'SAS-library';

proc sql;
connect to odbc as mycon
   (datasrc=ora7 user=testuser password=testpass);

create view samples.hires88 as
 select *
    from connection to mycon
      (select empid, lastname, firstname,
       hiredate, salary from sasdemo.employees
       where hiredate>='31.12.1988');

disconnect from mycon;
quit;
```

This example connects to Microsoft Access and creates a view NEWORDERS from all columns in the ORDERS table.

```
proc sql;
   connect to odbc as mydb
      (datasrc=MSAccess7);
   create view neworders as
    select * from connection to mydb
      (select * from orders);
```

```
disconnect from mydb;
quit;
```

This next example sends an SQL query to Microsoft SQL Server, configured under the data source name **`SQL Server`**, for processing. The results from the query serve as a virtual table for the PROC SQL FROM clause.

```
proc sql;
   connect to odbc as mydb
      (datasrc="SQL Server" user=testuser password=testpass);
   select * from connection to mydb
      (select CUSTOMER, NAME, COUNTRY
          from CUSTOMERS
          where COUNTRY <> 'USA');
quit;
```

This example returns a list of the columns in the CUSTOMERS table.

```
proc sql;
   connect to odbc as mydb
      (datasrc="SQL Server" user=testuser password=testpass);
   select * from connection to mydb
      (ODBC::SQLColumns (, , "CUSTOMERS"));
quit;
```

Special Catalog Queries

SAS/ACCESS Interface to ODBC supports the following special queries. Many databases provide or use system tables that allow queries to return the list of available tables, columns, procedures, and other useful information. ODBC provides much of this functionality through special application programming interfaces (APIs) to accommodate databases that do not follow the SQL table structure. You can use these special queries on SQL and non-SQL databases.

Here is the general format of the special queries:

ODBC::SQLAPI “*parameter 1*”,”*parameter n*”

ODBC::
: required to distinguish special queries from regular queries.

SQLAPI
: is the specific API that is being called. Neither ODBC:: nor SQLAPI are case sensitive.

"*parameter n*"
: a quoted string that is delimited by commas.

Within the quoted string, two characters are universally recognized: the percent sign (%) and the underscore (_). The percent sign matches any sequence of zero or more characters; the underscore represents any single character. Each driver also has an escape character that can be used to place characters within the string. See the driver documentation to determine the valid escape character.

The values for the special query arguments are DBMS-specific. For example, you supply the fully qualified table name for a “*Catalog*” argument. In dBase, the value of “*Catalog*” might be **`c:\dbase\tst.dbf`** and in SQL Server, the value might be **`test.customer`**. In addition, depending on the DBMS that you are using, valid values for a “*Schema*” argument might be a user ID, a database name, or a library. All arguments are optional. If you specify some but not all arguments within a parameter, use a comma to indicate the omitted arguments. If you do not specify any parameters, commas are not necessary. Special queries are not available for all ODBC drivers.

ODBC supports these special queries:

ODBC::SQLColumns <*"Catalog"*, *"Schema"*, *"Table-name"*, *"Column-name"*>
: returns a list of all columns that match the specified arguments. If no arguments are specified, all accessible column names and information are returned.

ODBC::SQLColumnPrivileges <*"Catalog"*, *"Schema"*, *"Table-name"*, *"Column-name"*>
: returns a list of all column privileges that match the specified arguments. If no arguments are specified, all accessible column names and privilege information are returned.

ODBC::SQLDataSources
: returns a list of database aliases to which ODBC is connected.

ODBC::SQLDBMSInfo
: returns a list of DB2 databases (DSNs) to which ODBC is connected. It returns one row with two columns that describe the DBMS name (such as SQL Server or Oracle) and the corresponding DBMS version.

ODBC::SQLForeignKeys <*"PK-catalog"*, *"PK-schema"*, *"PK-table-name"*, *"FK-catalog"*, *"FK-schema"*, *"FK-table-name"*>
: returns a list of all columns that comprise foreign keys that match the specified arguments. If no arguments are specified, all accessible foreign key columns and information are returned.

ODBC::SQLGetTypeInfo
: returns information about the data types that are supported in the data source.

ODBC::SQLPrimaryKeys <*"Catalog"*, *"Schema"*, *"Table-name"*>
: returns a list of all columns that compose the primary key that matches the specified table. A primary key can be composed of one or more columns. If no table name is specified, this special query fails.

ODBC::SQLProcedures <*"Catalog"*, *"Schema"*, *"Procedure-name"*>
: returns a list of all procedures that match the specified arguments. If no arguments are specified, all accessible procedures are returned.

ODBC::SQLProcedureColumns <*"Catalog"*, *"Schema"*, *"Procedure-name"*, *"Column-name"*>
: returns a list of all procedure columns that match the specified arguments. If no arguments are specified, all accessible procedure columns are returned.

ODBC::SQLSpecialColumns <*"Identifier-type"*, *"Catalog-name"*, *"Schema-name"*, *"Table-name"*, *"Scope"*, *"Nullable"*>
: returns a list of the optimal set of columns that uniquely identify a row in the specified table.

ODBC::SQLStatistics <*"Catalog"*, "*Schema"*, *"Table-name"*>
: returns a list of the statistics for the specified table name, with options of SQL_INDEX_ALL and SQL_ENSURE set in the SQLStatistics API call. If the table name argument is not specified, this special query fails.

ODBC::SQLTables <"*Catalog"*, *"Schema"*, *"Table-name"*, *"Type"*>
: returns a list of all tables that match the specified arguments. If no arguments are specified, all accessible table names and information are returned.

ODBC::SQLTablePrivileges <*"Catalog"*, *"Schema"*, *"Table-name"*>
: returns a list of all tables and associated privileges that match the specified arguments. If no arguments are specified, all accessible table names and associated privileges are returned.

Autopartitioning Scheme for ODBC

Overview

Autopartitioning for SAS/ACCESS Interface to ODBC is a modulo (MOD) function method. For general information about this feature, see “Autopartitioning Techniques in SAS/ACCESS” on page 63.

Autopartitioning Restrictions

SAS/ACCESS Interface to ODBC places additional restrictions on the columns that you can use for the partitioning column during the autopartitioning phase. Here is how columns are partitioned.

- SQL_INTEGER, SQL_BIT, SQL_SMALLINT, and SQL_TINYINT columns are given preference.
- You can use SQL_DECIMAL, SQL_DOUBLE, SQL_FLOAT, SQL_NUMERIC, and SQL_REAL columns for partitioning under these conditions:
  - The ODBC driver supports converting these types to SQL_INTEGER by using the INTEGER cast function.
  - The precision minus the scale of the column is greater than 0 but less than 10—that is, 0<(*precision-scale*)<10.

The exception to the above rule is for Oracle SQL_DECIMAL columns. As long as the scale of the SQL_DECIMAL column is 0, you can use the column as the partitioning column.

Nullable Columns

If you select a nullable column for autopartitioning, the **OR<*column-name*>IS NULL** SQL statement is appended at the end of the SQL code that is generated for the threaded Read. This ensures that any possible NULL values are returned in the result set. Also, if the column to be used for the partitioning is SQL_BIT, the number of threads are automatically changed to two, regardless of how the DBSLICEPARM= option is set.

Using WHERE Clauses

Autopartitioning does not select a column to be the partitioning column if it appears in the WHERE clause. For example, the following DATA step could not use a threaded Read to retrieve the data because all numeric columns in the table are in the WHERE clause:

```
data work.locemp;
  set trlib.MYEMPS;
  where EMPNUM<=30 and ISTENURE=0 and
        SALARY<=35000 and NUMCLASS>2;
run;
```

Using DBSLICEPARM=

SAS/ACCESS Interface to ODBC defaults to three threads when you use autopartitioning but do not specify a maximum number of threads in DBSLICEPARM= on page 139 to use for the threaded Read.

Using DBSLICE=

You might achieve the best possible performance when using threaded Reads by specifying the DBSLICE= option for ODBC in your SAS operation. This is especially true if your DBMS supports multiple database partitions and provides a mechanism to allow connections to individual partitions. If your DBMS supports this concept, you can configure an ODBC data source for each partition and use the DBSLICE= clause to specify both the data source and the WHERE clause for each partition, as shown in this example:

```
proc print data=trilib.MYEMPS(DBSLICE=(DSN1="EMPNUM BETWEEN 1 AND 33"
DSN2="EMPNUM BETWEEN 34 AND 66"
DSN3="EMPNUM BETWEEN 67 AND 100"));
run;
```

See your DBMS or ODBC driver documentation for more information about configuring for multiple partition access. You can also see "Configuring SQL Server Partitioned Views for Use with DBSLICE=" on page 658 for an example of configuring multiple partition access to a table.

Using the DATASOURCE= syntax is not required to use DBSLICE= with threaded Reads for the ODBC interface. The methods and examples described in DBSLICE= work well in cases where the table that you want to read is not stored in multiple partitions in your DBMS. These methods also give you flexibility in column selection. For example, if you know that the STATE column in your employee table only contains a few distinct values, you can customize your DBSLICE= clause accordingly:

```
datawork.locemp;
set trlib2.MYEMP(DBSLICE=("STATE='FL'" "STATE='GA'"
  "STATE='SC'" "STATE='VA'" "STATE='NC'"));
where EMPNUM<=30 and ISTENURE=0 and SALARY<=35000 and NUMCLASS>2;
run;
```

Configuring SQL Server Partitioned Views for Use with DBSLICE=

Microsoft SQL Server implements multiple partitioning by creating a global view across multiple instances of a Microsoft SQL Server database. For this example, assume that Microsoft SQL Server has been installed on three separate machines (SERVER1, SERVER2, SERVER3), and three ODBC data sources (SSPART1, SSPART2, SSPART3) have been configured against these servers. Also, a linked server definition for each of these servers has been defined. This example uses SAS to create the tables and associated views, but you can accomplish this outside of the SAS environment.

1. Create a local SAS table to build the Microsoft SQL Server tables.

   ```
   data work.MYEMPS;
   format HIREDATE mmddyy 0. SALARY 9.2
      NUMCLASS 6. GENDER $1. STATE $2. EMPNUM 10.;
   do EMPNUM=1 to 100;
   ```

```
        morf=mod(EMPNUM,2)+1;
        if(morf eq 1) then
            GENDER='F';
        else
            GENDER='M';
        SALARY=(ranuni(0)*5000);
        HIREDATE=int(ranuni(13131)*3650);
        whatstate=int(EMPNUM/5);
        if(whatstate eq 1) then
            STATE='FL';
        if(whatstate eq 2) then
            STATE='GA';
        if(whatstate eq 3) then
            STATE='SC';
        if(whatstate eq 4) then
            STATE='VA';
        else
            state='NC';
        ISTENURE=mod(EMPNUM,2);
        NUMCLASS=int(EMPNUM/5)+2;
        output;
   end;
   run;
```

2. Create a table on each of the SQL server databases with the same table structure, and insert one–third of the overall data into each table. These table definitions also use CHECK constraints to enforce the distribution of the data on each of the subtables of the target view.

```
libname trlib odbc user=ssuser pw=sspwd dsn=sspart1;
proc datasets library=trlib;
  delete MYEMPS1;run;
run;
data trlib.MYEMPS1(drop=morf whatstate
   DBTYPE=(HIREDATE="datetime" SALARY="numeric(8,2)"
   NUMCLASS="smallint" GENDER="char(1)" ISTENURE="bit" STATE="char(2)"
   EMPNUM="int NOT NULL Primary Key CHECK (EMPNUM BETWEEN 0 AND 33)"));
set work.MYEMPS;
where (EMPNUM BETWEEN 0 AND 33);
run;

libname trlib odbc user=ssuer pw=sspwd dsn=sspart2;
proc datasets library=trlib;
  delete MYEMPS2;run;
data trlib.MYEMPS2(drop=morf whatstate
   DBTYPE=(HIREDATE="datetime" SALARY="numeric(8,2)"
   NUMCLASS="smallint" GENDER="char(1)" ISTENURE="bit" STATE="char(2)"
   EMPNUM="int NOT NULL Primary Key CHECK (EMPNUM BETWEEN 34 AND 66)"));
set work.MYEMPS;
where (EMPNUM BETWEEN 34 AND 66);
run;

libname trlib odbc user=ssuer pw=sspwd dsn=sspart3;
proc datasets library=trlib;
  delete MYEMPS3;run;
data trlib.MYEMPS3(drop=morf whatstate
   DBTYPE=(HIREDATE="datetime" SALARY="numeric(8,2)"
```

```
   NUMCLASS="smallint" GENDER="char(1)" ISTENURE="bit" STATE="char(2)"
   EMPNUM="int NOT NULL Primary Key CHECK (EMPNUM BETWEEN 67 AND 100)"));
set work.MYEMPS;
where (EMPNUM BETWEEN 67 AND 100);
run;
```

3. Create a view using the UNION ALL construct on each Microsoft SQL Server instance that references the other two tables. This creates a global view that references the entire data set.

```
/*SERVER1,SSPART1*/
proc sql noerrorstop;
connect to odbc (UID=ssuser PWD=sspwd DSN=SSPART1);
execute (drop view MYEMPS) by odbc;
execute (create view MYEMPS AS
         SELECT * FROM users.ssuser.MYEMPS1
         UNION ALL
         SELECT * FROM SERVER2.users.ssuser.MYEMPS2
         UNION ALL
         SELECT * FROM SERVER3.users.ssuser.MYEMPS3) by odbc;
quit;

/*SERVER2,SSPART2*/
proc sql noerrorstop;
connect to odbc (UID=ssuser PWD=sspwd DSN=SSPART2);
execute (drop view MYEMPS) by odbc;
execute (create view MYEMPS AS
         SELECT * FROM users.ssuser.MYEMPS2
         UNION ALL
         SELECT * FROM SERVER1.users.ssuser.MYEMPS1
         UNION ALL
         SELECT * FROM SERVER3.users.ssuser.MYEMPS3) by odbc;
quit;

/*SERVER3,SSPART3*/
proc sql noerrorstop;
connect to odbc (UID=ssuser PWD=sspwd DSN=SSPART3);
execute (drop view MYEMPS) by odbc;
execute (create view MYEMPS AS
         SELECT * FROM users.ssuser.MYEMPS3
         UNION ALL
         SELECT * FROM SERVER2.users.ssuser.MYEMPS2
         UNION ALL
         SELECT * FROM SERVER1.users.ssuser.MYEMPS1) by odbc;
quit;
```

4. Set up your SAS operation to perform the threaded Read. The DBSLICE= option contains the Microsoft SQL Server partitioning information.

```
proc print data=trlib.MYEMPS(DBLICE=(sspart1="EMPNUM BETWEEN 1 AND 33"
sspart2="EMPNUM BETWEEN 34 AND 66"
sspart3="EMPNUM BETWEEN 67 AND 100"));
run;
```

This configuration lets the ODBC interface access the data for the MYEMPS view directly from each subtable on the corresponding Microsoft SQL Server instance. The data is inserted directly into each subtable, but this process can also be accomplished by using the global view to divide up the data. For example, you can create empty tables and then create the view as seen in the example with the UNION ALL construct. You can then insert the data into the view MYEMPS. The CHECK constraints allow the Microsoft SQL Server query processor to determine which subtables should receive the data.

Other tuning options are available when you configure Microsoft SQL Server to use partitioned data. For more information, see the "Creating a Partitioned View" and "Using Partitioned Views" sections in *Creating and Maintaining Databases (SQL Server 2000)*.

DBLOAD Procedure Specifics for ODBC

Overview

See DBLOAD Procedure on page 901 for general information about this feature. ODBC examples on page 662 are available.

SAS/ACCESS Interface to ODBC supports all DBLOAD procedure statements on page 902 (except ACCDESC=) in batch mode. Here are the DBLOAD procedure specifics for ODBC:

- The DBLOAD step DBMS= value is `ODBC`.
- Here are the database description statements that PROC DBLOAD uses:

DSN= <'>*ODBC-data-source*<'>;
: specifies the name of the data source in which you want to store the new ODBC table. The *data-source* is limited to eight characters.

  The data source that you specify must already exist. If the data source name contains the _, $, @, or # special character, you must enclose it in quotation marks. The ODBC standard recommends against using special characters in data source names, however.

USER= <'>*user name*<'>;
: lets you connect to an ODBC database with a user ID that is different from the default ID. USER= is optional in ODBC. If you specify USER=, you must also specify PASSWORD=. If USER= is omitted, your default user ID is used.

PASSWORD=<'>*password*<'>;
: specifies the ODBC password that is associated with your user ID.

  PASSWORD= is optional in ODBC because users have default user IDs. If you specify USER=, you must specify PASSWORD=.

Note: If you do not want to enter your ODBC password in uncoded text on this statement, see PROC PWENCODE in the *Base SAS Procedures Guide* for a method to encode it.

BULKCOPY= YES|NO;
: determines whether SAS uses the Microsoft Bulk Copy facility to insert data into a DBMS table (Microsoft SQL Server only). The default value is NO.

  The Microsoft Bulk Copy (BCP) facility lets you efficiently insert rows of data into a DBMS table as a unit. As the ODBC interface sends each row of data to BCP, the data is written to an input buffer. When you have inserted all rows or the buffer reaches a certain size (the DBCOMMIT= data set option determines this), all rows are inserted as a unit into the table, and the data is committed to the table.

You can also set the DBCOMMIT=*n* option to commit rows after every *n* insertions.

If an error occurs, a message is written to the SAS log, and any rows that have been inserted in the table before the error are rolled back.

Note: BULKCOPY= is not supported on UNIX.

- Here is the TABLE= statement:

  TABLE= <*authorization-id.*>*table-name*;
  identifies the table or view that you want to use to create an access descriptor. The TABLE= statement is required.

  The *authorization-id* is a user ID or group ID that is associated with the table.

- Here is the NULLS statement:

  NULLS *variable-identifier-1* =Y|N|D < . . . *variable-identifier-n* =Y|N|D >;
  enables you to specify whether the columns that are associated with the listed SAS variables allow NULL values. By default, all columns accept NULL values.

  The NULLS statement accepts any one of these three values:

  - Y – specifies that the column accepts NULL values. This is the default.
  - N – specifies that the column does not accept NULL values.
  - D – specifies that the column is defined as NOT NULL WITH DEFAULT.

Examples

This example creates a new ODBC table, TESTUSER.EXCHANGE, from the DLIB.RATEOFEX data file. You must be granted the appropriate privileges in order to create new ODBC tables or views.

```
proc dbload dbms=odbc data=dlib.rateofex;
   dsn=sample;
   user='testuser';
   password='testpass';
   table=exchange;
   rename fgnindol=fgnindollars
          4=dollarsinfgn;
   nulls updated=n fgnindollars=n
         dollarsinfgn=n country=n;
   load;
run;
```

This next example sends only an ODBC SQL GRANT statement to the SAMPLE database and does not create a new table. Therefore, the TABLE= and LOAD statements are omitted.

```
proc dbload dbms=odbc;
   user='testuser';
   password='testpass';
   dsn=sample;
   sql grant select on testuser.exchange
```

```
      to dbitest;
run;
```

Temporary Table Support for ODBC

Overview

For general information about this features, see “Temporary Table Support for SAS/ACCESS ” on page 42

Establishing a Temporary Table

When you want to use temporary tables that persist across SAS procedures and DATA steps with ODBC, you must use the CONNECTION=SHARED LIBNAME option. When you do this, the temporary table is available for processing until the libref is closed.

Terminating a Temporary Table

You can drop a temporary table at any time, or allow it to be implicitly dropped when the connection is terminated. Temporary tables do not persist beyond the scope of a single connection.

Examples

Using the Internat sample table, this example creates a temporary table, #LONDON, with Microsoft SQL Server that contains information about flights that flew to London. This table is then joined with a larger SQL Server table that lists all flights, March, but matched only on flights that flew to London.

```
libname samples odbc dsn=lupinss uid=dbitest pwd="dbigrp1" connection=shared;

data samples.'#LONDON'n;
   set work.internat;
   where dest='LON';
run;

proc sql;
   select b.flight, b.dates, b.depart, b.orig
          from samples.'#LONDON'n a, samples.march b
          where a.dest=b.dest;
quit;
```

In this example a temporary table called New is created with Microsoft SQL Server. The data from this table is then appended to an existing SQL Server table named Inventory.

```
libname samples odbc dsn=lupinss uid=dbitest pwd="dbigrp1" connection=shared;

data samples.inventory(DBTYPE=(itemnum='char(5)' item='varchar(30)'
                       quantity='numeric'));
   itemnum='12001';
      item='screwdriver';
```

```
      quantity=15;
      output;
   itemnum='12002';
      item='hammer';
      quantity=25:
      output;
   itemnum='12003';
      item='sledge hammer';
      quantity=10;
      output;
   itemnum='12004';
      item='saw';
      quantity=50;
      output;
   itemnum='12005';
      item='shovel';
      quantity=120;
      output;
run;

data samples.'#new'n(DBTYPE=(itemnum='char(5)' item='varchar(30)'
                     quantity='numeric'));
   itemnum='12006';
      item='snow shovel';
      quantity=5;
      output;
   itemnum='12007';
      item='nails';
      quantity=500;
      output;
run;

proc append base=samples.inventory data=samples.'#new'n;
run;

proc print data=samples.inventory;
run;
```

This example demonstrates the use of a temporary table using the SQL pass-through facility.

```
proc sql;
   connect to odbc as test (dsn=lupinss uid=dbitest
                            pwd=dbigrp1 connection=shared);
   execute (create table #FRANCE (flight char(3), dates datetime,
            dest char(3))) by test;

   execute (insert #FRANCE select flight, dates, dest from internat
            where dest like '%FRA%') by test;
   select * from connection to test (select * from #FRANCE);
quit;
```

Passing SAS Functions to ODBC

SAS/ACCESS Interface to ODBC passes the following SAS functions to the data source for processing if the DBMS server supports this function. Where the ODBC function name differs from the SAS SQL function name, the ODBC name appears in parentheses. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

ABS	LOG10
ARCOS	LOWCASE
ARSIN	MAX
ATAN	MIN
AVG	SIGN
CEIL	SIN
COS	SQRT
COUNT	STRIP
EXP	SUM
FLOOR	TAN
LOG	UPCASE

SQL_FUNCTIONS=ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. (See SQL_FUNCTIONS= on page 46.) Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to ODBC. Due to incompatibility in date and time functions between ODBC and SAS, ODBC might not process them correctly. Check your results to determine whether these functions are working as expected.

BYTE (CHAR)	REPEAT
COMPRESS (REPLACE)	SECOND
DATE (CURDATE)	SOUNDEX
DATEPART	SUBSTR (SUBSTRING)
DATETIME (NOW)	TIME (CURTIME)
DAY (DAYOFMONTH)	TIMEPART
HOUR	TODAY (CURDATE)
INDEX (LOCATE)	TRIMN (RTRIM)
LENGTH	TRANWRD (REPLACE)
MINUTE	WEEKDAY (DAYOFWEEK)
MONTH	YEAR
QTR (QUARTER)	

Passing Joins to ODBC

For a multiple libref join to pass to ODBC, all of these components of the LIBNAME statements must match exactly:

- user ID (USER=)
- password (PASSWORD=)
- data source (DATASRC=)
- catalog (QUALIFIER=)
- update isolation level (UPDATE_ISOLATION_LEVEL=, if specified)
- read isolation level (READ_ISOLATION_LEVEL=, if specified)
- prompt (PROMPT=, must *not* be specified)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Bulk Loading for ODBC

The BULKLOAD= on page 108 LIBNAME option calls the Bulk Copy (BCP) facility, which lets you efficiently insert rows of data into a DBMS table as a unit. BCP= is an alias for this option.

Note: The Bulk Copy facility is available only when you are accessing Microsoft SQL Server data on Windows platforms. To use this facility, your installation of Microsoft SQL Server must include the ODBCBCP.DLL file. BULKCOPY= is not available on UNIX.

As the ODBC interface sends rows of data to the Bulk Copy facility, data is written to an input buffer. When you send all rows or when the buffer reaches a certain size (DBCOMMIT= on page 124 determines this), all rows are inserted as a unit into the table and the data is committed to the table. You can set the DBCOMMIT= option to commit rows after a specified number of rows are inserted.

If an error occurs, a message is written to the SAS log, and any rows that were inserted before the error are rolled back.

Locking in the ODBC Interface

The following LIBNAME and data set options let you control how the ODBC interface handles locking. For general information about an option, see "LIBNAME Options for Relational Databases" on page 99.

READ_LOCK_TYPE= ROW | TABLE | NOLOCK

UPDATE_LOCK_TYPE= ROW | TABLE | NOLOCK

READ_ISOLATION_LEVEL= S | RR | RC | RU | V
The ODBC driver manager supports the S, RR, RC, RU, and V isolation levels that are defined in this table.

Table 24.3 *Isolation Levels for ODBC*

Isolation Level	Definition
S (serializable)	Does not allow dirty Reads, nonrepeatable Reads, or phantom Reads.
RR (repeatable read)	Does not allow dirty Reads or nonrepeatable Reads; does allow phantom Reads.
RC (read committed)	Does not allow dirty Reads or nonrepeatable Reads; does allow phantom Reads.
RU (read uncommitted)	Allows dirty Reads, nonrepeatable Reads, and phantom Reads.
V (versioning)	Does not allow dirty Reads, nonrepeatable Reads, or phantom Reads. These transactions are serializable but higher concurrency is possible than with the serializable isolation level. Typically, a nonlocking protocol is used.

Here are how the terms in the table are defined.

Dirty reads

A transaction that exhibits this phenomenon has very minimal isolation from concurrent transactions. In fact, it can see changes that are made by those concurrent transactions even before they commit.

For example, suppose that transaction T1 performs an update on a row, transaction T2 then retrieves that row, and transaction T1 then terminates with rollback. Transaction T2 has then seen a row that no longer exists.

Nonrepeatable reads

If a transaction exhibits this phenomenon, it is possible that it might read a row once and if it attempts to read that row again later in the course of the same transaction, the row might have been changed or even deleted by another concurrent transaction. Therefore, the read is not (necessarily) repeatable.

For example, suppose that transaction T1 retrieves a row, transaction T2 then updates that row, and transaction T1 then retrieves the same row again. Transaction T1 has now retrieved the same row twice but has seen two different values for it.

Phantom reads

When a transaction exhibits this phenomenon, a set of rows that it reads once might be a different set of rows if the transaction attempts to read them again.

For example, suppose that transaction T1 retrieves the set of all rows that satisfy some condition. Suppose that transaction T2 then inserts a new row that satisfies that same condition. If transaction T1 now repeats its retrieval request, it sees a row that did not previously exist, a phantom.

UPDATE_ISOLATION_LEVEL= S | RR | RC | V

The ODBC driver manager supports the S, RR, RC, and V isolation levels defined in the preceding table.

Naming Conventions for ODBC

For general information about this feature, see “SAS Names and Support for DBMS Names” on page 13.

Because ODBC is an application programming interface (API) rather than a database, table names and column names are determined at run time. Since SAS 7, table names and column names can be up to 32 characters long. SAS/ACCESS Interface to ODBC supports table names and column names that contain up to 32 characters. If DBMS column names are longer than 32 characters, SAS truncates them to 32 characters. If truncating a column name would result in identical names, SAS generates a unique name by replacing the last character with a number. DBMS table names must be 32 characters or less because SAS does *not* truncate a longer name. If you already have a table name that is greater than 32 characters, it is recommended that you create a table view.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options determine how SAS/ACCESS Interface to ODBC handles case sensitivity, spaces, and special characters. The default value for both options is YES for Microsoft Access, Microsoft Excel, and Microsoft SQL Server and NO for all others. For information about these options, see “Overview: LIBNAME Statement for Relational Databases ” on page 93.

This example specifies Sybase as the DBMS.

```
libname mydblib odbc user=TESTUSER password=testpass
        database=sybase;

data mydblib.a;
   x=1;
   y=2;
run;
```

Sybase is generally case sensitive. This example would therefore produce a Sybase table named **a** with columns named **x** and **y**.

If the DBMS being accessed was Oracle, which is not case sensitive, the example would produce an Oracle table named **A** and columns named **X** and **Y**. The object names would be normalized to uppercase.

Data Types for ODBC

Overview

Every column in a table has a name and a data type. The data type tells the DBMS how much physical storage to set aside for the column and the form in which the data is stored. This section includes information about ODBC null and default values and data conversions.

ODBC Null Values

Many relational database management systems have a special value called NULL. A DBMS NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a DBMS NULL value, it interprets it as a SAS missing value.

In most relational databases, columns can be defined as NOT NULL so that they require data (they cannot contain NULL values). When a column is defined as NOT NULL, the DBMS does not add a row to the table unless the row has a value for that column. When creating a DBMS table with SAS/ACCESS, you can use the “DBNULL= Data Set Option” on page 294 data set option to indicate whether NULL is a valid value for specified columns.

ODBC mirrors the behavior of the underlying DBMS with regard to NULL values. See the documentation for your DBMS for information about how it handles NULL values.

For more information about how SAS handles NULL values, see “Potential Result Set Differences When Processing Null Data ” on page 35.

To control how the DBMS handles SAS missing character values, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

This table shows all data types and default SAS formats that SAS/ACCESS Interface to ODBC supports. It does not explicitly define the data types as they exist for each DBMS. It lists the SQL types that each DBMS data type would map to. For example, a CHAR data type under DB2 would map to an ODBC data type of SQL_CHAR. All data types are supported.

Table 24.4 *ODBC Data Types and Default SAS Formats*

ODBC Data Type	Default SAS Format
SQL_CHAR	$*n*
SQL_VARCHAR	$*n*
SQL_LONGVARCHAR	$*n*
SQL_BINARY	$*n*.*
SQL_VARBINARY	$*n*.*
SQL_LONGVARBINARY	$*n*.*
SQL_DECIMAL	*m* or *m.n* or none if *m* and *n* are not specified
SQL_NUMERIC	*m* or *m.n* or none if *m* and *n* are not specified
SQL_INTEGER	11.
SQL_SMALLINT	6.
SQL_TINYINT	4.
SQL_BIT	1.
SQL_REAL	none

ODBC Data Type	Default SAS Format
SQL_FLOAT	none
SQL_DOUBLE	none
SQL_BIGINT	20.
SQL_INTERVAL	$n
SQL_GUID	$n
SQL_TYPE_DATE	DATE9.
SQL_TYPE_TIME	TIME8. ODBC cannot support fractions of seconds for time values
SQL_TYPE_TIMESTAMP	DATETIME*m.n* where *m* and *n* depend on precision

* Because the ODBC driver does the conversion, this field is displayed as if the $HEX*n*. format were applied.

The following table shows the default data types that SAS/ACCESS Interface to ODBC uses when creating tables. SAS/ACCESS Interface to ODBC lets you specify non-default data types by using the DBTYPE= on page 303 data set option.

Table 24.5 *Default ODBC Output Data Types*

SAS Variable Format	Default ODBC Data Type
m.n	SQL_DOUBLE or SQL_NUMERIC using *m.n* if the DBMS allows it
$*n*.	SQL_VARCHAR using *n*
datetime formats	SQL_TIMESTAMP
date formats	SQL_DATE
time formats	SQL_TIME

Chapter 25

SAS/ACCESS Interface to OLE DB

Introduction to SAS/ACCESS Interface to OLE DB

This section describes SAS/ACCESS Interface to OLE DB. For a list of SAS/ACCESS features that are available in this interface, see “SAS/ACCESS Interface to OLE DB: Supported Features” on page 86.

Microsoft OLE DB is an application programming interface (API) that provides access to data that can be in a database table, an e-mail file, a text file, or another type of file. This SAS/ACCESS interface accesses data from these sources through OLE DB data providers such as Microsoft Access, Microsoft SQL Server, and Oracle.

LIBNAME Statement Specifics for OLE DB

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to OLE DB supports and includes examples. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing OLE DB.

LIBNAME *libref* **oledb** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
: specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`oledb`
: specifies the SAS/ACCESS engine name for the OLE DB interface.

connection-options
: provide connection information and control how SAS manages the timing and concurrence of the connection to the data source. You can connect to a data source either by using OLE DB Services or by connecting directly to the provider. For details, see “Connecting with OLE DB Services” on page 677 and “Connecting Directly to a Data Provider” on page 678.

  These connection options are available with both connection methods. Here is how they are defined.

  USER=<'>*user-name*<'>
  : lets you connect to an OLE DB data source with a user ID that is different from the default ID. The default is your user ID.

  PASSWORD=<'>*password*<'>
  : specifies the OLE DB password that is associated with your user ID. If it contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. If you do not wish to enter your OLE DB password in uncoded text, see PROC PWENCODE in the *Base SAS Procedures Guide* for a method to encode it.

DATASOURCE=<'>*data-source*<'>
: identifies the data source object (such as a relational database server or a local file) to which you want to connect.

PROVIDER=<'>*provider-name*<'>
: specifies which OLE DB provider to use to connect to the data source. This option is required during batch processing. There is no restriction on the length of the *provider-name*. If the *provider-name* contains blank spaces or special characters, enclose it in quotation marks. If you do not specify a provider, an **OLE DB Services** dialog box prompts you for connection information. In batch mode, if you do not specify a provider the connection fails. If you are using the Microsoft Jet OLE DB 4.0 provider, specify PROVIDER=JET.

PROPERTIES=(<'>*property-1*<'>=<'>*value-1*<'> < . . . <'>*property-n*<'>=<'>*value-n*<'>>)
: specifies standard provider properties that enable you to connect to a data source and to define connection attributes. If a property name or value contains embedded spaces or special characters, enclose the name or value in quotation marks. Use a blank space to separate multiple properties. If your provider supports a password property, that value cannot be encoded. To use an encoded password, use the PASSWORD= option instead. See your provider documentation for a list and description of all properties that your provider supports. No properties are specified by default.

PROVIDER_STRING=<'>*extended-properties*<'>
: specifies provider-specific extended connection information, such as the file type of the data source. If the string contains blank spaces or special characters, enclose it in quotation marks. For example, the Microsoft Jet provider accepts strings that indicate file type, such as 'Excel 8.0'. The following example uses the Jet 4.0 provider to access the spreadsheet Y2KBUDGET.XLS. Specify the 'Excel 8.0' provider string so that Jet recognizes the file as an Excel 8.0 worksheet.

```
libname budget oledb provider=jet provider_string='Excel 8.0'
                     datasource='d:\excel80\Y2Kbudget.xls';
```

OLEDB_SERVICES=YES | NO
: determines whether SAS uses OLE DB Services to connect to the data source. Specify YES to use OLE DB Services or specify NO to use the provider to connect to the data source. When you specify PROMPT=YES and OLEDB_SERVICES=YES, you can set more options than you would otherwise be able to set by being prompted by the provider's dialog box. If OLEDB_SERVICES=NO, you must specify PROVIDER= first so that the provider's prompt dialog boxes are used. If PROVIDER= is omitted, SAS uses OLE DB Services, even if you specify OLEDB_SERVICES=NO. YES is the default. For Microsoft SQL Server data, if BULKLOAD=YES, OLEDB_SERVICES= is set to NO. When OLEDB_SERVICES=YES and a successful connection is made, the complete connection string is returned in the SYSDBMSG macro variable.

PROMPT =YES | NO
: determines whether one of these interactive dialog boxes is displayed to guide you through the connection process:

  - an **OLE DB provider** dialog box if OLEDB_SERVICES=NO and you specify a provider.
  - an **OLE DB Services** dialog box if OLEDB_SERVICES=YES or if you do not specify a provider.

Generally preferred over the provider's dialog box, the **OLE DB Services** dialog box lets you set options more easily. If you specify a provider and set OLEDB_SERVICES=NO, the default is PROMPT=NO. Otherwise, the default is PROMPT=YES. If OLEDB_SERVICES=YES or if you do not specify a provider, an **OLE DB Services** dialog box is displayed even if you specify PROMPT=NO. Specify no more than one of the following options on each LIBNAME statement: COMPLETE=, REQUIRED=, PROMPT=. Any properties that you specify in the PROPERTIES= option are displayed in the prompting interface, and you can edit any field.

UDL_FILE=<'>*path-and-file-name*<'>
: specifies the path and filename for a Microsoft universal data link (UDL). For example, you could specify **`UDL_FILE="C:\WinNT\profiles\me\desktop\MyDBLink.UDL"`**. This option does not support SAS filerefs. SYSDBMSG is *not* set on successful completion. For more information, see Microsoft documentation about the Data Link API. This option overrides any values that are set with the INIT_STRING=, PROVIDER=, and PROPERTIES= options.

This connection option is available only when you use OLE DB Services.

INIT_STRING='*property-1=value-1*<…;*property-n=value-n*>'
: specifies an initialization string, enabling you to bypass the interactive prompting interface yet still use OLE DB Services. (This option is not available if OLEDB_SERVICES=NO.) Use a semicolon to separate properties. After you connect to a data source, SAS returns the complete initialization string to the macro variable SYSDBMSG, which stores the connection information that you specify in the prompting window. You can reuse the initialization string to make automated connections or to specify connection information for batch jobs. For example, assume that you specify this initialization string:

```
init_string='Provider=SQLOLEDB;Password=dbmgr1;Persist
Security Info=True;User ID=rachel;Initial Catalog=users;
Data Source=dwtsrv1';
```

Here is what the content of the SYSDBMSG macro variable would be:

```
OLEDB:  Provider=SQLOLEDB;Password=dbmgr1;
Persist Security Info=True;User ID=rachel;
Initial Catalog=users;Data Source=dwtsrv1;
```

If you store this string for later use, delete the **`OLEDB:`** prefix and any initial spaces before the first listed option. There is no default value. However, if you specify a null value for this option, the OLE DB Provider for ODBC (MSDASQL) is used with your default data source and its properties. See your OLE DB documentation for more information about these default values. This option overrides any values that are set with the PROVIDER= and PROPERTIES= options. To write the initialization string to the SAS log, submit this code immediately after connecting to the data source: **`%put %superq(SYSDBMSG);`**

Only these connection options are available when you connect directly to a provider.

COMPLETE=YES | NO
: specifies whether SAS attempts to connect to the data source without prompting you for connection information. If you specify COMPLETE=YES and the connection information that you specify in your LIBNAME statement is sufficient, then SAS makes the connection and does not prompt you for additional information. If you specify COMPLETE=YES and the connection information that you specify in your LIBNAME statement is not sufficient, the

provider's dialog box prompts you for additional information. You can enter optional information as well as required information in the dialog box. NO is the default value. COMPLETE= is available only when you set OLEDB_SERVICES=NO and you specify a provider. It is not available in the SQL pass-through facility. Specify no more than one of these options on each LIBNAME statement: COMPLETE=, REQUIRED=, PROMPT=.

REQUIRED=YES | NO
: specifies whether SAS attempts to connect to the data source without prompting you for connection information and whether you can interactively specify optional connection information. If you specify REQUIRED=YES and the connection information that you specify in your LIBNAME statement is sufficient, SAS makes the connection and you are not prompted for additional information. If you specify REQUIRED=YES and the connection information that you specify in your LIBNAME statement is not sufficient, the provider's dialog box prompts you for the required connection information. You cannot enter optional connection information in the dialog box. NO is the default value. REQUIRED= is available only when you set OLEDB_SERVICES=NO and you specify a provider in the PROVIDER= option. It is not available in the SQL pass-through facility Specify no more than one of these options on each LIBNAME statement: COMPLETE=, REQUIRED=, PROMPT=.

LIBNAME-options
: define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to OLE DB, with the applicable default values. For more detail about these options, see “LIBNAME Options for Relational Databases” on page 99.

Table 25.1 *SAS/ACCESS LIBNAME Options for OLE DB*

Option	**Default Value**
ACCESS= on page 99	none
AUTHDOMAIN= on page 102	none
AUTOCOMMIT= on page 103	data-source specific
BL_KEEPIDENTITY= on page 104	NO
BL_KEEPNULLS= on page 105	YES
BL_OPTIONS= on page 106	not specified
BULKLOAD= on page 108	NO
CELLPROP= on page 111	VALUE
COMMAND_TIMEOUT= on page 113	0 (no time–out)
CONNECTION= on page 113	SHAREDREAD
CONNECTION_GROUP= on page 118	none

Option	Default Value
CURSOR_TYPE= on page 119	FORWARD_ONLY
DBCOMMIT= on page 124	1000 when inserting rows; 0 when updating rows
DBCONINIT= on page 125	none
DBCONTERM= on page 126	none
DBCREATE_TABLE_OPTS= on page 128	none
DBGEN_NAME= on page 128	DBMS
DBINDEX= on page 129	NO
DBLIBINIT= on page 130	none
DBLIBTERM= on page 131	none
DBMAX_TEXT= on page 133	1024
DBMSTEMP= on page 133	NO
DBNULLKEYS= on page 135	YES
DEFER= on page 141	NO
DELETE_MULT_ROWS= on page 143	NO
DIRECT_EXE= on page 144	none
DIRECT_SQL= on page 145	YES
IGNORE_READ_ONLY_COLUMNS= on page 150	NO
INSERT_SQL= on page 152	data-source specific
INSERTBUFF= on page 153	1
MULTI_DATASRC_OPT= on page 160	NONE
PRESERVE_COL_NAMES= on page 167	see "Naming Conventions for OLE DB" on page 692
PRESERVE_TAB_NAMES= on page 168	see "Naming Conventions for OLE DB" on page 692
QUALIFIER= on page 170	none
QUALIFY_ROWS= on page 171	NO

Option	Default Value
QUOTE_CHAR= on page 173	not set
READBUFF= on page 174	1
READ_ISOLATION_LEVEL= on page 175	not set (see “Locking in the OLE DB Interface” on page 689)
READ_LOCK_TYPE= on page 176	see “Locking in the OLE DB Interface” on page 689
REREAD_EXPOSURE= on page 179	NO
SCHEMA= on page 180	none
SPOOL= on page 184	YES
SQL_FUNCTIONS= on page 185	none
STRINGDATES= on page 190	NO
UPDATE_ISOLATION_LEVEL= on page 196	not set (see “Locking in the OLE DB Interface” on page 689)
UPDATE_LOCK TYPE= on page 197	ROW
UPDATE_MULT_ROWS= on page 199	NO
UTILCONN_TRANSIENT= on page 202	NO

Connecting with OLE DB Services

By default, SAS/ACCESS Interface to OLE DB uses OLE DB services because this is often the fastest and easiest way to connect to a data provider.

OLE DB Services provides performance optimizations and scaling features, including resource pooling. It also provides interactive prompting for the provider name and connection information.

Assume that you submit a simple LIBNAME statement, such as this one:

```
libname mydblib oledb;
```

SAS directs OLE DB Services to display a dialog box that contains tabs where you can enter the provider name and connection information.

After you make a successful connection using OLE DB Services, you can retrieve the connection information and reuse it in batch jobs and automated connections. For more information, see the connection options INIT_STRING= and OLEDB_SERVICES=.

Connecting Directly to a Data Provider

To connect to a data source, SAS/ACCESS Interface to OLE DB requires a provider name and provider-specific connection information such as the user ID, password, schema,, or server name. If you know all of this information, you can connect directly to a provider without using OLE DB services on page 677.

If you are connecting to Microsoft SQL Server and you specify the SAS/ACCESS BULKLOAD=YES option, you must connect directly to the provider by specifying this information:

- the name of the provider (PROVIDER=)
- any required connection information

After you connect to your provider, you can use the special OLE DB PROVIDER_INFO on page 682 query to make subsequent unprompted connections easier. You can submit this special query as part of a PROC SQL query to display all available provider names and properties. For an example, see "Examples of Special OLE DB Queries" on page 684.

If you know only the provider name and you are running an interactive SAS session, you can be prompted for the provider's properties. Specify PROMPT=YES to direct the provider to prompt you for properties and other connection information. Each provider displays its own prompting interface.

If you run SAS in a batch environment, specify only USER=, PASSWORD=, DATASOURCE=, PROVIDER=, and PROPERTIES=.

OLE DB LIBNAME Statement Examples

In the following example, the libref MYDBLIB uses the SAS/ACCESS OLE DB engine to connect to a Microsoft SQL Server database.

```
libname mydblib oledb user=username password=password
datasource=dept203 provider=sqloledb properties=('initial catalog'=mgronly);
proc print data=mydblib.customers;
   where state='CA';
run;
```

In the following example, the libref MYDBLIB uses the SAS/ACCESS engine for OLE DB to connect to an Oracle database. Because prompting is enabled, you can review and edit the user, password, and data source information in a dialog box.

```
libname mydblib oledb user=username password=password datasource=v2o7223.world
                      provider=msdaora prompt=yes;

proc print data=mydblib.customers;
   where state='CA';
run;
```

In the following example, you submit a basic LIBNAME statement, so an **OLE DB Services** dialog box prompts you for the provider name and property values.

```
libname mydblib oledb;
```

The advantage of being prompted is that you do not need to know any special syntax to set the values for the properties. Prompting also enables you to set more options than you might when you connect directly to the provider (and do not use OLE DB Services).

Data Set Options for OLE DB

All SAS/ACCESS data set options in this table are supported for OLE DB. Default values are provided where applicable. For general information about this feature, see Data Set Options for Relational Databases on page 207.

Table 25.2 *SAS/ACCESS Data Set Options for OLE DB*

Option	Default Value
BL_KEEPIDENTITY= on page 250	LIBNAME option setting
BL_KEEPNULLS= on page 250	LIBNAME option setting
BL_OPTIONS= on page 255	LIBNAME option setting
BULKLOAD= on page 274	NO
COMMAND_TIMEOUT= on page 278	LIBNAME option setting
CURSOR_TYPE= on page 278	LIBNAME option setting
DBCOMMIT= on page 280	LIBNAME option setting
DBCONDITION= on page 281	none
DBCREATE_TABLE_OPTS= on page 285	LIBNAME option setting
DBFORCE= on page 286	NO
DBGEN_NAME= on page 287	DBMS
DBINDEX= on page 288	LIBNAME option setting
DBKEY= on page 290	none
DBLABEL= on page 291	NO
DBMASTER= on page 293	none
DBMAX_TEXT= on page 293	1024
DBNULL= on page 294	_ALL_=YES
DBNULLKEYS= on page 295	LIBNAME option setting
DBSASLABEL= on page 137	COMPAT
DBSASTYPE= on page 298	see "Data Types for OLE DB " on page 693

Option	Default Value
DBTYPE= on page 303	see "Data Types for OLE DB " on page 693
ERRLIMIT= on page 308	1
IGNORE_READ_ONLY_COLUMNS= on page 311	NO
INSERT_SQL= on page 313	LIBNAME option setting
INSERTBUFF= on page 314	LIBNAME option setting
NULLCHAR= on page 329	SAS
NULLCHARVAL= on page 330	a blank character
PRESERVE_COL_NAMES= on page 336	LIBNAME option setting
PRESERVE_COL_NAMES= on page 338	LIBNAME option setting
READBUFF= on page 339	LIBNAME option setting
READ_ISOLATION_LEVEL= on page 340	LIBNAME option setting
SASDATEFMT= on page 342	not set
SCHEMA= on page 344	LIBNAME option setting
UPDATE_ISOLATION_LEVEL= on page 371	LIBNAME option setting
UPDATE_LOCK_TYPE= on page 371	LIBNAME option setting
UTILCONN_TRANSIENT= on page 202	YES

SQL Pass-Through Facility Specifics for OLE DB

Key Information

For general information about this feature, see "SQL Pass-Through Facility" on page 401. OLE DB examples are available.

Here are the SQL pass-through facility specifics for the OLE DB interface.

- The *dbms-name* is **OLEDB**.
- The CONNECT statement is required.
- PROC SQL supports multiple connections to OLE DB. If you use multiple simultaneous connections, you must use an *alias* to identify the different connections. If you do not specify an alias, the default alias, **OLEDB**, is used. The

functionality of multiple connections to the same OLE DB provider might be limited by a particular provider.

- The CONNECT statement *database-connection-arguments* are identical to the LIBNAME connection options. For some data sources, the connection options have default values and are therefore not required.

  Not all OLE DB providers support all connection options. See your provider documentation for more information.

- Here are the LIBNAME options that are available with the CONNECT statement:
  - AUTOCOMMIT= on page 103
  - CELLPROP= on page 111
  - COMMAND_TIMEOUT= on page 113
  - CURSOR_TYPE= on page 119
  - DBMAX_TEXT= on page 133
  - QUALIFY_ROWS= on page 171
  - READ_ISOLATION_LEVEL= on page 175
  - READ_LOCK_TYPE= on page 176
  - READBUFF= on page 174
  - STRINGDATES= on page 190

Examples

This example uses an alias to connect to a Microsoft SQL Server database and select a subset of data from the PAYROLL table. The SAS/ACCESS engine uses OLE DB Services to connect to OLE DB because this is the default action when the OLEDB_SERVICES= option is omitted.

```
proc sql;
connect to oledb as finance
  (user=username password=password datasource=dwtsrv1
   provider=sqloledb);

select * from connection to finance (select * from payroll
                                     where jobcode='FA3');
quit;
```

In this example, the CONNECT statement omits the provider name and properties. An **OLE DB Services** dialog box prompts you for the connection information.

```
proc sql;
connect to oledb;
quit;
```

This example uses OLE DB Services to connect to a provider that is configured under the data source name `User's Data` with the alias USER1. Note that the data source name can contain quotation marks and spaces.

```
proc sql;
   connect to oledb as user1
   (provider=JET datasource='c:\db1.mdb');;
```

Special Catalog Queries

Overview

SAS/ACCESS Interface to OLE DB supports the following special queries. Many databases provide or use system tables that allow queries to return the list of available tables, columns, procedures, and other useful information. OLE DB provides much of this functionality through special application programming interfaces (APIs) to accommodate databases that do not follow the SQL table structure. You can use these special queries on SQL and non-SQL databases.

Not all OLE DB providers support all queries. See your provider documentation for more information.

Here is the general format of the special queries:

OLEDB::*schema-rowset*("*parameter 1*","*parameter n*")

OLEDB::
: is required to distinguish special queries from regular queries.

schema-rowset
: is the specific schema rowset that is being called. All valid schema rowsets are listed under the IDBSchemaRowset Interface in the *Microsoft OLE DB Programmer's Reference*. Both OLEDB:: and *schema-rowset* are case sensitive.

"parameter n"
: is a quoted string that is enclosed by commas. The values for the special query arguments are specific to each data source. For example, you supply the fully qualified table name for a *"Qualifier"* argument. In dBase, the value of *"Qualifier"* might be **`c:\dbase\tst.dbf`**, and in SQL Server, the value might be **`test.customer`**. In addition, depending on the data source that you use, values for an *"Owner"* argument might be a user ID, a database name, or a library. All arguments are optional. If you specify some but not all arguments within a parameter, use commas to indicate omitted arguments. If you do not specify any parameters, no commas are necessary. These special queries might not be available for all OLE DB providers.

OLE DB supports these special queries:

OLEDB::ASSERTIONS(<*"Catalog", "Schema", "Constraint-Name"*>)
: returns assertions that are defined in the catalog that a given user owns.

OLEDB::CATALOGS(<*"Catalog"*>)
: returns physical attributes that are associated with catalogs that are accessible from the DBMS.

OLEDB::CHARACTER_SETS(<*"Catalog", "Schema","Character-Set-Name"*>)
: returns the character sets that are defined in the catalog that a given user can access.

OLEDB::CHECK_CONSTRAINTS(<*"Catalog", "Schema", "Constraint-Name"*>)
: returns check constraints that are defined in the catalog and that a given user owns.

OLEDB::COLLATIONS(<*"Catalog", "Schema", "Collation-Name"*>)
: returns the character collations that are defined in the catalog and that a given user can access.

OLEDB::COLUMN_DOMAIN_USAGE(<*"Catalog", "Schema", "Domain-Name", "Column-Name"*>)
: returns the columns that are defined in the catalog, are dependent on a domain that is defined in the catalog, and a given user owns.

OLEDB::COLUMN_PRIVILEGES(<*"Catalog", "Schema", "Table-Name", "Column-Name", "Grantor", "Grantee"*>)
: returns the privileges on columns of tables that are defined in the catalog that a given user grants or can access.

OLEDB::COLUMNS(<*"Catalog", "Schema", "Table-Name", "Column-Name"*>)
: returns the columns of tables that are defined in the catalogs that a given user can access.

OLEDB::CONSTRAINT_COLUMN_USAGE(<*"Catalog", "Schema", "Table-Name", "Column-Name"*>)
: returns the columns that referential constraints, unique constraints, check constraints, and assertions use that are defined in the catalog and that a given user owns.

OLEDB::CONSTRAINT_TABLE_USAGE(<*"Catalog", "Schema", "Table-Name"*>)
: returns the tables that referential constraints, unique constraints, check constraints, and assertions use that are defined in the catalog and that a given user owns.

OLEDB::FOREIGN_KEYS(<*"Primary-Key-Catalog", "Primary-Key-Schema", "Primary-Key-Table-Name", "Foreign-Key-Catalog", "Foreign-Key-Schema", "Foreign-Key-Table-Name"*>)
: returns the foreign key columns that a given user defined in the catalog.

OLEDB::INDEXES(<*"Catalog", "Schema", "Index-Name", "Type", "Table-Name"*>)
: returns the indexes that are defined in the catalog that a given user owns.

OLEDB::KEY_COLUMN_USAGE(<*"Constraint-Catalog", "Constraint-Schema", "Constraint-Name", "Table-Catalog", "Table-Schema", "Table-Name", "Column-Name"*>)
: returns the columns that are defined in the catalog and that a given user has constrained as keys.

OLEDB::PRIMARY_KEYS(<*"Catalog", "Schema", "Table-Name"*>)
: returns the primary key columns that a given user defined in the catalog.

OLEDB::PROCEDURE_COLUMNS(<*"Catalog", "Schema", "Procedure-Name", "Column-Name"*>)
: returns information about the columns of rowsets that procedures return.

OLEDB::PROCEDURE_PARAMETERS(<*"Catalog", "Schema", "Procedure-Name", "Parameter-Name"*>)
: returns information about the parameters and return codes of the procedures.

OLEDB::PROCEDURES(<*"Catalog", "Schema", "Procedure-Name", "Procedure-Type"*>)
: returns procedures that are defined in the catalog that a given user owns.

OLEDB::PROVIDER_INFO()
: returns output that contains these columns: PROVIDER_NAME, PROVIDER_DESCRIPTION, and PROVIDER_PROPERTIES. The PROVIDER_PROPERTIES column contains a list of all properties that the provider supports. A semicolon (;) separates the properties. See y“Examples of Special OLE DB Queries” on page 684.

OLEDB::PROVIDER_TYPES(<*"Data Type", "Best-Match"*>)
: returns information about the base data types that the data provider supports.

OLEDB::REFERENTIAL_CONSTRAINTS(<*"Catalog", "Schema", "Constraint-Name"*>)
: returns the referential constraints that are defined in the catalog that a given user owns.

OLEDB::SCHEMATA(<*"Catalog", "Schema", "Owner"*>)
: returns the schemas that a given user owns.

OLEDB::SQL_LANGUAGES()
: returns the conformance levels, options, and dialects that the SQL implementation processing data supports and that is defined in the catalog.

OLEDB::STATISTICS(<*"Catalog", "Schema", "Table-Name"*>)
: returns the statistics that is defined in the catalog that a given user owns.

OLEDB::TABLE_CONSTRAINTS(<*"Constraint-Catalog", "Constraint-Schema", "Constraint-Name", "Table-Catalog", "Table-Schema", "Table-Name", "Constraint-Type"*>)
: returns the table constraints that is defined in the catalog that a given user owns.

OLEDB::TABLE_PRIVILEGES(<*"Catalog", "Schema", "Table-Name", "Grantor", "Grantee"*>)
: returns the privileges on tables that are defined in the catalog that a given user grants or can access.

OLEDB::TABLES(<*"Catalog", "Schema", "Table-Name", "Table-Type"*>)
: returns the tables defined in the catalog that a given user grants and can access.

OLEDB::TRANSLATIONS(<*"Catalog", "Schema", "Translation-Name"*>)
: returns the character translations that are defined in the catalog and that are accessible to a given user.

OLEDB::USAGE_PRIVILEGES(<*"Catalog", "Schema", "Object-Name", "Object-Type", "Grantor", "Grantee"*>)
: returns the USAGE privileges on objects that are defined in the catalog and that a given user grants or can access.

OLEDB::VIEW_COLUMN_USAGE(<*"Catalog", "Schema", "View-Name"*>)
: returns the columns on which viewed tables depend that are defined in the catalog and that a given user owns.

OLEDB::VIEW_TABLE_USAGE(<*"Catalog", "Schema", "View-Name"*>)
: returns the tables on which viewed tables depend that are defined in the catalog and that a given user owns.

OLEDB::VIEWS(<*"Catalog", "Schema", "Table-Name"*>)
: returns the viewed tables that are defined in the catalog and that a given user can access.

For a complete description of each rowset and the columns that are defined in each rowset, see the *Microsoft OLE DB Programmer's Reference*.

Examples of Special OLE DB Queries

This example retrieves a rowset that displays all tables that the HRDEPT schema accesses:

```
proc sql;
  connect to oledb(provider=sqloledb properties=("User ID"=testuser
                 Password=testpass
     "Data Source"='dwtsrv1'));
  select * from connection to oledb
       (OLEDB::TABLES(,"HRDEPT"));
quit;
```

It uses the special query OLEDB::PROVIDER_INFO() to produce this output:

```
proc sql;
  connect to oledb(provider=msdaora properties=("User ID"=testuser
                   Password=testpass
     "Data Source"="Oraserver"));
  select * from connection to oledb
       (OLEDB::PROVIDER_INFO());
quit;
```

Output 25.1 *Provider and Properties Output*

```
PROVIDER_NAME   PROVIDER_DESCRIPTION   PROVIDER_PROPERTIES
-------------   --------------------   -------------------
MSDAORA         Microsoft OLE DB       Password;User ID;Data
                Provider for Oracle    Source;Window Handle;Locale
                                       Identifier;OLE DB Services;
                                       Prompt; Extended Properties;

SampProv        Microsoft OLE DB       Data Source;Window Handle;
                Sample Provider        Prompt;
```

You could then reference the output when automating a connection to the provider. For the previous result set, you could write this SAS/ACCESS LIBNAME statement:

```
libname mydblib oledb provider=msdaora
   props=('Data Source'=OraServer 'User ID'=scott 'Password'=tiger);
```

Temporary Table Support for OLE DB

Overview

For general information about this feature, see "Temporary Table Support for SAS/ACCESS " on page 42.

Establishing a Temporary Table

When you want to use temporary tables that persist across SAS procedures and DATA steps with OLE DB, you must use the CONNECTION=SHARED LIBNAME option. In doing so, the temporary table is available for processing until the libref is closed.

Terminating a Temporary Table

You can drop a temporary table at any time, or allow it to be implicitly dropped when the connection is terminated. Temporary tables do not persist beyond the scope of a singe connection.

Examples

Using the sample Internat table, this example creates a temporary table, #LONDON, with Microsoft SQL Server. It contains information about flights that flew to London. This table is then joined with a larger SQL Server table that lists all flights, March, but matched only on flights that flew to London.

```
libname samples oledb Provider=SQLOLEDB Password="dbigrp1" UID=dbitest
                      DSN='lupin\sql2000' connection=shared;

data samples.'#LONDON'n;
   set work.internat;
   where dest='LON';
run;

proc sql;
   select b.flight, b.dates, b.depart, b.orig
          from samples.'#LONDON'n a, samples.march b
          where a.dest=b.dest;
quit;
```

In this example, a temporary table, New, is created with Microsoft SQL Server. The data from this table is then appended to an existing SQL Server table, Inventory.

```
libname samples oledb provider=SQLOLEDB dsn=lupinss
                      uid=dbitest pwd="dbigrp1";

data samples.inventory(DBTYPE=(itemnum='char(5)' item='varchar(30)'
                       quantity='numeric'));
   itemnum='12001';
      item='screwdriver';
      quantity=15;
      output;
   itemnum='12002';
      item='hammer';
      quantity=25:
      output;
   itemnum='12003';
      item='sledge hammer';
      quantity=10;
      output;
   itemnum='12004';
      item='saw';
      quantity=50;
      output;
   itemnum='12005';
      item='shovel';
      quantity=120;
      output;
run;

data samples.'#new'n(DBTYPE=(itemnum='char(5)' item='varchar(30)'
                     quantity='numeric'));
   itemnum='12006';
      item='snow shovel';
      quantity=5;
      output;
   itemnum='12007';
      item='nails';
      quantity=500;
      output;
run;

proc append base=samples.inventory data=samples.'#new'n;
```

```
run;

proc print data=samples.inventory;
run;
```

This example demonstrates the use of a temporary table using the SQL pass-through facility.

```
proc sql;
   connect to oledb as test (provider=SQLOLEDB dsn=lupinss
                             uid=dbitest pwd=dbigrp1);
   execute (create table #FRANCE (flight char(3), dates datetime,
            dest char(3))) by test;

   execute (insert #FRANCE select flight, dates, dest from internat
            where dest like '%FRA%') by test;
   select * from connection to test (select * from #FRANCE);
quit;
```

Passing SAS Functions to OLE DB

SAS/ACCESS Interface to OLE DB passes the following SAS functions for OLE DB to DB2, Microsoft SQL Server, and Oracle for processing. Where the OLE DB function name differs from the SAS function name, the OLE DB name appears in parentheses. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

DAY	MINUTE
DTEXTDAY	MONTH
DTEXTMONTH	SECOND
DTEXTYEAR	WEEKDAY
DTEXTWEEKDAY	YEAR
HOUR	

SQL_FUNCTIONS=ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. (See SQL_FUNCTIONS= on page 185.) Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to OLE DB. Due to incompatibility in date and time functions between OLE DB and SAS, OLE DB might not process them correctly. Check your results to determine whether these functions are working as expected.

ABS	MAX
ARCOS (ACOS)	MIN
ARSIN (ASIN)	MOD
ATAN	QRT
AVG	REPEAT
BYTE	SIGN
CEIL	SIN
COMPRESS	SOUNDEX
COS	SQRT

COUNT	STRIP (TRIM)
DATE	SUBSTR
DATEPART	SUM
DATETIME	TAN
EXP	TIME
FLOOR	TIMEPART
INDEX	TODAY
LENGTH	TRANWRD
LOG	TRIM
LOG10	UPCASE
LOWCASE	

Passing Joins to OLE DB

For a multiple libref join to pass to OLE DB, all of these components of the LIBNAME statements must match exactly:

- user ID (USER=)
- password (PASSWORD=)
- data source (DATASOURCE=)
- provider (PROVIDER=)
- qualifier (QUALIFIER=, if specified)
- provider string (PROVIDER_STRING, if specified)
- path and filename (UDL_FILE=, if specified)
- initialization string (INIT_STRING=, if specified)
- read isolation level (READ_ISOLATION_LEVEL=, if specified)
- update isolation level (UPDATE_ISOLATION_LEVEL=, if specified)
- all properties (PROPERTIES=)
- prompt (PROMPT=, must *not* be specified)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see “Passing Joins to the DBMS” on page 47.

Bulk Loading for OLE DB

The BULKLOAD= on page 108 LIBNAME option calls the SQLOLEDB interface of IRowsetFastLoad so that you can efficiently insert rows of data into a Microsoft SQL Server database table as a unit. BCP= is an alias for this option.

Note: This functionality is available only when accessing Microsoft SQL Server data on Windows platforms using Microsoft SQL Server Version 7.0 or later.

As SAS/ACCESS sends rows of data to the bulk-load facility, the data is written to an input buffer. When you have sent all rows or when the buffer reaches a certain size

(DBCOMMIT= on page 124 determines this), all rows are inserted as a unit into the table and the data is committed to the table. You can also set DBCOMMIT= to commit rows after a specified number of rows are inserted.

If an error occurs, a message is written to the SAS log, and any rows that were inserted before the error are rolled back.

If you specify BULKLOAD=YES and the PROVIDER= option is set, SAS/ACCESS Interface to OLE DB uses the specified provider. If you specify BULKLOAD=YES and PROVIDER= is not set, the engine uses the PROVIDER=SQLOLEDB value.

If you specify BULKLOAD=YES, connections that are made through OLE DB Services or UDL files are not allowed.

Locking in the OLE DB Interface

The following LIBNAME and data set options let you control how the OLE DB interface handles locking. For general information about an option, see “LIBNAME Options for Relational Databases” on page 99.

- READ_LOCK_TYPE on page 176= ROW | NOLOCK
- UPDATE_LOCK_TYPE on page 197= ROW | NOLOCK
- READ_ISOLATION_LEVEL on page 175 = S | RR | RC | RU

  The data provider sets the default value. OLE DB supports the S, RR, RC, and RU isolation levels that are defined in this table.

  Table 25.3 *Isolation Levels for OLE DB*

Isolation Level	Definition
S (serializable)	Does not allow dirty Reads, nonrepeatable Reads, or phantom Reads.
RR (repeatable Read)	Does not allow dirty Reads or nonrepeatable Reads; does allow phantom Reads.
RC (committed Read)	Does not allow dirty Reads or nonrepeatable Reads; does allow phantom Reads.
RU (uncommitted Read)	Allows dirty Reads, nonrepeatable Reads, and phantom Reads.

  Here is how the terms in the table are defined.

  Dirty reads
  : A transaction that exhibits this phenomenon has very minimal isolation from concurrent transactions. In fact, it can see changes that are made by those concurrent transactions even before they commit.

    For example, suppose that transaction T1 performs an update on a row, transaction T2 then retrieves that row, and transaction T1 then terminates with rollback. Transaction T2 has then seen a row that no longer exists.

Nonrepeatable reads
If a transaction exhibits this phenomenon, it might read a row once. Then, if the same transaction attempts to read that row again, the row might have been changed or even deleted by another concurrent transaction. Therefore, the Read is not (necessarily) repeatable.

For example, suppose that transaction T1 retrieves a row, transaction T2 then updates that row, and transaction T1 then retrieves the same row again. Transaction T1 has now retrieved the same row twice but has seen two different values for it.

Phantom reads
When a transaction exhibits this phenomenon, a set of rows that it reads once might be a different set of rows if the transaction attempts to read them again.

For example, suppose that transaction T1 retrieves the set of all rows that satisfy some condition. Suppose that transaction T2 then inserts a new row that satisfies that same condition. If transaction T1 now repeats its retrieval request, it sees a row that did not previously exist, a phantom.

- UPDATE_ISOLATION_LEVEL on page 196= S | RR | RC

  The default value is set by the data provider. OLE DB supports the S, RR, and RC isolation levels defined in the preceding table. The RU isolation level is not allowed with this option.

Accessing OLE DB for OLAP Data

Overview

SAS/ACCESS Interface to OLE DB provides a facility for accessing OLE DB for OLAP data. You can specify a Multidimensional Expressions (MDX) statement through the SQL pass-through facility to access the data directly, or you can create an SQL view of the data. If your MDX statement specifies a data set with more than five axes (COLUMNS, ROWS, PAGES, SECTIONS, and CHAPTERS), SAS returns an error. See the Microsoft Data Access Components Software Developer's Kit for details about MDX syntax.

Note: This implementation provides Read-Only access to OLE DB for OLAP data. You cannot update or insert data with this facility.

Using the SQL Pass-Through Facility with OLAP Data

Overview

The main difference between normal OLE DB access using the SQL pass-through facility and the implementation for OLE DB for OLAP is the use of these additional identifiers to pass MDX statements to the OLE DB for OLAP data:

MDX::
identifies MDX statements that return a flattened data set from the multidimensional data.

MDX_DESCRIBE::
identifies MDX statements that return detailed column information.

An MDX_DESCRIBE:: identifier is used to obtain detailed information about each returned column. During the process of flattening multidimensional data, OLE DB for OLAP builds column names from each level of the given dimension. For example, for OLE DB for OLAP multidimensional data that contains CONTINENT, COUNTRY, REGION, and CITY dimensions, you could build a column with this name:

```
[NORTH AMERICA].[USA].[SOUTHEAST].[ATLANTA]
```

This name cannot be used as a SAS variable name because it has more than 32 characters. For this reason, the SAS/ACCESS engine for OLE DB creates a column name based on a shortened description, in this case, ATLANTA. However, since there could be an ATLANTA in some other combination of dimensions, you might need to know the complete OLE DB for OLAP column name. Using the MDX_DESCRIBE:: identifier returns a SAS data set that contains the SAS name for the returned column and its corresponding OLE DB for OLAP column name:

```
SASNAME          MDX_UNIQUE_NAME

ATLANTA          [NORTH AMERICA].[USA].[SOUTHEAST].[ATLANTA]
CHARLOTTE        [NORTH AMERICA].[USA].[SOUTHEAST].[CHARLOTTE]
   .                        .
   .                        .
   .                        .
```

If two or more SASNAME values are identical, a number is appended to the end of the second and later instances of the name (for example, ATLANTA , ATLANTA0 , ATLANTA1 , and so on). Also, depending on the value of the VALIDVARNAME= system option, illegal characters are converted to underscores in the SASNAME value.

Syntax

This facility uses the following general syntax. For more information about SQL pass-through facility syntax, see “SQL Pass-Through Facility” on page 401.

PROC SQL <*options*>;

CONNECT TO OLEDB (<*options*>);

<*non-SELECT SQL statement(s)*>

SELECT *column-identifier(s)* FROM CONNECTION TO OLEDB

(MDX:: | MDX_DESCRIBE:: <*MDX statement*>)

<*other SQL statement(s)*>

;

Examples

The following code uses the SQL pass-through facility to pass an MDX statement to a Microsoft SQL Server Decision Support Services (DSS) Cube. The provider used is the Microsoft OLE DB for OLAP provider named MSOLAP.

```
proc sql noerrorstop;
   connect to oledb (provider=msolap prompt=yes);
   select * from connection to oledb
      ( MDX::select {[Measures].[Units Shipped],
             [Measures].[Units Ordered]} on columns,
             NON EMPTY [Store].[Store Name].members on rows
              from Warehouse );
```

See the Microsoft Data Access Components Software Developer's Kit for details about MDX syntax.

The CONNECT statement requests prompting for connection information, which facilitates the connection process (especially with provider properties). The MDX:: prefix identifies the statement within the parentheses that follows the MDX statement syntax, and is not an SQL statement that is specific to OLAP. Partial output from this query might look like this:

```
Store              Units Shipped        Units Ordered

Store6             10,647               11,699
Store7             24,850               26,223
   .                  .                      .
   .                  .                      .
   .                  .                      .
```

You can use the same MDX statement with the MDX_DESCRIBE:: identifier to see the full description of each column:

```
proc sql noerrorstop;
connect to oledb (provider=msolap prompt=yes);
select * from connection to oledb
   ( MDX_DESCRIBE::select {[Measures].[Units Shipped],
                   [Measures].[Units Ordered]} on columns,
                  NON EMPTY [Store].[Store Name].members on rows
                  from Warehouse );
```

The next example creates a view of the OLAP data, which is then accessed using the PRINT procedure:

```
proc sql noerrorstop;
  connect to oledb(provider=msolap
                  props=('data source'=sqlserverdb
                     'user id'=myuserid password=mypassword));
  create view work.myview as
     select * from connection to oledb
        ( MDX::select {[MEASURES].[Unit Sales]} on columns,
               order(except([Promotion Media].[Media Type].members,
               {[Promotion Media].[Media Type].[No Media]}),
               [Measures].[Unit Sales],DESC) on rows
             from Sales )
;

proc print data=work.myview;
run;
```

In this example, full connection information is provided in the CONNECT statement, so the user is not prompted. The SQL view can be used in other PROC SQL statements, the DATA step, or in other procedures, but you cannot modify (that is, insert, update, or delete a row in) the view's underlying multidimensional data.

Naming Conventions for OLE DB

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

Because OLE DB is an application programming interface (API), data source names for files, tables, and columns are determined at run time. Since SAS 7, most SAS names can be up to 32 characters long. SAS/ACCESS Interface to OLE DB also supports file, table, and column names up to 32 characters long. If DBMS column names are longer than 32 characters, they are truncated to 32 characters. If truncating a name results in identical names, SAS generates unique names by replacing the last character with a number. For more information, see "SAS Names and Support for DBMS Names" on page 13.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= LIBNAME options determine how SAS/ACCESS Interface to OLE DB handles case sensitivity, spaces, and special characters. (For information about these options, see "Overview: LIBNAME Statement for Relational Databases " on page 93.) The default value for both options is NO for most data sources. The default value is YES for Microsoft Access, Microsoft Excel, and Microsoft SQL Server.

Data Types for OLE DB

Overview

Each data source column in a table has a name and a data type. The data type tells the data source how much physical storage to set aside for the column and the form in which the data is stored. This section includes information about OLE DB null and default values and data conversions.

OLE DB Null Values

Many relational database management systems have a special value called NULL. A DBMS NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a DBMS NULL value, it interprets it as a SAS missing value.

In most relational databases, columns can be defined as NOT NULL so that they require data (they cannot contain NULL values). When a column is defined as NOT NULL, the DBMS does not add a row to the table unless the row has a value for that column. When creating a DBMS table with SAS/ACCESS, you can use the DBNULL= on page 294 data set option to indicate whether NULL is a valid value for specified columns.

OLE DB mirrors the behavior of the underlying DBMS with regard to NULL values. See the documentation for your DBMS for information about how it handles NULL values.

For more information about how SAS handles NULL values, see "Potential Result Set Differences When Processing Null Data " on page 35.

To control how the DBMS handles SAS missing character values, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

This table shows all data types and default SAS formats that SAS/ACCESS Interface to OLE DB supports. It does not explicitly define the data types as they exist for each data source. It lists the types that each data source's data type might map to. For example, an INTEGER data type under DB2 might map to an OLE DB data type of DBTYPE_I4. All data types are supported.

Table 25.4 *OLE DB Data Types and Default SAS Formats*

OLE DB Data Type	Default SAS Format
DBTYPE_R8	none
DBTYPE_R4	none
DBTYPE_I8	none
DBTYPE_UI8	none
DBTYPE_I4	11.
DBTYPE_UI4	11.
DBTYPE_I2	6.
DBTYPE_UI2	6.
DBTYPE_I1	4.
DBTYPE_UI1	4.
DBTYPE_BOOL	1.
DBTYPE_NUMERIC	*m* or *m.n* or none, if *m* and *n* are not specified
DBTYPE_DECIMAL	*m* or *m.n* or none, if *m* and *n* are not specified
DBTYPE_CY	DOLLAR*m*.2
DBTYPE_BYTES	$*n*.
DBTYPE_STR	$*n*.
DBTYPE_BSTR	$*n*.
DBTYPE_WSTR	$*n*.
DBTYPE_VARIANT	$*n*.
DBTYPE_DBDATE	DATE9.
DBTYPE_DBTIME	TIME8.
DBTYPE_DBTIMESTAMP DBTYPE_DATE	DATETIME*m.n*, where *m* depends on precision and *n* depends on scale
DBTYPE_GUID	$*38*.

The following table shows the default data types that SAS/ACCESS Interface to OLE DB uses when creating DBMS tables. SAS/ACCESS Interface to OLE DB lets you specify non-default data types by using the DBTYPE= on page 303 data set option.

Table 25.5 *Default OLE DB Output Data Types*

SAS Variable Format	Default OLE DB Data Type
m.n	DBTYPE_R8 or DBTYPE_NUMERIC using *m.n* if the DBMS allows it
$*n*.	DBTYPE_STR using *n*
date formats	DBTYPE_DBDATE
time formats	DBTYPE_DBTIME
datetime formats	DBTYPE_DBTIMESTAMP

Chapter 26
SAS/ACCESS Interface to Oracle

Introduction to SAS/ACCESS Interface to Oracle

This section describes SAS/ACCESS Interface to Oracle. For a list of SAS/ACCESS features that are available in this interface, see “SAS/ACCESS Interface to Oracle: Supported Features” on page 87.

LIBNAME Statement Specifics for Oracle

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to Oracle supports and includes examples. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing Oracle.

LIBNAME *libref* **oracle** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`oracle`
specifies the SAS/ACCESS engine name for the Oracle interface.

connection-options
provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. Here is how these options are defined.

USER=<'>*Oracle-user-name*<'>
specifies an optional Oracle user name. If the user name contains blanks or national characters, enclose it in quotation marks. If you omit an Oracle user name and password, the default Oracle user ID OPS$*sysid* is used, if it is enabled. USER= must be used with PASSWORD=.

PASSWORD=<'>*Oracle-password*<'>
specifies an optional Oracle password that is associated with the Oracle user name. If you omit it, the password for the default Oracle user ID OPS$sysid is

used, if it is enabled. If the password contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. If you specify USER=, you must specify PASSWORD=.

PATH=<'>*Oracle-database-specification*<'>
: specifies the Oracle driver, node, and database. Aliases are required if you are using SQL*Net Version 2.0 or later. In some operating environments, you can enter the information that is required by the PATH= statement before invoking SAS.

SAS/ACCESS uses the same Oracle path designation that you use to connect to Oracle directly. See your database administrator to determine the databases that have been set up in your operating environment, and to determine the default values if you do not specify a database. On UNIX systems, the TWO_TASK environment variable is used, if set. If neither the PATH= nor the TWO_TASK values have been set, the default value is the local driver.

If you specify the appropriate system options or environment variables for Oracle, you can often omit the connection options from your LIBNAME statements. See your Oracle documentation for details.

LIBNAME-options
: define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to Oracle, with the applicable default values. For more detail about these options, see "LIBNAME Options for Relational Databases" on page 99.

***Table 26.1** SAS/ACCESS LIBNAME Options for Oracle*

Option	Default Value
ACCESS=	none
ADJUST_BYTE_SEMANTIC_COLUMN_LENGTHS=	conditional
ADJUST_NCHAR_COLUMN_LENGTHS=	YES
AUTHDOMAIN=	none
CONNECTION=	SHAREDREAD
CONNECTION_GROUP=	none
DB_LENGTH_SEMANTICS_BYTE=	YES
DB_OBJECTS= LIBNAME option	(TABLES VIEWS)
DBCLIENT_MAX_BYTES=	matches the maximum number of bytes per single character of the SAS session encoding
DBCOMMIT=	1000 when inserting rows; 0 when updating, deleting, or appending to an existing table

Option	Default Value
DBCONINIT=	none
DBCONTERM=	none
DBCREATE_TABLE_OPTS=	none
DBGEN_NAME=	DBMS
DBINDEX=	NO
DBLIBINIT=	none
DBLIBTERM=	none
DBLINK=	the local database
DBMAX_TEXT=	1024
DBMSTEMP=	NO
DBNULLKEYS=	YES
DBPROMPT=	NO
DBSERVER_MAX_BYTES=	usually 1
DBSLICEPARM=	THREADED_APPS,2
DEFER=	NO
DIRECT_EXE=	none
DIRECT_SQL=	YES
INSERTBUFF=	1 is the forced default when REREAD_EXPOSURE=YES; otherwise, 10
LOCKWAIT=	YES
MULTI_DATASRC_OPT=	NONE
OR_BINARY_DOUBLE=	
OR_ENABLE_INTERRUPT=	NO
OR_UPD_NOWHERE=	YES
PRESERVE_COL_NAMES=	NO
PRESERVE_TAB_NAMES=	NO

Option	Default Value
READBUFF=	250
READ_ISOLATION_LEVEL=<	see "Locking in the Oracle Interface" on page 718
READ_LOCK_TYPE=	NOLOCK
REREAD_EXPOSURE=	NO
SCHEMA=	SAS accesses objects in the default and public schemas
SHOW_SYNONYMS=	YES
SPOOL=	YES
SQL_FUNCTIONS=	none
SQL_FUNCTIONS_COPY=	none
SQLGENERATION=	DBMS
UPDATE_ISOLATION_LEVEL=	see "Locking in the Oracle Interface" on page 718
UPDATE_LOCK_TYPE=	NOLOCK
UPDATEBUFF=	1
UTILCONN_TRANSIENT=	NO

Oracle LIBNAME Statement Examples

In this first example, default settings are used for the connection options to make the connection. If you specify the appropriate system options or environment variables for Oracle, you can often omit the connection options from your LIBNAME statements. See your Oracle documentation for details.

```
libname myoralib oracle;
```

In the next example, the libref MYDBLIB uses SAS/ACCESS Interface to Oracle to connect to an Oracle database. The SAS/ACCESS connection options are USER=, PASSWORD=, and PATH=. PATH= specifies an alias for the database specification, which SQL*Net requires.

```
libname mydblib oracle user=testuser password=testpass path=hrdept_002;

proc print data=mydblib.employees;
   where dept='CSR010';
run;
```

Data Set Options for Oracle

All SAS/ACCESS data set options in this table are supported for Oracle. Default values are provided where applicable. For general information about this feature, see Data Set Options for Relational Databases on page 207.

Table 26.2 *SAS/ACCESS Data Set Options for Oracle*

Option	Default Value
BL_BADFILE=	creates a file in the current directory or with the default file specifications
BL_CONTROL=	creates a control file in the current directory or with the default file specifications
BL_DATAFILE=	creates a file in the current directory or with the default file specifications
BL_DEFAULT_DIR=	*<database-name>*
BL_DELETE_DATAFILE=	YES
BL_DELETE_ONLY_DATAFILE=	none
BL_DIRECT_PATH=	YES
BL_DISCARDFILE=	creates a file in the current directory or with the default file specifications
BL_INDEX_OPTIONS=	the current SQL*Loader Index options with bulk-loading
BL_LOAD_METHOD=	When loading an empty table, the default value is INSERT. When loading a table that contains data, the default value is APPEND.
BL_LOG=	If a log file does not already exist, it is created in the current directory or with the default file specifications. If a log file does already exist, the Oracle bulk loader reuses the file, replacing the contents with information from the new load.
BL_OPTIONS=	ERRORS=1000000
BL_PRESERVE_BLANKS=	NO
BL_RECOVERABLE=	YES

Option	Default Value
BL_RETURN_WARNINGS_AS_ERRORS=	NO
BL_SQLLDR_PATH=	sqldr
BL_SUPPRESS_NULLIF=	NO
BL_USE_PIPE=	NO
BULKLOAD=	NO
DBCOMMIT=	the current LIBNAME option setting
DB_ONE_CONNECT_PER_THREAD=	YES
DBCONDITION=	none
DBCREATE_TABLE_OPTS=	the current LIBNAME option setting
DBFORCE=	NO
DBGEN_NAME=	DBMS
DBINDEX=	the current LIBNAME option setting
DBKEY=	none
DBLABEL=	NO
DBLINK=	the current LIBNAME option setting
DBMASTER=	none
DBMAX_TEXT=	1024
DBNULL=	YES
DBNULLKEYS=	the current LIBNAME option setting
DBPROMPT=	the current LIBNAME option setting
DBSASLABEL=	COMPAT
DBSASTYPE=	see “Data Types for Oracle ” on page 720
DBSLICE=	none
DBSLICEPARM=	THREADED_APPS,2
DBTYPE=	see “LIBNAME Statement Data Conversions” on page 726

Option	Default Value
ERRLIMIT=	1
INSERTBUFF=	the current LIBNAME option setting
NULLCHAR=	SAS
NULLCHARVAL=	a blank character
OR_PARTITION=	an Oracle table partition name
OR_UPD_NOWHERE=	the current LIBNAME option setting
ORHINTS=	no hints
PRESERVE_COL_NAMES=	current LIBNAME option setting
READ_ISOLATION_LEVEL=	the current LIBNAME option setting
READ_LOCK_TYPE=	the current LIBNAME option setting
READBUFF=	the current LIBNAME option setting
SASDATEFORMAT=	DATETIME20.0
SCHEMA=	the current LIBNAME option setting
UPDATE_ISOLATION_LEVEL=	the current LIBNAME option setting
UPDATE_LOCK_TYPE=	the current LIBNAME option setting
UPDATEBUFF=	the current LIBNAME option setting

SQL Pass-Through Facility Specifics for Oracle

Key Information

For general information about this feature, see "SQL Pass-Through Facility" on page 401. Oracle examples are available.

Here are the SQL pass-through facility specifics for the Oracle interface.

- The *dbms-name* is **`oracle`**.
- The CONNECT statement is optional. If you omit it, an implicit connection is made with your OPS$*sysid*, if it is enabled. When you omit a CONNECT statement, an implicit connection is performed when the first EXECUTE statement or CONNECTION TO component is passed to Oracle. In this case you must use the default DBMS name **`oracle`**.

- The Oracle interface can connect to multiple databases (both local and remote) and to multiple user IDs. If you use multiple simultaneous connections, you must use an *alias* argument to identify each connection. If you do not specify an alias, the default alias, **`oracle`**, is used.
- Here are the *database-connection-arguments* for the CONNECT statement.

USER=<'>*Oracle-user-name*<'>
: specifies an optional Oracle user name. If you specify USER=, you must also specify PASSWORD=.

PASSWORD= <'>*Oracle-password*<'>
: specifies an optional Oracle password that is associated with the Oracle user name. If you omit an Oracle password, the default Oracle user ID OPS$*sysid* is used, if it is enabled. If you specify PASSWORD=, you must also specify USER=.

  ORAPW= is an alias for this option. If you do not wish to enter your Oracle password in uncoded text, see PROC PWENCODE in the *Base SAS Procedures Guide* for a method to encode it.

BUFFSIZE=*number-of-rows*
: specifies the number of rows to retrieve from an Oracle table or view with each fetch. Using this argument can improve the performance of any query to Oracle.

  By setting the value of the BUFFSIZE= argument in your SAS programs, you can find the optimal number of rows for a given query on a given table. The default buffer size is 250 rows per fetch. The value can be up to 2,147,483,647 rows per fetch, although a practical limit for most applications is less, depending on the available memory.

PRESERVE_COMMENTS
: lets you pass additional information (called *hints*) to Oracle for processing. These hints might direct the Oracle query optimizer to choose the best processing method based on your hint.

  You specify PRESERVE_COMMENTS as an argument in the CONNECT statement. You then specify the hints in the Oracle SQL query for the CONNECTION TO component. Hints are entered as comments in the SQL query and are passed to and processed by Oracle.

PATH=<'>*Oracle-database-specification*<'>
: specifies the Oracle driver, node, and database. Aliases are required if you are using SQL*Net Version 2.0 or later. In some operating environments, you can enter the information that is required by the PATH= statement before invoking SAS.

  SAS/ACCESS uses the same Oracle path designation that you use to connect to Oracle directly. See your database administrator to determine the path designations that have been set up in your operating environment, and to determine the default value if you do not specify a path designation. On UNIX systems, the TWO_TASK environment variable is used, if set. If neither PATH= nor TWO_TASK have been set, the default value is the local driver.

Examples

This example uses the alias DBCON for the DBMS connection (the connection alias is optional):

```
proc sql;
   connect to oracle as dbcon
       (user=testuser password=testpass buffsize=100
        path='myorapath');
quit;
```

This next example connects to Oracle and sends it two EXECUTE statements to process.

```
proc sql;
   connect to oracle (user=testuser password=testpass);
   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;
```

As shown in highlighted text, this example performs a query on the CUSTOMERS Oracle table:

```
proc sql;
connect to oracle (user=testuser password=testpass);
select *
   from connection to oracle
     (select * from customers
      where customer like '1%');
    disconnect from oracle;
quit;
```

In this example, the PRESERVE_COMMENTS argument is specified after the USER= and PASSWORD= arguments. The Oracle SQL query is enclosed in the required parentheses. The SQL INDX command identifies the index for the Oracle query optimizer to use to process the query. Multiple hints are separated with blanks.

```
proc sql;
connect to oracle as mycon(user=testuser
        password=testpass preserve_comments);
select *
   from connection to mycon
     (select /* +indx(empid) all_rows */
          count(*) from employees);
quit;
```

Hints are not preserved in this next example, which uses the prior style of syntax:

```
execute ( delete /*+ FIRST_ROWS */ from test2 where num2=1)
           by &db
```

Using the new syntax, hints are preserved in this example:

```
execute by &db
           ( delete /*+ FIRST_ROWS */ from test2 where num2=2);
```

Autopartitioning Scheme for Oracle

Overview

Without user-specified partitioning from the DBSLICE= option, SAS/ACCESS Interface to Oracle tries to use its own partitioning techniques. The technique that it chooses depends on whether the table is physically partitioned on the Oracle server.

For general information about this feature, see "Autopartitioning Techniques in SAS/ACCESS" on page 63.

Note: Threaded Reads for the Oracle engine on z/OS are not supported.

Partitioned Oracle Tables

If you are working with a partitioned Oracle table, it is recommended that you let the Oracle engine partition the table for you. The Oracle engine gathers all partition information needed to perform a threaded Read on the table.

A partitioned Oracle table is a good candidate for a threaded Read because each partition in the table can be read in parallel with little contention for disk resources. If the Oracle engine determines that the table is partitioned, it makes the same number of connections to the server as there are partitions, as long as the maximum number of threads that are allowed is higher than the number of partitions. Each connection retrieves rows from a single partition.

If the value of the maximum number of allowed threads is less than the number of partitions on the table, a single connection reads multiple partitions. Each connection retrieves rows from a single partition or multiple partitions. However, you can use the DB_ONE_CONNECT_PER_THREAD= data set option so that there is only one connection per thread.

The following example shows how to do this. First, create the SALES table in Oracle.

```
CREATE TABLE SALES (acct_no NUMBER(5),
 acct_name CHAR(30), amount_of_sale NUMBER(6), qtr_no INTEGER)
PARTITION BY RANGE (qtr_no)
(PARTITION sales1 VALUES LESS THAN (2) TABLESPACE ts0,
PARTITION sales2 VALUES LESS THAN (2) TABLESPACE ts1,
PARTITION sales3 VALUES LESS THAN (2) TABLESPACE ts2,
PARTITION sales4 VALUES LESS THAN (2) TABLESPACE ts3)
```

Performing a threaded Read on this table with the following code, SAS makes four separate connections to the Oracle server and each connection reads from each partition. Turning on SASTRACE= shows the SQL that is generated for each connection.

```
libname x oracle user=testuser path=oraserver;
data new;
set x.SALES (DBSLICEPARM=(ALL,10));
run;

ORACLE: SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE", "QTR_NO" FROM SALES
partition (SALES2)
ORACLE: SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE", "QTR_NO" FROM SALES
partition (SALES3)
```

```
ORACLE: SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE", "QTR_NO" FROM SALES
partition (SALES1)
ORACLE: SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE", "QTR_NO" FROM SALES
partition (SALES4)
```

Using the following code, SAS instead makes two separate connections to the Oracle server and each connection reads from two different partitions.

```
libname x oracle user=testuser path=oraserver;
data new;
set x.SALES (DBSLICEPARM=(ALL,2));
run;

ORACLE: SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE", "QTR_NO" FROM SALES
partition (SALES2) UNION ALL SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE",
"QTR_NO" FROM SALES partition (SALES3)
ORACLE: SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE", "QTR_NO" FROM SALES
partition (SALES1) UNION ALL  SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE",
"QTR_NO" FROM SALES partition (SALES4)
```

Using DB_ONE_CONNECT_PER_THREAD=NO, however, you can override the default behavior of limiting the number of connections to the number of threads. As shown below, SAS makes four separate connections to the Oracle server and each connection reads from each of the partition.

```
libname x oracle user=testuser path=oraserver;
data new;
set x.SALES (DBSLICEPARM=(ALL,2)  DB_ONE_CONNECT_PER_THREAD=NO );
run;

ORACLE: SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE", "QTR_NO" FROM SALES
partition (SALES2)
ORACLE: SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE", "QTR_NO" FROM SALES
partition (SALES3)
ORACLE: SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE", "QTR_NO" FROM SALES
partition (SALES1)
ORACLE: SELECT "ACCT_NO","ACCT_NAME", "AMOUNT_OF_SALE", "QTR_NO" FROM SALES
partition (SALES4)
```

The second parameter of the DBSLICEPARM= LIBNAME option determines the number of threads to read the table in parallel. The number of partitions on the table, the maximum number of allowed threads, and the value of DB_ONE_CONNECT_PER_THREAD= determine the number of connections to the Oracle server for retrieving rows from the table.

Nonpartitioned Oracle Tables

If the table is not partitioned, and the DBSLICE= option is not specified, Oracle resorts to the MOD function. (See "Autopartitioning Techniques in SAS/ACCESS" on page 63.) With this technique, the engine makes *N* connections, and each connection retrieves rows based on a WHERE clause as follows:

```
WHERE ABS(MOD(ModColumn,N))=R
```

- ModColumn is a column in the table of type integer and is not used in any user specified WHERE clauses. (The engine selects this column. If you do not think this is the ideal partitioning column, you can use the DBSLICE= data set option to override this default behavior.)

- R varies from 0 to (*N*-1) for each of the *N* WHERE clauses.
- *N* defaults to 2, and *N* can be overridden with the second parameter in the DBSLICEPARM= data set option.

The Oracle engine selects the ModColumn to use in this technique. Any numeric column with zero scale value can qualify as the ModColumn. However, if a primary key column is present, it is preferred over all others. Generally, values in the primary key column are in a serial order and yield an equal number of rows for each connection. This example illustrates the point:

```
create table employee (empno number(10) primary key,
  empname varchar2(20), hiredate date,
  salary number(8,2), gender char(1));
```

Performing a threaded Read on this table causes Oracle to make two separate connections to the Oracle server. SAS tracing shows the SQL generated for each connection:

```
data new;
set x.EMPLOYEE(DBSLICPARM=ALL);
run;
ORACLE: SELECT "EMPNO", "EMPNAME", "HIREDATE", "SALARY", "GENDER"
FROM EMPLOYEE WHERE ABS(MOD("EMPNO",2))=0
ORACLE: SELECT "EMPNO", "EMPNAME", "HIREDATE", "SALARY", "GENDER"
FROM EMPLOYEE WHERE ABS(MOD("EMPNO",2))=1
```

EMPNO, the primary key, is selected as the MOD column.

The success of MOD depends on the distribution of the values within the selected ModColumn and the value of *N*. Ideally, the rows are distributed evenly among the threads.

You can alter the *N* value by changing the second parameter of DBSLICEPARM= LIBNAME option.

Performance Summary

There are times that you might not see an improvement in performance with the MOD technique. It is possible that the engine might not be able to find a column that qualifies as a good MOD column. In these situations, you can explicitly specify DBSLICE= data set option to force a threaded Read and improve performance.

It is a good policy to let the engine autopartition and intervene with DBSLICE= only when necessary.

Temporary Table Support for Oracle

Overview

For general information about this feature, see "Temporary Table Support for SAS/ACCESS " on page 42.

Establishing a Temporary Table

A temporary table in Oracle persists just like a regular table, but contains either session-specific or transaction-specific data. Whether the data is session- or transaction-specific is determined by what is specified with the ON COMMIT keyword when you create the temporary table.

In the SAS context, you must use the LIBNAME option, CONNECTION=SHARED, before data in a temporary table persists over procedure and DATA step boundaries. Without this option, the temporary table persists but the data within it does not.

For data to persist between explicit SQL pass-through boundaries, you must use the LIBNAME option, CONNECTION=GLOBAL.

If you have a SAS data set and you want to join it with an Oracle table to generate a report, the join normally occurs in SAS. However, using a temporary table, you can also have the join occur on the Oracle server.

Syntax

Here is the syntax to create a temporary table for which the data is transaction-specific (default):

CREATE GLOBAL TEMPORARY TABLE *table name*
ON COMMIT DELETE ROWS

Here is the syntax to create a temporary table for which the data is session-specific:

CREATE GLOBAL TEMPORARY TABLE *table name*
ON COMMIT PRESERVE ROWS

Terminating a Temporary Table

You can drop a temporary table at any time, or allow it to be implicitly dropped when the connection is terminated. Temporary tables do not persist beyond the scope of a single connection.

Example

In this example, a temporary table, TEMPTRANS, is created in Oracle to match the TRANS SAS data set, using the SQL pass-through facility.

```
proc sql;
   connect to oracle (user=scott pw=tiger path=oraclev9);
   execute (create global temporary table TEMPTRANS
           (empid number, salary number)) by oracle;
quit;

libname ora oracle user=scott pw=tiger path=oracle9 connection=shared;

/* load the data from the TRANS table into the Oracle temporary table */
proc append base=ora.TEMPTRANS data=TRANS;
run;

proc sql;
/* do the join on the DBMS server */
```

```
   select lastname, firstname, salary
      from ora.EMPLOYEES T1, ora.TEMPTRANS  T2
      where  T1.empno=T2.empno;
quit;
```

ACCESS Procedure Specifics for Oracle

Overview

For general information about this feature, see Chapter 1, “ACCESS Procedure,” on page 881. Oracle examples are available.

The Oracle interface supports all ACCESS procedure statements in line and batch modes. See “About ACCESS Procedure Statements” on page 882.

Here are the ACCESS procedure specifics for Oracle.

- The PROC ACCESS step DBMS= value is **Oracle**.
- Here are the *database-description-statements* that PROC ACCESS uses:

  USER=<'>*Oracle-user-name*<'>
  : specifies an optional Oracle user name. If you omit an Oracle password and user name, the default Oracle user ID OPS$*sysid* is used if it is enabled. If you specify USER=, you must also specify ORAPW=.

  ORAPW= <'>*Oracle-password*<'>
  : specifies an optional Oracle password that is associated with the Oracle user name. If you omit ORAPW=, the password for the default Oracle user ID OPS $sysid is used, if it is enabled. If you specify ORAPW=, you must also specify USER=.

  PATH=<'>*Oracle-database-specification*<'>
  : specifies the Oracle driver, node, and database. Aliases are required if you are using SQL*Net Version 2.0 or later. In some operating environments, you can enter the information that is required by the PATH= statement before invoking SAS.

    SAS/ACCESS uses the same Oracle path designation that you use to connect to Oracle directly. See your database administrator to determine the path designations that are set up in your operating environment and to determine the default value if you do not specify a path designation. On UNIX systems, the TWO_TASK environment variable is used, if set. If neither PATH= nor TWO_TASK have been set, the default value is the local driver.

- Here is the PROC ACCESS step TABLE= statement:

  TABLE= <'><*Oracle-table-name*><'>;
  : specifies the name of the Oracle table or Oracle view on which the access descriptor is based. This statement is required. The *Oracle-table-name* argument can be up to 30 characters long and must be a valid Oracle table name. If the table name contains blanks or national characters, enclose it in quotation marks.

Examples

This example creates an access descriptor and a view descriptor based on Oracle data.

```
options linesize=80;

libname adlib 'SAS-library';
libname vlib 'SAS-library';

proc access dbms=oracle;

/* create access descriptor */

   create adlib.customer.access;
   user=testuser;
   orapw=testpass;
   table=customers;
   path='myorapath';
   assign=yes;
   rename customer=custnum;
   format firstorder date9.;
   list all;

/* create view descriptor */

   create vlib.usacust.view;
   select customer state zipcode name
          firstorder;
   subset where customer like '1%';
run;
```

This next example creates another view descriptor that is based on the ADLIB.CUSTOMER access descriptor. You can then print the view.

```
/* create socust view */

proc access dbms=oracle accdesc=adlib.customer;
   create vlib.socust.view;
   select customer state name contact;
   subset where state in ('NC', 'VA', 'TX');
run;

/* print socust view */

proc print data=vlib.socust;
title 'Customers in Southern States';
run;
```

DBLOAD Procedure Specifics for Oracle

Overview

For general information about this feature, see Chapter 2, “DBLOAD Procedure,” on page 901. Oracle examples are available.

The Oracle interface supports all DBLOAD procedure statements. See “About DBLOAD Procedure Statements” on page 902.

Here are the DBLOAD procedure specifics for Oracle.

- The PROC DBLOAD step DBMS= value is **`Oracle`**.
- Here are the *database-description-statements* that PROC DBLOAD uses:

USER=<'>*Oracle-user-name*<'>
: specifies an optional Oracle user name. If you omit an Oracle password and user name, the default Oracle user ID OPS$*sysid* is used if it is enabled. If you specify USER=, you must also specify ORAPW=.

ORAPW= <'>*Oracle-password*<'>
: specifies an optional Oracle password that is associated with the Oracle user name. If you omit ORAPW=, the password for the default Oracle user ID OPS $sysid is used, if it is enabled. If you specify ORAPW=, you must also specify USER=.

PATH=<'>*Oracle-database-specification*<'>
: specifies the Oracle driver, node, and database. Aliases are required if you are using SQL*Net Version 2.0 or later. In some operating environments, you can enter the information that is required by the PATH= statement before invoking SAS.

  SAS/ACCESS uses the same Oracle path designation that you use to connect to Oracle directly. See your database administrator to determine the path designations that are set up in your operating environment, and to determine the default value if you do not specify a path designation. On UNIX systems, the TWO_TASK environment variable is used, if set. If neither PATH= nor TWO_TASK have been set, the default value is the local driver.

TABLESPACE= <'>*Oracle-tablespace-name*<'>;
: specifies the name of the Oracle table space where you want to store the new table. The *Oracle-tablespace-name* argument can be up to 18 characters long and must be a valid Oracle table space name. If the name contains blanks or national characters, enclose the entire name in quotation marks.

  If TABLESPACE= is omitted, the table is created in your default table space that is defined by the Oracle database administrator at your site.

- Here is the PROC DBLOAD step TABLE= statement:

TABLE= <'><*Oracle-table-name*><'>;
: specifies the name of the Oracle table or Oracle view on which the access descriptor is based. This statement is required. The *Oracle-table-name* argument can be up to 30 characters long and must be a valid Oracle table name. If the table name contains blanks or national characters, enclose the name in quotation marks.

Examples

The following example creates a new Oracle table, EXCHANGE, from the DLIB.RATEOFEX data file. (The DLIB.RATEOFEX data set is included in the sample data shipped with your software.) An access descriptor, ADLIB.EXCHANGE, based on the new table, is also created. The PATH= statement uses an alias to connect to a remote Oracle 7 Server database.

The SQL statement in the second DBLOAD procedure sends an SQL GRANT statement to Oracle. You must be granted Oracle privileges to create new Oracle tables or to grant privileges to other users. The SQL statement is in a separate procedure because you cannot create a DBMS table and reference it within the same DBLOAD step. The new table is not created until the RUN statement is processed at the end of the first DBLOAD step.

```
libname adlib 'SAS-library';
libname dlib 'SAS-library';

proc dbload dbms=oracle data=dlib.rateofex;
   user=testuser;
   orapw=testpass;
   path='myorapath';
   table=exchange;
   accdesc=adlib.exchange;
   rename fgnindol=fgnindolar 4=dolrsinfgn;
   nulls updated=n fgnindol=n 4=n country=n;
   load;
run;

proc dbload dbms=oracle;
   user=testuser;
   orapw=testpass;
   path='myorapath';
   sql grant select on testuser.exchange to pham;
run;
```

This next example uses the APPEND option to append rows from the INVDATA data set, which was created previously, to an existing Oracle table named INVOICE.

```
proc dbload dbms=oracle data=invdata append;
  user=testuser;
  orapw=testpass;
  path='myorapath';
  table=invoice;
  load;
run;
```

Maximizing Oracle Performance

There are several measures that you can take to optimize performance when using SAS/ACCESS Interface to Oracle. For general information about improving performance when using SAS/ACCESS engines, see "Performance Considerations" on page 39.

SAS/ACCESS Interface to Oracle has several options that you can use to further improve performance.

- For tips on multi-row processing, see these LIBNAME options: INSERTBUFF, UPDATEBUFF, and READBUFF.
- For instructions on using the Oracle SQL*Loader to increase performance when loading rows of data into Oracle tables, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

If you choose the transactional inserting of rows (specify BULKLOAD=NO), you can improve performance by inserting multiple rows at a time. This performance enhancement is comparable to using the Oracle SQL*Loader Conventional Path Load. For more information about inserting multiple rows, see the INSERTBUFF= option.

Passing SAS Functions to Oracle

SAS/ACCESS Interface to Oracle passes the following SAS functions to Oracle for processing. Where the Oracle function name differs from the SAS function name, the Oracle name appears in parentheses. For more information, see “Passing Functions to the DBMS Using PROC SQL” on page 46.

ABS	LOG10
ARCOS (ACOS)	LOG2
ARSIN (ASIN)	LOWCASE (LCASE)
ATAN	MAX
AVG	MIN
CEIL	SIGN
COS	SIN
COSH	SINH
COUNT	SOUNDEX
DATEPART	SQRT
DATETIME (SYSDATE)	STRIP (TRIM)
DTEXTDAY	SUM
DTEXTMONTH	TAN
DTEXTYEAR	TRANSLATE
EXP	TRIMN (RTRIM)
FLOOR	UPCASE (UPPER)
LOG	

When the Oracle server is 9i or above, these additional functions are also passed.

COALESCE	MONTH (EXTRACT)
DAY (EXTRACT)	YEAR (EXTRACT)

SQL_FUNCTIONS=ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to Oracle. Due to incompatibility in date and time functions between Oracle and SAS, Oracle might not process them correctly. Check your results to determine whether these functions are working as expected. For more information, see SQL_FUNCTIONS= LIBNAME option on page 185.

DATE (TRUNC(SYSDATE))*	ROUND
DATEPART (TRUNC)*	SUBSTR
INDEX (INSTR)	TODAY (TRUNC(SYSDATE))*
LENGTH	TRANWRD (REPLACE)
MOD	TRIM (TRIMN)

*Only in WHERE or HAVING clauses.

Passing Joins to Oracle

Before a join can pass to Oracle, all of these components of the LIBNAME statements must match exactly:

- user ID (USER=)
- password (PASSWORD=)
- path (PATH=)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Bulk Loading for Oracle

Overview

SAS/ACCESS Interface to Oracle can call the Oracle SQL*Loader (SQLLDR) when you set the data set option BULKLOAD=YES. The Oracle bulk loader provides superior load performance, so you can rapidly move data from a SAS file into an Oracle table. Future releases of SAS/ACCESS software will continue to use powerful Oracle tools to improve load performance. An Oracle bulk-load example is available.

Here are the Oracle bulk-load data set options. For detailed information about these options, see Data Set Options for Relational Databases on page 207.

- BL_BADFILE=
- BL_CONTROL=
- BL_DATAFILE=
- BL_DELETE_DATAFILE=
- BL_DIRECT_PATH=
- BL_DISCARDFILE=
- BL_INDEX_OPTIONS=
- BL_LOAD_METHOD=
- BL_LOG=
- BL_OPTIONS= on page 255
- BL_PARFILE=
- BL_PRESERVE_BLANKS=
- BL_RECOVERABLE=
- BL_RETURN_WARNINGS_AS_ERRORS=
- BL_SQLLDR_PATH=
- BL_SUPPRESS_NULLIF=
- BULKLOAD= on page 274

BULKLOAD= calls the Oracle bulk loader so that the Oracle engine can move data from a SAS file into an Oracle table using SQL*Loader (SQLLDR).

Note: SQL*Loader direct-path load has a number of limitations. See your Oracle utilities documentation for details, including tips to boost performance. You can also view the SQL*Loader log file instead of the SAS log for information about the load when you use bulk load.

Interactions with Other Options

When BULKLOAD=YES, the following statements are true:

- The DBCOMMIT=, DBFORCE=, ERRLIMIT=, and INSERTBUFF= options are ignored.
- If NULLCHAR=SAS, and the NULLCHARVAL= value is blank, the SQL*Loader attempts to insert a NULL instead of a NULLCHARVAL value.
- If NULLCHAR=NO, and the NULLCHARVAL value is blank, the SQL*Loader attempts to insert a NULL even if the DBMS does not allow NULL.

  To avoid this result, set BL_PRESERVE_BLANKS=YES or set NULLCHARVAL to a non-blank value and then replace the non-blank value with blanks after processing, if necessary.

z/OS Specifics

When you use bulk load in the z/OS operating environment, the files that the SQL*Loader uses must conform to z/OS data set standards. The data sets can be either sequential data sets or partitioned data sets. Each filename that is supplied to the SQL*Loader are subject to extension and FNA processing.

If you do not specify filenames using data set options, default names in the form of *userid*.SAS.*data-set-extension* apply. The *userid* is the TSO prefix when running under TSO, and it is the PROFILE PREFIX in batch. The *data-set-extensions* are:

- BAD for the bad file
- CTL for the control file
- DAT for the data file
- DSC for the discard file
- LOG for the log file

If you want to specify filenames using data set options, you must use one of these forms:

- */DD/ddname*
- */DD/ddname(membername)*
- *Name*

For detailed information about these forms, see the SQL*Loader chapter in the Oracle user's guide for z/OS.

The Oracle engine runs the SQL*Loader by issuing a host-system command from within your SAS session. The data set where the SQLLDR executable file resides must be available to your TSO session or allocated to your batch job. Check with your system administrator if you do not know the name or availability of the data set that contains the SQLLDR executable file.

On z/OS, the bad file and the discard file are, by default, not created in the same format as the data file. This makes it difficult to load the contents of these files after making corrections. See the section on SQL*Loader file attributes in the SQL*Loader section in the Oracle user's guide for z/OS for information about overcoming this limitation.

Example

This example shows you how to create and use a SAS data set to create and load to a large Oracle table, FLIGHTS98. This load uses the SQL*Loader direct path method because you specified BULKLOAD=YES. BL_OPTIONS= passes the specified SQL*Loader options to SQL*Loader when it is invoked. In this example, you can use the ERRORS= option to have up to 899 errors in the load before it terminates and the LOAD= option loads the first 5,000 rows of the input data set, SASFLT.FLT98.

```
options yearcutoff=1925;   /* included for Year 2000 compliance */

libname sasflt 'SAS-library';
libname ora_air oracle user=testuser password=testpass
   path='ora8_flt' schema=statsdiv;

data sasflt.flt98;
   input flight $3. +5 dates date7. +3 depart time5. +2 orig $3.
         +3 dest $3.  +7 miles +6 boarded +6 capacity;
   format dates date9. depart time5.;
   informat dates date7. depart time5.;
   datalines;
114     01JAN98    7:10  LGA   LAX       2475       172       210
202     01JAN98   10:43  LGA   ORD        740       151       210
219     01JAN98    9:31  LGA   LON       3442       198       250

<…10,000 more observations…>

proc sql;
create table ora_air.flights98
(BULKLOAD=YES BL_OPTIONS='ERRORS=899,LOAD=5000') as
   select * from sasflt.flt98;
quit;
```

During a load, certain SQL*Loader files are created, such as the data, log, and control files. Unless otherwise specified, they are given a default name and written to the current directory. For this example, the default names would be **bl_flights98.dat**, **bl_flights98.log**, and **bl_flights98.ctl**.

Locking in the Oracle Interface

The following LIBNAME and data set options let you control how the Oracle interface handles locking. For general information about an option, see "LIBNAME Options for Relational Databases" on page 99.

READ_LOCK_TYPE= NOLOCK | ROW | TABLE
The default value is NOLOCK. Here are the valid values for this option:

- NOLOCK — table locking is not used during the reading of tables and views.

- ROW — the Oracle ROW SHARE table lock is used during the reading of tables and views.
- TABLE — the Oracle SHARE table lock is used during the reading of tables and views.

If you set READ_LOCK_TYPE= to either TABLE or ROW, you must also set the CONNECTION= option to UNIQUE. If not, an error occurs.

UPDATE_LOCK_TYPE= NOLOCK | ROW | TABLE
The default value is NOLOCK. Here are the valid values for this option:

- ROW — the Oracle ROW SHARE table lock is used during the reading of tables and views for update.
- TABLE — the Oracle EXCLUSIVE table lock is used during the reading of tables and views for update.
- NOLOCK — table locking is not used during the reading of tables and views for update.
  - If OR_UPD_NOWHERE=YES, updates are performed using serializable transactions.
  - If OR_UPD_NOWHERE=NO, updates are performed using an extra WHERE clause to ensure that the row has not been updated since it was first read. Updates might fail under these conditions, because other users might modify a row after the row was read for update.

READ_ISOLATION_LEVEL= READCOMMITTED | SERIALIZABLE
Oracle supports the READCOMMITTED and SERIALIZABLE read isolation levels, as defined in the following table. The SPOOL= option overrides the READ_ISOLATION_LEVEL= option. The READ_ISOLATION_LEVEL= option should be rarely needed because the SAS/ACCESS engine chooses the appropriate isolation level based on other locking options.

Table 26.3 *Isolation Levels for Oracle*

Isolation Level	Definition
SERIALIZABLE	Does not allow dirty Reads, nonrepeatable Reads, or phantom Reads.
READCOMMITTED	Does not allow dirty Reads; does allow nonrepeatable Reads and phantom Reads

UPDATE_ISOLATION_LEVEL= READCOMMITTED | SERIALIZABLE
Oracle supports the READCOMMITTED and SERIALIZABLE isolation levels, as defined in the preceding table, for updates.

This option should be rarely needed because the SAS/ACCESS engine chooses the appropriate isolation level based on other locking options.

Naming Conventions for Oracle

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= LIBNAME options determine how SAS/ACCESS Interface to Oracle handles case sensitivity, spaces, and special characters. For information about these options, see “Overview: LIBNAME Statement for Relational Databases ” on page 93.

You can name such Oracle objects as tables, views, columns, and indexes. For the Oracle 7 Server, objects also include database triggers, procedures, and stored functions. They follow these naming conventions.

- A name must be from 1 to 30 characters long. Database names are limited to 8 characters, and link names are limited to 128 characters.
- A name must begin with a letter. However, if you enclose the name in double quotation marks, it can begin with any character.
- A name can contain the letters A through Z, the digits 0 through 9, the underscore (_), $, and #. If the name appears within double quotation marks, it can contain any characters, except double quotation marks.
- Names are not case sensitive. For example, **`CUSTOMER`** and **`Customer`** are the same. However, if you enclose an object names in double quotation marks, it is case sensitive.
- A name cannot be an Oracle reserved word.
- A name cannot be the same as another Oracle object in the same schema.

Data Types for Oracle

Overview

Every column in a table has a name and a data type. The data type tells Oracle how much physical storage to set aside for the column and the form in which the data is stored. This section includes information about Oracle data types, null and default values, and data conversions.

For detailed information about Oracle data types, see the *Oracle Database SQL Reference*.

SAS/ACCESS Interface to Oracle does not support the Oracle MLSLABEL data type.

Character Data

CHAR (*n*)
: contains fixed-length character string data with a length of *n*, where *n* must be at least 1 and cannot exceed 255 characters. (The limit is 2,000 characters with an Oracle8 Server.) The Oracle 7 Server CHAR data type is not equivalent to the Oracle Version 6 CHAR data type. The Oracle 7 Server CHAR data type is new with the Oracle 7 Server and uses blank-padded comparison semantics.

CLOB (character large object)
: contains varying-length character string data that is similar to type VARCHAR2. Type CLOB is character data of variable length with a maximum length of 2 gigabytes. You can define only one CLOB column per table. Available memory considerations might also limit the size of a CLOB data type.

VARCHAR2(*n*)
: contains character string data with a length of *n*, where *n* must be at least 1 and cannot exceed 2000 characters. (The limit is 4,000 characters with an Oracle8 Server.) The VARCHAR2 data type is equivalent to the Oracle Version 6 CHAR data type except for the difference in maximum lengths. The VARCHAR2 data type uses nonpadded comparison semantics.

Numeric Data

BINARY_DOUBLE
: specifies a floating-point double binary with a precision of 38. A floating-point value can either specify a decimal point anywhere from the first to the last digit or omit the decimal point. A scale value does not apply to floating-point double binaries because there is no restriction on the number of digits that can appear after the decimal point. Compared to the NUMBER data type, BINARY_DOUBLE provides substantially faster calculations, plus tighter integration with XML and Java environments.

BINARY_FLOAT
: specifies a floating-point single binary with a precision of 38. A floating-point value can either specify a decimal point anywhere from the first to the last digit or omit the decimal point. A scale value does not apply to floating-point single binaries because there is no restriction on the number of digits that can appear after the decimal point. Compared to the NUMBER data type, BINARY_FLOAT provides substantially faster calculations, plus tighter integration with XML and Java environments.

NUMBER
: specifies a floating-point number with a precision of 38. A floating-point value can either specify a decimal point anywhere from the first to the last digit or omit the decimal point. A scale value does not apply to floating-point numbers because there is no restriction on the number of digits that can appear after the decimal point.

NUMBER(*p*)
: specifies an integer of precision *p* that can range from 1 to 38 and a scale of 0.

NUMBER(*p,s*)
: specifies a fixed-point number with an implicit decimal point, where *p* is the total number of digits (precision) and can range from 1 to 38, and *s* is the number of digits to the right of the decimal point (scale) and can range from -84 to 127.

Date, Timestamp, and Interval Data

Overview

DATE
: contains date values. Valid dates are from January 1, 4712 BC to December 31, 4712 AD. The default format is DD-MON-YY, for example '05-OCT-98'.

TIMESTAMP
: contains double binary data that represents the SAS DATETIME value, where *d* is the fractional second precision that you specify on the column and *w* is derived from the value of *d*. The default value of *d* is 6. Although you can override the default format to view more than six decimal places, the accuracy of th value is not guaranteed. When you update or insert TIMESTAMP into SAS, the value is converted to a string value with the form of DDMONYYYY:HH24:MI:SS:SS, where the fractional second precision defaults to *d* in the SAS DATETIME format. This value is then inserted into Oracle, using this string:

TO_TIMESTAMP(:"TS",'DDMONYYYY:HH24:MI:SSXFF',
'NLS_DATE_LANGUAGE=American')

TIMESTAMP WITH TIME ZONE
contains a character string that is *w* characters long, where *w* is derived from the fractional second precision that you specify on the column and the additional width needed to specify the TIMEZONE value. When you update or insert TIMESTAMP into SAS, the value is inserted into the column. The NLS_TIMESTAMP_TZ_FORMAT parameter determines the expected format. An error results if users do not ensure that the string matches the expected (default) format.

TIMESTAMP WITH LOCAL TIME ZONE
contains double binary data that represents the SAS DATETIME value. (This data type is the same as TIMESTAMP.) SAS returns whatever Oracle returns. When you update or insert TIMESTAMP into SAS, the value is assumed to be a number representing the number of months.

Note: A fix for Oracle Bug 2422838 is available in Oracle 9.2.0.5 and above.

INTERVAL YEAR TO MONTH
contains double binary data that represents the number of months, where *w* is based on the Year precision value that you specify on the column: INTERVAL YEAR*(p)* TO MONTH. When you update or insert TIMESTAMP into SAS, the value is assumed to be a number representing the number of months.

INTERVAL DAY TO SECOND
contains double binary data that represents the number of seconds, where *d* is the same as the fractional second precision that you specify on the column: INTERVAL DAY*(p)* TO SECOND*(d)*. The width *w* is derived based on the values for DAY precision *(p)* and SECOND *d* precision.

For compatibility with other DBMSs, Oracle supports the syntax for a wide variety of numeric data types, including DECIMAL, INTEGER, REAL, DOUBLE-PRECISION, and SMALLINT. All forms of numeric data types are actually stored in the same internal Oracle NUMBER format. The additional numeric data types are variations of precision and scale. A null scale implies a floating-point number, and a non-null scale implies a fixed-point number.

Examples

Here is a TIMESTAMP example.

```
%let PTCONN= %str(user=scott pw=tiger path=oraclev10);
%let CONN= %str(user=scott pw=tiger  path=oraclev10);

options sastrace=",,," sastraceloc=saslog nostsuffix;

proc sql;
connect to oracle ( &PTCONN);

/*execute ( drop table EMP_ATTENDANCE) by oracle;*/

execute ( create table EMP_ATTENDANCE (  EMP_NAME VARCHAR2(10),
   arrival_timestamp TIMESTAMP, departure_timestamp TIMESTAMP ) ) by oracle;
execute ( insert into EMP_ATTENDANCE values
   ('John Doe', systimestamp, systimestamp+.2) ) by oracle;
execute ( insert into EMP_ATTENDANCE values
   ('Sue  Day', TIMESTAMP'1980-1-12 10:13:23.33',
       TIMESTAMP'1980-1-12 17:13:23.33' )) by oracle;
```

```
quit;

libname ora  oracle &CONN

proc contents data=ora.EMP_ATTENDANCE; run;

proc sql;
 /* reading TIMESTAMP datatype */
select * from ora.EMP_ATTENDANCE;
quit;

 /* appending to TIMESTAMP datatype */
data work.new;
 EMP_NAME='New  Bee1';
 ARRIVAL_TIMESTAMP='30sep1998:14:00:35.00'dt;
 DEPARTURE_TIMESTAMP='30sep1998:17:00:14.44'dt;  output;
 EMP_NAME='New  Bee2';
 ARRIVAL_TIMESTAMP='30sep1998:11:00:25.11'dt;
 DEPARTURE_TIMESTAMP='30sep1998:14:00:35.27'dt;  output;
 EMP_NAME='New  Bee3';
 ARRIVAL_TIMESTAMP='30sep1998:08:00:35.33'dt;
 DEPARTURE_TIMESTAMP='30sep1998:17:00:35.10'dt; output;
 format ARRIVAL_TIMESTAMP datetime23.2;
 format DEPARTURE_TIMESTAMP datetime23.2;
run;

title2 'After append';
proc append data=work.new base=ora.EMP_ATTENDANCE ; run;
proc print data=ora.EMP_ATTENDANCE ; run;

 /* updating TIMESTAMP datatype */
proc sql;
update ora.EMP_ATTENDANCE set ARRIVAL_TIMESTAMP=.
   where EMP_NAME like '%Bee2%' ;

select * from ora.EMP_ATTENDANCE ;

delete from ora.EMP_ATTENDANCE where EMP_NAME like '%Bee2%' ;

select * from ora.EMP_ATTENDANCE ;

 /* OUTPUT: Creating a brand new table using Data Step*/
data work.sasdsfsec; c_ts='30sep1998:14:00:35.16'dt; k=1; output;
                     c_ts='.'dt; k=2; output;
                     format c_ts datetime23.2; run;

/* picks default TIMESTAMP type */
options sastrace=",,,d" sastraceloc=saslog nostsuffix;
data ora.tab_tsfsec; set work.sasdsfsec; run;
options sastrace=",,," sastraceloc=saslog nostsuffix;
proc datasets library=ora;
  delete tab_tsfsec;run;

/* Override the default datatype */
options sastrace=",,,d" sastraceloc=saslog nostsuffix;
data ora.tab_tsfsec (dbtype=(c_ts='timestamp(3)'));
```

```
 c_ts='30sep1998:14:00:35'dt;
 format c_ts datetime23.; run;
options sastrace=",,," sastraceloc=saslog nostsuffix;
proc datasets library=ora;
  delete tab_tsfsec;run;

proc print data=ora.tab_tsfsec; run;

/* OUTPUT: Brand new table creation with bulkload=yes */
title2 'Test OUTPUT with bulkloader';
proc datasets library=ora;
  delete tab_tsfsec;run;

/* picks default TIMESTAMP type */
data ora.tab_tsfsec (bulkload=yes); set work.sasdsfsec; run;
proc print data=ora.tab_tsfsec; run;
```

Here is an INTERVAL YEAR TO MONTH example.

```
proc sql;
connect to oracle ( &PTCONN);
execute ( drop table PRODUCT_INFO) by oracle;

execute (
 create table PRODUCT_INFO (  PRODUCT VARCHAR2(20),
    LIST_PRICE number(8,2),
    WARRANTY_PERIOD INTERVAL YEAR(2) TO MONTH )
)by oracle;
execute (
  insert into PRODUCT_INFO values ('Dish Washer', 4000, '02-00')
)by Oracle;
execute (
  insert into PRODUCT_INFO values ('TV', 6000, '03-06')
)by Oracle;
quit;

proc contents data=ora.PRODUCT_INFO; run;

 /* Shows WARRANTY_PERIOD as number of months */
proc print data=ora.PRODUCT_INFO; run;

 /* Shows WARRANTY_PERIOD in a format just like in Oracle*/
proc print
   data=ora.PRODUCT_INFO(dbsastype=(WARRANTY_PERIOD='CHAR(6)')); run;

 /* Add a new product */
data new_prods;
   PRODUCT='Dryer';  LIST_PRICE=2000;WARRANTY_PERIOD=12;
run;

proc sql;
insert into ora.PRODUCT_INFO select * from new_prods;
select * from ora.PRODUCT_INFO;
select * from ora.PRODUCT_INFO  where WARRANTY_PERIOD > 24;
quit;
```

Here is an INTERVAL DAY TO SECOND.

```
proc sql;
connect to oracle ( &PTCONN);
execute ( drop table PERF_TESTS) by oracle;

execute (
 create table PERF_TESTS (  TEST_NUMBER number(4) primary key,
    TIME_TAKEN INTERVAL DAY TO SECOND )
)by oracle;

execute (
  insert into PERF_TESTS values (1,  '0 00:01:05.000200000')
)by Oracle;

execute (
  insert into PERF_TESTS values (2,  '0 00:01:03.400000000')
)by Oracle;

quit;

proc contents data=ora.PERF_TESTS; run;

 /* Shows TIME_TAKEN as number of seconds */
proc print data=ora.PERF_TESTS; run;

 /* Shows TIME_TAKEN in a format just like in Oracle*/
proc print
  data=ora.PERF_TESTS(dbsastype=(TIME_TAKEN='CHAR(25)')); run;

 /* Add a new test*/
data new_tests;
  TEST_NUMBER=3; TIME_TAKEN=50;
run;

proc sql;
insert into ora.PERF_TESTS select * from new_tests;
select * from ora.PERF_TESTS;

select * from ora.PERF_TESTS  where TIME_TAKEN < 60;
quit;
```

Binary Data

RAW(*n*)
: contains raw binary data, where *n* must be at least 1 and cannot exceed 255 bytes. (In Oracle Version 8, the limit is 2,000 bytes.) Values entered into columns of this type must be inserted as character strings in hexadecimal notation. You must specify *n* for this data type.

BLOB
: contains raw binary data of variable length up to 2 gigabytes. Values entered into columns of this type must be inserted as character strings in hexadecimal notation.

Oracle Null and Default Values

Oracle has a special value called NULL. An Oracle NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads an Oracle NULL value, it interprets it as a SAS missing value.

By default, Oracle columns accept NULL values. However, you can define columns so that they cannot contain NULL data. NOT NULL tells Oracle not to add a row to the table unless the row has a value for that column. When creating an Oracle table with SAS/ACCESS, you can use the DBNULL= data set option to indicate whether NULL is a valid value for specified columns.

To control how SAS missing character values are handled, use the NULLCHAR= and NULLCHARVAL= data set options.

For more information about how SAS handles NULL values, see "Potential Result Set Differences When Processing Null Data " on page 35.

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to Oracle assigns to SAS variables when using the LIBNAME statement on page 93 to read from an Oracle table. These default formats are based on Oracle column attributes.

Table 26.4 *LIBNAME Statement: Default SAS Formats for Oracle Data Types*

Oracle Data Type	Default SAS Format
CHAR(*n*) *	$*w*.* (where*w* is the minimum of *n* and the value of the DBMAX_TEXT= option)
VARCHAR2(*n*)	$*w*. (where *w* is the minimum of *n* and the value of the DBMAX_TEXT= option)
LONG	$*w*. (where *w* is the minimum of 32767 and the value of the DBMAX_TEXT= option)
CLOB	$*w*.* (where *w* is the minimum of 32767 and the value of the DBMAX_TEXT= option)
RAW(*n*)	$HEXw.* (where w/2 is the minimum of *n* and the value of the DBMAX_TEXT= option)
LONG RAW	$HEXw. (where w/2 is the minimum of 32767 and the value of the DBMAX_TEXT= option)
BLOB RAW	$HEXw. (where w/2 is the minimum of 32767 and the value of the DBMAX_TEXT= option)
BINARY_DOUBLE	none
BINARY_FLOAT	none
NUMBER	none

Oracle Data Type	Default SAS Format
NUMBER(*p*)	*w*.
NUMBER(*p*,*s*)	*w.d*
DATE	DATETIME20.
TIMESTAMP	DATETIME*w.d* (where *d* is derived from the fractional-second precision)
TIMESTAMP WITH LOCAL TIMEZONE	DATETIME*w.d* (where *d* is derived from the fractional-second precision)
TIMESTAMP WITH TIMEZONE	$*w*)
INTERVAL YEAR TO MONTH	*w*. (where *w* is derived from the year precision)
INTERVAL DAY TO SECOND	*w.d* (where *w* is derived from the fractional-second precision)

* The value of the DBMAX_TEXT= option can override these values.

SAS/ACCESS does not support Oracle data types that do not appear in this table.

If Oracle data falls outside valid SAS data ranges, the values are usually counted as missing.

SAS automatically converts Oracle NUMBER types to SAS number formats by using an algorithm that determines the correct scale and precision. When the scale and precision cannot be determined, SAS/ACCESS allows the procedure or application to determine the format. You can also convert numeric data to character data by using the SQL pass-through facility with the Oracle TO_CHAR function. See your Oracle documentation for more details.

The following table shows the default Oracle data types that SAS/ACCESS assigns to SAS variable formats during output operations when you use the LIBNAME statement.

Table 26.5 *LIBNAME Statement: Default Oracle Data Types for SAS Formats*

SAS Variable Format	Oracle Data Type
$*w*.	VARCHAR2(*w*)
$*w*. (where *w* > 4000)	CLOB
w.d	NUMBER(*p*,*s*)
any date, time, or datetime format without fractional parts of a second	DATE
any date, time, or datetime format without fractional parts of a second	TIMESTAMP

To override these data types, use the DBTYPE= data set option during output processing.

ACCESS Procedure Data Conversions

The following table shows the default SAS variable formats that SAS/ACCESS assigns to Oracle data types when you use the ACCESS procedure on page 881.

Table 26.6 *PROC ACCESS: Default SAS Formats for Oracle Data Types*

Oracle Data Type	Default SAS Format
CHAR(*n*)	$*n*. (*n* <= 200) $200. (*n* > 200)
VARCHAR2(*n*)	$*n*. (*n* <= 200) $200. (*n* > 200)
FLOAT	BEST22.
NUMBER	BEST22.
NUMBER(*p*)	*w*.
NUMBER(*p*, *s*)	*w.d*
DATE	DATETIME16.
CLOB	$200.
RAW(*n*)	$*n*. (*n* < 200) $200. (*n* > 200)
BLOB RAW	$200.

Oracle data types that are omitted from this table are not supported by SAS/ACCESS. If Oracle data falls outside valid SAS data ranges, the values are usually counted as missing.

The following table shows the correlation between the Oracle NUMBER data types and the default SAS formats that are created from that data type.

Table 26.7 *Default SAS Formats for Oracle NUMBER Data Types*

Oracle NUMBER Data Type	Rules	Default SAS Format
NUMBER(*p*)	$0 < p <= 32$	$(p + 1).0$
NUMBER(*p*,*s*)	$p > 0, s < 0, \|s\| < p$	$(p + \|s\| + 1).0$
NUMBER(*p*,*s*)	$p > 0, s < 0, \|s\| >= p$	$(p + \|s\| + 1).0$
NUMBER(*p*,*s*)	$p > 0, s > 0, s < p$	$(p + 2).s$
NUMBER(*p*,*s*)	$p > 0, s > 0, s >= p$	$(s + 3).s$

Oracle NUMBER Data Type	Rules	Default SAS Format
NUMBER(*p*)	*p* > 32	BEST22. SAS selects format
NUMBER	*p*, *s* unspecified	BEST22. SAS selects format

The general form of an Oracle number is NUMBER(*p,s*) where *p* is the precision and *s* is the scale of the number. Oracle defines precision as the total number of digits, with a valid range of -84 to 127. However, a negative scale means that the number is rounded to the specified number of places to the left of the decimal. For example, if the number 1,234.56 is specified as data type NUMBER(8,-2), it is rounded to the nearest hundred and stored as 1,200.

DBLOAD Procedure Data Conversions

The following table shows the default Oracle data types that SAS/ACCESS assigns to SAS variable formats when you use the DBLOAD procedure on page 901.

Table 26.8 *PROC DBLOAD: Default Oracle Data Types for SAS Formats*

SAS Variable Format	Oracle Data Type
$*w*.	CHAR(*n*)
w.	NUMBER(*p*)
w.d	NUMBER(*p,s*)
all other numerics *	NUMBER
datetime*w.d*	DATE
date*w*.	DATE
time. **	NUMBER

* Includes all SAS numeric formats, such as BINARY8 and E10.0.

** Includes all SAS time formats, such as TOD*w,d* and HHMM*w,d*.

Chapter 27

SAS/ACCESS Interface to Sybase

Introduction to SAS/ACCESS Interface to Sybase

This section describes SAS/ACCESS Interface to Sybase. For a list of SAS/ACCESS features that are available in this interface, see “SAS/ACCESS Interface to Sybase: Supported Features” on page 88.

For information about Sybase IQ, see “SAS/ACCESS Interface to Sybase IQ” on page 758.

LIBNAME Statement Specifics for Sybase

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to Sybase supports. A Sybase example is available. For details about this feature, see the IBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing Sybase.

LIBNAME *libref* **sybase** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`sybase`
the SAS/ACCESS engine name for the Sybase interface.

connection-options
provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. Here are the connection options for Sybase.

USER=<'>*SYBASE-user-name*<'>
: specifies the Sybase user name (also called the login name) that you use to connect to your database. If the user name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

PASSWORD=<'>*SYBASE-password*<'>
: specifies the password that is associated with your Sybase user name. If you omit the password, a default password of NULL is used. If the password contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. PASSWORD= can also be specified with the SYBPW=, PASS=, and PW= aliases.

DATABASE=<'>*database-name*<'>
: specifies the name of the Sybase database that contains the tables and views that you want to access. If the database name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. If you omit DATABASE=, the default database for your Sybase user name is used.

  Alias: DB=

SERVER=<'>*server-name*<'>
: specifies the server that you want to connect to. This server accesses the database that contains the tables and views that you want to access. If the server name contains lowercase, spaces, or nonalphanumeric characters, you must enclose it in quotation marks. If you omit SERVER=, the default action for your operating system occurs. On UNIX systems, the value of the environment variable DSQUERY is used if it has been set.

IP_CURSOR= YES | NO
: specifies whether implicit PROC SQL pass-through processes multiple result sets simultaneously. IP_CURSOR is set to NO by default. Setting it to YES allows this type of extended processing. However, it decreases performance because cursors, not result sets, are being used. Do not set to YES unless needed.

If you specify the appropriate system options or environment variables for your database, you can often omit the connection options. See your Sybase documentation for details.

LIBNAME-options
: define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to Sybase, with the applicable default values. For more detail about these options, see “LIBNAME Options for Relational Databases” on page 99.

Table 27.1 *SAS/ACCESS LIBNAME Options for Sybase*

Option	Default Value
ACCESS= on page 99	none
AUTHDOMAIN= on page 102	none
AUTOCOMMIT= on page 103	YES
CONNECTION= on page 113	SHAREDREAD
CONNECTION_GROUP= on page 118	none

Option	Default Value
DBCOMMIT= on page 124	1000 when inserting rows; 0 when updating rows
DBCONINIT= on page 125	none
DBCONTERM= on page 126	none
DBCREATE_TABLE_OPTS= on page 128	none
DBGEN_NAME= on page 128	DBMS
DBINDEX= on page 129	NO
DBLIBINIT= on page 130	none
DBLIBTERM= on page 131	none
DBLINK= on page 132	the local database
DBMAX_TEXT= on page 133	1024
DBPROMPT= on page 136	NO
DBSASLABEL= on page 137	COMPAT
DBSERVER_MAX_BYTES= on page 138	COMPAT
DBSLICEPARM= on page 139	THREADED_APPS,2 or 3
DEFER= on page 141	NO
DIRECT_EXE= on page 144	none
DIRECT_SQL= on page 145	YES
ENABLE_BULK= on page 147	YES
INTERFACE= on page 154	none
MAX_CONNECTS= on page 159	25
MULTI_DATASRC_OPT= on page 160	none
PACKETSIZE= on page 164	server setting
QUOTED_IDENTIFIER= on page 174	NO
READBUFF= on page 174	100
READ_ISOLATION_LEVEL= on page 175	1 (see "Locking in the Sybase Interface" on page 747)

Option	Default Value
READ_LOCK_TYPE= on page 176	NOLOCK (see "Locking in the Sybase Interface" on page 747)
REREAD_EXPOSURE= on page 179	NO
SCHEMA= on page 180	none
SPOOL= on page 184	YES
SQL_FUNCTIONS= on page 185	none
SQL_FUNCTIONS_COPY= on page 188	none
SQL_OJ_ANSI= on page 188	NO
UPDATE_ISOLATION_LEVEL= on page 196	1 (see "Locking in the Sybase Interface" on page 747)
UPDATE_LOCK_TYPE= on page 197	PAGE (see "Locking in the Sybase Interface" on page 747)
UTILCONN_TRANSIENT= on page 202	NO

Sybase LIBNAME Statement Example

In the following example, the libref MYDBLIB uses the Sybase engine to connect to a Sybase database. USER= and PASSWORD= are connection options.

```
libname mydblib sybase user=testuser password=testpass;
```

If you specify the appropriate system options or environment variables for your database, you can often omit the connection options. See your Sybase documentation for details.

Data Set Options for Sybase

All SAS/ACCESS data set options in this table are supported for Sybase. Default values are provided where applicable. For general information about this feature, see "Overview" on page 207.

Table 27.2 *SAS/ACCESS Data Set Options for Sybase*

Option	Default Value
AUTOCOMMIT= on page 208	LIBNAME option setting
BULK_BUFFER= on page 273	100

Option	Default Value
BULKLOAD= on page 274	NO
DBCOMMIT= on page 280	LIBNAME setting
DBCONDITION= on page 281	none
DBCREATE_TABLE_OPTS= on page 285	LIBNAME setting
DBFORCE= on page 286	NO
DBGEN_NAME= on page 287	LIBNAME option setting
DBINDEX= on page 288	LIBNAME option setting
DBKEY= on page 290	none
DBLABEL= on page 291	NO
DBLINK= on page 292	LIBNAME option setting
DBMASTER= on page 293	none
DBMAX_TEXT= on page 293	LIBNAME option setting
DBNULL= on page 294	_ALL_YES
DBPROMPT= on page 296	LIBNAME option setting
DBSASLABEL= on page 297	COMPAT
DBSLICE= on page 299	none
DBSLICEPARM= on page 301	THREADED_APPS,2 or 3
DBTYPE= on page 303	see “Data Types for Sybase ” on page 749
ERRLIMIT= on page 308	1
NULLCHAR= on page 329	SAS
NULLCHARVAL= on page 330	a blank character
READBUFF= on page 339	LIBNAME option setting
READ_ISOLATION_LEVEL= on page 340	LIBNAME option setting
READ_LOCK_TYPE= on page 341	LIBNAME option setting
SASDATEFMT= on page 342	DATETIME22.3
SCHEMA= on page 344	LIBNAME option setting

Option	Default Value
SEGMENT_NAME= on page 346	none
UPDATE_ISOLATION_LEVEL= on page 371	LIBNAME option setting
UPDATE_LOCK_TYPE= on page 371	LIBNAME option setting

SQL Pass-Through Facility Specifics for Sybase

Key Information

For general information about this feature, see “SQL Pass-Through Facility” on page 401. A Sybase example is available.

Here are the SQL pass-through facility specifics for the Sybase interface.

- The *dbms-name* is **SYBASE**.
- The CONNECT statement is optional. If you omit the CONNECT statement, an implicit connection is made using the default values for all connection options.
- The interface can connect multiple times to one or more servers.
- Here are the *database-connection-arguments* for the CONNECT statement.

  USER=<'>*SYBASE-user-name*<'>
  : specifies the Sybase user name (also called the login name) that you use to connect to your database. If the user name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

  PASSWORD=<'>*SYBASE-password*<'>
  : specifies the password that is associated with the Sybase user name.

    If you omit the password, a default password of NULL is used. If the password contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

    PASSWORD= can also be specified with the SYBPW=, PASS=, and PW= aliases. If you do not wish to enter your Sybase password in uncoded text, see PROC PWENCODE in the *Base SAS Procedures Guide* for a method to encode it.

  DATABASE=<'>*database-name*<'>
  : specifies the name of the Sybase database that contains the tables and views that you want to access.

    If the database name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. If you omit DATABASE=, the default database for your Sybase user name is used.

    Alias: DB=

  SERVER=<'>*server-name*<'>
  : specifies the server that you want to connect to. This server accesses the database that contains the tables and views that you want to access.

If the server name contains lowercase, spaces, or nonalphanumeric characters, you must enclose it in quotation marks.

If you omit SERVER=, the default action for your operating system occurs. On UNIX systems, the value of the environment variable DSQUERY is used if it has been set.

INTERFACE=*filename*
: specifies the name and location of the Sybase interfaces file. The interfaces file contains the names and network addresses of all available servers on the network.

If you omit this statement, the default action for your operating system occurs. INTERFACE= is not used in some operating environments. Contact your database administrator to determine whether it applies to your operating environment.

SYBBUFSZ=*number-of-rows*
: specifies the number of rows of DBMS data to write to the buffer. If this statement is used, the SAS/ACCESS interface view engine creates a buffer that is large enough to hold the specified number of rows. This buffer is created when the associated database table is read. The interface view engine uses SYBBUFSZ= to improve performance.

If you omit this statement, no data is written to the buffer.

Connection options for Sybase are all case sensitive. They are passed to Sybase exactly as you enter them.

- Here are the LIBNAME options that are available with the CONNECT statement.
  - DBMAX_TEXT=
  - MAX_CONNECTS=
  - READBUFF=
  - PACKETSIZE=

Example

This example retrieves a subset of rows from the Sybase INVOICE table. Because the WHERE clause is specified in the DBMS query (the inner SELECT statement), the DBMS processes the WHERE expression and returns a subset of rows to SAS.

```
proc sql;
connect to sybase(server=SERVER1
                  database=INVENTORY
                  user=testuser password=testpass);
%put &sqlxmsg;

select * from connection to sybase
   (select * from INVOICE where BILLEDBY=457232);
%put &sqlxmsg;
```

The SELECT statement that is enclosed in parentheses is sent directly to the database and therefore must be specified using valid database variable names and syntax.

Autopartitioning Scheme for Sybase

Overview

For general information about this feature, see "Autopartitioning Techniques in SAS/ACCESS" on page 63.

Sybase autopartitioning uses the Sybase MOD function (%) to create multiple SELECT statements with WHERE clauses. In the optimum scenario, the WHERE clauses divide the result set into equal chunks: one chunk per thread. For example, assume that your original SQL statement was **`SELECT * FROM DBTAB`**, and assume that DBTAB has a primary key column PKCOL of type integer and that you want it partitioned into three threads. Here is how the autopartitioning scheme would break up the table into three SQL statements:

```
select * from DBTAB where (abs(PKCOL))%3=0
select * from DBTAB where (abs(PKCOL))%3=1
select * from DBTAB where (abs(PKCOL))%3=2
```

Since PKCOL is a primary key column, you should receive a fairly even distribution among the three partitions, which is the primary goal.

Indexes

An index on a SAS partitioning column increases performance of the threaded Read. If a primary key is not defined for the table, an index should be placed on the partitioning column in order to attain similar benefits. To achieve optimum database performance, it is essential to understand and follow the recommendations in the *Sybase ASE Performance and Tuning Guide* for creating and using indexes. Here is the order of column selection for the partitioning column.

1. Identity column
2. Primary key column (integer or numeric)
3. integer, numeric, or bit; not nullable
4. integer, numeric, or bit; nullable

If the column selected is a bit type, only two partitions are created because the only values are 0 and 1.

Partitioning Criteria

The most efficient partitioning column is an Identity column, which is usually identified as a primary key column. Identity columns usually lead to evenly partitioned result sets because of the sequential values that they store.

The least efficient partitioning column is a numeric, decimal, or float column that is NULLABLE, and does not have an index defined.

Given equivalent selection criteria, columns defined at the beginning of the table definition that meet the selection criteria takes precedence over columns defined toward the end of the table definition.

Data Types

These data types are supported in partitioning column selection:

- INTEGER
- TINYINT
- SMALLINT
- NUMERIC
- DECIMAL
- FLOAT
- BIT

Examples

Here are examples of generated SELECT statements involving various column data types.

COL1 is numeric, decimal, or float. This example uses three threads (the default) and COL1 is NOT NULL.

```
select * from DBTAB where (abs(convert(INTEGER, COL1)))%3=0
select * from DBTAB where (abs(convert(INTEGER, COL1)))%3=1
select * from DBTAB where (abs(convert(INTEGER, COL1)))%3=2
```

COL1 is bit, integer, smallint, or tinyint. This example uses two threads (the default) and COL1 is NOT NULL.

```
select * from DBTAB where (abs(COL1))%3=0
select * from DBTAB where (abs(COL1))%3=1
```

COL1 is and integer and is nullable.

```
select * from DBTAB where (abs(COL1))%3=0 OR COL1 IS NULL
select * from DBTAB where (abs(COL1))%3=1
```

Temporary Table Support for Sybase

Overview

For general information about this feature, see “Temporary Table Support for SAS/ACCESS ” on page 42.

Establishing a Temporary Table

When you specify CONNECTION=GLOBAL, you can reference a temporary table throughout a SAS session, in both DATA steps and procedures. The name of the table MUST start with the character '#'. To reference it, use the SAS convention of an *n* literal, as in mylib.'#foo'n.

Terminating a Temporary Table

You can drop a temporary table at any time, or allow it to be implicitly dropped when the connection is terminated. Temporary tables do not persist beyond the scope of a singe connection.

Example

This example shows how to use temporary tables.

```
/* clear any connection */
libname x clear;

libname x sybase user=test pass=test connection=global;

/* create the temp table. You can even use bulk copy */
/* Notice how the name is specified: '#mytemp'n */
   data x.'#mytemp'n (bulk=yes);
   x=55;
   output;
   x=44;
   output;
run;

/* print it */
proc print data=x.'#mytemp'n;
run ;

/* The same temp table persists in PROC SQL, */
/* with the global connection specified */
proc sql;
   connect to sybase (user=austin pass=austin connection=global);
   select * from connection to sybase (select * from #mytemp);
quit;

/* use the temp table again in a procedure */
proc means data=x.'#mytemp'n;
run;

/* drop the connection, the temp table is automatically dropped */
libname x clear;

/* to convince yourself it's gone, try to access it */
libname x sybase user=austin password=austin connection=global;

/* it's not there */
proc print data=x.'#mytemp'n;
run;
```

ACCESS Procedure Specifics for Sybase

Overview

For general information about this feature, see Chapter 1, “ACCESS Procedure,” on page 881. A Sybase example is available.

SAS/ACCESS for Sybase supports all ACCESS procedure statements on page 882. Here are the ACCESS Procedure specifics for Sybase.

- The DBMS= value for PROC ACCESS is **SYBASE**.
- The *database-description-statements* that PROC ACCESS uses are identical to the database-connection-arguments on page 737 in the CONNECT statement for the SQL pass-through facility.
- Here is the TABLE= statement for PROC ACCESS.

  TABLE= <'>*table-name*<'>;
  : specifies the name of the Sybase table or Sybase view on which the access descriptor is based.

Example

The following example creates access descriptors and view descriptors for the EMPLOYEES and INVOICE tables. These tables have different owners and are stored in PERSONNEL and INVENTORY databases that reside on different machines. The USER= and PASSWORD= statements identify the owners of the Sybase tables and their passwords.

```
libname vlib 'sas-library';

proc access dbms=sybase;
   create work.employee.access;
      server='server1';
      database='personnel';
      user='testuser1';
      password='testpass1';
      table=EMPLOYEES;
   create vlib.emp_acc.view;
      select all;
      format empid 6.;
      subset where DEPT like 'ACC%';
run;

proc access dbms=sybase;
   create work.invoice.access;
      server='server2';
      database='inventory';
      user='testuser2';
      password='testpass2';
      table=INVOICE;
      rename invoicenum=invnum;
```

```
      format invoicenum 6. billedon date9.
        paidon date9.;
   create vlib.sainv.view;
      select all;
      subset where COUNTRY in ('Argentina','Brazil');
run;

options linesize=120;
title 'South American Invoices and
        Who Submitted Them';

proc sql;
   select invnum, country, billedon, paidon,
          billedby, lastname, firstnam
      from vlib.emp_acc, vlib.sainv
      where emp_acc.empid=sainv.billedby;
```

Sybase is a case-sensitive database. The PROC ACCESS database identification statements and the Sybase column names in all statements except SUBSET are converted to uppercase unless the names are enclosed in quotation marks. The SUBSET statements are passed to Sybase exactly as you enter them, so you must use the correct case for the Sybase column names.

DBLOAD Procedure Specifics for Sybase

Overview

For general information about this feature, see Chapter 2, "DBLOAD Procedure," on page 901. A Sybase example is available.

The Sybase interface supports all DBLOAD procedure statements on page 902. Here are the Sybase interface specifics for the DBLOAD procedure.

- The DBMS= value for PROC DBLOAD is **SYBASE**.
- The TABLE= statement for PROC DBLOAD is:

  TABLE= <'>*table-name*<'>;
- PROC DBLOAD uses these *database-description-statements*.

  USER=<'>*SYBASE-user-name*<'>
  : specifies the Sybase user name (also called the login name) that you use to connect to your database. If the user name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

  PASSWORD=<'>*SYBASE-password*<'>
  : specifies the password that is associated with the Sybase user name.

    If you omit the password, a default password of NULL is used. If the password contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

    PASSWORD= can also be specified with the SYBPW=, PASS=, and PW= aliases.

DATABASE=<'>*database-name*<'>
: specifies the name of the Sybase database that contains the tables and views that you want to access.

  If the database name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. If you omit DATABASE=, the default database for your Sybase user name is used.

  Alias: DB=

SERVER=<'>*server-name*<'>
: specifies the server that you want to connect to. This server accesses the database that contains the tables and views that you want to access.

  If the server name contains lowercase, spaces, or nonalphanumeric characters, you must enclose it in quotation marks.

  If you omit SERVER=, the default action for your operating system occurs. On UNIX systems, the value of the environment variable DSQUERY is used if it has been set.

INTERFACE=*filename*
: specifies the name and location of the Sybase interfaces file. The interfaces file contains the names and network addresses of all available servers on the network.

  If you omit this statement, the default action for your operating system occurs. INTERFACE= is not used in some operating environments. Contact your database administrator to determine whether it applies to your operating environment.

BULKCOPY= Y|N;
: uses the Sybase bulk copy utility to insert rows into a Sybase table. The default value is N.

  If you specify BULKCOPY=Y, BULKCOPY= calls the Sybase bulk copy utility in order to load data into a Sybase table. This utility groups rows so that they are inserted as a unit into the new table. Using the bulk copy utility can improve performance.

  You use the COMMIT= statement to specify the number of rows in each group (this argument must be a positive integer). After each group of rows is inserted, the rows are permanently saved in the table. As each group is being inserted, if one row in the group is rejected, all rows in that group are rejected.

  If you specify BULKCOPY=N, rows are inserted into the new table using Transact-SQL INSERT statements. See your Sybase documentation for more information about the bulk copy utility.

Example

The following example creates a new Sybase table, EXCHANGE, from the DLIB.RATEOFEX data file. (The DLIB.RATEOFEX data set is included in the sample data that is shipped with your software.) An access descriptor ADLIB.EXCHANGE is also created, and it is based on the new table. The DBLOAD procedure sends a Transact-SQL GRANT statement to Sybase. You must be granted Sybase privileges to create new Sybase tables or to grant privileges to other users.

```
libname adlib 'SAS-library';
libname dlib 'SAS-library';
```

```
proc dbload dbms=sybase data=dlib.rateofex;
   server='server1';
   database='testdb';
   user='testuser';
   password='testpass';
   table=EXCHANGE;
   accdesc=adlib.exchange;
   rename fgnindol=fgnindolar 4=dolrsinfgn;
   nulls updated=n fgnindol=n 4=n country=n;
   load;
run;
```

Passing SAS Functions to Sybase

SAS/ACCESS Interface to Sybase passes the following SAS functions to Sybase for processing if the DBMS driver or client that you are using supports the function. Where the Sybase function name differs from the SAS function name, the Sybase name appears in parentheses. See "Passing Functions to the DBMS Using PROC SQL" on page 46 for information.

ABS	LOG
ARCOS (ACOS)	LOWCASE (LOWER)
ARSIN (ASIN)	MAX
ATAN	MIN
AVG	MINUTE
CEIL (CEILING)	MONTH
COS	SECOND
COUNT	SIGN
DATETIME (GETDATE())	SIN
DAY	SQRT
DTEXTDAY	STRIP (RTRIM(LTRIM))
DTEXTMONTH	SUM
DTEXTWEEKDAY	TAN
DTEXTYEAR	TRIMN (RTRIM)
EXP	UPCASE (UPPER)
FLOOR	WEEKDAY
HOUR	YEAR

SQL_FUNCTIONS= on page 185ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to Sybase. Due to incompatibility in date and time functions between Sybase and SAS, Sybase might not process them correctly. Check your results to determine whether these functions are working as expected.

DATEPART	TIMEPART
ROUND	

Passing Joins to Sybase

For a multiple libref join to pass to Sybase, all of these components of the LIBNAME statements must match exactly:

- user ID (USER= on page 733)
- password (PASSWORD= on page 733)
- database (DATABASE= on page 733)
- server (SERVER= on page 733)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Bulk Loading for Sybase

Bulk loading is the fastest way to insert large numbers of rows into a Sybase table. To use the bulk-load facility, specify BULKLOAD=YES. The bulk-load facility uses the Sybase bulk copy facility to move data from SAS to Sybase. See the ENABLE_BULK= LIBNAME option on page 147.

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 DBCOMMIT= data set option determines the commit intervals.

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 are detected, even if the ERRLIMIT= value is greater than one. In addition, all rows before the error are committed, even if DBCOMMIT= is larger than the number of the erroneous row.

Here are the Sybase bulk-load data set options. For detailed information about these options, see Chapter 11, "Data Set Options for Relational Databases," on page 203.

- BULK_BUFFER=
- BULKLOAD=

Reading Multiple Sybase Tables

SAS opens multiple Sybase tables for simultaneous reading in these situations:

- When you are using PROC COMPARE. Here is an example:

  ```
  proc compare base=syb.data1 compare=syb.data2;
  ```

- When you are running an SCL program that reads from more than one Sybase table simultaneously.
- When you are joining Sybase tables in SAS—namely, when implicit pass-through is not used (DIRECT_SQL=NO). Here are four examples:

```
proc sql ;
   select * from syb.table1, syb.table2 where table1.x=table2.x;

proc sql;
   select * from syb.table1 where table1.x = (select x from syb.table2
   where y = 33);

proc sql;
   select empname from syb.employee where empyears > all (select empyears
   from syb.employee where emptitle = 'salesrep');
proc sql ;
   create view myview as
      select * from employee where empyears > all (select empyears from
      syb.employee where emptitle = 'salesrep');
proc print data=myview ;
```

To read two or more Sybase tables simultaneously, you must specify either the LIBNAME option CONNECTION=UNIQUE or the LIBNAME option READLOCK_TYPE=PAGE. Because READLOCK_TYPE=PAGE can degrade performance, it is generally recommended that you use CONNECTION=UNIQUE (unless there is a concern about the number of connections that are opened on the database).

Locking in the Sybase Interface

Overview

The following LIBNAME and data set options let you control how the Sybase interface handles locking. For general information about an option, see "LIBNAME Options for Relational Databases" on page 99.

READ_LOCK_TYPE= PAGE | NOLOCK
: The default value for Sybase is NOLOCK.

UPDATE_LOCK_TYPE= PAGE | NOLOCK

PAGE
: SAS/ACCESS uses a cursor that you can update. PAGE is the default value for Sybase. When you use this setting, you cannot use the SCHEMA= option, and it is also recommended that the table have a defined primary key.

NOLOCK
: SAS/ACCESS uses Sybase browse mode updating, in which the table that is being updated must have a primary key and timestamp.

READ_ISOLATION_LEVEL= 1 | 2 | 3
: For reads, Sybase supports isolation levels 1, 2, and 3, as defined in the following table. See your Sybase documentation for more information.

Table 27.3 *Isolation Levels for Sybase*

Isolation Level	Definition
1	Prevents dirty Reads. This is the default transaction isolation level.

Isolation Level	Definition
2	Uses serialized Reads.
3	Also uses serialized Reads.

UPDATE_ISOLATION_LEVEL= 1 | 3
Sybase uses a shared or update lock on base table pages that contain rows representing a current cursor position. This option applies to updates only when UPDATE_LOCK_TYPE=PAGE because cursor updating is in effect. It does not apply when UPDATE_LOCK_TYPE=NOLOCK.

For updates, Sybase supports isolation levels 1 and 3, as defined in the preceding table. See your Sybase documentation for more information.

Understanding Sybase Update Rules

To avoid data integrity problems when updating and deleting data in Sybase tables, take these precautionary measures:

- Always define a primary key.
- If the updates are not taking place through cursor processing, define a timestamp column.

It is not always obvious whether updates are using cursor processing. Cursor processing is *never* used for LIBNAME statement updates if UPDATE_LOCK_TYPE=NOLOCK. Cursor processing is *always* used in these situations:

- Updates using the LIBNAME statement with UPDATE_LOCK_TYPE=PAGE. This is the default setting for this option.
- Updates using PROC SQL views.
- Updates using PROC ACCESS view descriptors.

Naming Conventions for Sybase

For general information about this feature, see "SAS Names and Support for DBMS Names" on page 13.

Sybase database objects include tables, views, columns, indexes, and database procedures. They follow these naming conventions:

- A name must be from 1 to 30 characters long—or 28 characters, if you enclose the name in quotation marks.
- A name must begin with an alphabetic character (A to Z) or an underscore (_) unless you enclose the name in quotation marks.
- After the first character, a name can contain letters (A to Z) in uppercase or lowercase, numbers from 0 to 9, underscore (_), dollar sign ($), pound sign (#), at sign (@), yen sign (¥), and monetary pound sign (£).
- Embedded spaces are not allowed unless you enclose the name in quotation marks.

- Embedded quotation marks are not allowed.
- Case sensitivity is set when a server is installed. By default, the names of database objects are case sensitive. For example, the names **`CUSTOMER`** and **`customer`** are different on a case-sensitive server.
- A name cannot be a reserved word in Sybase unless the name is enclosed in quotation marks. See your Sybase documentation for more information about reserved words.
- Database names must be unique. For each owner within a database, names of database objects must be unique. Column names and index names must be unique within a table.

By default, Sybase does not enclose column names and table names in quotations marks. To enclose these in quotation marks, you must use the QUOTED_IDENTIFIER= LIBNAME option when you assign a libref.

When you use the DATASETS procedure to list your Sybase tables, the table names appear exactly as they exist in the Sybase data dictionary. If you specified the SCHEMA= LIBNAME option, SAS/ACCESS lists the tables for the specified schema user name.

To reference a table or other named object that you own, or for the specified schema, use the table name (for example, CUSTOMERS). If you use the DBLINK= LIBNAME option, all references to the libref refer to the specified database.

Data Types for Sybase

Overview

Every column in a table has a name and a data type. The data type indicates to the DBMS how much physical storage to reserve for the column and the format in which the data is stored. This section includes information about Sybase data types, null values, and data conversions, and also explains how to insert text into Sybase from SAS.

SAS/ACCESS does not support these Sybase data types: BINARY, VARBINARY, IMAGE, NCHAR(*n*), and NVARCHAR(*n*). SAS/ACCESS provides an error message when it tries to read a table that has at least one column that uses an unsupported data type.

Character Data

You must enclose all character data in single or double quotation marks.

CHAR(*n*)
: CHAR(*n*) is a character string that can contain letters, symbols, and numbers. Use *n* to specify the maximum length of the string, which is the currently set value for the **Adaptive Server** page size (2K, 4K, 8K, or 16K). Storage size is also *n*, regardless of the actual entry length.

VARCHAR(*n*)
: VARCHAR(*n*) is a varying-length character string that can contain letters, symbols, and numbers. Use *n* to specify the maximum length of the string, which is the currently set value for the **Adaptive Server** page size (2K, 4K, 8K, or 16K). Storage size is the actual entry length.

TEXT
: TEXT stores character data of variable length up to two gigabytes. Although SAS supports the TEXT data type that Sybase provides, it allows a maximum of only 32,767 bytes of character data.

Numeric Data

NUMERIC(*p,s*), DECIMAL(*p,s*)
: Exact numeric values have specified degrees of precision (*p*) and scale (*s*). NUMERIC data can have a precision of 1 to 38 and scale of 0 to 38, where the value of *s* must be less or equal to than the value of *p*. The DECIMAL data type is identical to the NUMERIC data type. The default precision and scale are (18,0) for the DECIMAL data type.

REAL, FLOAT
: Floating-point values consist of an integer part, a decimal point, and a fraction part, or scientific notation. The exact format for REAL and FLOAT data depends on the number of significant digits and the precision that your machine supports. You can use all arithmetic operations and aggregate functions with REAL and FLOAT except modulus. The REAL (4-byte) range is approximately 3.4E−38 to 3.4E+38, with 7-digit precision. The FLOAT (8-byte) range is approximately 1.7E−308 to 1.7E+308, with 15-digit precision.

TINYINT, SMALLINT, INT
: Integers contain no fractional part. The three-integer data types are TINYINT (1 byte), which has a range of 0 to 255; SMALLINT (2 bytes), which has a range of –32,768 to +32,767; and INT (4 bytes), which has a range of –2,147,483,648 to +2,147,483,647.

BIT
: BIT data has a storage size of one bit and holds either a 0 or a 1. Other integer values are accepted but are interpreted as 1. BIT data cannot be NULL and cannot have indexes defined on it.

Date, Time, and Money Data

Sybase date and money data types are abstract data types. See your documentation on Transact-SQL for more information about abstract data types.

DATE
: DATE data is 4 bytes long and represents dates from January 1, 0001, to December 31, 9999.

TIME
: TIME data is 4 byes long and represents times from 12:00:00 AM to 11:59:59:999 PM.

SMALLDATETIME
: SMALLDATETIME data is 4 bytes long. It consists of one small integer that represents the number of days after January 1, 1900, and one small integer that represents the number of minutes past midnight. The date range is from January 1, 1900, to December 31, 2079.

DATETIME
: DATETIME data has two 4-byte integers. The first integer represents the number of days after January 1, 1900, and the second integer represents the number of

milliseconds past midnight. Values can range from January 1, 1753, to December 31, 9999.

You must enter DATETIME values as quoted character strings in various alphabetic or numeric formats. You must enter time data in the prescribed order (hours, minutes, seconds, milliseconds, AM, am, PM, pm), and you must include either a colon or an AM/PM designator. Case is ignored, and spaces can be inserted anywhere within the value.

When you input DATETIME values, the national language setting determines how the date values are interpreted. You can change the default date order with the SET DATEFORMAT statement. See your Transact-SQL documentation for more information.

You can use Sybase built-in date functions to perform some arithmetic calculations on DATETIME values.

TIMESTAMP
: SAS uses TIMESTAMP data in UPDATE mode. If you select a column that contains TIMESTAMP data for input into SAS, values are displayed in hexadecimal format.

SMALLMONEY
: SMALLMONEY data is 4 bytes long and can range from –214,748.3648 to 214,748.3647. When it is displayed, it is rounded up to two places.

MONEY
: MONEY data is 8 bytes long and can range from –922,337,203,685,477.5808 to 922,337,203,685,477.5807. You must include a dollar sign ($) before the MONEY value. For negative values, you must include the minus sign after the dollar sign. Commas are not allowed.

MONEY values are accurate to a ten-thousandth of a monetary unit. However, when they are displayed, the dollar sign is omitted and MONEY values are rounded up to two places. A comma is inserted after every three digits.

You can store values for currencies other than U.S. dollars, but no form of conversion is provided.

User-Defined Data

You can supplement the Sybase system data types by defining your own data types with the Sybase system procedure **`sp_addtype`**. When you define your own data type for a column, you can specify a default value (other than NULL) for it and define a range of allowable values for it.

Sybase Null Values

Sybase has a special value that is called NULL. A This value indicates an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a Sybase NULL value, it interprets it as a SAS missing value.

By default, Sybase columns are defined as NOT NULL. NOT NULL tells Sybase not to add a row to the table unless the row has a value for the specified column.

If you want a column to accept NULL values, you must explicitly define it as NULL. Here is an example of a CREATE TABLE statement that defines all table columns as NULL except CUSTOMER. In this case, Sybase accepts a row only if it contains a value for CUSTOMER.

```
create table CUSTOMERS
   (CUSTOMER        char(8)    not null,
    STATE           char(2)        null,
    ZIPCODE         char(5)        null,
    COUNTRY         char(20)       null,
    TELEPHONE       char(12)       null,
    NAME            char(60)       null,
    CONTACT         char(30)       null,
    STREETADDRESS   char(40)       null,
    CITY            char(25)       null,
    FIRSTORDERDATE  datetime       null);
```

When you create a Sybase table with SAS/ACCESS, you can use the DBNULL= on page 294 data set option to indicate whether NULL is a valid value for specified columns.

For more information about how SAS handles NULL values, see "Potential Result Set Differences When Processing Null Data " on page 35.

To control how SAS missing character values are handled, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to Sybase assigns to SAS variables when using the LIBNAME statement on page 93 to read from a Sybase table. These default formats are based on Sybase column attributes.

Table 27.4 *LIBNAME Statement: Default SAS Formats for Sybase Server Data Types*

Sybase Column Type	SAS Data Type	Default SAS Format
CHAR(*n*)	character	$*n***
VARCHAR(*n*)	character	$*n***
TEXT	character	$*n*.** (where *n* is the value of the DBMAX_TEXT= on page 293 option)
BIT	numeric	1.0
TINYINT	numeric	4.0
SMALLINT	numeric	6.0
INT	numeric	11.0
NUMERIC	numeric	*w, w.d* (if possible)
DECIMAL	numeric	*w, w.d* (if possible)
FLOAT	numeric	
REAL	numeric	
SMALLMONEY	numeric	DOLLAR12.2

Sybase Column Type	SAS Data Type	Default SAS Format
MONEY	numeric	DOLLAR24.2
DATE*	numeric	DATE9.
TIME*	numeric	TIME12.
SMALLDATETIME	numeric	DATETIME22.3
DATETIME	numeric	DATETIME22.3
TIMESTAMP	hexadecimal	$HEX*w*

* If a conflict might occur between the Sybase and SAS value for this data type, use SASDATEFMT= to specify the SAS format.

** *n* specifies the current value for the **Adaptive Server** page size.

The following table shows the default Sybase data types that SAS/ACCESS assigns to SAS variable formats during output operations when you use the LIBNAME statement.

***Table 27.5** LIBNAME STATEMENT: Default Sybase Data Types for SAS Variable Formats*

SAS Variable Format	Sybase Data Type
$*w*., $CHAR*w*, $VARYING*w*., $HEX*w*.	VARCHAR(*w*)
DOLLAR*w.d*	SMALLMONEY (where *w* < 6) MONEY (where *w* >= 6)
datetime format	DATETIME
date format	DATE
time format	TIME
any numeric with a SAS format name of *w.d* (where *d* > 0 and *w* > 10) or *w*.	NUMERIC(p,s)
any numeric with a SAS format name of *w.d* (where *d* = 0 and *w* < 10)	TINYINT (where *w* < 3) SMALLINT (where *w* < 5) INT (where *w* < 10)
any other numeric	FLOAT

You can override these default data types by using the DBTYPE= on page 303 data set option.

ACCESS Procedure Data Conversions

The following table shows the default SAS variable formats that SAS/ACCESS assigns to Sybase data types when you use the ACCESS procedure on page 881.

***Table 27.6** PROC ACCESS: Default SAS Formats for Sybase Server Data Types*

Sybase Column Type	SAS Data Type	Default SAS Format
CHAR(*n*)	character	$*n*. (*n* <= 200) $200. (*n* > 200)
VARCHAR(*n*)	character	$*n*. (*n* <= 200) $200. (*n* > 200)
BIT	numeric	1.0
TINYINT	numeric	4.0
SMALLINT	numeric	6.0
INT	numeric	11.0
FLOAT	numeric	BEST22.
REAL	numeric	BEST11.
SMALLMONEY	numeric	DOLLAR12.2
MONEY	numeric	DOLLAR24.2
SMALLDATETIME	numeric	DATETIME21.2
DATETIME	numeric	DATETIME21.2

The ACCESS procedure also supports Sybase user-defined data types. The ACCESS procedure uses the Sybase data type on which a user-defined data type is based in order to assign a default SAS format for columns.

The DECIMAL, NUMERIC, and TEXT data types are not supported in PROC ACCESS. The TIMESTAMP data type is not displayed in PROC ACCESS.

DBLOAD Procedure Data Conversions

The following table shows the default Sybase data types that SAS/ACCESS assigns to SAS variable formats when you use the DBLOAD procedure on page 901.

Table 27.7 *PROC DBLOAD: Default Sybase Data Types for SAS Variable Formats*

SAS Variable Format	Sybase Data Type
$*w*., $CHAR*w*., $VARYING*w*., $HEX*w*.	CHAR(*w*)
w.	TINYINT
w.	SMALLINT
w.	INT
w.	FLOAT
w.d	FLOAT
IB*w.d*, PIB*w.d*	INT
FRACT, E format, and other numeric formats	FLOAT
DOLLAR*w.d*, *w*<=12	SMALLMONEY
DOLLAR*w.d*, *w*>12	MONEY
any datetime, date, or time format	DATETIME

The DBLOAD procedure also supports Sybase user-defined data types. Use the TYPE= on page 913 statement to specify a user-defined data type.

Data Returned as SAS Binary Data with Default Format $HEX

- BINARY
- VARBINARY
- IMAGE

Data Returned as SAS Character Data

- NCHAR
- NVARCHAR

Inserting TEXT into Sybase from SAS

You can insert only TEXT data into a Sybase table by using the BULKLOAD= on page 274 data set option, as in this example:

```
data yourlib.newtable(bulkload=yes);
   set work.sasbigtext;
run;
```

If you do not use the BULKLOAD= option, you receive this error message:

```
ERROR: Object not found in database. Error Code: -2782
An untyped variable in the PREPARE statement 'S401bcf78'
is being resolved to a TEXT or IMAGE type.
This is illegal in a dynamic PREPARE statement.
```

Case Sensitivity in Sybase

SAS names can be entered in either uppercase or lowercase. When you reference Sybase objects through the SAS/ACCESS interface, objects are case sensitive and require no quotation marks.

However, Sybase is generally set for case sensitivity. Give special consideration to the names of such objects as tables and columns when the SAS ACCESS or DBLOAD procedures are to use them. The ACCESS procedure converts Sybase object names to uppercase unless they are enclosed in quotation marks. Any Sybase objects that were given lowercase names, or whose names contain national or special characters, must be enclosed in quotation marks. The only exceptions are the SUBSET statement in the ACCESS procedure and the SQL statement in the DBLOAD procedure. Arguments or values from these statements are passed to Sybase exactly as you enter them, with the case preserved.

In the SQL pass-through facility, all Sybase object names are case sensitive. The names are passed to Sybase exactly as they are entered.

For more information about case sensitivity and Sybase names, see “Naming Conventions for Sybase ” on page 748.

National Language Support for Sybase

To support output and update processing from SAS into Sybase in languages other than English, special setup steps are required so that date, time, and datetime values can be processed correctly. In SAS, you must ensure that the DFLANG= system option is set to the correct language. A system administrator can set this globally administrator or a user can set it within a single SAS session. In Sybase, the default client language, set in the *locales.dat* file, must match the language that is used in SAS.

Chapter 28

SAS/ACCESS Interface to Sybase IQ

Introduction to SAS/ACCESS Interface to Sybase IQ

This section describes SAS/ACCESS Interface to Sybase IQ. For a list of SAS/ACCESS features that are available for this interface, see “SAS/ACCESS Interface to Sybase IQ: Supported Features” on page 89.

For information about Sybase, see “SAS/ACCESS Interface to Sybase” on page 732.

LIBNAME Statement Specifics for Sybase IQ

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to Sybase IQ supports and includes examples. For details about this feature, see the LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing Sybase IQ.

LIBNAME *libref* **sybaseiq** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
: specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`sybaseiq`
: specifies the SAS/ACCESS engine name for the SybaseIQ interface.

connection-options
: provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. When you use the LIBNAME statement, you can connect to the Sybase IQ database in two ways. Specify *only one* of these methods for each connection because they are mutually exclusive.

- HOST=, SERVER=, DATABASE=, PORT=, USER=, PASSWORD=
- DSN=, USER=, PASSWORD=

Here is how these options are defined.

HOST=<'>*server-name*<'>
: specifies the host name or IP address where the Sybase IQ database is running. If the server name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

SERVER=<'>*server-name*<'>
: specifies the Sybase IQ server name, also known as the engine name. If the server name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

DATABASE=<'>*database-name*<'>
: specifies the Sybase IQ database that contains the tables and views that you want to access. If the database name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

  Alias: DB=

PORT=*port*
: specifies the port number that is used to connect to the specified Sybase IQ database. If you do not specify a port, the default is 2638.

USER=<'>*Sybase IQ-user-name*<'>
: specifies the Sybase IQ user name (also called the user ID) that you use to connect to your database. If the user name contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

PASSWORD=<'>*Sybase IQ-password*<'>
: specifies the password that is associated with your Sybase IQ user name. If the password contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. You can also specify PASSWORD= with the PWD=, PASS=, and PW= aliases.

DSN=<'>*Sybase IQ-data-source*<'>
: specifies the configured Sybase IQ ODBC data source to which you want to connect. Use this option if you have existing Sybase IQ ODBC data sources that are configured on your client. This method requires additional setup—either through the ODBC Administrator control panel on Windows platforms or through the odbc.ini file or a similarly named configuration file on UNIX platforms. So it is recommended that you use this connection method only if you have existing, functioning data sources that have been defined.

LIBNAME-options
: define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to Sybase IQ, with the applicable default values. For more detail about these options, see "LIBNAME Options for Relational Databases" on page 99.

Table 28.1 *SAS/ACCESS LIBNAME Options for Sybase IQ*

Option	Default Value
ACCESS= on page 99	none
AUTHDOMAIN= on page 102	none
AUTOCOMMIT= on page 103	operation-specific
CONNECTION= on page 113	SHAREDREAD
CONNECTION_GROUP= on page 118	none
DBCOMMIT= on page 124	1000 when inserting rows; 0 when updating rows
DBCONINIT= on page 125	none

Option	Default Value
DBCONTERM= on page 126	none
DBCREATE_TABLE_OPTS= on page 128	none
DBGEN_NAME= on page 128	DBMS
DBINDEX= on page 129	YES
DBLIBINIT= on page 130	none
DBLIBTERM on page 131	none
DBMAX_TEXT= on page 133	1024
DBMSTEMP= on page 133	NO
DBNULLKEYS= on page 135	YES
DBMPROMPT= on page 136	NO
DBSASLABEL= on page 137	COMPAT
DBSLICEPARM= on page 139	THREADED_APPS,2 or 3
DEFER= on page 141	NO
DELETE_MULT_ROWS= on page 143	NO
DIRECT_EXE= on page 144	none
DIRECT_SQL= on page 145	YES
IGNORE_READ_ONLY_COLUMNS= on page 150	NO
INSERTBUFF= on page 153	automatically calculated based on row length
LOGIN_TIMEOUT= on page 159	0
MULTI_DATASRC_OPT= on page 160	none
PRESERVE_COL_NAMES= on page 167	see "Naming Conventions for Sybase IQ" on page 770
PRESERVER_TAB_NAMES= on page 168	see "Naming Conventions for Sybase IQ" on page 770
QUERY_TIMEOUT= on page 172	0
QUOTE_CHAR= on page 173	none

Option	Default Value
READ_ISOLATION_LEVEL= on page 175	RC (see "Locking in the Sybase IQ Interface" on page 769)
READ_LOCK_TYPE= on page 176	ROW
READBUFF= on page 174	automatically calculated based on row length
REREAD_EXPOSURE= on page 179	NO
SCHEMA= on page 180	none
SPOOL= on page 184	YES
SQL_FUNCTIONS= on page 185	none
SQL_FUNCTIONS_COPY= on page 188	none
STRINGDATES= on page 190	NO
TRACE= on page 195	NO
TRACEFILE= on page 196	none
UDPATE_ISOLATION_LEVEL= on page 196	RC (see "Locking in the Sybase IQ Interface" on page 769)
UPDATE_LOCK_TYPE= on page 197	ROW
UDPATE_MULT_ROW= on page 199	NO
UTILCONN_TRANSIENT= on page 202	NO

Sybase IQ LIBNAME Statement Example

In this example, HOST=, SERVER=, DATABASE=, USER=, and PASSWORD= are connection options.

```
libname mydblib sybaseiq host=iqsvr1 server=iqsrv1_users
   db=users user=iqusr1 password=iqpwd1;

proc print data=mydblib.customers;
   where state='CA';
run;
```

In the next example, DSN=, USER=, and PASSWORD= are connection options. The SybaseIQ SQL data source is configured in the ODBC Administrator Control Panel on Windows platforms or in the odbc.ini file or a similarly named configuration file on UNIX platforms.

```
libname mydblib sybaseiq DSN=SybaseIQSQL user=iqusr1 password=iqpwd1;
```

```
proc print data=mydblib.customers;
   where state='CA';
run;
```

Data Set Options for Sybase IQ

All SAS/ACCESS data set options in this table are supported for Sybase IQ. Default values are provided where applicable. For details about this feature, see "Overview" on page 207.

Table 28.2 *SAS/ACCESS Data Set Options for Sybase IQ*

Option	Default Value
BL_CLIENT_DATAFILE= on page 211	none
BL_DATAFILE= on page 216	When BL_USE_PIPE=NO, creates a file in the current directory or with the default file specifications.
BL_DELETE_DATAFILE= on page 233	YES (only when BL_USE_PIPE=NO)
BL_DELIMITER= on page 235	| (the pipe symbol)
BL_OPTIONS= on page 255	none
BL_SERVER_DATAFILE= on page 265	creates a data file in the current directory or with the default file specifications (same as for BL_DATAFILE=)
BL_USE_PIPE= on page 271	YES
BULKLOAD= on page 274	NO
DBCOMMIT= on page 280	LIBNAME option setting
DBCONDITION= on page 281	none
DBCREATE_TABLE_OPTS= on page 285	LIBNAME option setting
DBFORCE= on page 286	NO
DBGEN_NAME= on page 287	DBMS
DBINDEX= on page 288	LIBNAME option setting
DBKEY= on page 290	none
DBLABEL= on page 291	NO
DBMASTER= on page 293	none

Option	Default Value
DBMAX_TEXT= on page 293	1024
DBNULL= on page 294	YES
DBNULLKEYS= on page 295	LIBNAME option setting
DBPROMPT= on page 296	LIBNAME option setting
DBSASTYPE= on page 298	see “Data Types for Sybase IQ ” on page 771
DBSLICE= on page 299	none
DBSLICEPARM= on page 301	THREADED_APPS,2 or 3
DBTYPE= on page 303	see “Data Types for Sybase IQ ” on page 771
ERRLIMIT= on page 308	1
IGNORE_READ_ONLY_COLUMNS= on page 311	NO
INSERTBUFF= on page 314	LIBNAME option setting
NULLCHAR= on page 329	SAS
NULLCHARVAL= on page 330	a blank character
PRESERVE_COL_NAMES= on page 336	LIBNAME option setting
QUERY_TIMEOUT= on page 339	LIBNAME option setting
READ_ISOLATION_LEVEL= on page 340	LIBNAME option setting
READ_LOCK_TYPE= on page 341	LIBNAME option setting
READBUFF= on page 339	LIBNAME option setting
SASDATEFMT= on page 342	none
SCHEMA= on page 344	LIBNAME option setting

SQL Pass-Through Facility Specifics for Sybase IQ

Key Information

For general information about this feature, see "Overview of SQL Procedure Interactions with SAS/ACCESS" on page 401. A Sybase IQ example is available.

Here are the SQL pass-through facility specifics for the Sybase IQ interface.

- The *dbms-name* is **SYBASEIQ**.
- The CONNECT statement is required.
- PROC SQL supports multiple connections to Sybase IQ. If you use multiple simultaneous connections, you must use the *alias* argument to identify the different connections. If you do not specify an alias, the default **sybaseiq** alias is used.
- The CONNECT statement *database-connection-arguments* are identical to its LIBNAME connection-options.

CONNECT Statement Example

This example uses the DBCON alias to connection to the **iqsrv1** Sybase IQ database and execute a query. The connection alias is optional.

```
proc sql;
   connect to sybaseiq as dbcon
   (host=iqsvr1 server=iqsrv1_users db=users user=iqusr1 password=iqpwd1);
select * from connection to dbcon
   (select * from customers where customer like '1%');
quit;
```

Special Catalog Queries

SAS/ACCESS Interface to Sybase IQ supports the following special queries. You can the queries use to call the ODBC-style catalog function application programming interfaces (APIs). Here is the general format of the special queries:

SIQ::SQLAPI "*parameter 1*","*parameter n*"

SIQ::
: is required to distinguish special queries from regular queries. SIQ:: is not case sensitive.

SQLAPI
: is the specific API that is being called. SQLAPI is not case sensitive.

"*parameter n*"
: a quoted string that is delimited by commas.

Within the quoted string, two characters are universally recognized: the percent sign (%) and the underscore (_). The percent sign matches any sequence of zero or more characters, and the underscore represents any single character. To use either character as a literal value, you can use the backslash character (\) to escape the match characters. For example, this call to SQLTables usually matches table names such as myatest and my_test:

```
select * from connection to sybaseiq (SIQ::SQLTables "test","","my_test");
```

Use the escape character to search only for the my_test table:

```
select * from connection to sybaseiq (SIQ::SQLTables "test","","my\_test");
```

SAS/ACCESS Interface to Sybase IQ supports these special queries.

SIQ::SQLTables <*"Catalog"*, *"Schema"*, *"Table-name"*, *"Type"*>
: returns a list of all tables that match the specified arguments. If you do not specify any arguments, all accessible table names and information are returned.

SIQ::SQLColumns <*"Catalog"*, *"Schema"*, *"Table-name"*, *"Column-name"*>
: returns a list of all columns that match the specified arguments. If you do not specify any argument, all accessible column names and information are returned.

SIQ::SQLPrimaryKeys <*"Catalog"*, *"Schema"*, *"Table-name"*>
: returns a list of all columns that compose the primary key that matches the specified table. A primary key can be composed of one or more columns. If you do not specify any table name, this special query fails.

SIQ::SQLSpecialColumns <*"Identifier-type"*, *"Catalog-name"*, *"Schema-name"*, *"Table-name"*, *"Scope"*, *"Nullable"*>
: returns a list of the optimal set of columns that uniquely identify a row in the specified table.

SIQ::SQLStatistics <*"Catalog"*, *"Schema"*, *"Table-name"*>
: returns a list of the statistics for the specified table name, with options of SQL_INDEX_ALL and SQL_ENSURE set in the SQLStatistics API call. If you do not specify any table name argument, this special query fails.

SIQ::SQLGetTypeInfo
: returns information about the data types that the Sybase IQ database supports.

Autopartitioning Scheme for Sybase IQ

Overview

Autopartitioning for SAS/ACCESS Interface to Sybase IQ is a modulo (MOD) function method. For general information about this feature, see "Autopartitioning Techniques in SAS/ACCESS" on page 63.

Autopartitioning Restrictions

SAS/ACCESS Interface to Sybase IQ places additional restrictions on the columns that you can use for the partitioning column during the autopartitioning phase. Here is how columns are partitioned.

- INTEGER, SMALLINT, and TINYINT columns are given preference.
- You can use DECIMAL, DOUBLE, FLOAT, NUMERIC, or NUMERIC columns for partitioning if the precision minus the scale of the column is greater than 0 but less than 10—namely, 0<(*precision-scale*)<10.

Nullable Columns

If you select a nullable column for autopartitioning, the **OR<*column-name*>IS NULL** SQL statement is appended at the end of the SQL code that is generated for the threaded Read. This ensures that any possible NULL values are returned in the result set. Also, if the column to be used for partitioning is defined as BIT, the number of threads are automatically changed to two, regardless how DBSLICEPARM= is set.

Using WHERE Clauses

Autopartitioning does not select a column to be the partitioning column if it appears in a SAS WHERE clause. For example, this DATA step cannot use a threaded Read to retrieve the data because all numeric columns in the table are in the WHERE clause:

```
data work.locemp;
set iqlib.MYEMPS;
where EMPNUM<=30 and ISTENURE=0 and
 SALARY<=35000 and NUMCLASS>2;
run;
```

Using DBSLICEPARM=

Although SAS/ACCESS Interface to Sybase IQ defaults to three threads when you use autopartitioning, do not specify a maximum number of threads for the threaded Read in the DBSLICEPARM= LIBNAME option on page 139.

Using DBSLICE=

You might achieve the best possible performance when using threaded Reads by specifying the DBSLICE= data set option on page 299 for Sybase IQ in your SAS operation. This is especially true if you defined an index on one of the columns in the table. SAS/ACCESS Interface to Sybase IQ selects only the first integer-type column in the table. This column might not be the same column where the index is defined. If so, you can specify the indexed column using DBSLICE=, as shown in this example.

```
proc print data=iqlib.MYEMPS(DBSLICE=("EMPNUM BETWEEN 1 AND 33"
"EMPNUM BETWEEN 34 AND 66" "EMPNUM BETWEEN 67 AND 100"));
run;
```

Using DBSLICE= also gives you flexibility in column selection. For example, if you know that the STATE column in your employee table contains only a few distinct values, you can customize your DBSLICE= clause accordingly.

```
datawork.locemp;
set iqlib2.MYEMP(DBSLICE=("STATE='FL'" "STATE='GA'"
  "STATE='SC'" "STATE='VA'" "STATE='NC'"));
where EMPNUM<=30 and ISTENURE=0 and SALARY<=35000 and NUMCLASS>2;
run;
```

Passing SAS Functions to Sybase IQ

SAS/ACCESS Interface to Sybase IQ passes the following SAS functions to Sybase IQ for processing. Where the Sybase IQ function name differs from the SAS function name, the Sybase IQ name appears in parentheses. For more information, see "Passing Functions to the DBMS Using PROC SQL" on page 46.

ABS	MIN
ACOS (ACOS)	MINUTE
ARSIN (ASIN)	MOD
ATAN	MONTH
AVG	QTR (QUARTER)
BYTE (CHAR)	REPEAT
CEIL	SECOND
COALESCE	SIGN
COS	SIN
COUNT	SQRT
DAY	STRIP (TRIM)
EXP	SUBSTR (SUBSTRING)
FLOOR	SUM
HOUR	TAN
INDEX (LOCATE)	TRANWRD (REPLACE)
LOG	TRIMN (RTRIM)
LOG10	UPCASE (UPPER)
LOWCASE (LOWER)	WEEKDAY (DOW)
MAX	YEAR

SQL_FUNCTIONS=ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to Sybase IQ. Due to incompatibility in date and time functions between Sybase IQ and SAS, Sybase IQ might not process them correctly. Check your results to determine whether these functions are working as expected. For more information, see "SQL_FUNCTIONS= LIBNAME Option" on page 185.

COMPRESS (REPLACE)	SOUNDEX
DATE (CURRENT_DATE)	TIME (CURRENT_TIME)
DATEPART (DATE)	TIMEPART (TIME)
DATETIME (CURRENT_TIMESTAMP)	TODAY (CURRENT_DATE)
LENGTH (BYTE_LENGTH)	TRIM

Passing Joins to Sybase IQ

For a multiple libref join to pass to Sybase IQ, all of these components of the LIBNAME statements must match exactly.

- user ID (USER=)
- password (PASSWORD=)
- host (HOST=)
- server (SERVER=)
- database (DATABASE=)
- port (PORT=)
- data source (DSN=, if specified)
- SQL functions (SQL_FUNCTIONS=)

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Bulk Loading for Sybase IQ

Loading

Bulk loading is the fastest way to insert large numbers of rows into a Sybase IQ table. To use the bulk-load facility, specify BULKLOAD=YES. The bulk-load facility uses the Sybase IQ LOAD TABLE command to move data from the client to the Sybase IQ database.

Here are the Sybase IQ bulk-load data set options. For detailed information about these options, see "Overview" on page 207.

- BL_CLIENT_DATAFILE=
- BL_DATAFILE=
- BL_DELETE_DATAFILE=
- BL_DELIMITER=
- BL_OPTIONS= on page 255
- BL_SERVER_DATAFILE=
- BL_USE_PIPE=
- BULKLOAD=

Examples

In this example, the SASFLT.FLT98 SAS data set creates and loads FLIGHTS98, a large Sybase IQ table. For Sybase IQ 12.x, this works only when the Sybase IQ server is on the same server as your SAS session.

```
libname sasflt 'SAS-library';
libname mydblib sybaseiq host=iqsvr1 server=iqsrv1_users
    db=users user=iqusr1 password=iqpwd1;

proc sql;
.create table mydblib.flights98
       (bulkload=YES)
```

```
        as select * from sasflt.flt98;
quit;
```

When the Sybase IQ server and your SAS session are not on the same server, you need to include additional options, as shown in this example.

```
libname sasflt 'SAS-library';
libname mydblib sybaseiq host=iqsvr1 server=iqsrv1_users
    db=users user=iqusr1 password=iqpwd1;
proc sql;
create table mydblib.flights98
 (  BULKLOAD=YES
   BL_USE_PIPE=NO
    BL_SERVER_DATAFILE='/tmp/fltdata.dat'
    BL_CLIENT_DATAFILE='/tmp/fltdata.dat' )
 as select * from sasflt.flt98;
quit;
```

In this example, you can append the SASFLT.FLT98 SAS data set to the existing Sybase IQ table, ALLFLIGHTS. The BL_USE_PIPE=NO option forces SAS/ACCESS Interface to Sybase IQ to write data to a flat file, as specified in the BL_DATAFILE= option. Rather than deleting the data file, BL_DELETE_DATAFILE=NO causes the engine to leave it after the load has completed.

```
proc append base=mydblib.allflights
     (BULKLOAD=YES
       BL_DATAFILE='/tmp/fltdata.dat'
       BL_USE_PIPE=NO
       BL_DELETE_DATAFILE=NO)
data=sasflt.flt98;
run;
```

Locking in the Sybase IQ Interface

The following LIBNAME and data set options let you control how the Sybase IQ interface handles locking. For general information about an option, see "LIBNAME Options for Relational Databases" on page 99.

READ_LOCK_TYPE= ROW | TABLE

UPDATE_LOCK_TYPE= ROW | TABLE

READ_ISOLATION_LEVEL= S | RR | RC | RU
Sybase IQ supports the S, RR, RC, and RU isolation levels that are defined in this table.

Table 28.3 *Isolation Levels for Sybase IQ*

Isolation Level	Definition
S (serializable)	Does not allow dirty Reads, nonrepeatable Reads, or phantom Reads.
RR (repeatable read)	Does not allow dirty Reads or nonrepeatable Reads; does allow phantom Reads.

Isolation Level	Definition
RC (read committed)	Does not allow dirty Reads or nonrepeatable Reads; does allow phantom Reads.
RU (read uncommitted)	Allows dirty Reads, nonrepeatable Reads, and phantom Reads.

Here are how the terms in the table are defined.

Dirty reads
A transaction that exhibits this phenomenon has very minimal isolation from concurrent transactions. In fact, it can see changes that are made by those concurrent transactions even before they commit.

For example, suppose that transaction T1 performs an update on a row, transaction T2 then retrieves that row, and transaction T1 then terminates with rollback. Transaction T2 has then seen a row that no longer exists.

Nonrepeatable reads
If a transaction exhibits this phenomenon, it is possible that it might read a row once and if it attempts to read that row again later in the course of the same transaction, the row might have been changed or even deleted by another concurrent transaction. Therefore, the read is not (necessarily) repeatable.

For example, suppose that transaction T1 retrieves a row, transaction T2 then updates that row, and transaction T1 then retrieves the same row again. Transaction T1 has now retrieved the same row twice but has seen two different values for it.

Phantom reads
When a transaction exhibits this phenomenon, a set of rows that it reads once might be a different set of rows if the transaction attempts to read them again.

For example, suppose that transaction T1 retrieves the set of all rows that satisfy some condition. Suppose that transaction T2 then inserts a new row that satisfies that same condition. If transaction T1 now repeats its retrieval request, it sees a row that did not previously exist, a phantom.

UPDATE_ISOLATION_LEVEL= S | RR | RC
Sybase IQ supports the S, RR, and RC isolation levels defined in the preceding table.

Naming Conventions for Sybase IQ

For general information about this feature, see “SAS Names and Support for DBMS Names” on page 13.

Since SAS 7, most SAS names can be up to 32 characters long. SAS/ACCESS Interface to Sybase IQ supports table names and column names that contain up to 32 characters. If DBMS column names are longer than 32 characters, they are truncated to 32 characters. If truncating a column name would result in identical names, SAS generates a unique name by replacing the last character with a number. DBMS table names must be 32 characters or less because SAS does not truncate a longer name. If you already have a table name that is greater than 32 characters, it is recommended that you create a table view. For more information, see "SAS Names and Support for DBMS Names" on page 13.

The PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options determine how SAS/ACCESS Interface to Sybase IQ handles case sensitivity. (For information about these options, see "Overview: LIBNAME Statement for Relational Databases " on page 93.) Sybase IQ is not case sensitive, so all names default to lowercase.

Sybase IQ objects include tables, views, and columns. They follow these naming conventions.

- A name must be from 1 to 128 characters long.
- A name must begin with a letter (A through Z), underscore (_), at sign (@), dollar sign ($), or pound sign (#).
- Names are not case sensitive. For example, CUSTOMER and `Customer` are the same, but object names are converted to lowercase when they are stored in the Sybase IQ database. However, if you enclose a name in quotation marks, it is case sensitive.
- A name cannot be a Sybase IQ reserved word, such as WHERE or VIEW.
- A name cannot be the same as another Sybase IQ object that has the same type.

For more information, see your *Sybase IQ Reference Manual*.

Data Types for Sybase IQ

Overview

Every column in a table has a name and a data type. The data type tells Sybase IQ how much physical storage to set aside for the column and the form in which the data is stored. This information includes information about Sybase IQ data types, null and default values, and data conversions.

For more information about Sybase IQ data types and to determine which data types are available for your version of Sybase IQ, see your *Sybase IQ Reference Manual*.

SAS/ACCESS Interface to Sybase IQ does not directly support any data types that are not listed below. Any columns using these types are read into SAS as character strings.

String Data

CHAR(*n*)
: specifies a fixed-length column for character string data. The maximum length is 32,768 characters. If the length is greater than 254, the column is a long-string column. SQL imposes some restrictions on referencing long-string columns. For more information about these restrictions, see your Sybase IQ documentation.

VARCHAR(*n*)
: specifies a varying-length column for character string data. The maximum length is 32,768 characters. If the length is greater than 254, the column is a long-string column. SQL imposes some restrictions on referencing long-string columns. For more information about these restrictions, see your Sybase IQ documentation.

LONG VARCHAR(*n*)
: specifies a varying-length column for character string data. The maximum size is limited by the maximum size of the database file, which is currently 2 gigabytes.

Numeric Data

BIGINT
: specifies a big integer. Values in a column of this type can range from –9223372036854775808 to +9223372036854775807.

SMALLINT
: specifies a small integer. Values in a column of this type can range from –32768 through +32767.

INTEGER
: specifies a large integer. Values in a column of this type can range from –2147483648 through +2147483647.

TINYINT
: specifies a tiny integer. Values in a column of this type can range from 0 to 255.

BIT
: specifies a Boolean type. Values in a column of this type can be either 0 or 1. Inserting any nonzero value into a BIT column stores a 1 in the column.

DOUBLE | DOUBLE PRECISION
: specifies a floating-point number that is 64 bits long. Values in a column of this type can range from –1.79769E+308 to –2.225E-307 or +2.225E-307 to +1.79769E+308, or they can be 0. This data type is stored the same way that SAS stores its numeric data type. Therefore, numeric columns of this type require the least processing when SAS accesses them.

REAL
: specifies a floating-point number that is 32 bits long. Values in a column of this type can range from approximately –3.4E38 to –1.17E-38 and +1.17E-38 to +3.4E38.

FLOAT
: specifies a floating-point number. If you do not supply the precision, the FLOAT data type is the same as the REAL data type. If you supply the precision, the FLOAT data type is the same as the REAL or DOUBLE data type, depending on the value of the precision. The cutoff between REAL and DOUBLE is platform-dependent, and it is the number of bits that are used in the mantissa of the single-precision floating-point number on the platform.

DECIMAL | DEC | NUMERIC
: specifies a fixed-point decimal number. The precision and scale of the number determines the position of the decimal point. The numbers to the right of the decimal point are the scale, and the scale cannot be negative or greater than the precision. The maximum precision is 126 digits.

Date, Time, and Timestamp Data

SQL date and time data types are collectively called datetime values. The SQL data types for dates, times, and timestamps are listed here. Be aware that columns of these data types can contain data values that are out of range for SAS.

DATE
: specifies date values. The range is 01-01-0001 to 12-31-9999. The default format is *YYYY-MM-DD*. An example is 1961-06-13.

TIME
: specifies time values in hours, minutes, and seconds to six decimal positions: hh:mm:ss[.nnnnnn]. The range is 00:00:00.000000 to 23:59:59.999999. However, due to the ODBC-style interface that SAS/ACCESS Interface to Sybase IQ uses to communicate with the Sybase IQ Performance Server, any fractional seconds are lost in the transfer of data from server to client.

TIMESTAMP
: combines a date and time in the default format of *yyyy-mm-dd hh:mm:ss[.nnnnnn]*. For example, a timestamp for precisely 2:25 p.m. on January 25, 1991, would be 1991-01-25-14.25.00.000000. Values in a column of this type have the same ranges as described for DATE and TIME.

Sybase IQ Null Values

Sybase IQ has a special value called NULL. A Sybase IQ NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a Sybase IQ NULL value, it interprets it as a SAS missing value.

You can define a column in a Sybase IQ table so that it requires data. To do this in SQL, you specify a column as NOT NULL. This tells SQL to allow only a row to be added to a table if a value exists for the field. For example, NOT NULL assigned to the CUSTOMER field in the SASDEMO.CUSTOMER table does not allow a row to be added unless there is a value for CUSTOMER. When creating a table with SAS/ACCESS, you can use the DBNULL= data set option to indicate whether NULL is a valid value for specified columns.

You can also define Sybase IQ columns as NOT NULL DEFAULT. For more information about using the NOT NULL DEFAULT value, see your *Sybase IQ Reference Manual*.

Once you know whether a Sybase IQ column enables NULLs or the host system supplies a default value for a column that is defined as NOT NULL WITH DEFAULT, you can write selection criteria and enter values to update a table. Unless a column is defined as NOT NULL or NOT NULL DEFAULT, it allows NULL values.

For more information about how SAS handles NULL values, see “Potential Result Set Differences When Processing Null Data ” on page 35.

To control how SAS missing character values are handled, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to Sybase IQ assigns to SAS variables to read from a Sybase IQ table when using the “Overview: LIBNAME Statement for Relational Databases ” on page 93. These default formats are based on Sybase IQ column attributes.

Table 28.4 LIBNAME Statement: Default SAS Formats for Sybase IQ Data Types

Sybase IQ Data Type	SAS Data Type	Default SAS Format
CHAR(*n*)*	character	$*n*.
VARCHAR(*n*)*	character	$*n*.
LONG VARCHAR(*n*)*	character	$*n*.
BIGINT	numeric	20.
SMALLINT	numeric	6.
TINYINT	numeric	4.
INTEGER	numeric	11.
BIT	numeric	1.
DOUBLE	numeric	none
REAL	numeric	none
FLOAT	numeric	none
DECIMAL(*p,s*)	numeric	*m.n*
NUMERIC(*p,s*)	numeric	*m.n*
TIME	numeric	TIME8.
DATE	numeric	DATE9.
TIMESTAMP	numeric	DATETIME*25.6*

* *n* in Sybase IQ data types is equivalent to *w* in SAS formats.

This table shows the default Sybase IQ data types that SAS/ACCESS assigns to SAS variable formats during output operations when you use the LIBNAME statement.

Table 28.5 LIBNAME Statement: Default Sybase IQ Data Types for SAS Variable Formats

SAS Variable Format	Sybase IQ Data Type
m.n	DECIMAL(*p,s*)
other numerics	DOUBLE
$*n*.	VARCHAR(*n*)*
datetime formats	TIMESTAMP
date formats	DATE

SAS Variable Format	Sybase IQ Data Type
time formats	TIME

* *n* in Sybase IQ data types is equivalent to *w* in SAS formats.

Chapter 29
SAS/ACCESS Interface to Teradata

Introduction to SAS/ACCESS Interface to Teradata

Overview

This section describes SAS/ACCESS Interface to Teradata. For a list of SAS/ACCESS features that are available for this interface, see “SAS/ACCESS Interface to Teradata: Supported Features” on page 90.

Note: SAS/ACCESS Interface to Teradata does not support the DBLOAD and ACCESS procedures. The LIBNAME engine technology enhances and replaces the functionality of these procedures. Therefore, you must revise SAS jobs that were written for a different SAS/ACCESS interface and that include DBLOAD or ACCESS procedures before you can run them with SAS/ACCESS Interface to Teradata.

The SAS/ACCESS Teradata Client

Teradata is a massively parallel (MPP) RDBMS. A high-end Teradata server supports many users. It simultaneously loads and extracts table data and processes complex queries.

Because Teradata customers run many processors at the same time for queries of the database, users enjoy excellent DBMS *server* performance. The challenge to client software, such as SAS, is to leverage Teradata performance by rapidly extracting and loading table data. SAS/ACCESS Interface to Teradata meets this challenge by letting you optimize extracts and loads (reads and creates).

Information throughout this document explains how you can use the SAS/ACCESS interface to optimize DBMS operations:

- It can create and update Teradata tables. It supports a FastLoad interface that rapidly creates new table. It can also potentially optimize table reads by using FastExport for the highest possible read performance.
- It supports MultiLoad, which loads both empty and existing Teradata tables and greatly accelerates the speed of insertion into Teradata tables.
- It supports the Teradata Parallel Transporter (TPT) API on Windows and UNIX. This API uses the Teradata Load, Update, and Stream operator to load data and the export operator to read data.

LIBNAME Statement Specifics for Teradata

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to Teradata supports and includes examples. For a complete description of this feature, see LIBNAME statement on page 93.

Here is the LIBNAME statement syntax for accessing Teradata.

LIBNAME *libref* **teradata** <*connection-options*> <*LIBNAME-options*>;

Arguments

libref
: specifies any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

`teradata`
: specifies the SAS/ACCESS engine name for the Teradata interface.

connection-options
: provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. Here are the connection options for the Teradata interface.

USER=<'>*Teradata-user-name*<'> | <">*ldapid*@LDAP<"> | <">*ldapid*@LDAP*realm-name*<">
: specifies a required connection option that specifies a Teradata user name. If the name contains blanks or national characters, enclose it in quotation marks. For LDAP authentication with either a NULL or single realm, append only the @LDAP token to the Teradata user name. In this case, no realm name is needed. If you append a realm name, the LDAP authentication server ignores it and authentication proceeds. However, if multiple realms exist, you must append the realm name to the @LDAP token. In this case, an LDAP server must already be configured to accept authentication requests from the Teradata server.

PASSWORD=<'>*Teradata-password*<'>
: specifies a required connection option that specifies a Teradata password. The password that you specify must be correct for your USER= value. If the password contains spaces or nonalphanumeric characters, you must enclose it in quotation marks. If you do not want to enter your Teradata password in clear text on this statement, see PROC PWENCODE in the *Base SAS Procedures Guide* for a method for encoding it. For LDAP authentication, you use this password option to specify the authentication string or password. If the authentication string or password includes an embedded @ symbol, a backslash (\) is required and it must precede the @ symbol. See "Teradata LIBNAME Statement Examples" on page 784.

ACCOUNT=<'>*account_ID*<'>
: is an optional connection option that specifies the account number that you want to charge for the Teradata session.

TDPID=<'>*dbcname*<'>
: Alias: SERVER=

: specifies a required connection option if you run more than one Teradata server. TDPID= operates differently for network-attached and channel-attached systems, as described below.

- For NETWORK-ATTACHED systems (PC and UNIX), *dbcname* specifies an entry in your (client) HOSTS file that provides an IP address for a database server connection.

  By default, SAS/ACCESS connects to the Teradata server that corresponds to the *dbccop1* entry in your HOSTS file. When you run only one Teradata server and your HOSTS file defines the *dbccop1* entry correctly, you do not need to specify TDPID=.

  However, if you run more than one Teradata server, you must use the TDPID= option to specifying a *dbcname* of eight characters or less. SAS/ACCESS adds the specified *dbcname* to the login string that it submits to Teradata. (Teradata documentation refers to this name as the tdpid component of the login string.)

  After SAS/ACCESS submits a *dbcname* to Teradata, Teradata searches your HOSTS file for all entries that begin with the same *dbcname*. For Teradata to recognize the HOSTS file entry, the *dbcname* suffix must be COP*x* (*x* is a number). If there is only one entry that matches the *dbcname*, *x* must be 1. If there are multiple entries for the *dbcname*, *x* must begin with 1 and increment sequentially for each related entry. (See the example HOSTS file entries below.)

  When there are multiple, matching entries for a *dbcname* in your HOSTS file, Teradata does simple load balancing by selecting one of the Teradata servers specified for login. Teradata distributes your queries across these servers so that it can return your results as fast as possible.

The TDPID= examples below assume that your HOSTS file contains these *dbcname* entries and IP addresses.

- Example 1: The TDPID= option is not specified, establishing a login to the Teradata server that runs at 10.25.20.34.

```
dbccop1 10.25.20.34
```

- Example 2: Using TDPID= myserver *or* SERVER=myserver, you specify a login to the Teradata server that runs at 130.96.8.207.

```
myservercop1 130.96.8.207
```

- Example 3: Using TDPID=xyz *or* SERVER=xyz, you specify a login to a Teradata server that runs at 11.22.33.44 or to a Teradata server that runs at 33.44.55.66.

```
xyzcop1 33.44.55.66
```

or

```
xyzcop2 11.22.33.44
```

- For CHANNEL-ATTACHED systems (z/OS), TDPID= specifies the subsystem name. This name must be TDPx, where x can be 0–9, A–Z (not case sensitive), or $, # or @. If there is only one Teradata server, and your z/OS system administrator has set up the HSISPB and HSHSPB modules, you do not need to specify TDPID=. For further information, see your Teradata TDPID documentation for z/OS.

DATABASE=<'>*database-name*<'>
Alias: DB=

specifies an optional connection option that specifies the name of the Teradata database that you want to access. This option enables you to view or modify a different user's Teradata DBMS tables or views, if you have the required privileges. (For example, to read another user's tables, you must have the Teradata privilege SELECT for that user's tables.) If you do not specify DATABASE=, the libref points to your default Teradata database, which is often named the same as your user name. If the database value that you specify contains spaces or nonalphanumeric characters, you must enclose it in quotation marks.

SCHEMA=<'>*database-name*<'>
specifies an optional connection option that specifies the database name to use to qualify any database objects that the LIBNAME can reference.

LIBNAME-options
define how SAS processes DBMS objects. Some LIBNAME options can enhance performance, and others determine locking or naming behavior. The following table describes the LIBNAME options for SAS/ACCESS Interface to Teradata, with the applicable default values. For more detail about these options, see “LIBNAME Options for Relational Databases” on page 99.

Table 29.1 *SAS/ACCESS LIBNAME Options for Teradata*

Option	Default Value
ACCESS=	none

Option	Default Value
AUTHDOMAIN=	none
BULKLOAD=	NO
CAST=	none
CAST_OVERHEAD_MAXPERCENT=	20%
CONNECTION=	SHAREDREAD for channel-attached systems (z/OS); UNIQUE for network attached systems (UNIX and PC platforms)
CONNECTION_GROUP=	none
DATABASE= (see SCHEMA=)	none
DBCOMMIT=	1000 when inserting rows; 0 when updating rows
DBCONINIT=	none
DBCONTERM=	none
DBCREATE_TABLE_OPTS=	none
DBGEN_NAME=	DBMS
DBINDEX=	NO
DBLIBINIT=	none
DBLIBTERM=	none
DBMSTEMP=	NO
DBPROMPT=	NO
DBSASLABEL=	COMPAT
DBSLICEPARM=	THREADED_APPS,2
DEFER=	NO
DIRECT_EXE=	
DIRECT_SQL=	YES
ERRLIMIT=	1 million
FASTEXPORT=	NO

Option	Default Value
LOGDB=	Default Teradata database for the libref
MODE=	ANSI
MULTISTMT=	NO
MULTI_DATASRC_OPT=	IN_CLAUSE
PREFETCH=	not enabled
PRESERVE_COL_NAMES=	YES
PRESERVE_TAB_NAMES=	YES
QUERY_BAND=	none
READ_ISOLATION_LEVEL=	see "Locking in the Teradata Interface" on page 814
READ_LOCK_TYPE=	none
READ_MODE_WAIT=	none
REREAD_EXPOSURE=	NO
SCHEMA=	your default Teradata database
SESSIONS=	none
SLEEP=	6
SPOOL=	YES
SQL_FUNCTIONS=	none
SQL_FUNCTIONS_COPY=	none
SQLGENERATION=	DBMS
TEMPORAL_QUALIFIER=	CURRENT VALIDTIME for valid-time column; CURRENT TRANSACTIONTIME for transaction-time column
TENACITY=	0 for FastLoad; 4 for FASTEXPORT= and MultiLoad
TPT=	YES
TR_ENABLE_INTERRUPT=	NO
UPDATE_ISOLATION_LEVEL=	see "Locking in the Teradata Interface" on page 814

Option	Default Value
"UPDATE_LOCK_TYPE= LIBNAME Option"	none
UPDATE_MODE_WAIT=	none
UTILCONN_TRANSIENT=	NO

Teradata LIBNAME Statement Examples

These examples show how to make the proper connection by using the USER= and PASSWORD= connection options on page 779. Teradata requires these options, and you must use them together.

This example shows how to connect to a single or NULL realm.

```
libname x teradata user="johndoe@LDAP" password="johndoeworld";
```

Here is an example of how to make the connection to a specific realm where multiple realms are configured.

```
libname x teradata user="johndoe@LDAPjsrealm" password="johndoeworld";
```

Here is an example of a configuration with a single or NULL realm that contains a password with an embedded @ symbol. The password must contain a required backslash (\), which must precede the embedded @ symbol.

```
libname x teradata user="johndoe@LDAP" password="johndoe\@world"
```

Data Set Options for Teradata

All SAS/ACCESS data set options in this table are supported for Teradata. Default values are provided where applicable. For details about this feature, see "Overview" on page 207.

Table 29.2 *SAS/ACCESS Data Set Options for Teradata*

Option	Default Value
BL_CONTROL=	creates a FastExport control file in the current directory with a platform-specific name
BL_DATAFILE=	creates a MultiLoad script file in the current directory or with a platform-specific name
BL_LOG=	FastLoad errors are logged in Teradata tables named TableName_SE1_*randnum* and TableName_SE2_*randnum*. TableName consists of up to the first 15 characters of the target table name, and *randnum* is a randomly generated number.

Option	Default Value
BUFFERS=	2
BULKLOAD=	NO
CAST=	none
CAST_OVERHEAD_MAXPERCENT=	20%
DBCOMMIT=	the current LIBNAME option setting
DBCONDITION=	none
DBCREATE_TABLE_OPTS=	the current LIBNAME option setting
DBFORCE=	NO
DBGEN_NAME=	DBMS
DBINDEX=	the current LIBNAME option setting
DBKEY=	none
DBLABEL=	NO
DBMASTER=	none
DBNULL=	none
DBSASLABEL=	COMPAT
DBSASTYPE=	see "Data Types for Teradata " on page 820
DBSLICE=	none
DBSLICEPARM=	THREADED_APPS,2
DBTYPE=	see "Data Types for Teradata " on page 820
ERRLIMIT=	1
MBUFFSIZE=	0
ML_CHECKPOINT=	none
ML_ERROR1=	none
ML_ERROR2=	none
ML_LOG=	none

Option	Default Value
ML_RESTART=	none
ML_WORK=	none
MULTILOAD=	NO
MULTISTMT=	NO
NULLCHAR=	SAS
NULLCHARVAL=	a blank character
PRESERVE_COL_NAMES=	YES
QUERY_BAND=	none
READ_ISOLATION_LEVEL=	the current LIBNAME option setting
READ_LOCK_TYPE=	the current LIBNAME option setting
READ_MODE_WAIT=	the current LIBNAME option setting
SASDATEFMT=	none
SCHEMA=	the current LIBNAME option setting
SESSIONS=	none
SET=	NO
SLEEP=	6
TEMPORAL_QUALIFIER=	CURRENT VALIDTIME [valid-time column]CURRENT TRANSACTIONTIME [transaction-time column]
TENACITY=	0 [FastLoad], 4 [FASTEXPORT=, MultiLoad]
TPT=	YES
TPT_APPL_PHASE=	NO
TPT_BUFFER_SIZE=	64
TPT_CHECKPOINT_DATA=	none
TPT_DATA_ENCRYPTION=	none
TPT_ERROR_TABLE_1=	table_name_ET

Option	Default Value
TPT_ERROR_TABLE_2=	table_name_UV
TPT_LOG_TABLE=	table_name_RS
TPT_MAX_SESSIONS=	1 session per available Access Module Processor (AMP)
TPT_MIN_SESSION=	1 session
TPT_PACK=	20
TPT_PACKMAXIMUM=	NO
TPT_RESTART=	NO
TPT_TRACE_LEVEL=	1
TPT_TRACE_LEVEL_INF=	1
TPT_TRACE_OUTPUT=	driver name, followed by a timestamp
TPT_WORK_TABLE=	table_name_WT
UPDATE_ISOLATION_LEVEL=	the current LIBNAME option setting
UPDATE_LOCK_TYPE=	the current LIBNAME option setting
UPDATE_MODE_WAIT=	the current LIBNAME option setting

SQL Pass-Through Facility Specifics for Teradata

Key Information

For general information about this feature, see "SQL Pass-Through Facility" on page 401. Teradata examples are available.

Here are the SQL pass-through facility specifics for the Teradata interface.

- The *dbms-name* is **`TERADATA`**.
- The CONNECT statement is required.
- The Teradata interface can connect to multiple Teradata servers and to multiple Teradata databases. However, if you use multiple simultaneous connections, you must use an *alias* argument to identify each connection.
- The CONNECT statement *database-connection-arguments* are identical to the LIBNAME connection options on page 779.

  The MODE= LIBNAME option is available with the CONNECT statement. By default, SAS/ACCESS opens Teradata connections in ANSI mode. In contrast, most

Teradata tools, such as BTEQ, run in Teradata mode. If you specify MODE=TERADATA, pass-through connections open in Teradata mode, forcing Teradata mode rules for all SQL requests that are passed to the Teradata DBMS. For example, MODE= impacts transaction behavior and case sensitivity. See your Teradata SQL reference documentation for a complete discussion of ANSI versus Teradata mode.

- By default, SAS/ACCESS opens Teradata in ANSI mode. You must therefore use one of these techniques when you write PROC SQL steps that use the SQL pass-through facility.
  - Specify an explicit COMMIT statement to close a transaction. You must also specify an explicit COMMIT statement after any data definition language (DDL) statement. The examples below demonstrate these rules. For further information about ANSI mode and DDL statements, see your Teradata SQL reference documentation.
  - Specify MODE=TERADATA in your CONNECT statement. When MODE=TERADATA, you do not specify explicit COMMIT statements as described above. When MODE=TERADATA, data processing is not case sensitive. This option is available when you use the LIBNAME statement and also with the SQL pass-through facility.

CAUTION:

Do not issue a Teradata DATABASE statement within the EXECUTE statement in PROC SQL. Add the SCHEMA= option to your CONNECT statement if you must change the default Teradata database.

Examples

In this example, SAS/ACCESS connects to the Teradata DBMS using the **dbcon** alias.

```
proc sql;
   connect to teradata as dbcon (user=testuser pass=testpass);
quit;
```

In the next example, SAS/ACCESS connects to the Teradata DBMS using the **tera** alias, drops and re-creates the SALARY table, inserts two rows, and disconnects from the Teradata DBMS. Note that COMMIT must follow each DDL statement. DROP TABLE and CREATE TABLE are DDL statements. The COMMIT statement that follows the INSERT statement is also required. Otherwise, Teradata rolls back the inserted rows.

```
proc sql;
   connect to teradata as tera ( user=testuser password=testpass );
   execute (drop table salary) by tera;
   execute (commit) by tera;
   execute (create table salary (current_salary float, name char(10)))
           by tera;
   execute (commit) by tera;
   execute (insert into salary values (35335.00, 'Dan J.')) by tera;
   execute (insert into salary values (40300.00, 'Irma L.')) by tera;
   execute (commit) by tera;
   disconnect from tera;
quit;
```

For this example, SAS/ACCESS connects to the Teradata DBMS using the **tera** alias, updates a row, and disconnects from the Teradata DBMS. The COMMIT statement causes Teradata to commit the update request. Without the COMMIT statement, Teradata rolls back the update.

```
 proc sql;
   connect to teradata as tera ( user=testuser password=testpass );
   execute (update salary set current_salary=45000
            where (name='Irma L.')) by tera;
   execute (commit) by tera;
   disconnect from tera;
  quit;
```

In this example, SAS/ACCESS uses the **tera2** alias to connect to the Teradata database, selects all rows in the SALARY table, displays them using PROC SQL, and disconnects from the Teradata database. No COMMIT statement is needed in this example because the operations are reading only data. No changes are made to the database.

```
proc sql;
   connect to teradata as tera2 ( user=testuser password=testpass );
   select * from connection to tera2 (select * from salary);
   disconnect from tera2;
  quit;
```

In this next example, MODE=TERADATA is specified to avoid case-insensitive behavior. Because Teradata Mode is used, SQL COMMIT statements are not required.

```
/* Create & populate the table in Teradata mode (case insensitive). */
proc sql;
   connect to teradata (user=testuser pass=testpass mode=teradata);
   execute(create table casetest(x char(28)) ) by teradata;
   execute(insert into casetest values('Case Insensitivity Desired') )
      by teradata;
quit;
/* Query the table in Teradata mode (for case-insensitive match). */
proc sql;
   connect to teradata (user=testuser pass=testpass mode=teradata);
   select * from connection to teradata (select * from
   casetest where x='case insensitivity desired');
quit;
```

Autopartitioning Scheme for Teradata

Overview

For general information about this feature, see “Autopartitioning Techniques in SAS/ACCESS” on page 63.

The FastExport Utility is the fastest available way to read large Teradata tables. FastExport is NCR-provided software that delivers data over multiple Teradata connections or sessions. If FastExport is available, SAS threaded Reads use it. If FastExport is not available, SAS threaded Reads generate partitioning WHERE clauses. Using the DBSLICE= option overrides FastExport. So if you have FastExport available and want to use it, do not use DBSLICE=. To use FastExport everywhere possible, use DBSLICEPARM=ALL.

Note: FastExport is supported only on z/OS and UNIX. Whether automatically generated or created by using DBSLICE=, partitioning WHERE clauses is not supported.

FastExport and Case Sensitivity

In certain situations Teradata returns different row results to SAS when using FastExport, compared to reading normally without FastExport. The difference arises only when all of these conditions are met:

- A WHERE clause is asserted that compares a character column with a character literal.
- The column definition is NOT CASESPECIFIC.

  Unless you specify otherwise, most Teradata native utilities create NOT CASESPECIFIC character columns. On the other hand, SAS/ACCESS Interface to Teradata creates CASESPECIFIC columns. In general, this means that you do not see result differences with tables that SAS creates. However, but you might see result differences with tables that Teradata utilities create, which are frequently many of your tables. To determine how Teradata creates a table, look at your column declarations with the Teradata SHOW TABLE statement.
- A character literal matches to a column value that differs only in case.

  You can see differences in the rows returned if your character column has mixed-case data that is otherwise identical. For example, 'Top' and 'top' are identical except for case.

Case sensitivity is an issue when SAS generates SQL code that contains a WHERE clause with one or more character comparisons. It is also an issue when you supply the Teradata SQL yourself with the explicit SQL feature of PROC SQL. The following examples illustrate each scenario, using DBSLICEPARM=ALL to start FastExport instead of the normal SAS read:

```
/* SAS generates the SQL for you. */
libname trlib teradata user=username password=userpwd dbsliceparm=all;
proc print data=trlib.employees;
where lastname='lovell';
run;

/* Use explicit SQL with PROC SQL & supply the
SQL yourself, also starting FastExport. */
proc sql;
   connect to teradata(user=username password=userpwd dbsliceparm=all);
select * from connection to teradata
   (select * from sales where gender='f' and salesamt>1000);
quit;
```

For more information about case sensitivity, see your Teradata documentation.

FastExport Password Security

FastExport requires passwords to be in clear text. Because this poses a security risk, users must specify the full pathname so that the file path is in a protected directory:

- Windows users should specify *BL_CONTROL="PROTECTED-DIR/myscr.ctl"*. SAS/ACCESS creates the myscr.ctl script file in the protected directory with PROTECTED-DIR as the path.

- UNIX users can specify a similar pathname.
- MVS users must specify a middle-level qualifier such as *BL_CONTROL="MYSCR.TEST1"* so that the system generates the USERID.MYSCR.TEST1.CTL script file.
- Users can also use RACF to protect the USERID.MYSCR* profile.

FastExport Setup

There are three requirements for using FastExport with SAS:

- You must have the Teradata FastExport Utility present on your system. If you do not have FastExport and want to use it with SAS, contact NCR to obtain the Utility.
- SAS must be able to locate the FastExport Utility on your system.
- The FastExport Utility must be able to locate the SasAxsm access module, which is supplied with your SAS/ACCESS Interface to Teradata product. SasAxsm is in the SAS directory tree, in the same location as the sasiotra component.

Assuming you have the Teradata FastExport Utility, perform this setup, which varies by system:

- *Windows:* As needed, modify your Path environment variable to include *both* the directories containing Fexp.exe (FastExport) and SasAxsm. Place these directory specifications last in your path.
- *UNIX:* As needed, modify your library path environment variable to include the directory containing sasaxsm.sl (HP) or sasaxsm.so (Solaris and AIX). These shared objects are delivered in the $SASROOT/sasexe directory. You can copy these modules where you want, but make sure that the directory into which you copy them is in the appropriate shared library path environment variable. On Solaris, the library path variable is LD_LIBRARY_PATH. On HP-UX, it is SHLIB_PATH. On AIX, it is LIBPATH. Also, make sure that the directory containing the Teradata FastExport Utility (fexp), is included in the PATH environment variable. FastExport is usually installed in the /usr/bin directory.
- *z/OS:* No action is needed when starting FastExport under TSO. When starting FastExport with a batch JCL, the SAS source statements must be assigned to a DD name other than SYSIN. This can be done by passing a parameter such as SYSIN=SASIN in the JCL where all SAS source statements are assigned to the DD name SASIN.

Keep in mind that future releases of SAS might require an updated version of SasAxsm. Therefore, when upgrading to a new SAS version, you should update the path for SAS on Windows and the library path for SAS on UNIX.

Using FastExport

To use FastExport, SAS writes a specialized script to a disk that the FastExport Utility reads. SAS might also log FastExport log lines to another disk file. SAS creates and deletes these files on your behalf, so no intervention is required. Sockets deliver the data from FastExport to SAS, so you do not need to do anything except install the SasAxsm access module that enables data transfer.

On Windows, when the FastExport Utility is active, a DOS window appears minimized as an icon on your toolbar. You can maximize the DOS window, but do not close it. After a FastExport operation is complete, SAS closes the window for you.

This example shows how to create a SAS data set that is a subset of a Teradata table that uses FastExport to transfer the data:

```
libname trlib teradata user=username password=userpwd;
data saslocal(keep=EMPID SALARY);
set trlib.employees(dbsliceparm=all);
run;
```

FastExport and Explicit SQL

FastExport is also supported for the explicit SQL feature of PROC SQL.

The following example shows how to create a SAS data set that is a subset of a Teradata table by using explicit SQL and FastExport to transfer the data:

```
proc sql;
connect to teradata as pro1 (user=username password=userpwd dbsliceparm=all);
create table saslocal as select * from connection to pro1
   (select EMPID, SALARY from employees);
quit;
```

FastExport for explicit SQL is a Teradata extension only, for optimizing Read operations, and is not covered in the threaded Read documentation.

Exceptions to Using FastExport

With the Teradata FastExport Utility and the SasAxsm module in place that SAS supplies, FastExport works automatically for all SAS steps that have threaded Reads enabled, except for one situation. FastExport does not handle single Access Module Processor (AMP) queries. In this case, SAS/ACCESS simply reverts to a normal single connection read. For information about FastExport and single AMP queries, see your Teradata documentation.

To determine whether FastExport worked, turn on SAS tracing in advance of the step that attempts to use FastExport. If you use FastExport, you receive this (English only) message, which is written to your SAS log:

```
sasiotra/tryottrm(): SELECT was processed with FastExport.
```

To turn on SAS tracing, run this statement:

```
options sastrace=',,,d' sastraceloc=saslog;
```

Threaded Reads with Partitioning WHERE Clauses

If FastExport is unavailable, threaded Reads use partitioning WHERE clauses. You can create your own partitioning WHERE clauses using the DBSLICE= option. Otherwise, SAS/ACCESS to Teradata attempts to generate them on your behalf. Like other SAS/ACCESS interfaces, this partitioning is based on the MOD function. To generate partitioning WHERE clauses, SAS/ACCESS to Teradata must locate a table column suitable for applying MOD. These types are eligible:

- BYTEINT
- SMALLINT
- INTEGER
- DATE

- DECIMAL (integral DECIMAL columns only)

A DECIMAL column is eligible only if the column definition restricts it to integer values. In other words, the DECIMAL column must be defined with a scale of zero.

If the table that you are reading contains more than one column of the above mentioned types, SAS/ACCESS to Teradata applies some nominal intelligence to select a best choice. Top priority is given to the primary index, if it is MOD-eligible. Otherwise, preference is given to any column that is defined as NOT NULL. Since this is an unsophisticated set of selection rules, you might want to supply your own partitioning using the DBSLICE= option.

To view your table's column definitions, use the Teradata SHOW TABLE statement.

Note: Partitioning WHERE clauses, either automatically generated or created by using DBSLICE=, are not supported on z/OS. Whether automatically generated or created by using DBSLICE=, partitioning WHERE clauses is not supported on z/OS and UNIX.

FastExport versus Partitioning WHERE Clauses

Partitioning WHERE clauses are innately less efficient than FastExport. The Teradata DBMS must process separate SQL statements that vary in the WHERE clause. In contrast, FastExport is optimal because only one SQL statement is transmitted to the Teradata DBMS. However, older editions of the Teradata DBMS place severe restrictions on the system-wide number of simultaneous FastExport operations that are allowed. Even with newer versions of Teradata, your database administrator might be concerned about large numbers of FastExport operations.

Threaded Reads with partitioning WHERE clauses also place higher workload on Teradata and might not be appropriate on a widespread basis. Both technologies expedite throughput between SAS and the Teradata DBMS, but should be used judiciously. For this reason, only SAS threaded applications are eligible for threaded Read by default. To enable more threaded Reads or to turn them off entirely, use the DBSLICEPARM= option.

Even when FastExport is available, you can force SAS/ACCESS to Teradata to generate partitioning WHERE clauses on your behalf. This is accomplished with the DBI argument to the DBSLICEPARM= option (DBSLICEPARM=DBI). This feature is available primarily to enable comparisons of these techniques. In general, you should use FastExport if it is available.

The explicit SQL feature of PROC SQL supports FastExport. Partitioning of WHERE clauses is not supported for explicit SQL.

Temporary Table Support for Teradata

Overview

For general information about this feature, see “Temporary Table Support for SAS/ACCESS ” on page 42.

Establishing a Temporary Table

When you specify CONNECTION=GLOBAL, you can reference a temporary table throughout a SAS session, in both DATA steps and procedures. Due to a Teradata limitation, FastLoad and FastExport do *not* support use of temporary tables at this time.

Teradata supports two types of temporary tables, global and volatile. With the use of global temporary tables, the rows are deleted after the connection is closed but the table definition itself remains. With volatile temporary tables, the table (and all rows) are dropped when the connection is closed.

When accessing a volatile table with a LIBNAME statement, it is recommended that you do *not* use these options:

- DATABASE= (as a LIBNAME option)
- SCHEMA= (as a data set or LIBNAME option)

If you use DATABASE= or SCHEMA=, you *must* specify DBMSTEMP=YES in the LIBNAME statement. This denotes that all tables accessed through it and all tables that it creates are volatile tables.

DBMSTEMP= also causes all table names to be not fully qualified for either SCHEMA= or DATABASE=. In this case, you should use the LIBNAME statement only to access tables—either permanent or volatile—within your default database or schema.

Terminating a Temporary Table

You can drop a temporary table at any time, or allow it to be implicitly dropped when the connection is terminated. Temporary tables do not persist beyond the scope of a single connection.

Examples

This example shows how to use a temporary table.

```
/* Set global connection for all tables. */
libname x teradata user=test pw=test server=boom connection=global;

/* Create global temporary table & store in the current database schema. */
proc sql;
   connect to teradata(user=test pw=test server=boom connection=global);
   execute (CREATE GLOBAL TEMPORARY TABLE temp1 (col1 INT )
            ON COMMIT PRESERVE ROWS) by teradata;
   execute (COMMIT WORK) by teradata;
quit;

/* Insert 1 row into the temporary table to specify the table. */
proc sql;
   connect to teradata(user=test pw=test server=boom connection=global);
   execute (INSERT INTO temp1 VALUES(1)) by teradata;
   execute (COMMIT WORK) by teradata;
quit;

/* Access the temporary table through the global libref. */
data work.new_temp1;
set x.temp1;
```

```
run;

/* Access the temporary table through the global connection. */
proc sql;
   connect to teradata (user=test pw=test server=boom connection=global);
   select * from connection to teradata (select * from temp1);
quit;

/* Drop the temporary table. */
proc sql;
   connect to teradata(user=prboni pw=prboni server=boom connection=global);
   execute (DROP TABLE temp1) by teradata;
   execute (COMMIT WORK) by teradata;
quit;
```

This is an example of how to use a volatile table.

```
/* Set global connection for all tables. */
libname x teradata user=test pw=test server=boom connection=global;

/* Create a volatile table. */
proc sql;
   connect to teradata(user=test pw=test server=boom connection=global);
   execute (CREATE VOLATILE TABLE temp1 (col1 INT)
            ON COMMIT PRESERVE ROWS) by teradata;
   execute (COMMIT WORK) by teradata;
quit;

/* Insert 1 row into the volatile table. */
proc sql;
   connect to teradata(user=test pw=test server=boom connection=global);
   execute (INSERT INTO temp1 VALUES(1)) by teradata;
   execute (COMMIT WORK) by teradata;
quit;

/* Access the temporary table through the global libref. */
data _null_;
   set x.temp1;
   put _all_;
run;

/* Access the volatile table through the global connection. */
proc sql;
   connect to teradata (user=test pw=test server=boom connection=global);
   select * from connection to teradata (select * from temp1);
quit;

/* Drop the connection & the volatile table is automatically dropped. */
libname x clear;

/* To confirm that it is gone, try to access it. */
libname x teradata user=test pw=test server=boom connection=global;

/* It is not there. */
proc print data=x.temp1;
run;
```

Passing SAS Functions to Teradata

SAS/ACCESS Interface to Teradata passes the following SAS functions to Teradata for processing. Where the Teradata function name differs from the SAS function name, the Teradata name appears in parentheses. For more information, see “Passing Functions to the DBMS Using PROC SQL” on page 46.

ABS	LOWCASE (LCASE)
ACOS	MAX
ARCOSH (ACOSH)	MIN
ARSINH (ASINH)	MINUTE
ASIN	MOD
ATAN	MONTH
ATAN2	SECOND
AVG	SIN
COALESCE	SINH
COS	SQRT
COSH	STD (STDDEV_SAMP)
COUNT	STRIP (TRIM)
DAY	SUBSTR
DTEXTDAY	SUM4
DTEXTMONTH	TAN
DTEXTYEAR	TANH
EXP	TIMEPART
HOUR	TRIM
INDEX (POSITION)	UPCASE
LENGTH (CHARACTER_LENGTH)	VAR (VAR_SAMP)
LOG	YEAR
LOG10	

ALL allows for SAS functions that have slightly different behavior from corresponding database functions that are passed down to the database. Only when SQL_FUNCTIONS=ALL can SAS/ACCESS Interface to Teradata also pass these SAS SQL functions to Teradata. Due to incompatibility in date and time functions between Teradata and SAS, Teradata might not process them correctly. Check your results to determine whether these functions are working as expected. For more information, see SQL_FUNCTIONS=LIBNAME Option on page 185.

DATE	SOUNDEX
DATETIME (current_timestamp)	TIME (current_time)
LEFT (TRIM	TODAY
LENGTH (CHARACTER_LENGTH)	TRIM

DATETIME, SOUNDEX, and TIME are not entirely compatible with the corresponding SAS functions. Also, for SOUNDEX, although Teradata always returns 4 characters, SAS might return more or less than 4 characters.

Passing Joins to Teradata

For a multiple libref join to pass to Teradata, all of these components of the LIBNAME statements must match exactly:

- user ID (USER=)
- password (PASSWORD=)
- account ID (ACCOUNT=)
- server (TDPID= or SERVER=)

You must specify the SCHEMA= LIBNAME option to fully qualify each table name in a join for each LIBNAME that you reference.

For more information about when and how SAS/ACCESS passes joins to the DBMS, see "Passing Joins to the DBMS" on page 47.

Maximizing Teradata Read Performance

Overview

A major objective of SAS/ACCESS when you are reading DBMS tables is to take advantage of the Teradata rate of data transfer. The DBINDEX=, SPOOL="SPOOL= LIBNAME Option", and PREFETCH= options can help you achieve optimal read performance. This section provides detailed information about PREFETCH as a LIBNAME option and PREFETCH as a global option.

Using the PreFetch Facility

Overview

PreFetch is a SAS/ACCESS Interface to Teradata facility that speeds up a SAS job by exploiting the parallel processing capability of Teradata. To obtain benefit from the facility, your SAS job must run more than once and have these characteristics:

- use SAS/ACCESS to query Teradata DBMS tables
- should *not* contain SAS statements that create, update, or delete Teradata DBMS tables
- run SAS code that changes infrequently or not at all.

In brief, the ideal job is a stable read-only SAS job.

Use of PreFetch is optional. To use the facility, you must explicitly enable it with the PREFETCH= LIBNAME option.

How PreFetch Works

When reading DBMS tables,SAS/ACCESS submits SQL statements on your behalf to Teradata. Each SQL statement that is submitted has an execution cost: the amount of time Teradata spends processing the statement before it returns the requested data to SAS/ACCESS.

When PreFetch is enabled the first time you run your SAS job, SAS/ACCESS identifies and selects statements with a high execution cost. SAS/ACCESS then stores (caches) the selected SQL statements to one or more Teradata macros that it creates.

On subsequent runs of the job, when PreFetch is enabled, SAS/ACCESS extracts statements from the cache and submits them to Teradata in advance. The rows that these SQL statements select are immediately available to SAS/ACCESS because Teradata *prefetches* them. Your SAS job runs faster because PreFetch reduces the wait for SQL statements with a high execution cost. However, PreFetch improves elapsed time only on subsequent runs of a SAS job. During the first run, SAS/ACCESS only creates the SQL cache and stores selected SQL statements; no prefetching is performed.

PreFetch Option Arguments

unique_storename

As mentioned, when PreFetch is enabled, SAS/ACCESS creates one or more Teradata macros to store the selected SQL statements that PreFetch caches. You can easily distinguish a PreFetch macro from other Teradata macros. The PreFetch Teradata macro contains a comment that is prefaced with this text:

```
"SAS/ACCESS PreFetch Cache"
```

The name that the PreFetch facility assigns for the macro is the value that you enter for the *unique_storename* argument. The *unique_storename* must be unique. Do not specify a name that exists in the Teradata DBMS already for a DBMS table, view, or macro. Also, do not enter a name that exists already in another SAS job that uses the Prefetch facility.

#sessions

This argument specifies how many cached SQL statements SAS/ACCESS submits in parallel to Teradata. In general, your SAS job completes faster if you increase the number of statements that Teradata works on in advance. However, a large number (too many sessions) can strain client and server resources. A valid value is 1 through 9. If you do not specify a value for this argument, the default is 3.

In addition to the specified number of sessions, SAS/ACCESS adds an additional session for submitting SQL statements that are not stored in the PreFetch cache. Therefore, if the default is 3, SAS/ACCESS actually opens up to four sessions on the Teradata server.

algorithm

This argument is present to handle future enhancements. Currently PreFetch only supports one algorithm, SEQUENTIAL.

When and Why Use PreFetch

If you have a read-only SAS job that runs frequently, this is an ideal candidate for PreFetch. An example is a daily job that extracts data from Teradata tables. To help you decide when to use PreFetch, consider these daily jobs:

- *Job 1*

  Reads and collects data from the Teradata DBMS.

- *Job 2*

  Contains a WHERE clause that reads in values from an external, variable data source. As a result, the SQL code that the job submits through a Teradata LIBNAME statement or through PROC SQL changes from run to run.

In these examples, Job 1 is an excellent candidate for the facility. In contrast, Job 2 is not. Using PreFetch with Job 2 does not return incorrect results, but can impose a performance penalty. PreFetch uses stored SQL statements. Therefore, Job 2 is not a good candidate because the SQL statements that the job generates with the WHERE clause change each time the job is run. Consequently, the SQL statements that the job generates never match the statements that are stored.

The impact of Prefetch on processing performance varies by SAS job. Some jobs improve elapsed time 5% or less; others improve elapsed time 25% or more.

Possible Unexpected Results

It is unlikely, but possible, to write a SAS job that delivers unexpected or incorrect results. This can occur if the job contains code that waits on some Teradata or system event before proceeding. For example, SAS code that pauses the SAS job until another user updates a given data item in a Teradata table. Another examples is SAS code that pauses the SAS job until a given time, such as 5:00 p.m. In both cases, PreFetch would generate SQL statements in advance. But, table results from these SQL statements would not reflect data changes that are made by the scheduled Teradata or system event.

PreFetch Processing of Unusual Conditions

PreFetch is designed to handle unusual conditions gracefully. Here are some of the unusual conditions that are included:

Condition: Your job contains SAS code that creates updates, or deletes Teradata tables.
: PreFetch is designed only for Read operations and is disabled when it encounters a nonread operation. The facility returns a performance benefit up to the point where the first nonread operation is encountered. After that, SAS/ACCESS disables the PreFetch facility and continues processing.

Condition: Your SQL cache name (*unique_storename* value) is identical to the name of a Teradata table.
: PreFetch issues a warning message. SAS/ACCESS disables the PreFetch facility and continues processing.

Condition: You change your SAS code for a job that has PreFetch enabled.
: PreFetch detects that the SQL statements for the job changed and deletes the cache. SAS/ACCESS disables Prefetch and continues processing. The next time you run the job, PreFetch creates a fresh cache.

Condition: Your SAS job encounters a PreFetch cache that was created by a different SAS job.
: PreFetch deletes the cache. SAS/ACCESS disables Prefetch and continues processing. The next time you run the job, PreFetch creates a fresh cache.

Condition: You remove the PreFetch option from an existing job.
: Prefetch is disabled. Even if the SQL cache (Teradata macro) still exists in your database, SAS/ACCESS ignores it.

Condition: You accidentally delete the SQL cache (the Teradata macro created by PreFetch) for a SAS job that has enabled PreFetch.
: SAS/ACCESS simply rebuilds the cache on the next run of the job. In subsequent job runs, PreFetch continues to enhance performance.

Using PreFetch as a LIBNAME Option

If you specify the PREFETCH= option in a LIBNAME statement, PreFetch applies the option to tables read by the libref.

If you have more than one LIBNAME in your SAS job, and you specify PREFETCH= for each LIBNAME, remember to make the SQL cache name for each LIBNAME unique.

This example applies PREFETCH= to one of two librefs. During the first job run, PreFetch stores SQL statements for tables referenced by the libref ONE in a Teradata macro named PF_STORE1 for reuse later.

```
libname one teradata user=testuser password=testpass
 prefetch='pf_store1';
libname two teradata user=larry password=riley;
```

This example applies PREFETCH= to multiple librefs. During the first job run, PreFetch stores SQL statements for tables that the EMP libref referenced to a Teradata macro named EMP_SAS_MACRO and SQL statements for tables that the SALE libref referenced to a Teradata macro named SALE_SAS_MACRO.

```
libname emp teradata user=testuser password=testpass
 prefetch='emp_sas_macro';
libname sale teradata user=larry password=riley
 prefetch='sale_sas_macro';
```

Using Prefetch as a Global Option

Unlike other Teradata LIBNAME options, you can also invoke PreFetch globally for a SAS job. To do this, place the OPTION DEBUG= statement in your SAS program before all LIBNAME statements and PROC SQL steps. If your job contains multiple LIBNAME statements, the global PreFetch invocation creates a uniquely named SQL cache name for each of the librefs.

Do not be confused by the DEBUG= option here. It is merely a mechanism to deliver the PreFetch capability globally. PreFetch is not for debugging; it is a supported feature of SAS/ACCESS Interface to Teradata.

In this example the first time you run the job with PreFetch enabled, the facility creates three Teradata macros: UNIQUE_MAC1, UNIQUE_MAC2, and UNIQUE_MAC3. In subsequent runs of the job, PreFetch extracts SQL statements from these Teradata macros, enhancing the job performance across all three librefs referenced by the job.

```
option debug="PREFETCH(unique_mac,2,SEQUENTIAL)";
libname one teradata user=kamdar password=ellis;
libname two teradata user=kamdar password=ellis
                          database=larry;
libname three teradata user=kamdar password=ellis
                            database=wayne;
proc print data=one.kamdar_goods;
run;
proc print data=two.larry_services;
run;
proc print data=three.wayne_miscellaneous;
run;
```

In this example PreFetch selects the algorithm, that is, the order of the SQL statements. (The OPTION DEBUG= statement must be the first statement in your SAS job.)

```
option debug='prefetch(pf_unique_sas,3)';
```

In this example the user specifies for PreFetch to use the SEQUENTIAL algorithm. (The OPTION DEBUG= statement must be the first statement in your SAS job.)

```
option debug='prefetch(sas_pf_store,3,sequential)';
```

Maximizing Teradata Load Performance

Overview

To significantly improve performance when loading data, SAS/ACCESS Interface to Teradata provides these facilities. These correspond to native Teradata utilities.

- FastLoad
- MultiLoad
- Multi-Statement

SAS/ACCESS also supports the Teradata Protocol Transporter application programming interface on page 804 (TPT API), which you can also use with these facilities.

Using FastLoad

FastLoad Supported Features and Restrictions

SAS/ACCESS Interface to Teradata supports a bulk-load capability called FastLoad that greatly accelerates insertion of data into empty Teradata tables. For general information about using FastLoad and error recovery, see the Teradata FastLoad documentation. SAS/ACCESS examples are available.

Note: Implementation of SAS/ACCESS FastLoad facility will change in a future release of SAS. So you might need to change SAS programming statements and options that you specify to enable this feature in the future.

The SAS/ACCESS FastLoad facility is similar to the native Teradata FastLoad Utility. They share these limitations:

- FastLoad can load only empty tables; it cannot append to a table that already contains data. If you attempt to use FastLoad when appending to a table that contains rows, the append step fails.
- Both the Teradata FastLoad Utility and the SAS/ACCESS FastLoad facility log data errors to tables. Error recovery can be difficult. To find the error that corresponds to the code that is stored in the error table, see the Teradata FastLoad documentation.
- FastLoad does not load duplicate rows (rows where all corresponding fields contain identical data) into a Teradata table. If your SAS data set contains duplicate rows, you can use the normal insert (load) process.

Starting FastLoad

If you do not specify FastLoad, your Teradata tables are loaded normally (slowly). To start FastLoad in the SAS/ACCESS interface, you can use one of these items:

- the BULKLOAD=YES data set option in a processing step that populates an empty Teradata table
- the BULKLOAD=YES LIBNAME option on the destination libref (the Teradata DBMS library where one or more intended tables are to be created and loaded)
- the FASTLOAD= alias for either of these options

FastLoad Data Set Options

Here are the data set options that you can use with the FastLoad facility.

- BL_LOG= specifies the names of error tables that are created when you use the SAS/ACCESS FastLoad facility. By default, FastLoad errors are logged in Teradata tables named SAS_FASTLOAD_ERRS1_*randnum* and SAS_FASTLOAD_ERRS2_*randnum*, where *randnum* is a randomly generated number. For example, if you specify **`BL_LOG=my_load_errors`**, errors are logged in tables *my_load_errors1* and *my_load_errors2*. If you specify **`BL_LOG=errtab`**, errors are logged in tables name *errtab1* and *errtab2*.

  Note: SAS/ACCESS automatically deletes the error tables if no errors are logged. If errors occur, the tables are retained and SAS/ACCESS issues a warning message that includes the names of the error tables.

- DBCOMMIT=*n* causes a Teradata “checkpoint” after each group of *n* rows is transmitted. Using checkpoints slows performance but provides known synchronization points if failure occurs during the loading process. Checkpoints are not used by default if you do not explicitly set DBCOMMIT= and BULKLOAD=YES. The Teradata alias for this option is CHECKPOINT=.

For details, see the data set options on page 207section of this document.

Using MultiLoad

MultiLoad Supported Features and Restrictions

SAS/ACCESS Interface to Teradata supports a bulk-load capability called MultiLoad that greatly accelerates insertion of data into Teradata tables. For general information about using MultiLoad with Teradata tables and for information about error recovery, see the Teradata MultiLoad documentation. SAS/ACCESS examples are available.

Unlike FastLoad, which only loads empty tables, MultiLoad loads both empty and existing Teradata tables. If you do not specify MultiLoad, your Teradata tables are loaded normally (inserts are sent one row at a time).

The SAS/ACCESS MultiLoad facility loads both empty and existing Teradata tables. SAS/ACCESS supports these features:

- You can load only one target table at a time.
- Only Insert operations are supported.

Because the SAS/ACCESS MultiLoad facility is similar to the native Teradata MultiLoad utility, they share a limitation: You must drop these items onto the target tables before the load:

- unique secondary indexes
- foreign key references
- join indexes

Both the Teradata MultiLoad utility and the SAS/ACCESS MultiLoad facility log data errors to tables. Error recovery can be difficult, but the ability to restart from the last checkpoint is possible. To find the error that corresponds to the code that is stored in the error table, see the Teradata MultiLoad documentation.

MultiLoad Setup

Here are the requirements for using the MultiLoad bulk-load capability in SAS.

- The native Teradata MultiLoad utility must be present on your system. If you do not have the Teradata MultiLoad utility and you want to use it with SAS, contact Teradata to obtain the utility.
- SAS must be able to locate the Teradata MultiLoad utility on your system.
- The Teradata MultiLoad utility must be able to locate the SASMlam access module and the SasMlne exit routine. They are supplied with SAS/ACCESS Interface to Teradata.
- SAS MultiLoad requires Teradata client TTU 8.2 or later.

If not already done during post-installation configuration, see the SAS configuration documentation for your system for information about how to configure SAS to work with MultiLoad.

MultiLoad Data Set Options

Call the SAS/ACCESS MultiLoad facility by specifying MULTILOAD=YES. See the MULTILOAD= data set option for detailed information and examples on loading data and recovering from errors during the load process.

Here are the data set options that are available for use with the MultiLoad facility. For detailed information about these options, see "Overview" on page 207.

- MBUFSIZE=
- ML_CHECKPOINT=
- ML_ERROR1= lets the user name the error table that MultiLoad uses for tracking errors from the acquisition phase. See the Teradata MultiLoad reference for more information about what is stored in this table. By default, the acquisition error table is named SAS_ML_ET_*randnum* where *randnum* is a random number. When restarting a failed MultiLoad job, you need to specify the same acquisition table from the earlier run so that the MultiLoad job can restart correctly. Note that the same log table, application error table, and work table must also be specified upon restarting, using ML_RESTART, ML_ERROR2, and ML_WORK data set options. ML_ERROR1 and ML_LOG are mutually exclusive and cannot be specified together.
- ML_ERROR2=
- ML_LOG= specifies a prefix for the temporary tables that the Teradata MultiLoad utility uses during the load process. The MultiLoad utility uses a log table, two error tables, and a work table when loading data to the target table. These tables are named by default as SAS_ML_RS_*randnum*, SAS_ML_ET_*randnum*, SAS_ML_UT_*randnum*, and SAS_ML_WT_*randnum* where *randnum* is a randomly generated number. ML_LOG= is used to override the default names used. For example, if you specify **`ML_LOG=MY_LOAD`** the log table is named **`MY_LOAD_RS`**. Errors are logged in tables **`MY_LOAD_ET`** and **`MY_LOAD_UT`**. The work table is named **`MY_LOAD_WT`**.
- ML_RESTART= lets the user name the log table that MultiLoad uses for tracking checkpoint information. By default, the log table is named SAS_ML_RS_*randnum* where *randnum* is a random number. When restarting a failed MultiLoad job, you need to specify the same log table from the earlier run so that the MultiLoad job can restart correctly. Note that the same error tables and work table must also be specified upon restarting the job, using ML_ERROR1, ML_ERROR2, and ML_WORK data set options. ML_RESTART and ML_LOG are mutually exclusive and cannot be specified together.

- ML_WORK= lets the user name the work table that MultiLoad uses for loading the target table. See the Teradata MultiLoad reference for more information about what is stored in this table. By default, the work table is named SAS_ML_WT_*randnum* where *randnum* is a random number. When restarting a failed MultiLoad job, you need to specify the same work table from the earlier run so that the MultiLoad job can restart correctly. Note that the same log table, acquisition error table and application error table must also be specified upon restarting the job using ML_RESTART, ML_ERROR1, and ML_ERROR2 data set options. ML_WORK and ML_LOG are mutually exclusive and cannot be specified together.
- SLEEP= specifies the number of minutes that MultiLoad waits before it retries a logon operation when the maximum number of utilities are already running on the Teradata database. The default value is 6. SLEEP= functions very much like the SLEEP run-time option of the native Teradata MultiLoad utility.
- TENACITY= specifies the number of hours that MultiLoad tries to log on when the maximum number of utilities are already running on the Teradata database. The default value is 4. TENACITY= functions very much like the TENACITY run-time option of the native Teradata MultiLoad utility.

Be aware that these options are disabled while you are using the SAS/ACCESS MultiLoad facility.

- The DBCOMMIT= LIBNAME and data set options are disabled because DBCOMMIT= functions very differently from CHECKPOINT of the native Teradata MultiLoad utility.
- The ERRLIMIT= data set option is disabled because the number of errors is not known until all records have been sent to MultiLoad. The default value of ERRLIMIT=1 is not honored.

To see whether threaded Reads are actually generated, turn on SAS tracing by setting OPTIONS SASTRACE=",,,d" in your program.

Using the TPT API

TPT API Supported Features and Restrictions

SAS/ACCESS Interface to Teradata supports the TPT API for loading data. The TPT API provides a consistent interface for Fastload, MultiLoad, and Multi-Statement insert. TPT API documentation refers to Fastload as the *Load operator*, MultiLoad as the *Update operator*, and Multi-Statement insert as the *Stream operator*. SAS supports all three load methods and can restart loading from checkpoints when you use the TPT API with any of them.

TPT API Setup

Here are the requirements for using the TPT API in SAS for loading SAS.

- Loading data from SAS to Teradata using the TPT API requires Teradata client TTU 8.2 or later. Verify that you have applied all of the latest Teradata eFixes.
- This feature is supported only on platforms for which Teradata provides the TPT API.
- The native TPT API infrastructure must be present on your system. Contact Teradata if you do not already have it but want to use it with SAS.
- These Teradata privileges are needed to load a Teradata table using the TPT API. For more information, see the Teradata TPT API documentation.

- CREATE TABLE
- DROP TABLE
- CREATE MACRO (Multi-Statement Stream operator)
- DROP MACRO (Multi-Statement Stream operator)
- INSERT
- DELETE
- SELECT

The SAS configuration document for your system contains information about how to configure SAS to work with the TPT API. However, those steps might already have been completed as part of the post-installation configuration process for your site.

TPT API LIBNAME Options

These LIBNAME options are common to all three supported load methods:

- LOGDB=
- SLEEP=
- TENACITY=
- TPT=

If SAS cannot use the TPT API, it reverts to using Fastload, MultiLoad, or Multi-Statement insert, depending on which method of loading was requested without generating any errors.

TPT API Data Set Options

These data set options are common to all three supported load methods:

- SLEEP=
- TENACITY=
- TPT=
- TPT_CHECKPOINT_DATA=
- TPT_DATA_ENCRYPTION=
- TPT_LOG_TABLE=
- TPT_MAX_SESSIONS=
- TPT_MIN_SESSIONS=
- TPT_RESTART=
- TPT_TRACE_LEVEL=
- TPT_TRACE_LEVEL_INF=
- TPT_TRACE_OUTPUT=

You can use the DBCOMMIT= LIBNAME option and the CHECKPOINT= data set option to specify checkpoint frequency with TPT for Fastload, Multiload, and Multi-Statement insert. These options are disabled only for the non-TPT MultiLoad interface and are not relevant with TPT FastExport.

TPT API FastLoad Supported Features and Restrictions

SAS/ACCESS Interface to Teradata supports the TPT API for FastLoad, also known as the *Load operator*, SAS/ACCESS works by interfacing with the Load operator through the TPT API, which in turn uses the Teradata Fastload protocol for loading data. See your Teradata documentation for more information about the Load operator.

This is the default FastLoad method. If SAS cannot find the Teradata modules that are required for the TPT API or TPT=NO, SAS/ACCESS uses the old method of Fastload. SAS/ACCESS can restart Fastload from checkpoints when FastLoad uses the TPT API. The SAS/ACCESS FastLoad facility using the TPT API is similar to the native Teradata FastLoad utility. They share these limitations.

- FastLoad can load only empty tables. It cannot append to a table that already contains data. If you try to use FastLoad when appending to a table that contains rows, the append step fails.
- Data errors are logged in Teradata tables. Error recovery can be difficult if you do not TPT_CHECKPOINT_DATA= to enable restart from the last checkpoint. To find the error that corresponds to the code that is stored in the error table, see your Teradata documentation. You can restart a failed job for the last checkpoint by following the instructions in the SAS error log.
- FastLoad does not load duplicate rows (those where all corresponding fields contain identical data) into a Teradata table. If your SAS data set contains duplicate rows, you can use other load methods.

Starting FastLoad with the TPT API

See the SAS configuration document for instructions on setting up the environment so that SAS can find the TPT API modules.

You can use one of these options to start FastLoad in the SAS/ACCESS interface using the TPT API:

- the TPT=YES data set option in a processing step that populates an empty Teradata table
- the TPT=YES LIBNAME option on the destination libref (the Teradata DBMS library where one or more tables are to be created and loaded)

FastLoad with TPT API Data Set Options

These data set options are specific to FastLoad using the TPT API:

- TPT_BUFFER_SIZE=
- TPT_ERROR_TABLE_1=
- TPT_ERROR_TABLE_2=

TPT API MultiLoad Supported Features and Restrictions

SAS/ACCESS Interface to Teradata supports the TPT API for MultiLoad, also known as the *Update operator*. SAS/ACCESS works by interfacing with the Update operator through the TPT API. This API then uses the Teradata Multiload protocol for loading data. See your Teradata documentation for more information about the Update operator.

This is the default MultiLoad method. If SAS cannot find the Teradata modules that are required for the TPT API or TPT=NO, SAS/ACCESS uses the old method of MultiLoad. SAS/ACCESS can restart Multiload from checkpoints when MultiLoad uses the TPT API.

The SAS/ACCESS MultiLoad facility loads both empty and existing Teradata tables. SAS/ACCESS supports only Insert operations and loading only one target table at a time.

The SAS/ACCESS MultLoad facility using the TPT API is similar to the native Teradata MultiLoad utility. A common limitation that they share is that you must drop these items onto target tables before the load:

- unique secondary indexes
- foreign key references
- join indexes

Errors are logged to Teradata tables. Error recovery can be difficult if you do not set TPT_CHECKPOINT_DATA= to enable restart from the last checkpoint. To find the error that corresponds to the code that is stored in the error table, see your Teradata documentation. You can restart a failed job for the last checkpoint by following the instructions in the SAS error log.

Starting MultiLoad with the TPT API

See the SAS configuration document for instructions on setting up the environment so that SAS can find the TPT API modules.

You can use one of these options to start MultiLoad in the SAS/ACCESS interface using the TPT API:

- the TPT=YES data set option in a processing step that populates an empty Teradata table
- the TPT=YES LIBNAME option on the destination libref (the Teradata DBMS library where one or more tables are to be created and loaded)

MultiLoad with TPT API LIBNAME Options

The LOGDB= LIBNAME option is specific to MultiLoad using the TPT API.

MultiLoad with TPT API Data Set Options

These data set options are specific to MultiLoad using the TPT API:

- TPT_BUFFER_SIZE=
- TPT_ERROR_TABLE_1=
- TPT_ERROR_TABLE_2=

TPT API Multi-Statement Insert Supported Features and Restrictions

SAS/ACCESS Interface to Teradata supports the TPT API for Multi-Statement insert, also known as the *Stream operator*. SAS/ACCESS works by interfacing with the Stream operator through the TPT API, which in turn uses the Teradata Multi-Statement insert (TPump) protocol for loading data. See your Teradata documentation for more information about the Stream operator.

This is the default Multi-Statement insert method. If SAS cannot find the Teradata modules that are required for the TPT API or TPT=NO, SAS/ACCESS uses the old method of Multi-Statement insert. SAS/ACCESS can restart Multi-Statement insert from checkpoints when Multi-Statement insert uses the TPT API.

The SAS/ACCESS Multi-Statement insert facility loads both empty and existing Teradata tables. SAS/ACCESS supports only Insert operations and loading only one target table at time.

Errors are logged to Teradata tables. Error recovery can be difficult if you do not set TPT_CHECKPOINT_DATA= to enable restart from the last checkpoint. To find the error that corresponds to the code that is stored in the error table, see your Teradata documentation. You can restart a failed job for the last checkpoint by following the instructions on the SAS error log.

Starting Multi-Statement Insert with the TPT API

See the SAS configuration document for instructions on setting up the environment so that SAS can find the TPT API modules.

You can use one of these options to start Multi-Statement in the SAS/ACCESS interface using the TPT API:

- the TPT=YES data set option in a processing step that populates an empty Teradata table
- the TPT=YES LIBNAME option on the destination libref (the Teradata DBMS library where one or more tables are to be created and loaded)

Multi-Statement Insert with TPT API Data Set Options

These data set options are specific to Multi-Statement insert using the TPT API.

- TPT_PACK=
- TPT_PACKMAXIMUM=

Examples

This example starts the FastLoad facility.

```
libname fload teradata user=testuser password=testpass;
data fload.nffloat(bulkload=yes);
     do x=1 to 1000000;
        output;
     end;
run;
```

This next example uses FastLoad to append SAS data to an empty Teradata table and specifies the BL_LOG= option to name the error tables **`Append_Err1`** and **`Append_Err2`**. In practice, applications typically append many rows.

```
/* Create the empty Teradata table. */
proc sql;
     connect to teradata as tera(user=testuser password=testpass);
     execute (create table performers
             (userid int, salary decimal(10,2), job_desc char(50)))
              by tera;
     execute (commit) by tera;
quit;

/* Create the SAS data to load. */
data local;
     input userid 5. salary 9. job_desc $50.;
           datalines;
             0433 35993.00 grounds keeper
             4432 44339.92 code groomer
             3288 59000.00 manager
             ;
```

```
/* Append the SAS data & name the Teradata error tables. */
libname tera teradata user=testuser password=testpass;

proc append data=local base=tera.performers
     (bulkload=yes bl_log=append_err);
run;
```

This example starts the MultiLoad facility.

```
libname trlib teradata user=testuser pw=testpass server=dbc;

/* Use MultiLoad to load a table with 2000 rows. */
data trlib.mlfloat(MultiLoad=yes);
     do x=1 to 2000;
        output;
     end;
run;

/* Append another 1000 rows. */
data work.testdata;
  do x=2001 to 3000;
     output;
  end;
run;

/* Append the SAS data to the Teradata table. */
proc append data=work.testdata base=trlib.mlfload
     (MultiLoad=yes);
run;
```

This example loads data using TPT FastLoad.

```
/* Check the SAS log for this message to verify that the TPT API was used.
NOTE:  Teradata connection:  TPT Fastload has inserted 100 rows.
*/
data trlib.load(TPT=YES FASTLOAD=YES);
        do x=1 to 1000;
                output;
        end;
run;
```

This example restarts a MultiLoad step that recorded checkpoints and failed after loading 2000 rows of data.

```
proc append data=trlib.load(TPT=YES MULTILOAD=YES
    TPT_RESTART=YES TPT_CHECKPOINT_DATA=2000)
data=work.inputdata(FIRSTOBS=2001);
run;
```

Teradata Processing Tips for SAS Users

Reading from and Inserting to the Same Teradata Table

If you use SAS/ACCESS to read rows from a Teradata table and then attempt to insert these rows into the same table, you can hang (suspend) your SAS session.

Here is what happens:

- A SAS/ACCESS connection requests a standard Teradata READ lock for the Read operation.
- A SAS/ACCESS connection then requests a standard Teradata WRITE lock for the Insert operation.
- The WRITE lock request suspends because the read connection already holds a READ lock on the table. Consequently, your SAS session stops responding (is suspended).

Here is what happens in the next example:

- SAS/ACCESS creates a read connection to Teradata to fetch the rows selected (select *) from TRA.SAMETABLE, requiring a standard Teradata READ lock; Teradata issues a READ lock.
- SAS/ACCESS creates an insert connection to Teradata to insert the rows into TRA.SAMETABLE, requiring a standard Teradata WRITE lock. But the WRITE lock request suspends because the table is locked already by the READ lock.
- Your SAS/ACCESS session stops responding.

```
libname tra teradata user=testuser password=testpass;
proc sql;
insert into tra.sametable
   select * from tra.sametable;
```

To avoid this situation, use the SAS/ACCESS locking options. For details, see “Locking in the Teradata Interface” on page 814.

Using a BY Clause to Order Query Results

SAS/ACCESS returns table results from a query in random order because Teradata returns the rows to SAS/ACCESS randomly. In contrast, traditional SAS processing returns SAS data set observations in the same order during every run of your job. If maintaining row order is important, you should add a BY clause to your SAS statements. A BY clause ensures consistent ordering of the table results from Teradata.

In this example, the Teradata ORD table has NAME and NUMBER columns. The PROC PRINT statements illustrate consistent and inconsistent ordering when it displays ORD table rows.

```
libname prt teradata user=testuser password=testpass;

  proc print data=prt.ORD;
  var name number;
  run;
```

If this statement is run several times, it yields inconsistent ordering, meaning that ORD rows are likely to be arranged differently each time. This happens because SAS/ACCESS displays the rows in the order in which Teradata returns them—that is, randomly.

```
proc print data=prt.ORD;
var name number;
by name;
run;
```

This statement achieves more consistent ordering because it orders PROC PRINT output by the NAME value. However, on successive runs of the statement, rows display of rows with a different number and an identical name can vary, as shown here.

Output 29.1 *PROC PRINT Display 1*

```
Rita Calvin 2222
Rita Calvin 199
```

Output 29.2 *PROC PRINT Display 2*

```
Rita Calvin 199
Rita Calvin 2222
```

```
proc print data=prt.ORD;
var name number;
by name number;
run;
```

The above statement always yields identical ordering because every column is specified in the BY clause. So your PROC PRINT output always looks the same.

Using TIME and TIMESTAMP

This example creates a Teradata table and assigns the SAS TIME8. format to the TRXTIME0 column. Teradata creates the TRXTIME0 column as the equivalent Teradata data type, TIME(0), with the value of 12:30:55.

```
libname mylib teradata user=testuser password=testpass;

data mylib.trxtimes;
   format trxtime0 time8.;
   trxtime0 = '12:30:55't;
run;
```

This example creates a Teradata column that specifies very precise time values. The format TIME(5) is specified for the TRXTIME5 column. When SAS reads this column, it assigns the equivalent SAS format TIME14.5.

```
libname mylib teradata user=testuser password=testpass;

proc sql noerrorstop;
   connect to teradata (user=testuser password=testpass);
   execute (create table trxtimes (trxtime5 time(5)
      )) by teradata;
```

```
   execute (commit) by teradata;
   execute (insert into trxtimes
            values (cast('12:12:12' as time(5))
      )) by teradata;
   execute (commit) by teradata;
quit;

/* You can print the value that is read with SAS/ACCESS. */
proc print data =mylib.trxtimes;
run;
```

SAS might not preserve more than four digits of fractional precision for Teradata TIMESTAMP.

This next example creates a Teradata table and specifies a simple timestamp column with no digits of precision. Teradata stores the value 2000-01-01 00:00:00. SAS assigns the default format DATETIME19. to the TRSTAMP0 column generating the corresponding SAS value of 01JAN2000:00:00:00.

```
proc sql noerrorstop;
   connect to teradata (user=testuser password=testpass);
   execute (create table stamps (tstamp0 timestamp(0)
      )) by teradata;
   execute (commit) by teradata;
   execute (insert into stamps
            values (cast('2000-01-01 00:00:00' as
                    timestamp(0))
      )) by teradata;
   execute (commit) by teradata;
quit;
```

This example creates a Teradata table and assigns the SAS format DATETIME23.3 to the TSTAMP3 column, generating the value 13APR1961:12:30:55.123. Teradata creates the TSTAMP3 column as the equivalent data type TIMESTAMP(3) with the value 1961-04-13 12:30:55.123.

```
libname mylib teradata user=testuser password=testpass;

data mylib.stamps;
format tstamp3 datetime23.3;
tstamp3 = '13apr1961:12:30:55.123'dt;
run;
```

This next example illustrates how the SAS engine passes the literal value for TIMESTAMP in a WHERE statement to Teradata for processing. Note that the value is passed without being rounded or truncated so that Teradata can handle the rounding or truncation during processing. This example would also work in a DATA step.

```
proc sql ;
  select * from trlib.flytime where col1 = '22Aug1995 12:30:00.557'dt ;
  quit;
```

In SAS 8, the Teradata interface did not create TIME and TIMESTAMP data types. Instead, the interface generated FLOAT values for SAS times and dates. This example shows how to format a column that contains a FLOAT representation of a SAS datetime into a readable SAS datetime.

```
libname mylib teradata user=testuser password=testpass;
```

```
proc print data=mylib.stampv80;
format stamp080 datetime25.0;
run;
```

Here, the old Teradata table STAMPV80 contains the FLOAT column, STAMP080, which stores SAS datetime values. The FORMAT statement displays the FLOAT as a SAS datetime value.

Replacing PROC SORT with a BY Clause

In general, PROC SORT steps are not useful to output a Teradata table. In traditional SAS processing, PROC SORT is used to order observations in a SAS data set. Subsequent SAS steps that use the sorted data set receive and process the observations in the sorted order. Teradata does not store output rows in the sorted order. Therefore, do not sort rows with PROC SORT if the destination sorted file is a Teradata table.

The following example illustrates a PROC SORT statement found in typical SAS processing. You cannot use this statement in SAS/ACCESS Interface to Teradata.

```
libname sortprt '.';
proc sort data=sortprt.salaries;
by income;
proc print data=sortprt.salaries;
```

This example removes the PROC SORT statement shown in the previous example. It instead uses a BY clause with a VAR clause with PROC PRINT. The BY clause returns Teradata rows ordered by the INCOME column.

```
libname sortprt teradata user=testuser password=testpass;
proc print data=sortprt.salaries;
var income;
by income;
```

Reducing Workload on Teradata By Sampling

The OBS= option triggers SAS/ACCESS to add a SAMPLE clause to generated SQL. In this example, 10 rows are printed from dbc.ChildrenX:

```
Libname tra teradata user=sasdxs pass=****** database=dbc;
Proc print data=tra.ChildrenX (obs=10);
run;
```

The SQL passed to Teradata is:

```
SELECT "Child","Parent" FROM "ChildrenX"  SAMPLE 10
```

Especially against large Teradata tables, small values for OBS= reduce workload and spool space consumption on Teradata and your queries complete much sooner. See the SAMPLE clause in your Teradata documentation for more information.

Locking in the Teradata Interface

Overview

The following LIBNAME and data set options let you control how the Teradata interface handles locking. For general information about an option, see "LIBNAME Options for Relational Databases" on page 99.

Use SAS/ACCESS locking options only when Teradata standard locking is undesirable. For tips on using these options, see "Understanding SAS/ACCESS Locking Options" on page 815 and "When to Use SAS/ACCESS Locking Options" on page 816. Teradata examples are available.

READ_LOCK_TYPE= TABLE | VIEW

UPDATE_LOCK_TYPE= TABLE | VIEW

READ_MODE_WAIT= YES | NO

UPDATE_MODE_WAIT= YES | NO

READ_ISOLATION_LEVEL= ACCESS | READ | WRITE
Here are the valid values for this option.

Table 29.3 *Read Isolation Levels for Teradata*

Isolation Level	Definition
ACCESS	Obtains an ACCESS lock by ignoring other users' ACCESS, READ, and WRITE locks. Permits other users to obtain a lock on the table or view. Can return inconsistent or unusual results.
READ	Obtains a READ lock if no other user holds a WRITE or EXCLUSIVE lock. Does not prevent other users from reading the object. Specify this isolation level whenever possible, it is usually adequate for most SAS/ACCESS processing.
WRITE	Obtains a WRITE lock on the table or view if no other user has a READ, WRITE, or EXCLUSIVE lock on the resource. You cannot explicitly release a WRITE lock. It is released only when the table is closed. Prevents other users from acquiring any lock but ACCESS. This is unnecessarily restrictive, because it locks the entire table until the Read operation is finished.

UPDATE_ISOLATION_LEVEL= ACCESS | READ | WRITE
The valid values for this option, ACCESS, READ, and WRITE, are defined in the following table.

Table 29.4 *Update Isolation Levels for Teradata*

Isolation Level	Definition
ACCESS	Obtains an ACCESS lock by ignoring other users' ACCESS, READ, and WRITE locks. Avoids a potential deadlock but can cause data corruption if another user is updating the same data.
READ	Obtains a READ lock if no other user holds a WRITE or EXCLUSIVE lock. Prevents other users from being granted a WRITE or EXCLUSIVE lock. Locks the entire table or view, allowing other users to acquire READ locks. Can lead to deadlock situations.
WRITE	Obtains a WRITE lock on the table or view if no other user has a READ, WRITE, or EXCLUSIVE lock on the resource. You cannot explicitly release a WRITE lock. It is released only when the table is closed. Prevents other users from acquiring any lock but ACCESS. Prevents all users, except those with ACCESS locks, from accessing the table. Prevents the possibility of a deadlock, but limits concurrent use of the table.

These locking options cause the LIBNAME engine to transmit a locking request to the DBMS; Teradata performs all data-locking. If you correctly specify a set of SAS/ACCESS read or update locking options, SAS/ACCESS generates locking modifiers that override the Teradata standard locking.

If you specify an incomplete set of locking options, SAS/ACCESS returns an error message. If you do not use SAS/ACCESS locking options, Teradata lock defaults are in effect. For a complete description of Teradata locking, see the LOCKING statement in your Teradata SQL reference documentation.

Understanding SAS/ACCESS Locking Options

SAS/ACCESS locking options modify Teradata standard locking. Teradata usually locks at the row level; SAS/ACCESS lock options lock at the table or view level. The change in the scope of the lock from row to table affects concurrent access to DBMS objects. Specifically, READ and WRITE table locks increase the time that other users must wait to access the table and can decrease overall system performance. These measures help minimize these negative effects.

- Apply READ or WRITE locks *only* when you must apply special locking on Teradata tables.

  SAS/ACCESS locking options can be appropriate for special situations, as described in "When to Use SAS/ACCESS Locking Options" on page 816. If SAS/ACCESS locking options do not meet your specialized needs, you can use additional Teradata locking features using views. See CREATE VIEW in your Teradata SQL reference documentation for details.

- Limit the span of the locks by using data set locking options instead of LIBNAME locking options whenever possible. (LIBNAME options affect all tables that you open that your libref references. Data set options apply only to the specified table.)

If you specify these read locking options, SAS/ACCESS generates and submits to Teradata locking modifiers that contain the values that you specify for the three read lock options:

- READ_ISOLATION_LEVEL= specifies the level of isolation from other table users that is required during SAS/ACCESS Read operations.
- READ_LOCK_TYPE= specifies and changes the scope of the Teradata lock during SAS/ACCESS Read operations.
- READ_MODE_WAIT= specifies during SAS/ACCESS Read operations whether Teradata should wait to acquire a lock or fail your request when the DBMS resource is locked by a different user.

If you specify these update lock options, SAS/ACCESS generates and submits to Teradata locking modifiers that contain the values that you specify for the three update lock options:

- UPDATE_ISOLATION_LEVEL= specifies the level of isolation from other table users that is required as SAS/ACCESS reads Teradata rows in preparation for updating the rows.
- UPDATE_LOCK_TYPE= specifies and changes the scope of the Teradata lock during SAS/ACCESS Update operations.
- UPDATE_MODE_WAIT= specifies during SAS/ACCESS Update operations whether Teradata should wait to acquire a lock or fail your request when the DBMS resource is locked by a different user.

When to Use SAS/ACCESS Locking Options

This section describes situations that might require SAS/ACCESS lock options instead of the standard locking that Teradata provides.

- Use SAS/ACCESS locking options to reduce the isolation level for a Read operation.

  When you lock a table using a READ option, you can lock out both yourself and other users from updating or inserting into the table. Conversely, when other users update or insert into the table, they can lock you out from reading the table. In this situation, you want to reduce the isolation level during a Read operation. To do this, you specify these read SAS/ACCESS lock options and values.

  - READ_ISOLATION_LEVEL=ACCESS
  - READ_LOCK_TYPE=TABLE
  - READ_MODE_WAIT=YES

  One of these situations can result from the options and settings in this situation:

  - Specify ACCESS locking, eliminating a lock out of yourself and other users. Because ACCESS can return inconsistent results to a table reader, specify ACCESS only if you are casually browsing data, not if you require precise data.
  - Change the scope of the lock from row-level to the entire table.
  - Request that Teradata wait if it attempts to secure your lock and finds the resource already locked.

- Use SAS/ACCESS lock options to avoid contention.

  When you read or update a table, contention can occur: the DBMS is waiting for other users to release their locks on the table that you want to access. This contention suspends your SAS/ACCESS session. In this situation, to avoid contention during a Read operation, you specify these SAS/ACCESS Read lock options and values.

- READ_ISOLATION_LEVEL=READ
- READ_LOCK_TYPE=TABLE
- READ_MODE_WAIT=NO

One of these situations can result from the options and settings in this situation.

- Specify a READ lock.
- Change the scope of the lock. Because SAS/ACCESS does not support row locking when you obtain the lock requested, you lock the entire table until your Read operation finishes.
- Tell SAS/ACCESS to fail the job step if Teradata cannot immediately obtain the READ lock.

Examples

Setting the Isolation Level to ACCESS for Teradata Tables

```
  /* This generates a quick survey of unusual customer purchases. */
libname cust teradata user=testuser password=testpass
               READ_ISOLATION_LEVEL=ACCESS
               READ_LOCK_TYPE=TABLE
               READ_MODE_WAIT=YES
               CONNECTION=UNIQUE;

proc print data=cust.purchases(where= (bill<2));
run;

data local;
 set cust.purchases (where= (quantity>1000));
run;
```

Here is what SAS/ACCESS does in the above example.

- Connects to the Teradata DBMS and specifies the three SAS/ACCESS LIBNAME read lock options.
- Opens the PURCHASES table and obtains an ACCESS lock if a different user does not hold an EXCLUSIVE lock on the table.
- Reads and displays table rows with a value less than 2 in the BILL column.
- Closes the PURCHASES table and releases the ACCESS lock.
- Opens the PURCHASES table again and obtains an ACCESS lock if a different user does not hold an EXCLUSIVE lock on the table.
- Reads table rows with a value greater than 1000 in the QUANTITY column.
- Closes the PURCHASES table and releases the ACCESS lock.

Setting Isolation Level to WRITE to Update a Teradata Table

```
  /* This updates the critical Rebate row. */
libname cust teradata user=testuser password=testpass;

proc sql;
```

```
  update cust.purchases(UPDATE_ISOLATION_LEVEL=WRITE
        UPDATE_MODE_WAIT=YES
        UPDATE_LOCK_TYPE=TABLE)
  set rebate=10 where bill>100;
quit;
```

In this example here is what SAS/ACCESS does:

- Connects to the Teradata DBMS and specifies the three SAS/ACCESS data set update lock options.
- Opens the PURCHASES table and obtains a WRITE lock if a different user does not hold a READ, WRITE, or EXCLUSIVE lock on the table.
- Updates table rows with BILL greater than 100 and sets the REBATE column to 10.
- Closes the PURCHASES table and releases the WRITE lock.

Preventing a Hung SAS Session When Reading and Inserting to the Same Table

```
  /* SAS/ACCESS lock options prevent the session hang */
  /* that occurs when reading & inserting into the same table. */
libname tra teradata user=testuser password=testpass  connection=unique;

proc sql;
insert into tra.sametable
   select * from tra.sametable(read_isolation_level=access
          read_mode_wait=yes
          read_lock_type=table);
```

Here is what SAS/ACCESS does in the above example:

- Creates a read connection to fetch the rows selected (SELECT *) from TRA.SAMETABLE and specifies an ACCESS lock (READ_ISOLATION_LEVEL=ACCESS). Teradata grants the ACCESS lock.
- Creates an insert connection to Teradata to process the Insert operation to TRA.SAMETABLE. Because the ACCESS lock that is already on the table permits access to the table, Teradata grants a WRITE lock.
- Performs the Insert operation without hanging (suspending) your SAS session.

Naming Conventions for Teradata

Teradata Conventions

For general information about this feature, see “SAS Names and Support for DBMS Names” on page 13.

You can use these conventions to name such Teradata objects as include tables, views, columns, indexes, and macros.

- A name must be from 1 to 30 characters long.
- A name must begin with a letter unless you enclose it in double quotation marks.

- A name can contain letters (A to Z), numbers from 0 to 9, underscore (_), dollar sign ($), and the number or pound sign (#). A name in double quotation marks can contain any characters except double quotation marks.
- A name, even when enclosed in double quotation marks, is not case sensitive. For example, **`CUSTOMER`** and **`Customer`** are the same.
- A name cannot be a Teradata reserved word.
- Because the name must be unique between objects, a view and table in the same database cannot have an identical name.

SAS Naming Conventions

Use these conventions when naming a SAS object:

- A name must be from 1 to 32 characters long.
- A name must begin with a letter (A to Z) or an underscore (_).
- A name can contain letters (A to Z), numbers from 0 to 9, and an underscore (_).
- Names are not case sensitive. For example, **`CUSTOMER`** and **`Customer`** are the same.
- A name cannot be enclosed in double quotation marks.
- A name need not be unique between object types.

Naming Objects to Meet Teradata and SAS Conventions

To easily share objects between SAS and the DBMS, create names that meet both SAS and Teradata naming conventions:

- Start with a letter.
- Include only letters, digits, and underscores.
- Use a length of 1 to 30 characters.

Accessing Teradata Objects That Do Not Meet SAS Naming Conventions

Overview

The following SAS/ACCESS code examples can help you access Teradata objects (existing Teradata DBMS tables and columns) that have names that do not follow SAS naming conventions.

Example 1: Unusual Teradata Table Name

```
libname unusual teradata user=testuser password=testpass;
proc sql dquote=ansi;
  create view myview as
  select * from unusual."More names";
proc print data=myview;run;
```

Example 2: Unusual Teradata Column Names

SAS/ACCESS automatically converts Teradata column names that are not valid for SAS, mapping such characters to underscores. It also appends numeric suffixes to identical names to ensure that column names are unique.

```
create table unusual_names( Name$ char(20), Name# char(20),
                             "Other strange name" char(20))
```

In this example SAS/ACCESS converts the spaces found in the Teradata column name, OTHER STRANGE NAME, to Other_strange_name. After the automatic conversion, SAS programs can then reference the table as usual.

```
libname unusual teradata user=testuser password=testpass;
proc print data=unusual.unusual_names; run;
```

Output 29.3 *PROC PRINT Display*

```
        Name_          Name_0         Other_strange_name
```

Data Types for Teradata

Overview

Every column in a table has a name and data type. The data type tells Teradata how much physical storage to set aside for the column, as well as the form in which to store the data. This section includes information about Teradata data types, null values, and data conversions.

SAS/ACCESS 9 does not support these Teradata data types: GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC.

Binary String Data

BYTE (*n*)
: specifies a fixed-length column of length *n* for binary string data. The maximum for *n* is 64,000.

VARBYTE (*n*)
: specifies a varying-length column of length *n* for binary string data. The maximum for *n* is 64,000.

Character String Data

CHAR (*n*)
: specifies a fixed-length column of length *n* for character string data. The maximum for *n* is 64,000.

VARCHAR (*n*)
: specifies a varying-length column of length *n* for character string data. The maximum for *n* is 64,000. VARCHAR is also known as CHARACTER VARYING.

LONG VARCHAR
specifies a varying-length column, of the maximum length, for character string data. LONG VARCHAR is equivalent to VARCHAR(32000) or VARCHAR(64000) depending on which Teradata version your server is running.

Date, Time, and Timestamp Data

The date type in Teradata is similar to the SAS date value. It is stored internally as a numeric value and is displayed in a site-defined format. Date type columns might contain Teradata values that are out of range for SAS, which handles dates from A.D. 1582 through A.D. 20,000. If SAS/ACCESS encounters an unsupported date (for example, a date earlier than A.D. 1582), it returns an error message and displays the date as a missing value.

See "Using TIME and TIMESTAMP" on page 811 for examples.

The Teradata date/time types that SAS supports are listed here.

DATE
specifies date values in the default format YYYY-MM-DD. For example, January 25, 1989, is input as 1989-01-25. Values for this type can range from 0001-01-01 through 9999-12-31.

TIME (*n*)
specifies time values in the format HH:MM:SS.SS. In the time, SS.SS is the number of seconds ranging from 00 to 59 with the fraction of a second following the decimal point.

n is a number from 0 to 6 that represents the number of digits (precision) of the fractional second. For example, TIME(5) is 11:37:58.12345 and TIME(0) is 11:37:58. This type is supported for Teradata Version 2, Release 3 and later.

TIMESTAMP (*n*)
specifies date/time values in the format YYYY-MM-DD HH:MM:SS.SS. In the timestamp, SS.SS is the number of seconds ranging from 00 through 59 with the fraction of a section following the decimal point.

n is a number from 0 to 6 that represents the number of digits (precision) of the fractional second. For example, TIMESTAMP(5) is 1999-01-01 23:59:59.99999 and TIMESTAMP(0) is 1999-01-01 23:59:59. This type is supported for Teradata Version 2, Release 3 and later.

CAUTION:
When processing WHERE statements (using PROC SQL or the DATA step) that contain *literal* values for TIME or TIMESTAMP, the SAS engine passes the values to Teradata exactly as they were entered, without being rounded or truncated. This is done so that Teradata can handle the rounding or truncation during processing.

Numeric Data

When reading Teradata data, SAS/ACCESS converts all Teradata numeric data types to the SAS internal format, floating-point.

BYTEINT
specifies a single-byte signed binary integer. Values can range from –128 to +127.

DECIMAL(*n*,*m*)
specifies a packed-decimal number. *n* is the total number of digits (precision). *m* is the number of digits to the right of the decimal point (scale). The range for precision is 1 through 18. The range for scale is 0 through *n*.

If *m* is omitted, 0 is assigned and *n* can also be omitted. Omitting both *n* and *m* results in the default DECIMAL(5,0). DECIMAL is also known as NUMERIC.

CAUTION:
Because SAS stores numbers in floating-point format, a Teradata DECIMAL number with very high precision can lose precision. For example, when SAS/ACCESS running on a UNIX MP-RAS client reads a Teradata column specified as DECIMAL (18,18), it maintains only 13 digits of precision. This can cause problems. A large DECIMAL number can cause the WHERE clause that SAS/ACCESS generates to perform improperly (fail to select the expected rows). There are other potential problems. For this reason, *use carefully* large precision DECIMAL data types for Teradata columns that SAS/ACCESS accesses.

FLOAT
specifies a 64-bit Institute of Electrical and Electronics Engineers (IEEE) floating-point number in sign-and-magnitude form. Values can range from approximately 2.226×10^{-308} to 1.797×10^{308}. FLOAT is also known as REAL or DOUBLE PRECISION.

When the SAS/ACCESS client internal floating point format is IEEE, Teradata FLOAT numbers convert precisely to SAS numbers. Exact conversion applies to SAS/ACCESS Interface to Teradata running under UNIX MP-RAS. However, if you are running SAS/ACCESS Interface to Teradata under z/OS, there can be minor precision and magnitude discrepancies.

INTEGER
specifies a large integer. Values can range from −2,147,483,648 through +2,147,483,647.

SMALLINT
specifies a small integer. Values can range from −32,768 through +32,767.

Teradata Null Values

Teradata has a special value that is called NULL. A Teradata NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a Teradata NULL value, it interprets it as a SAS missing value.

By default, Teradata columns accept NULL values. However, you can define columns so that they do not contain NULL values. For example, when you create a SALES table, define the CUSTOMER column as NOT NULL. This tells Teradata not to add a row to the table unless the CUSTOMER column for the row has a value. When creating a Teradata table with SAS/ACCESS, you can use the DBNULL= data set option to indicate whether NULL is a valid value for specified columns.

For more information about how SAS handles NULL values, see “Potential Result Set Differences When Processing Null Data ” on page 35.

To control how SAS missing character values are handled, use the NULLCHAR= and NULLCHARVAL= data set options.

LIBNAME Statement Data Conversions

This table shows the default formats that SAS/ACCESS Interface to Teradata assigns to SAS variables when using the LIBNAME statement to read from a Teradata table. SAS/ACCESS does not use Teradata table column attributes when it assigns defaults.

Table 29.5 *Default SAS Formats for Teradata*

Teradata Data Type	Default SAS Format
CHAR(*n*)	$*n* (*n*<= 32,767)
CHAR(*n*)	$32767.(*n*>32,767) [*]
VARCHAR(*n*)	$*n* (*n*<= 32,767)
VARCHAR(*n*)	$32767.(*n*> 32,767) [*]
LONG VARCHAR(*n*)	$32767. [*]
BYTE(*n*)	$HEX*n*. (*n*<= 32,767)
BYTE(*n*)[*]	$HEX32767.(*n*> 32,767)
VARBYTE(*n*)	$HEX*n*. (*n*<= 32,767)
VARBYTE(*n*)	$HEX32767.(*n*> 32,767)
INTEGER	11.0
SMALLINT	6.0
BYTEINT	4.0
DECIMAL(*n*, *m*)[**]	(*n*+2).(*m*)
FLOAT	none
DATE[***]	DATE9.
TIME(*n*)[†]	for *n*=0, TIME8. for *n*>0, TIME9+*n.n*
TIMESTAMP(*n*)[†]	for *n*=0, DATETIME19. for *n*>0, DATETIME20+*n.n*
TRIM(LEADING FROM c)	LEFT(c)
CHARACTER_ LENGTH(TRIM(TRAILING FROM c)	LENGTH(c)
(v MOD d)	MOD(v,d)

Teradata Data Type	Default SAS Format
TRIMN(c)	TRIM(TRAILING FROM c)

* When reading Teradata data into SAS, DBMS columns that exceed 32,767 bytes are truncated. The maximum size for a SAS character column is 32,767 bytes.

** If the DECIMAL number is extremely large, SAS can lose precision. For details, see "Numeric Data" on page 821.

*** See the topic "Date/Time Data" for how SAS/ACCESS handles dates that are outside the valid SAS date range.

† TIME and TIMESTAMP are supported for Teradata Version 2, Release 3 and later. The TIME with TIMEZONE, TIMESTAMP with TIMEZONE, and INTERVAL types are presented as SAS character strings, Therefore, they are harder to use.

When you *create* Teradata tables, the default Teradata columns that SAS/ACCESS creates are based on the type and format of the SAS column. The following table shows the default Teradata data types that SAS/ACCESS assigns to the SAS formats during output processing when you use the LIBNAME statement.

Table 29.6 *Default Output Teradata Data Types*

SAS Data Type	SAS Format	Teradata Data Type
Character	$*w*. $CHAR*w*. $VARYING*w*.	CHAR[*w*]
Character	$HEX*w*.	BYTE[*w*]
Numeric	A date format	DATE
Numeric	TIME*w.d*	TIME(*d*)
Numeric	DATETIME*w.d*	TIMESTAMP(*d*)
Numeric	*w*.(*w*≤2)	BYTEINT
Numeric	*w*.(3≤*w*≤4)	SMALLINT
Numeric	*w*.(5≤*w*≤9)	INTEGER
Numeric	*w*.(≤10*w*≤18)	DECIMAL
Numeric	*w*.(*w*≥19)	FLOAT
Numeric	*w.d*	DECIMAL(*w*-1,*d*)
Numeric	All other numeric formats	FLOAT

To override any default output type, use the DBTYPE= data set option.

Data Returned as SAS Binary Data with Default Format $HEX

- BYTE

- VARBYTE
- LONGVARBYTE
- GRAPHIC
- VARGRAPHIC
- LONG VARGRAPHIC

Temporal Data Types for Teradata

Overview

Teradata provides built-in support for *temporal data*. Temporal data makes it easy to build and maintain applications where information changes over time. For example, consider a merchandise application that must store the price of an item along with the dates during which that price is valid. As shown below, traditional table design uses two date columns to store the beginning and end dates during which the price is valid.

```
CREATE TABLE price
(
  Item_UPC    BIGINT,
  Price       FLOAT,
  Begin_date  DATE,
  End_Date    DATE
)
```

```
   Item_UPC           Price       Begin_Date        End_Date
   --------           -----       ----------        --------
   123456789012      $5        2010-01-05       9999-12-31
```

When the price of the item changes, the end date of the current price must be updated, and a new row must be inserted with the new price.

```
   Item_UPC           Price       Begin_Date        End_Date
   --------           -----       ----------        --------
   123456789012       $5        2010-01-05       2011-05-01
   123456789012       $4        2011-05-01       9999-12-31
```

Teradata temporal support can simplify this process. You can use the PERIOD(DATE) data type to represent the time period during which the price is valid. Here is how you can create the table instead.

```
CREATE TABLE price
(
  Item_UPC       BIGINT,
  Price          FLOAT,
  Price_validity PERIOD(DATE) NOT NULL AS VALIDTIME
)
```

```
   Item_UPC           Price       Price_validity
   --------           -----       --------------
   123456789012      $5        (2010-01-05, 9999-12-31)
```

For additional examples and information about features, see the Teradata temporal table support documentation.

Supported Temporal Data Types

SAS/ACCESS Interface to Teradata supports these temporal data types.

- PERIOD(TIME)
- PERIOD(DATE)
- PERIOD(TIMESTAMP)
- PERIOD(DATE) VALIDTIME/TRANSACTIONTIME
- PERIOD(TIMESTAMP) VALIDTIME/TRANSACTIONTIME

Specifying Transaction Time and Valid Time

For true temporal support, you must specify the transaction-time and valid-time attributes on the PERIOD data type.

The transaction-time attribute on a PERIOD column makes the table a transaction-time table. Teradata automatically maintains tables with transaction-time columns. It tracks when a row is first made known to the table. When a row is inserted, it is considered to be an open row because it is currently in effect until the end of time. If the row is deleted, Teradata marks it as a closed row that is no longer in effect. However, the table can be queried to obtain rows that were open at a particular point in time even though the row is not currently valid. Similarly, when a row is modified, the current row is closed and a new row is opened and made effective.

A user cannot set or modify a transaction-time column.

The valid-time attribute indicates the time period during which the information is in effect. If valid time is specified with the PERIOD data type, Teradata maintains how the time period is in effect if the row is updated or deleted. As in the example about prices, when a row is inserted with a new price, Teradata maintains the end date of the original row. The row with the old price is updated with an end date and the new row is inserted.

A row in a valid-time transaction table can be a history row, a current row, or a future row. The history row is no longer valid with respect to current time. Its end-time period is before the current time. A current row has a time period that straddles the current time.

Creating a Table from SAS with the PERIOD Data Type

To create a Teradata table with temporal data types from SAS, use the DBTYPE= data set option. SAS does not have an equivalent data type for PERIOD. The value is represented in SAS as a character string. In this example, when the character string that represents the period ID is inserted into Teradata, it is implicitly converted to a PERIOD data type.

```
data x.mytest(DBTYPE=(validity='PERIOD(DATE) VALIDTIME'));
       i=1;
       validity='(1973-02-03, 9999-12-31)';
       output;
run;
```

Reading in a PERIOD Data Type

A Teradata PERIOD data-type column can be read into SAS like any other column. It is represented in SAS as a character string, such as **'(1973-02-03, 9999-12-31)'**.

Temporal Qualifiers

Temporal tables contain rows that can be current, history, or future in the valid-time dimension. In the transaction-time dimension, rows can be open or closed. Temporal qualifiers specify what data is needed. As a LIBNAME and a data set option, TEMPORAL_QUALIFIER= lets you qualify queries in the valid-time or transaction-time dimension. For example, to fetch rows that are valid as of **'2009-01-01'** in a table, you must specify **TEMPORAL_QUALIFIER='VALIDTIME AS OF DATE '2009-01-01' '** as a LIBNAME or a data set option when you query temporal tables.

The option that you specify for TEMPORAL_QUALIFIER= is free-form text. Here are some examples.

```
TEMPORAL_QUALIFIER='CURRENT VALIDTIME'
TEMPORAL_QUALIFIER='VALIDTIME AS OF DATE '2009-01-01' '
TEMPORAL_QUALIFIER='NONSEQUENCED VALIDTIME'
TEMPORAL_QUALIFIER='   SEQUENCED VALIDTIME'
TEMPORAL_QUALIFIER='NONSEQUENCED VALIDTIME PERIOD '(2007-01-01, 2008-03-01)''
TEMPORAL_QUALIFIER='   SEQUENCED VALIDTIME PERIOD '(2007-01-01, 2008-03-01)''
TEMPORAL_QUALIFIER='CURRENT TRANSACTIONTIME'
TEMPORAL_QUALIFIER='TRANSACTIONTIME AS OF TIMESTAMP '2009-01-01 01:02:03.123456' '
```

If you specify the temporal qualifier on the LIBNAME, it applies to the entire session because it is implemented by issuing session commands at connect time. For example, if you specify **TEMPORAL_QUALIFIER='ASOF PERIOD '(1999-01-01, 2099-01-05)' '** on the LIBNAME, here is the Teradata SET SESSION command that is issued at connect time. The SQL is submitted as usual.

```
.SET SESSION ASOF PERIOD '(1999-01-01, 2099-01-05)'
```

If you submit the above command, the temporal qualifier is added as a prefix, as shown below.

```
ASOF PERIOD '(1999-01-01, 2099-01-05)'
SELECT * from TEMPORAL_TABLE;
```

```
Sample code
-----------

/* PERIOD data types require the Teradata V13 server. */
libname x teradata user=testuser pw=testpw server=td13;

/* Create a table with the PERIOD(DATE) data type.
   Note:  This is not a temporal table. */
data x.mytest(DBTYPE=(validity='PERIOD(DATE)'));
i=1; validity='(1973-02-03, 9999-12-31)'; output;
run;

/* Read from a table with a PERIOD data type? */
proc print data=x.mytest;
run;
```

```
/* Use Fastload to load a table with a PERIOD data type. */
proc datasets library=x;
  delete mytest;run;

data x.mytest(DBTYPE=(validity='PERIOD(TIMESTAMP)') FASTLOAD=YES TPT=NO);
i=1; validity='(1970-01-05 01:02:03.123, 1970-01-05 05:06:07.456)'; output;
run;

/* Tempral support starts in Teradata V13.10. */
libname x teradata user=testuser pw=testpw server=td1310;

/* Create a table with the PERIOD(DATE) data type. */
data x.mytest(DBTYPE=(validity='PERIOD(DATE) VALIDTIME'));
i=1; validity='(1973-02-03, 1999-12-31)'; output;
i=2; validity='(2000-01-01, 2011-01-01)'; output;
i=3; validity='(2011-01-02, 9999-12-31)'; output;
run;

/* Can we read a PERIOD data type?
   You must select the row with i=2. */
proc print data=x.mytest(TEMPORAL_QUALIFIER='CURRENT VALIDTIME');
run;

/* Consider data as of 1995-01-01. */
libname x teradata user=testuser pw=testpw server=td1310
          TEMPORAL_QUALIFIER='VALIDTIME AS OF DATE '1995-01-01' '

/* Row with i=1 is returned. */
proc print data=x.mytest(DBSLICEPARM=ALL);
run;
```

Part 4

Sample Code

Chapter 30

Accessing DBMS Data with the LIBNAME Statement

Overview: LIBNAME Statement Sample Code

Examples in this section show how to use the LIBNAME statement on page 93 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 SAS/ACCESS relational interfaces. However, you might need to make some adjustments for some interfaces, such as using different or additional information. For interface-specific details, see the comments within the sample code and also check for related SAS notes, which are available at **`http://support.sas.com/notes`**.

Follow these steps to run the examples.

1. Modify and submit the ACCAUTO.SAS file, which creates 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 cannot locate this library, contact your SAS support consultant.For information about the tables that are in sample code, see "Descriptions of the Sample Data" on page 861.

Note: Before you rerun an example that *updates* DBMS data, resubmit the ACCDATA.sas program to re-create the DBMS tables.

Examples: Create SAS Data Sets from DBMS Data

Overview

After you associate a SAS/ACCESS libref with your DBMS data, you can use the libref just as you would use any SAS libref. These examples show basic uses of the DATA step with librefs that reference DBMS data.

Example 1: Use the PRINT Procedure with DBMS Data

In this example, the DB2 interface creates the MyDbLib libref and associates it 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 Staff DB2 table. Information for staff from states other than New Jersey is not printed, and the Staff DB2 table 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 30.1 *Use the PRINT Procedure with DBMS Data*

```
                    New Jersey Phone List

     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
```

Example 2: Combine DBMS Data and SAS Data

This example shows how to read DBMS data into SAS and create additional variables to perform calculations or subsetting operations on the data. It creates the Work.HighWage SAS data set from the Payroll DB2 table, and it adds a new variable, Category. This variable is based on the value of the salary column in the Payroll DB2 table. 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 30.2 *Combine DBMS Data and SAS Data*

```
                    Salary Analysis

        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
```

Example 3: Read Data from Multiple DBMS Tables

You can use the DATA step to read data from multiple data sets. This example merges data from the Staff and SuperV Oracle tables in the Work.Combined SAS data set.

```
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 30.3 *Read Data from Multiple DBMS Tables*

Supervisor Information

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

Example 4: Use the DATA Step UPDATE Statement with DBMS Data

You can also use the DATA step UPDATE statement to create a SAS data set with DBMS data. This example creates the Work.Payroll SAS data set with data from the Payroll and Payroll2 Oracle tables. The Oracle tables are not modified.

The columns in the two Oracle tables must match, but 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 30.4 *Create a SAS Data Set with DBMS Data By Using the UPDATE Statement*

Updated Payroll Data

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

Examples: Use the SQL Procedure with DBMS Data

Overview

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. These examples use the SQL procedure to query, update, and create DBMS tables.

Examples: Query a DBMS Table

Example 1: Query a DBMS Table

This example uses the SQL procedure to query the Payroll Oracle table. 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 30.5 *Query 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
```

Example 2: Query a DBMS Table with a WHERE Clause

The next example uses the SQL procedure to query flight information from the Delay Oracle table. 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;
```

To optimize performance, SAS/ACCESS passes the entire join to the DBMS for processing by default. For more information, see "Overview: Optimizing Your SQL Usage" on page 45.

Output 30.6 *Query 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
```

Example 3: Query a DBMS Table with SAS Extensions

This example uses the SQL procedure to query the InterNat DB2 table for information about international flights with more than 200 passengers. A PROC SQL query sorts output. Also, 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 More Than 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 30.7 *Query a DBMS Table with SAS Extensions*

```
             International Flights by Flight Number
                   with More Than 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
```

Examples: Query Multiple DBMS Tables

Example 1: Query Multiple Oracle Tables

You can also retrieve data from multiple DBMS tables in a single query by using the SQL procedure. This example joins the Staff and Payroll Oracle tables 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;
```

To optimize performance, SAS/ACCESS passes the entire join to the DBMS for processing by default. For more information, see "Passing Joins to the DBMS" on page 47.

Output 30.8 *Query 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
```

Example 2: Query Multiple DB2 Tables

This example uses the SQL procedure to join and query the March, Delay, and Flight DB2 tables. 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: To optimize performance, SAS/ACCESS passes the entire join to the DBMS for processing by default. For more information, see “Passing Joins to the DBMS” on page 47.

Output 30.9 *Query 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
```

Example 3: Query Multiple DBMS Tables

This example uses the SQL procedure to retrieve the combined results of two queries to the Payroll and Payroll2 Oracle tables. 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 30.10 *Query Multiple DBMS Tables*

```
               Payrolls 1 and 2

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
```

Examples: Update DBMS Data

Example 1: Add to 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. This example adds a new row to the SuperV DB2 table.

```
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 30.11 *Add to DBMS Data*

```
   New Row in AIRLINE.SUPERV

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
```

Example 2: Delete DBMS Data

This example deletes data from the Staff DB2 table for all employees who work in Connecticut.

```
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 30.12 *Delete DBMS Data*

```
              AIRLINE.STAFF After Deleting Connecticut Employees

    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
```

Example 3: Create 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 GTForty Oracle table by using data from the Staff and Payroll Oracle 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 greater than $40,000';
  format salary dollar10.2;
run;
```

Output 30.13 *Create a DBMS Table*

```
     Employees with salaries greater than $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
```

Examples: Use Other SAS Procedures with DBMS Data

Overview

Examples in this section show basic uses of other SAS procedures with librefs that refer to DBMS data.

Example 1: Use the MEANS Procedure

This example uses the PRINT and MEANS procedures on a SAS data set created from the March Oracle table. 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 30.14 *Use 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
```

Example 2: Use the DATASETS Procedure

This example uses the DATASETS procedure to view a list of DBMS table—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 30.15 *Use 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
```

Example 3: Use the CONTENTS Procedure

This example shows output from the CONTENTS procedure when it is run on a DBMS table. PROC CONTENTS shows all SAS metadata that the SAS/ACCESS interface derives from the DBMS table.

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

title 'Contents of the DELAY Table';

proc contents data=mydblib.delay;
run;
```

Output 30.16 *Use 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
```

Example 4: Use the RANK Procedure

This example uses the RANK procedure to rank flights in the Delay DB2 table by number of minutes that flights are 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 30.17 *Use the RANK Procedure*

```
                         Ranking of Delayed Flights

 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
```

Example 5: Use the TABULATE Procedure

This example uses the TABULATE procedure on the Payroll Oracle table 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 30.18 *Use the TABULATE Procedure*

```
                      Number of Employees by Jobcode

        -----------------------------------------------------------------
        |                            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|
        -----------------------------------------------------------------
```

Example 6: Use the APPEND Procedure

In this example, the Payroll2 DB2 table is appended to the Payroll DB2 table with the APPEND procedure. The Payroll table is updated on DB2.

When you append data to a DBMS table, you are actually inserting rows into a table. You can insert rows 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: When a DBMS table that you are using is in the same database space as a table that you are creating or updating, use the CONNECTION=SHARED on page 113 LIBNAME option to prevent a deadlock.

Output 30.19 *Use the APPEND Procedure*

PAYROLL After Appending PAYROLL2

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

Example: Calculate Statistics from DBMS Data

This example uses the FREQ procedure to calculate statistics on the Invoices DB2 table.

```
libname mydblib db2 ssid=db2;

proc freq data=mydblib.invoices(keep=invnum country);
  tables country;
  title 'Invoice Frequency by Country';
run;
```

This output shows the one-way frequency table that this example generates.

Output 30.20 *Use the FREQ Procedure*

```
                    Invoice Frequency by Country
                      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
```

Example: Select and Combine 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 30.21 *Use a WHERE Statement*

High Bills--Not Paid

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

Example: Join DBMS and SAS Data

This example shows how to combine SAS and DBMS data using the SAS/ACCESS LIBNAME statement. It creates an SQL view, Work.Emp_Csr, from the Employees DB2 table 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 30.22 *Combine an SQL View with a SAS Data Set*

Interns Who Are Family Members of Employees

lastname	firstnam	empid	familyid	gender	dept	hiredate
SMITH	ROBERT	765112	234967	M	CSR010	04MAY1998
NISHIMATSU-LYNCH	RICHARD	765111	677890	M	CSR011	04MAY1998

Chapter 31

Accessing DBMS Data with the SQL Pass-Through Facility

Overview: SQL Pass-Through Facility Sample Code

Examples in this section show how to use the SQL pass-through facility on page 403 to access and update DBMS data. You can use the SQL pass-through facility to read and write data between SAS and a DBMS. However, it is recommended that you use the LIBNAME statement on page 93 to access your DBMS data more easily and directly.

To run these examples, follow these steps.

1. Modify and submit the ACCAUTO.SAS file, which creates 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 861 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.

Example: Retrieve DBMS Data with a Pass-Through Query

This section describes how to retrieve DBMS data by using the statements and components of the SQL pass-through facility.

This 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. The example accesses Oracle data.

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 MyDb alias. It then lists the columns to select from the Oracle tables in the PROC SQL SELECT clause.

If you want, 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 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 Customers primary key column.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) is displayed 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 SQL pass-through facility. The DISCONNECT statement terminates the Oracle connection and the QUIT statement ends the SQL procedure.

This output shows the results of the pass-through query.

Output 31.1 *Data That a Pass-Through Query Retrieves*

```
                              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 next example changes the pass-through query into an SQL view. It adds a CREATE VIEW statement to the query, removes the ORDER BY clause from the CONNECTION TO component, and adds the ORDER BY clause to a separate SELECT statement that prints only the new SQL view. [1]

```
libname samples 'your-SAS-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';
```

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.

```
select * from samples.brief
   order by billedon, invnum;

quit;
```

The output from the Samples.Brief view is the same as shown in Output 31.1 on page 855.

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.

Example: Combine an SQL View with a SAS Data Set

This 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 31.2 *Combine 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
```

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. Yet there is another way to increase efficiency: extract the DBMS data, place the data in a new SAS data file, assign SAS indexes to the appropriate variables, and join the two SAS data files.

Example: Use 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-library';

/* Create SQL view */
proc sql;

   create view samples.allemp as
      select * from mydblib.employees;

quit;

/* Use the SQL 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. 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 31.3 *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 in the PROC SQL statement provides statistics on the SAS process.)

Log 31.1 *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 32

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 on page 831 and SQL pass-through facility on page 853 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 PROC CONTENTS output excerpts in this section describe DBMS tables and SAS data sets that are used in the sample code.

Output 32.1 *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 32.2 *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 32.3 *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 32.4 *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 32.5 *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 32.6 *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 32.7 *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 32.8 *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 32.9 *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 32.10 *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 32.11 *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 32.12 *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 32.13 *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 5

Converting SAS/ACCESS Descriptors to PROC SQL Views

Chapter 33

CV2VIEW Procedure

Overview: CV2VIEW Procedure

The CV2VIEW procedure converts SAS/ACCESS view descriptors into SQL views. Consider converting your descriptors for these reasons

- Descriptors are no longer the recommended method for accessing relational database data. By converting to SQL views, you can use the LIBNAME statement, which is the preferred method. The LIBNAME statement provides greater control over such DBMS operations as locking, spooling, and data type conversions. The LIBNAME statement can also handle long field names, but descriptors cannot.
- SQL views are platform-independent. SAS/ACCESS descriptors are not.

The CV2VIEW procedure in SAS 9.1 can convert both of these descriptors.

- 64-bit SAS/ACCESS view descriptors that were created in either 64-bit SAS 8 or 64-bit SAS 9.1
- 32-bit SAS/ACCESS view descriptors that were created in 32-bit SAS 6 and SAS 8

If the descriptor that you want to convert is READ-, WRITE-, or ALTER-protected, 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 part of the

LIBNAME statement is also not visible to prevent generated SQL statements from being submitted manually without modification.

Syntax: CV2VIEW Procedure

See: Chapter 33, “CV2VIEW Procedure,” on page 869

PROC CV2VIEW DBMS=*dbms-name* | ALL;
FROM_VIEW=*libref.input-descriptor*;
FROM_LIBREF=*input-library*;
TO_VIEW= *libref.output-view*;
TYPE= SQL | VIEW | ACCESS;
TO_LIBREF= *name of output library*;
SAVEAS=*external filename*;
SUBMIT;
REPLACE=ALL | VIEW | FILE;

Statement	Task
“PROC CV2VIEW Statement”	Convert SAS/ACCESS view descriptors into SQL views
“FROM_LIBREF= Statement”	Specify the library containing the view descriptors or access descriptors that you want to convert
“FROM_VIEW= Statement”	Specify the view descriptor or access descriptor that you want to convert
“REPLACE= Statement”	Specify whether existing views and files are replaced
	Save the generated PROC SQL statements to a file
“SAVEAS= Statement”	Submit the generated PROC SQL statements when you specify the SAVEAS= statement
	Specify the name of the new (converted) SQL view
“TO_VIEW= Statement”	Specify the library that contains the new (converted) SQL views
“TYPE= Statement”	Specify what type of conversion should occur

PROC CV2VIEW Statement

Converts SAS/ACCESS view descriptors into SQL views

Syntax

PROC CV2VIEW DBMS=*dbms-name* | ALL;

Required 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

Restriction: If you specify DBMS=ALL, you cannot use the FROM_VIEW= statement.

Requirements: You must specify either the FROM_VIEW= statement or the FROM_LIBREF= statement.

FROM_VIEW= and TO_VIEW= statements are always used together.

Syntax

FROM_VIEW=*libref.input-descriptor*;

Required 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

Requirements: You must specify either the FROM_VIEW= statement or the FROM_LIBREF= statement.

FROM_LIBREF= and TO_LIBREF= statements are always used together.

Syntax

FROM_LIBREF= *input-library*;

Required 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, 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

Syntax

REPLACE=ALL | FILE | VIEW;

Required 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 is 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 on page 873.

Syntax

SAVEAS=*external-filename*;

Required Argument

external-filename
lets you 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, the output view has the same level of security. However, the file that contains the SQL statements does not show password values. The password in the LIBNAME statement also does not show password values.

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 on page 872, PROC CV2VIEW automatically submits the generated SQL. So you need not specify the SUBMIT statement.

Syntax

SUBMIT;

TO_VIEW= Statement

Specifies the name of the new (converted) SQL view

Restriction: If you specify DBMS=ALL, you cannot use the TO_VIEW= statement.

Requirements: You must specify either the TO_VIEW= statement or the TO_LIBREF= statement.

FROM_VIEW= and TO_VIEW= statements are always used together.

Interaction: Use the REPLACE= statement on page 872 to control whether the output file is overwritten or appended if it already exists.

Syntax

TO_VIEW=*libref.output-view*;

Required Arguments

libref
specifies the libref where you want to 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

Requirements: You must specify either the TO_VIEW= statement or the TO_LIBREF= statement.

The FROM_LIBREF= and TO_LIBREF= statements are always used together.

Interaction: Use the REPLACE= statement on page 872 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.

Syntax

TO_LIBREF= *output-library*;

Required 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 on page 871 and the TO_VIEW= statement on page 873.

TYPE= Statement

Specifies what type of conversion should occur

Syntax

TYPE=SQL | VIEW | ACCESS;

Required 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. It is most useful in the 32-bit to 64-bit case. It does not convert view descriptors across different operating systems.

ACCESS
specifies that PROC CV2VIEW converts access descriptors to native access descriptor format. It is most useful in the 32-bit to 64-bit case. It does not convert access descriptors across different operating systems.

Details

If TYPE=VIEW or TYPE=ACCESS, then SAVEAS=, SUBMIT, and REPLACE= or REPLACE_FILE= are not valid options.

Examples: CV2VIEW Procedure

Example 1: Converting an Individual View Descriptor

Features: Statements
PROC CV2VIEW
FROM_VIEW=
TO_VIEW=
SAVEAS=
SUBMIT
REPLACE

In this example, PROC CV2VIEW converts the MYVIEW view descriptor to the SQL view NEWVIEW. When you use ALTER, READ, and WRITE, the MYVIEW view descriptor is protected again alteration, reading, and writing. The PROC SQL statements that PROC CV2VIEW generates are submitted and saved to an external file named SQL.SAS.

Details

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. You can then use CV2VIEW to access those tables with 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.

```
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 these 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;
```

Example 2: Converting a Library of View Descriptors for a Single DBMS

Features: Statements
- PROC CV2VIEW
- FROM_LIBREF=
- TO_LIBREF=
- SAVEAS=
- SUBMIT

In this example PROC CV2VIEW converts all 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 PROC CV2VIEW generates are both submitted and saved to an external file named SQL.SAS.

The SAVEAS= statement causes all generated SQL for all 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. You can then use CV2VIEW to access those tables with mixed-case or embedded-blank table names.

```
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 these 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;
```

Example 3: Converting a Library of View Descriptors for All Supported DBMSs

Features: PROC CV2VIEW option
  DBMS=ALL

Statements
  PROC CV2VIEW
  FROM_LIBREF=
  TO_LIBREF=

In this example PROC CV2VIEW converts all 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, the procedure then moves to the next view. The PROC SQL statements that PROC CV2VIEW generates are automatically submitted but are not saved to an external file because the SAVEAS= statement on page 872 is not used.

```
libname input '/username/descriptors/';
libname output '/username/sqlviews/';

proc cv2view dbms=all;
from_libref = input;
to_libref = output;
run;
```

Part 6

Appendixes

Appendix 1
ACCESS Procedure

Overview: ACCESS Procedure

Accessing DBMS Data

The ACCESS procedure on page 881 is still supported for the database systems and environments on which it was available in SAS 6. However, 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 SAS/ACCESS LIBNAME statement on page 93 or the SQL pass-through facility on page 69.

Not all SAS/ACCESS interfaces support this feature. See features by host on page 77 to determine whether this feature is available in your environment.

Along with the DBLOAD procedure and an interface view engine, the ACCESS procedure creates an interface between SAS and data in other vendors' databases. You can use the ACCESS procedure to create and update descriptors on page 896

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 SAS/ACCESS documentation for your DBMS.
- *Creating and updating statements* are CREATE statement on page 886 and UPDATE statement on page 895.
- *Table and editing statements* include ASSIGN on page 886, DROP on page 888, FORMAT on page 888, LIST on page 889, QUIT on page 890, RENAME on page 890, RESET on page 891, SELECT on page 892, SUBSET on page 893, TABLE= on page 894, and UNIQUE on page 894.

This table summarizes PROC ACCESS options and statements that are required to accomplish common tasks.

Table A1.1 *Statement Sequence for Accomplishing Tasks with the ACCESS Procedure*

Task	Statements and Options to Use
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**;

Task	Statements and Options to Use
Update an access descriptor	**PROC ACCESS** *statement-options*; **UPDATE** *libref.member-name*.ACCESS; *database-connection-statements*; *editing-statements*; **RUN**;
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**;

DBMS Specifics: ACCESS Procedure

These SAS/ACCESS relational interfaces support the ACCESS procedure. Select your DBMS for details.

- DB2 z/OS on page 479
- Oracle on page 711
- Sybase on page 742

Syntax: ACCESS Procedure

See: DB2 z/OS on page 479, Oracle on page 711, or Sybase on page 742 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;

Statement	Task
"PROC ACCESS Statement"	Access relational DBMS data
"Database Connection Statement"	Provide DBMS-specific connection information
"ASSIGN Statement"	Indicate whether SAS variable names and formats are generated
"CREATE Statement"	Create a SAS/ACCESS descriptor file
"DROP Statement"	Drop a column so that it cannot be selected in a view descriptor
"FORMAT Statement"	Change a SAS format for a DBMS column
"LIST Statement"	List columns in the descriptor and give information about them
"QUIT Statement"	Terminate the procedure
"RENAME Statement"	Modify the SAS variable name

Statement	Task
"RESET Statement"	Reset DBMS columns to their default settings
"SELECT Statement"	Select DBMS columns for the view descriptor
"SUBSET Statement"	Add or modify selection criteria for a view descriptor
"TABLE= Statement"	Identify the DBMS table on which the access descriptor is based
"UNIQUE Statement"	Generate SAS variable names based on DBMS column names
"UPDATE Statement"	Update a SAS/ACCESS descriptor file

PROC ACCESS Statement

Accesses relational DBMS data

Syntax

PROC ACCESS <*options*>;

Optional Arguments

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 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 Statement

Provides DBMS-specific connection information

See: *Database connection statements* are used to connect to your DBMS. For the statements to use with your DBMS, see SAS/ACCESS documentation for your interface.

Syntax

database-connection-statements;

ASSIGN Statement

Indicates whether SAS variable names and formats are generated

Default: NO

Interaction: FORMAT, RENAME, RESET, UNIQUE

Syntax

ASSIGN <=> YES | NO | Y | N;

Required Arguments

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
lets you 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 how SAS variable names and formats are assigned:

- SAS automatically generates SAS variable names.
- 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.

When you create an access descriptor, use the RENAME statement to change SAS variable names and the FORMAT statement to change SAS formats.

When you specify **YES**, SAS generates names according to these rules:

- You can change the SAS variable names only in the access descriptor.
- 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

Syntax

CREATE *libref.member-name*.ACCESS | VIEW <*password-option*>;

Required Arguments

libref.member-name
identifies the libref of the SAS library where you want to store the descriptor and identifies the descriptor name.

ACCESS
specifies an access descriptor.

VIEW
specifies a view descriptor.

Optional Argument

password-option
specifies a password.

Details

The CREATE statement is required. It names the access descriptor or view descriptor that you are creating. Use a three-level name:

- The first level identifies the libref of the SAS library where you want to store the descriptor,
- The second level is the descriptor name,
- 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 882 for the appropriate sequence of statements for creating access and view descriptors.

Example

This 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

Interaction: RESET, SELECT

Syntax

DROP <'>*column-identifier-1*<'> <…<'>*column-identifier-n*<'>>;

Required Argument

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 this 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 column attributes (such as the SAS variable name and format) to their default values.

FORMAT Statement

Changes a SAS format for a DBMS column

Interaction: ASSIGN, DROP, RESET

Syntax

FORMAT <'>*column-identifier-1*<'>
<=>*SAS-format-name-1* <…<'>*column-identifier-n*<'><=>*SAS-format-name-n*>;

Required Arguments

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 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, submit this statement to associate the DATE9. format with the BIRTHDATE column and with the second column in the access descriptor:

```
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

Default: ALL

Syntax

LIST <ALL | VIEW | <'>*column-identifier*<'>>;

Optional Arguments

ALL
lists all DBMS columns in the table, positional equivalents, SAS variable names, and SAS variable formats that are available for a descriptor.

VIEW
lists all 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 this 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.

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 create or update multiple descriptors, specify LIST before each CREATE or UPDATE statement to list information about all descriptors that you are creating or updating.

QUIT Statement

Terminates the procedure

Syntax

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

Interaction: ASSIGN, RESET

Syntax

RENAME <'>*column-identifier-1*<'> <=> *SAS-variable-name-1* <…<'>
column-identifier-n<'><=>*SAS-variable-name-n*>;

Required Arguments

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 type of descriptor that 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 when an ACCESS procedure executes. For example, if you rename the CUSTOMER column to CUSTNUM when you create an access descriptor, the column is still named CUSTNUM when you select it in a view descriptor unless you specify another RESET or RENAME statement.

  When you create a view descriptor that is based on this access descriptor, you can specify the RESET statement or another RENAME statement to rename the variable. However, 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 this statement:

```
rename
7 birthdy  'firstname'=fname;
```

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

Interaction: ASSIGN, DROP, FORMAT, RENAME, SELECT

Syntax

RESET ALL | <'>*column-identifier-1*<'> <…<'>*column-identifier-n*<'>>;

Required Arguments

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 this 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.

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, default names are the first eight characters of each DBMS column name.

For *view descriptors*, the RESET statement clears (deselects) any columns that were included in the SELECT statement. When you create a view descriptor that is based on an access descriptor that is 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

Interaction: RESET

Syntax

SELECT ALL | <'>*column-identifier-1*<'> <…<'>*column-identifier-n*<'>>;

Required Arguments

ALL
includes in the view descriptor all 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. The positional equivalent is the number that represents where the column is located in the access descriptor on which the view is based. For example, to select the first three columns, submit this 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 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

Syntax

SUBSET *selection-criteria*;

Required Argument

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 data (rows) in the DBMS table.

For example, you could submit the following SUBSET statement for a view descriptor that retrieves rows from a DBMS table:

```
subset where firstorder is not null;
```

If you have multiple selection criteria, enter them all in one SUBSET statement, as shown in this example:

```
subset where firstorder is not null
  and country = 'USA'
  order by country;
```

Unlike other ACCESS procedure statements, the SUBSET statement is case sensitive. The SQL statement is sent to the DBMS exactly as you enter it. Therefore, you must use the correct case for any DBMS object names. See 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

Syntax

TABLE= <'>*table-name*<'>;

Required Argument

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 SAS/ACCESS documentation for your DBMS for details about4 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

Interaction: ASSIGN

Syntax

UNIQUE <=> YES | NO | Y | N;

Required Arguments

YES
: causes the SAS/ACCESS interface to append numbers to any duplicate SAS variable names, 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 890 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.

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

Syntax

UPDATE *libref.member-name*.ACCESS | VIEW <*password-option*>;

Required Arguments

libref.member-name
identifies the libref of the SAS library where you want to store the descriptor and identifies the descriptor name.

ACCESS
specifies an access descriptor.

VIEW
specifies a view descriptor.

Optional Argument

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.

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 so that they 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.

The following statements are not supported when you use the UPDATE statement: ASSIGN, RESET, SELECT, and UNIQUE.

SeeTable A1.1 on page 882 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.

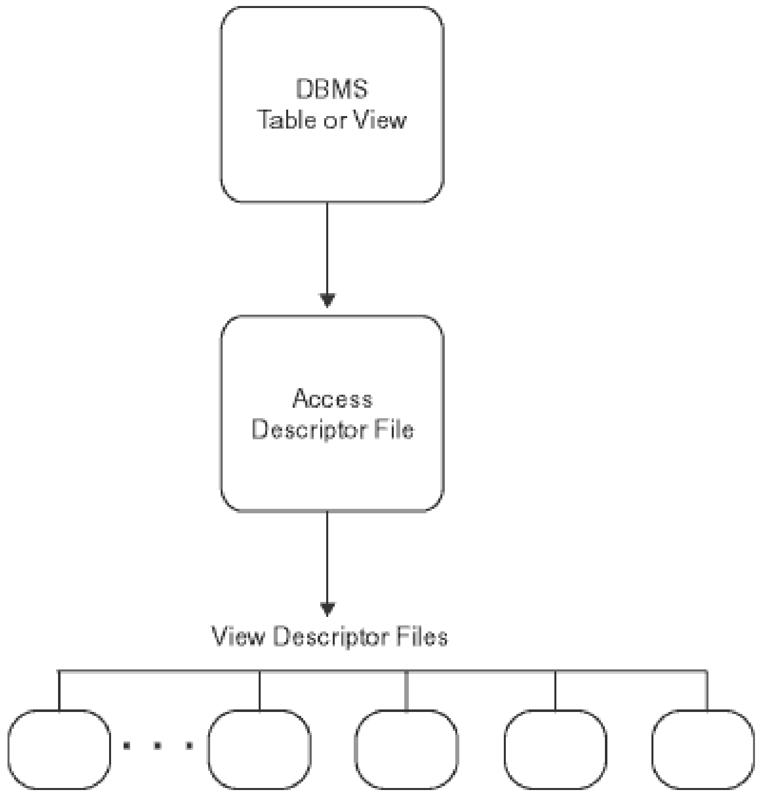
There are two types of descriptors, *access descriptors* and *view descriptors*. Access descriptors provide SAS with information about the structure and attributes of a DBMS table or view. An access descriptor, in turn, is used to create one or more view descriptors, or SAS data views, of the DBMS data.

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 that 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 A1.1 *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, which is called *extracting* the data. When you need to use DBMS data in several procedures or DATA steps, you might use fewer resources by extracting the data into a SAS data file instead of repeatedly accessing the data directly.

The SAS/ACCESS interface view engine usually tries to pass WHERE conditions to the DBMS for processing. 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 lets you 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 password, if one exists. In this case, you are *not* assigning a password to the view descriptor that is being created or updated. Instead, using the password grants you permission to use the access descriptor to create or update the view descriptor. Here is an 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: ACCESS Procedure

Example 1: Update an Access Descriptor

Features: Statements
PROC ACCESS
UPDATE
DROP
LIST

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.

You can use the LIST statement to write all variables to the SAS log so that you can see the complete access descriptor before you update it.

```
proc access dbms=oracle ad=adlaib.employ;

   /* update access descriptor  */

   update adlib.employ.access;
   drop salary birthdate;
   list all;
run;
```

Example 2: Create a View Descriptor

Features: Statements
PROC ACCESS
CREATE
SELECT
FORMAT
SUBSET

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.

The following example re-creates a view descriptor, VLIB.EMP1204, which is based on an access descriptor, ADLIB.EMPLOY, which 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;
```

Appendix 2
DBLOAD Procedure

Overview: DBLOAD Procedure

Sending Data from SAS to a DBMS

The DBLOAD procedure is still supported for the database systems and environments on which it was available in SAS 6. However, it 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 SAS/ACCESS LIBNAME statement on page 93 or the SQL pass-through facility on page 403.

Not all SAS/ACCESS interfaces support this feature. See features by host on page 77 to determine whether this feature is available in your environment.

Properties of the DBLOAD Procedure

Along with the ACCESS procedure and an interface view engine, the DBLOAD procedure 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 these items:

- 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:

- Use *database connection statements* to connect to your DBMS. See the DBMS-specific reference in this document for details about your DBMS.
- *Creating and loading statements* are LOAD on page 909 and RUN.
- Use *table and editing statements* ACCDESC= on page 906, COMMIT= on page 906, DELETE on page 907, ERRLIMIT= on page 907, LABEL on page 908, LIMIT= on page 908, LIST on page 908, NULLS on page 910, QUIT on page 910, RENAME on page 911, RESET on page 911, SQL on page 912, TABLE= on page 913, TYPE on page 913, and WHERE on page 914.

This table summarizes PROC DBLOAD options and statements that are required to accomplish common tasks.

Table A2.1 *Statement Sequence for Accomplishing Common Tasks with the DBLOAD Procedure*

Task	Options and Statements to Use
Create and load a DBMS table	**PROC DBLOAD** *statement-options*; *database-connection-options*; **TABLE=** <'>*table-name*<'>; **LOAD**; **RUN**;

Task	Options and Statements to Use
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**;

LOAD must appear before RUN to create and load a table or append data to a table.

DBMS Specifics: DBLOAD Procedure

These SAS/ACCESS relational interfaces support the DBLOAD procedure. Select your DBMS for details.

- DB2 under UNIX and PC Hosts on page 450
- DB2 under z/OS on page 481
- Microsoft SQL Server on page 602
- ODBC on page 661
- Oracle on page 712
- Sybase on page 743

Syntax: DBLOAD Procedure

See: DB2 under UNIX and PC Hosts on page 450, DB2 under z/OS on page 481, Microsoft SQL Server on page 602, ODBC on page 661, Oracle on page 712, Sybase on page 743 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;

Statement	Task
"PROC DBLOAD Statement"	Send data from SAS to a DBMS
"Database Connection Statement"	Provide DBMS connection information
"ACCDESC= Statement"	Create an access descriptor based on the new DBMS table
"COMMIT= Statement"	Issue a commit or save rows after a specified number of inserts
"DELETE Statement"	Specify variables that should not be loaded into the new table
"ERRLIMIT= Statement"	Stop the loading of data after a specified number of errors
"LABEL Statement"	Cause DBMS column names to default to SAS labels
"LIMIT= Statement"	Limit the number of observations that are loaded
"LIST Statement"	List information about the variables to be loaded
"LOAD Statement"	Create and load the new DBMS table
"NULLS Statement"	Specify whether DBMS columns accept NULL values

Statement	Task
"QUIT Statement"	Terminate the procedure
"RENAME Statement"	Rename DBMS columns
"RESET Statement"	Reset column names and data types to their default values
"SQL Statement"	Submit a DBMS-specific SQL statement to the DBMS
"TABLE= Statement"	Name the DBMS table to be created and loaded
"TYPE Statement"	Change default DBMS data types in the new table
"WHERE Statement"	Load a subset of data into the new table

PROC DBLOAD Statement

Sends data from SAS to a DBMS

Syntax

PROC DBLOAD <*options*>;

Optional Arguments

DBMS=*database-management-system*
specifies which database management system you want to access. This DBMS-specific option is required. See the DBMS-specific reference in this document for details about your DBMS.

DATA=<*libref.*>*SAS-data-set*
specifies the input data set. You can retrieve input data from a SAS data file, an SQL view, a DATA step view, a SAS/ACCESS view descriptor, or another DBMS table to which a SAS/ACCESS libref points. 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;`**

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 Statement

Provides DBMS connection information

Note: These statements are used to connect to your DBMS and vary depending on which SAS/ACCESS interface you are using. See the DBMS-specific reference in this document for details about your DBMS. Examples include USER=, PASSWORD=, and DATABASE=.

Syntax

database-connection-statements

ACCDESC= Statement

Creates an access descriptor based on the new DBMS table

Syntax

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

Syntax

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 nonnegative integer.

DELETE Statement

Does not load specified variables into the new table

Syntax

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 this 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 DBMS-specific details for possible exceptions)

Syntax

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 nonnegative 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

Default: DBMS column names default to SAS variable names

Interaction: RESET

Syntax

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.

You must use the RESET on page 911 statement *after* the LABEL statement for the LABEL statement to take effect.

LIMIT= Statement

Limits the number of observations that are loaded

Default: 5000

Syntax

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 nonnegative integer. To load all observations from your input data set, specify LIMIT=0.

LIST Statement

Lists information about the variables to be loaded

Default: ALL

Syntax

LIST <ALL | FIELD | *variable-identifier*>;

Optional Arguments

ALL
: lists information about all variables in the input SAS data set, despite whether 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 this statement: **`list 3;`**

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.

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

Restriction: in the DBLOAD procedure (required statement for loading or appending data)

Syntax

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 in Table A2.1 on page 902.

Example

This 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

Syntax

NULLS *variable-identifier-1* = Y | N <…*variable-identifier-n* = Y | N>;

Details

Some DBMSs have three valid values for this statement, Y, N, and D. See the DBMS-specific reference in this document for details about your DBMS.

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, no observations that contain missing values in the corresponding SAS variable are loaded into the table. A message is written to the SAS log, and the current error count increases by one for each observation that is not loaded. See ERRLIMIT= statement on page 907 for more information.

If a character column contains blanks (the SAS missing value) and you have specified N for the DBMS column, 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 this 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

Restriction: Valid in the DBLOAD procedure (control statement)

Syntax

QUIT;

Details

The QUIT statement terminates the DBLOAD procedure without further processing.

RENAME Statement

Renames DBMS columns

Interaction: DELETE, LABEL, RESET

Syntax

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 DBMS column names default to the corresponding SAS variable names unless you specify the LABEL statement.

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 where to place the variable in the data set. For example, submit this statement if you want to rename the column associated with the third SAS variable:

```
rename 3=employeename;
```

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 this syntax: **`rename 3='"employeename"'`**

The RENAME statement enables you to include variables that you have previously deleted. For example, suppose you submit these 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 on page 908 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

Interaction: DELETE, LABEL, RENAME, TYPE

Syntax

RESET ALL | *variable-identifier-1* <*…variable-identifier-n*>;

Details

The RESET statement resets columns that are associated with the listed SAS variables to default values for the 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. Here are the default values.

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 this statement:

```
reset 3;
```

You must use the RESET statement *after* the LABEL on page 908 statement for the LABEL statement to take effect.

SQL Statement

Submits a DBMS-specific SQL statement to the DBMS

Syntax

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, despite whether 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

This 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

Syntax

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 DBMS-specific reference in this document for the syntax for your DBMS.) 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

Syntax

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 this 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 DBMS-specific reference in this document for a complete list of default conversion data types for the DBLOAD procedure for your DBMS.

WHERE Statement

Loads a subset of data into the new table

Syntax

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. This 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 System Options: Reference*.

Example: Append a Data Set to a DBMS Table

Features: PROC DBLOAD option
- APPEND

Statements
- PROC DBLOAD
- TABLE=
- COMMIT=
- ERRLIMIT=

Database connection statements
- USER
- password
- PATH

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.

The following example appends new employee data from the NewEmp SAS data set to the Employees DBMS table. 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;
```

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 SAS, by a database management system, or by a PC-based software application such as Microsoft Excel, Lotus 1-2-3, or dBASE. After creating an access descriptor, you can use it as the basis for creating one or more view descriptors.

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
an application that requests either resources or services from a server, possibly over a network.

column
a vertical component of a table. Each column has a unique name, contains data of a specific type, and has particular 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 that makes permanent any changes to the database that the user made during the transaction.

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.

DATA step view
a type of SAS data set that consists of a stored DATA step program. 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
an attribute of every column in a table or database. The data type tells the operating system how much physical storage to set aside for the column, and specifies what type of data the column will contain. It is similar to the type attribute of SAS variables.

data value
a unit of character, numeric, or alphanumeric information that is stored as a single item in a data record.

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
a software application that enables you to create and manipulate data that is stored in the form of databases. Short form: DBMS.

DB2 catalog
a set of tables that DB2 maintains for its own use. These tables contain information about the other objects your DB2 system manages.

DBMS
a software application that enables you to create and manipulate data that is stored in the form of databases. Short form: DBMS.

engine
a component of SAS software that reads from or writes to a file. Various engines enable SAS to access different types of file formats.

explicit pass-through
a form of the SQL pass-through facility that passes the user-written, DBMS-specific SQL query code directly to the particular DBMS for processing.

foreign key
a SAS/ACCESS file that describes data that is managed by SAS, by a database management system, or by a PC-based software application such as Microsoft Excel, Lotus 1-2-3, or dBASE. After creating an access descriptor, you can use it as the basis for creating one or more view descriptors.

format
a type of SAS language element that is used to write or display data values according to the data's type: numeric, character, date, time, or timestamp. Short form: format.

function
a type of SAS language element that is used to process one or more arguments and then to return a result that can be used in either an assignment statement or an expression.

Hadoop Distributed File System
a framework for managing files as blocks of equal size, which are replicated across the machines in a Hadoop cluster to provide fault tolerance.

HDFS
See Hadoop Distributed File System.

implicit pass-through
a form of the SQL pass-through facility that translates SAS SQL query code to the DBMS-specific SQL code, enabling the translated code to be passed to the particular DBMS for processing.

index
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.

informat
a type of SAS language element that is used to read data values according to the data's type: numeric, character, date, time, or timestamp. Short form: informat.

interactive line mode
a method of running SAS programs in which you enter one line of a SAS program at a time at the SAS session prompt. SAS processes each line immediately after you press the ENTER or RETURN key. Procedure output and informative messages are returned directly to your display device.

interface view engine
a type of SAS engine that SAS/ACCESS software uses 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).

key column
in Oracle, a column in a table that is used to create an index on the table. The column contains a unique data value for each row in a table.

libref
a SAS name that is associated with the location of a SAS library. For example, in the name MYLIB.MYFILE, MYLIB is the libref, and MYFILE is a file in the SAS library.

line mode
a method of running SAS programs in which you enter one line of a SAS program at a time at the SAS session prompt. SAS processes each line immediately after you press the ENTER or RETURN key. Procedure output and informative messages are returned directly to your display device.

member
a SAS file in a SAS library.

member name
a name that is assigned to a SAS file in a SAS library.

member type
a SAS name that identifies the type of information that is stored in a SAS file. Member types include ACCESS, AUDIT, DMBD, DATA, CATALOG, FDB, INDEX, ITEMSTOR, MDDB, PROGRAM, UTILITY, and VIEW.

missing value
a type of value for a variable that contains no data for a particular row or column. By default, SAS writes a missing numeric value as a single period and a missing character value as a blank space.

null value
a special value that indicates the absence of information. Null values are analogous to SAS missing values.

object
any entity that can be manipulated by the commands of a programming language. Examples are values, variables, functions, and data structures.

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 either one data value or a missing-value indicator for each variable.

pass-through facility
the technology that enables SQL query code to be passed to a particular DBMS for processing. Short form: pass-through facility.

predicate
a component of a SAS WHERE statement or a component of an SQL WHERE or HAVING clause that is used in view descriptor selection criteria.

PROC SQL view
a SAS data set 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. The output of a PROC SQL view 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.

RDBMS
a database management system that organizes and accesses data according to relationships between data items. The main characteristic of a relational database management system is the two-dimensional table. Examples of relational database management systems are DB2, Oracle, Sybase, and Microsoft SQL Server.

referential integrity
a set of rules that a DBMS uses to ensure that a change to a data value in one table also results in a change 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. The main characteristic of a relational database management system is the two-dimensional table. Examples of relational database management systems are DB2, Oracle, Sybase, and Microsoft SQL Server.

rollback
a data recovery process that restores a database after a hardware or software failure, or that returns it to a state before changes were made.

row
in relational database management systems, the horizontal component of a table. A row is analogous to a SAS observation.

SAS/ACCESS view
a type of file that retrieves data values from files that are stored in other software vendors' file formats. You use the ACCESS procedure of SAS/ACCESS software to create SAS/ACCESS views.

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.

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 view
a type of SAS data set that retrieves data values from other files. A SAS data view contains only descriptor information such as the data types and lengths of the variables (columns) plus other information that is required for retrieving data values from other SAS data sets or from files that are stored in other software vendors' file formats. Short form: data view.

SAS file
a type of SAS data set that retrieves data values from other files. A SAS data view contains only descriptor information such as the data types and lengths of the variables (columns) plus other information that is required for retrieving data values from other SAS data sets or from files that are stored in other software vendors' file formats. Short form: data view.

SAS format
a type of SAS language element that is used to write or display data values according to the data's type: numeric, character, date, time, or timestamp. Short form: format.

SAS function
a type of SAS language element that is used to process one or more arguments and then to return a result that can be used in either an assignment statement or an expression.

SAS informat
a type of SAS language element that is used to read data values according to the data's type: numeric, character, date, time, or timestamp. Short form: informat.

SAS library
one or more files that are defined, recognized, and accessible by SAS, and that are referenced and stored as a unit. Each file is a member of the library.

SAS metadata
metadata that is created by SAS software. Metadata that is in SAS Open Metadata Architecture format is one example.

SAS variable
a column in a SAS data set or in a SAS data view. The data values for each variable describe a single characteristic for all observations (rows).

serde
an interface that enables serialization or deserialization of one or more file formats. [plural: serdes]

server
software that provides either resources or services to requesting clients, possibly over a network.

SQL
a standardized, high-level query language that is used in relational database management systems to create and manipulate objects in a database management system. SAS implements SQL through the SQL procedure. Short form: SQL.

SQL pass-through facility
the technology that enables SQL query code to be passed to a particular DBMS for processing. Short form: pass-through facility.

SSID
the unique name that identifies each DB2 subsystem on your machine. Short form: SSID.

Structured Query Language (SQL)
a standardized, high-level query language that is used in relational database management systems to create and manipulate objects in a database management system. SAS implements SQL through the SQL procedure. Short form: SQL.

subsystem ID
the unique name that identifies each DB2 subsystem on your machine. Short form: SSID.

table space
a named DB2 object that identifies sets of pages where records for one or more tables are stored. (A page is a 4K-byte or 32K-byte unit of storage within a table space.)

temporal data
event data that occurs at a particular date and time, such as an account inquiry. Temporal data is often referred to as time-sensitive data.

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.

term
an attribute of every column in a table or database. The data type tells the operating system how much physical storage to set aside for the column, and specifies what type of data the column will contain. It is similar to the type attribute of SAS variables.

variable
a column in a SAS data set or in a SAS data view. The data values for each variable describe a single characteristic for all observations (rows).

view
a definition of a virtual data set that is named and stored for later use. A view contains no data; it merely describes or defines data that is stored elsewhere.

view descriptor
a SAS/ACCESS file that defines part or all of the DBMS data that is described by an access descriptor.

Index

B

E

J

M

P

T