SAS/ACCESS® for Relational Databases with SAS® Viya™ 3.2: Reference

SAS® Documentation

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *SAS/ACCESS® for Relational Databases with SAS® Viya™ 3.2: Reference*. Cary, NC: SAS Institute Inc.

SAS/ACCESS® for Relational Databases with SAS® Viya™ 3.2: Reference

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

May 2017

3.2-P1:acreldb

Contents

Part 1

Concepts

Chapter 1

Overview of SAS/ACCESS Interface to Relational Databases

About the SAS/ACCESS Documentation

This documentation provides conceptual, reference, and usage information for SAS/ACCESS software for data sources for relational database management systems (DBMSs), data warehouse appliances, and distributed systems. The current release supports the SAS/ACCESS Interface to Hadoop. However, some conceptual information refers generically to DBMSs and data sources. Additional SAS/ACCESS interfaces are coming in future releases.

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).
- 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 websites 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. 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. SQL statements are passed directly to the data source for processing.

For information about when to use each method, see the following section.

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 or the SQL pass-through facility. Before processing complex or data-intensive operations, you might want to test different 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, 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 Optimizing Your SQL Usage on page 31.
- 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 that 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. Summary functions include AVG, COUNT, GROUP BY clauses, or columns that are created by expressions, such as those that use the COMPUTED function. The DBMS optimizer can use indexes on DBMS columns to process queries more rapidly.
- 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
Create DBMS objects, such as tables	LIBNAME statement*
	SQL pass-through facility EXECUTE statement
Update, delete, or insert rows into DBMS tables	LIBNAME statement*
	SQL pass-through facility EXECUTE statement
Append data to DBMS tables	LIBNAME statement and APPEND procedure*
	SQL pass-through facility EXECUTE statement
	SQL pass-through facility INSERT statement

Task	SAS/ACCESS Features
List DBMS tables	LIBNAME statement
	LIBNAME statement and DATASETS procedure*
	LIBNAME statement and CONTENTS procedure*
	LIBNAME statement and SQL procedure dictionary tables*

* *LIBNAME statement* refers to the SAS/ACCESS LIBNAME statement.

SAS Views of DBMS Data

SAS/ACCESS lets you create a SAS view of data that exists in a DBMS. 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.

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 one of the following is true:

- The view is based on only one DBMS table.
- The view is based on a DBMS view that is based on only one DBMS table, and 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 meets 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 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 $r \neq 0.0$. 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 Linux environments, the output from this DATA step is 4024333333333*333* (8/10-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 Linux (high-precision formatting).

```
x=-47.8
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.

Your Options When Choosing Your Needed Degree of Precision

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. For example, you can use this EQFUZZ 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. Alternatively, you can select a numeric DBMS data type to match the numeric SAS data type. For example, select the DOUBLE type in Hadoop (precise up to 15 digits) or the INT type instead of the BIGINT type (precise up to 38 digits).

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

This option retrieves numbers as character strings and preserves precision beyond 15 digits. For details, see the DBSASTYPE= data set option on page 83.

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 September 23, 2013.
- 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 September 23, 2013.

National Language Support

SAS/ACCESS provides National Language Support (NLS) in a variety of ways.

These data types allow for more flexible adjustment of column lengths.

- CHAR
- VARCHAR

For more NLS information, see these resources.

- For NLS limitations that are specific to Hive, see “Naming Conventions for SAS and Hive” on page 126.
- For additional NLS considerations, see the technical paper, “Processing Multilingual Data with the SAS 9.2 Unicode Server.” This paper is available at this URL:

  http://support.sas.com/resources/papers/92unicodesrvr.pdf
- For more comprehensive NLS information, see *SAS Viya National Language Support: Reference Guide*.

Chapter 2

Using SAS/ACCESS with SAS Viya

How SAS/ACCESS Works within SAS Viya

When you use SAS Viya, most of your analysis takes place using SAS Cloud Analytic Services (CAS). CAS holds your data in memory for fast access, and the CAS engine enables distributed processing of large amounts of data. Because your data must be loaded into CAS, use your data connector to load your data into a caslib that you add in a CASLIB statement. If you have licensed a data connect accelerator for your interface, use it to load data in parallel into CAS.

Your traditional SAS/ACCESS interface enables you to establish a connection, via a libref that you define in a LIBNAME statement, between the SAS client and your external data source. After you define this connection to your data source, you can send SQL commands by using the SQL procedure to manipulate the data in your external data source. You can also write data back to your data source with a DATA step.

Note: It is possible to read data into the SAS client by means of a SAS/ACCESS LIBNAME statement. However, you still must load your data into CAS for processing. Therefore, loading data using a LIBNAME statement is not recommended.

You also define a libref to connect the SAS client and the CAS server. This connection enables you to access any results that you generate using the CAS server. If you want to write any data from your CAS session back to your DBMS, you must establish this connection between the SAS client and the CAS server.

SAS/ACCESS Usage in SAS Viya

Tools That Are Available

SAS/ACCESS interface
: enables you to define a connection to your external data source. Use the LIBNAME statement to connect to a data source location.

  The SAS/ACCESS interface also enables you to define a connection to the CAS server so that you can access data in CAS tables. Use the CAS LIBNAME statement to connect to a CAS server.

data connector or data connect accelerator
: enables you to load data efficiently into a caslib. For more information, see "Where to Specify Data Connector Options" in *SAS Cloud Analytic Services: Language Reference*.

SQL procedure
: enables you to manipulate data in your external data source. Use PROC SQL to perform joins or filter your data before loading it into a caslib. For more information, see the information about using the SQL pass-through facility for your data source or *SAS Viya SQL Procedure User's Guide*.

Suggested Workflow

Here are the general steps to manage and manipulate your data from an external data source.

1. Perform any necessary data preparation before loading data into CAS. For example, if you want to load the join result of two Hive tables into CAS, create a Hive table that is the result of the join. You can then load the resulting table into CAS. Perform the data preparation by interacting directly with your data source or by using the SQL procedure in SAS.
2. Load the data into CAS using a data connector or data connect accelerator. For more information, see "Where to Specify Data Connector Options" in *SAS Cloud Analytic Services: Language Reference*.
3. Perform analysis on your data. You can use the DATA step, CAS actions, and procedures on your data. Save results to CAS tables in a caslib. For more information, see *SAS Viya Quick Start* and *SAS Cloud Analytic Services: Fundamentals*.
4. (Optional) Use a DATA step to write results back to your external data source. For example, assume that you have defined a libref C that connects the SAS client to the CAS server, and you have defined a libref H that connects the SAS client to an external Hadoop data source. You can then use a result table from libref C, Casresults, as input to a DATA step that writes back to the Hadoop libref H in a table called Myresults.

```
/* add a caslib for loading and analyzing Hadoop data */
caslib casdata datasource=(srctype='hadoop',
                           dataTransferMode='parallel',
                           server='HiveServer',
                           username='user1'
```

```
                          hadoopJarPath="//root/myJars",
                          hadoopConfigDir="//sasroot/myConfig");

/* define a libref to connect the SAS client with the CAS server */
/* by default, the connection goes to the active caslib           */
libname c cas;

/* define a libref to connect the Hadoop database with the SAS client */
libname h hadoop user=user1 pwd=mypwd1 server='HiveServer' schema=statsdiv;

/* add code to load data and run analyses here */

/* write results table back out to Hadoop */
data h.myresults;
   set c.casresults;
run;
```

Chapter 3

SAS Names and Support for DBMS Names

DBMS-Specific Naming Conventions

Some DBMSs allow case-sensitive names and names with special characters. As a result, keep the considerations in this chapter in mind when you use the names of DBMS objects such as tables and columns with SAS/ACCESS features.

For information about how SAS handles your DBMS names, see the DBMS-specific reference section for your SAS/ACCESS interface.

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.

Some SAS names can be up to 32 characters, depending on the SAS session encoding and the limits of the object name length of the DBMS. These SAS names can be up to 32 characters:

- 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
- variable names in SAS/ACCESS access descriptors

Case Sensitivity

When SAS encounters mixed-case or case-sensitive names in SAS code, SAS stores and displays the names as they are specified. In the following example, two SAS variables (Flight and dates) are defined in mixed case.

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

SAS then displays the variable names as defined, and the column headings appear as defined.

Output 3.1 *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 that is followed by the letter **n**. By using name literals, you can 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 a name literal 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 100.

SAS/ACCESS Default Naming Behaviors

Modification and Truncation of Column Names

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 column 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.

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
: This option applies only to creating DBMS tables. When set to YES, it preserves spaces, special characters, and mixed case in DBMS column names. For more information, see the PRESERVE_COL_NAMES= LIBNAME and data set options.

PRESERVE_TAB_NAMES=YES
: 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 you time when you are coding. For more information, see the PRESERVE_TAB_NAMES= LIBNAME option.

VALIDVARNAME=ANY
: This global system option can override SAS naming conventions. For more information, see the VALIDVARNAME= system option on page 100.

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 do not need to specify these options because the option default values are DBMS-specific. For details, see the DBMS-specific reference section for your SAS/ACCESS interface.

Table 3.1 *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 3.2 *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 SQL DQUOTE=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 do not need to specify these options because the option default values are DBMS-specific. For details, see the documentation for your DBMS.

Table 3.3 *SAS Variable Names to DBMS Column Names When Creating Tables*

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

Table 3.4 *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

Chapter 4

Data Integrity and Security

DBMS Security

Privileges

Database administrators control who has privileges to access or update DBMS objects. They also control who can create objects, and object creators control who can access the objects. Users cannot use DBMS facilities to access DBMS objects through SAS/ACCESS software unless they have the appropriate DBMS privileges or authority on those objects. You can grant privileges on the DBMS side by using the SQL pass-through facility to EXECUTE an SQL statement.

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.

For more information about ensuring security on the DBMS side of the interface, see your DBMS documentation.

SAS Security

Securing Data

SAS preserves the data security provided by your DBMS and operating system. However, SAS/ACCESS does not override the security of your DBMS. You can secure

DBMS data from accidental update or deletion from the SAS side of the interface by creating SQL views, and you can protect them from unauthorized access by applying passwords.

These and other approaches are discussed in detail in subsequent sections.

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 between Hive and SAS/ACCESS Interface to Hadoop is to use Kerberos authentication. When you use Kerberos, there are no sensitive user names and passwords to protect. For more information, see "Hadoop Configuration" on page 116.

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. To restrict access to some columns, specify that those columns be dropped. Columns that are dropped from views do not affect the underlying DBMS table and can be reselected for later use.

Controlling DBMS Connections

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

The CONNECTION= LIBNAME option enables you to specify how many connections are executed when the library is used and which operations on tables are shared within a connection. For many DBMSs, the default 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 PROC SQL queries the table, the query uses a second connection to the DBMS.

```
libname myhadlib hadoop server=hadoopsvr user=myusr1 password=mypwd1;

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

```
proc sql;
   select date1 from myhadlib.mytable;
quit;
```

If you do not want a connection to occur when a 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 myhadlib hadoop server=hadoopsvr user=user1 pass=mypwd1 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. The processing is faster because the connections to the DBMS are not made every time SAS is invoked.

Customizing a DBMS Connection

To specify DBMS commands or stored procedures to run immediately after a DBMS connection, use the DBCONINIT= LIBNAME option.

When a libref is assigned, the SAS/ACCESS engine connects to the DBMS and passes a command that you specify to the DBMS to execute. By default, a new connection to the DBMS is made for every table that is opened for updating. Therefore, the command is executed an additional time after a connection is made to update a table.

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 results 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. For more information, see "Sorting DBMS Data" on page 29.

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

```
libname myhadlib hadoop user=myusr1 password=mypwd1;
data myhadlib.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=myhadlib.table;
   where x<0;
run;
```

The log indicates that the WHERE clause selected no observations. Hadoop 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= LIBNAME option to control whether SAS or the DBMS handles processing.
- Use the SQL pass-through facility 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.

You can use the first of these strategies to force SAS to process the data in this example.

```
libname myhadlib hadoop user=myusr1 password=mypwd1
        direct_sql=nowhere;   /* forces SAS to process WHERE clauses */

data myhadlib.table;
x=.;       /*create a missing value */
run;
```

You can then print the data set using a WHERE clause

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

```
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 5

Performance Considerations

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 limit 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 Employees table.

```
libname myhadlib hadoop user=user1 password=mypwd1 server=hadoopsvr;

proc print data=myhadlib.employees (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 columns 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 myhadlib hadoop user=testid password=testpass server=hadoopsvr;

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

Here is how you can use the KEEP= data set option to retrieve only the SALARY column.

```
data temp;
   set myhadlib.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. SAS data files are organized to provide optimal performance with PROC and DATA steps. Programs that use SAS data files often outperform SAS programs that read DBMS data.

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.

You can extract DBMS data to a SAS data file by using the DATA step or the OUT= option in a SAS procedure.

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 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. This includes those specified by the DBCONDITION= option, in a WHERE clause, and in the selection criteria in a view descriptor. In the following example, the Exec_Employees data set includes a BY statement that sorts data by the HIREDATE variable. However, when that data set is used in the following PROC SQL query, the data is ordered by the SALARY column, not by HIREDATE.

```
libname mydblib hadoop server=hadoopsvr database=compdata user=user1
           pass=mypwd1;
data exec_employees;
   set mydblib.employees (keep=lastname firstname empid salary hiredate);
   by hiredate;
   where salary >= 75000;
run;

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

- 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.
- Do not use the SORTSEQ= system option because this option has no effect on the sort order of the data. Whenever possible, SAS allows sorting to be performed by the DBMS to improve performance. The values for the SORTSEQ= system option apply to processing that is performed by SAS. Therefore, the option has no impact when data is sorted by the DBMS.
- 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 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.

The value of DBMS temporary table support in SAS is increased performance potential. By pushing the processing to the DBMS in certain situations, you can achieve an overall performance gain. The processes in this section 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.

Chapter 6

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 number of rows. SAS/ACCESS sends operations to the DBMS for processing in these situations.

- When you use the SQL pass-through facility, you submit DBMS-specific SQL statements that are sent directly to the DBMS for execution.
- When SAS/ACCESS can translate the operations into the SQL of the DBMS. When you use the SAS/ACCESS LIBNAME statement, 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. For information about passing WHERE clauses to the DBMS, see “Passing the WHERE Clause to the DBMS”.

Note: Some conditions prevent operations from being passed to the DBMS. For example, when you use an INTO clause or most data set options, 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 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.

Passing Functions to the DBMS Using PROC SQL

Overview of Passing Functions to a DBMS

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 Hadoop Emp table that is being passed to the DBMS for processing.

```
libname mydblib hadoop server=hadoopsvr database=company user=user1 password=mypwd1;
proc sql;
   select count(*) from mydblib.emp;
quit;
```

This code causes the DBMS to process this query.

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

When a function is located in the WHERE clause of the SQL statement, SAS/ACCESS can 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 Hadoop UPPER function.

```
libname mydblib hadoop server=hadoopsvr database=company user=user1 password=mypwd1;
proc sql;
   select customer from mydblib.customers
   where upcase(country)="USA";
quit;
```

Here is the translated query that the DBMS processes.

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

See the SQL pass-through section in the DBMS-specific reference for your SAS/ACCESS interface for a list of functions that it translates.

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 44 for the librefs. Certain criteria must be met for the join to proceed. See the passing joins section in the DBMS-specific

reference for your SAS/ACCESS interface to see the criteria that it requires before PROC SQL can pass the join.

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.

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 data sources (DBMS engines), 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 most data set options on a table that is referenced in an SQL query prohibits SAS from successfully passing the statement to the DBMS for direct processing.

Note: The PRE_STMT_OPTS=, POST_STMT_OPTS=, PRE_TABLE_OPTS=, and POST_TABLE_OPTS= data set options are exceptions, because SAS passes the arguments for these options on to the DBMS for 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 default setting for the DIRECT_SQL= LIBNAME option 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= 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= 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.

Passing the WHERE Clause to the DBMS

General Guidelines for WHERE Clauses

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

Table 6.1 *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`**

Guideline	Inefficient	Efficient
Avoid LIKE predicates that begin with % or _ .	Inefficient: **`where COUNTRY like '%INA'`**	Efficient: **`where COUNTRY like 'A %INA'`**
Avoid arithmetic expressions in a predicate.	Inefficient: **`where SALARY>12*4000.00`**	Efficient: **`where SALARY>48000.00`**
Use MULTI_DATASRC_OPT= when appropriate.		

Whenever possible, SAS/ACCESS passes WHERE clauses to the DBMS. SAS translates the WHERE clauses into generated SQL code. The performance impact is 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 31.

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.

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.

See the DBMS-specific reference section for your SAS/ACCESS interface to determine which functions it translates.

Using the MULTI_DATASRC_OPT= Option

The MULTI_DATASRC_OPT= option applies to an equijoin in PROC SQL that is processed in SAS and that is not passed to the DBMS. A common example is an equijoin of a SAS data set and a DBMS table. An *equijoin* is a join in which the join conditions check for equivalence. For example, you might use the following WHERE clause in an equijoin:

```
where bigtab.deptno=smallds.deptno
```

Without the MULTI_DATASRC_OPT= option, SAS retrieves all rows of the DBMS table into SAS and then performs the equijoin. If the DBMS table is very large, the join could take a long time. If only a small percentage of the DBMS rows match to the SAS data set, then MULTI_DATASRC_OPT= speeds the join operation. When you specify MULTI_DATASRC_OPT=IN_CLAUSE, PROC SQL constructs an SQL IN clause from

the unique join key values of the SAS data set. The IN clause retrieves only the matching database table rows.

In some situations, PROC SQL ignores MULTI_DATASRC_OPT=. For example, PROC SQL ignores MULTI_DATASRC_OPT= when all tables being joined are DBMS tables. In this case, PROC SQL pushes the join to the DBMS.

In the following example, the MULTI_DATASRC_OPT= option is used to improve the performance of an SQL join. PROC SQL creates an IN clause from the unique values of the SAS DEPTNO variable. As a result, only rows that match the IN clause are retrieved from the large DBMS table Dept.

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

libname dblib hadoop user=myusr1 password=mypwd1
     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;
```

Chapter 7

How SAS/ACCESS Works

Introduction to How SAS/ACCESS Works

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.

You can call a SAS/ACCESS relational DBMS interface by using either a LIBNAME statement or a PROC SQL statement.

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 are the steps.

1. Start a SAS/ACCESS interface by specifying a DBMS engine name and the appropriate connection options in a LIBNAME statement.
2. 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 customized for the DBMS.

Processing Queries, Joins, and Data Functions

To enhance performance, SAS/ACCESS can 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 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.

For details about tasks that SAS/ACCESS can pass to the DBMS, see the DBMS-specific reference section for your SAS/ACCESS interface.

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 (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.
3. Use a PROC SQL EXECUTE statement to pass any dynamic, nonquery SQL statements (such as INSERT) 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.
4. End the connection with the DISCONNECT statement.

For more details, see the SQL pass-through facility on page 104.

Part 2

General Reference

Chapter 8

The LIBNAME Statement

Overview: LIBNAME Statement for Relational Databases

Assigning Librefs

The SAS/ACCESS LIBNAME statement extends the SAS global LIBNAME statement so that you can 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.

For details about the syntax, see "LIBNAME Statement Syntax for Relational Databases" on page 45. For the engine name, connection options, and LIBNAME options for your SAS/ACCESS interface, see the DBMS-specific reference section for your SAS/ACCESS interface.

For details about defining and naming librefs, see "LIBNAME Statement" in *SAS Viya Statements: Reference*.

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 in 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 being 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 SAS translates into 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 Viya Functions and CALL Routines: Reference*.

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"

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
specifies 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
specifies the SAS/ACCESS engine name for your DBMS, such as **`hadoop`**. 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. For example, to connect to a Hadoop database, your connection options are USER=, PASSWORD=, and SERVER=:

```
libname mydblib hadoop user=myusr1 password=mypwd1 server=hadoopsvr;
```

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 connection option details, see the DBMS-specific information for your SAS/ACCESS interface.

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. See the DBMS-specific reference section for LIBNAME options that are available for your SAS/ACCESS interface. For general information, see LIBNAME Options for Relational Databases on page 48.

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 Hadoop table Dept is created. Whenever you use this view in a SAS program, the Hadlib 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 hadlib.dept
          using libname hadlib hadoop
                user=scott pw=tiger datasrc=hadsrv;
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 *SAS Viya SQL Procedure User's Guide*.

Assigning a Libref with a SAS/ACCESS LIBNAME Statement

This statement creates a libref, MYDBLIB, that uses the SAS/ACCESS Interface to Hadoop.

```
libname mydblib hadoop server=hadoopsvr user=user1 password=mypwd1;
```

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

```
libname mydblib hadoop user=myusr1 password=mypwd1
        server=hadoopsvr 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 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 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;
```

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 71. For general information, see "LIBNAME Statement Syntax for Relational Databases" on page 45.See the DBMS-specific reference section for LIBNAME options that are available for your SAS/ACCESS interface.

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, see "SQL Pass-Through Facility" on page 103.

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

Category: Data Access

Default: none

Data source: Hadoop

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.

ANALYZE= LIBNAME Option

Lets SAS improve performance when a single Hive store table is queried.

Valid in: SAS/ACCESS LIBNAME statement

Category: Data Access

Default: NO

Data source: Hadoop

See: ANALYZE= data set option, READ_METHOD= LIBNAME option, READ_METHOD= data set option

Syntax

ANALYZE=YES | NO

Syntax Description

YES
specifies that SAS might run an ANALYZE TABLE command to update table statistics. Current table statistics might improve SAS Read performance when a single table is queried. This operation is considered a hint, so if statistics are up-to-date, SAS might not perform the operation. The format of the ANALYZE TABLE command is subject to change in future releases of SAS as needed.

NO
specifies that SAS does not perform an additional operation to update table statistics.

Details

Performance improves when Hive statistics are up-to-date. When the Hive Boolean variable hive.stats.autogather is set to TRUE, in most cases Hive automatically gathers statistics. This option can be useful when hive.stats.autogather is set to FALSE or when statistics are not being computed. Specifically, Hive statistics are not generated when loading a target table with a file format of TEXT. Users can check the state of Hive statistics using the Hive DESCRIBE FORMATTED command.

BL_PORT= LIBNAME Option

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

Valid in: SAS/ACCESS LIBNAME statement

Categories: Bulk Loading
Data Set Control

Alias: BLPORT=, BULKLOAD_PORT=

Default: 8020

Restriction: This option is required if Hadoop HDFS service is running on a port other than 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

Use this option to specify only the port number that bulk loading uses to write data to the Hadoop cluster using the HDFS streaming server.

BULKLOAD= LIBNAME Option

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

Valid in: SAS/ACCESS LIBNAME statement

Categories: Bulk Loading
Data Set Control

Default: NO

Data source: Hadoop

See: BULKLOAD= 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.

CONFIGDIR= LIBNAME Option

Specifies a directory path to search for the required XML configuration files to use for bulk-load options.

Valid in: SAS/ACCESS LIBNAME statement

Category: Data Set Control

Aliases: CFGDIR=
HD_CONFIGDIR=

Default: none (SAS_HADOOP_JAR_PATH is the default when you set only BULKLOAD=YES)

Requirement: You must enclose the value in quotation marks.

Data source: Hadoop

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

Syntax

CONFIGDIR='*config-dir*'

Required Argument

config-dir
specifies the directory path to search for the required configuration files (core-site.xml and hdfs-site.xml) for bulk-load options that contain the required host, port, and Kerberos principal for HDFS.

CONNECTION= LIBNAME Option

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

Valid in: SAS/ACCESS LIBNAME statement
CONNECT statement

Category: Data Set Control

Default: UNIQUE

Interaction: When the default is CONNECTION=UNIQUE, the LIBNAME connection can fail when you use SQL_FUNCTIONS= to store the external SQL dictionary.

Data source: Hadoop

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

Syntax

CONNECTION=UNIQUE | SHARED

Syntax Description

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

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

SHARED
specifies that *all* operations that access DBMS tables *in a single libref* share a single connection if the conditions for sharing a connection are met. For more information, see "Conditions for a Shared DBMS Connection".

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

Tip 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 tablespace. This happens only in certain output processing situations and is the only recommended use for CONNECTION=SHARED.

Details

Overview of the CONNECTION= LIBNAME Option

The main reason for using the CONNECTION= LIBNAME option is to control the number of physical connections to your DBMS. When you specify that you want to share DBMS connections, you enable SAS to use one physical connection across multiple DATA steps and procedure calls. In this way, you limit the number of physical connections for your SAS session. Sharing a connection also enables you to share access to temporary tables across DATA steps and procedure calls.

For most SAS/ACCESS interfaces, there must be a connection, also known as an *attach*, to the DBMS server before a user can access any data. Typically, a 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, SQL pass-through CONNECT statement, or both.

Conditions for a Shared DBMS Connection

If you want to share a connection across librefs, the critical connection options that you specify must be the same. You can specify the connection options in a LIBNAME statement or in the CONNECT statement in an SQL procedure call. When SAS/ACCESS compares connection option values, it does not matter whether you use optional quotation marks across libref declarations.

Here are the conditions that must be met to share a physical connection to your DBMS:

- These connection options must have the same value for each libref declaration:

  USER=
  DATABASE=
  PASSWORD=
  SCHEMA=
  SERVER=

- These LIBNAME options must have the same value for each libref declaration:

  DATABASE=
  DBCONTERM=
  SCHEMA=
  DBLIBINIT=
  CONNECTION=
  DBLIBTERM=
  CONNECTION_GROUP=
  SQL_FUNCTIONS=
  DBCONINIT=

If any of these conditions are not met, SAS/ACCESS automatically creates additional physical connections to the DBMS.

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

Category: Data Set Control

Default: none

Data source: Hadoop

See: SCHEMA= LIBNAME option, SCHEMA= data set 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).

Example: Treat Backslash Characters as Literals

In this example, specify that a backslash character ('\') should be read as a literal character rather than as an escape character. By default, the DBMS variable that controls how the backslash is read is disabled, resulting in a backslash being treated as an escape character. If this is not the desired behavior (such as when specifying a directory path), you can change the behavior.

DBCREATE_TABLE_EXTERNAL= LIBNAME Option

Specifies the type of table to create and how associated data files are handled.

Valid in: SAS/ACCESS LIBNAME statement

Category: Data Set Control

Aliases: 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
creates an *external* table—one that is stored outside of the Hive warehouse.

NO
creates a *managed* table—one that is managed within the Hive warehouse.

Details

When a managed table is dropped, its data is also deleted. When an external table is dropped, its data is preserved. Create an EXTERNAL table if you want to preserve table data if the table is dropped. SAS issues a DROP TABLE statement when PROC DELETE references a Hive table and also with the DROP TABLE statement in PROC SQL.

Example: Protect Data from DROP TABLE

In this example, DBCREATE_TABLE_LOCATION= stores the table data outside of the Hive warehouse. DBCREATE_TABLE_EXTERNAL=YES protects the data from being deleted if the table is dropped.

```
LIBNAME db HADOOP SERVER=mysrv1 USER=myusr1 DB=myschema1;
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 and CONNECT statements

Category: Data Set Control

Alias: POST_STMT_OPTS=

Default: none

Data source: Hadoop

Tips: If you are already using DBTYPE= within an SQL CREATE TABLE statement, you can also use it to include column modifiers.

If you want all output tables to be in the default (non-TEXTFILE) format, see the examples in this section.

See: DBCREATE_TABLE_EXTERNAL= LIBNAME option, DBCREATE_TABLE_EXTERNAL= data set option, DBCREATE_TABLE_LOCATION= data set option, DBCREATE_TABLE_OPTS= data set option, DBTYPE= data set option, POST_STMT_OPTS= data set option, POST_TABLE_OPTS= data set option, PRE_STMT_OPTS= data set option, PRE_TABLE_OPTS= data set option

Syntax

DBCREATE_TABLE_OPTS='*DBMS-SQL-clauses*'

Required Argument

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

Details

You can use this option 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. 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 that is associated with DBMS data.

If you need to add an option in a location other than at the end of your CREATE TABLE statement, use one of these data set options: POST_TABLE_OPTS=, PRE_STMT_OPTS=, and PRE_TABLE_OPTS=.

Examples

Example 1: Create All Hive Tables in ORC Format

```
libname x hadoop … DBCREATE_TABLE_OPTS="stored as ORC";
```

Example 2: Create All Hive Tables in RCFILE Format

```
libname x hadoop … DBCREATE_TABLE_OPTS="stored as RCFILE";
```

Example 3: Create All Hive Tables in SEQUENCEFILE Format

```
libname x hadoop … DBCREATE_TABLE_OPTS="stored as SEQUENCEFILE";
```

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
CONNECT statement

Category: Data Set Control

Default: DBMS

Data source: Hadoop

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.

Example

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

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
CONNECT statement

Category: Data Set Control

Default: 32767

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

Requirement: You must set the value to 4000 when you are using procedures that work with SAS High-Performance Analytics Server.

Data source: Hadoop

Note: If you set the value of DBMAX_TEXT= so that data in a table is truncated, the data load fails for that table. The number of bytes that are used to store characters might vary and is based on your session encoding.

See: DBMAX_TEXT= data set option

Syntax

DBMAX_TEXT=*integer*

Syntax Description

integer
an integer between 1 and 32,767.

Details

This option applies for the STRING data type.

DBMSTEMP= LIBNAME Option

Specifies whether SAS creates temporary or permanent tables.

Valid in: SAS/ACCESS LIBNAME statement

Category: Data Set Control

Default: NO

Requirement: To specify this option, you must first specify CONNECTION=GLOBAL.

Interaction: To access temporary tables, set DBMSTEMP= to YES and set the CONNECTION= LIBNAME option to GLOBAL.

Data source: Hadoop

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

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

DEFER= LIBNAME Option

Specifies when the connection to the DBMS occurs.

Valid in: SAS/ACCESS LIBNAME statement

CONNECT statement

Category: Data Access

Default: NO

Interaction: The DEFER= option is ignored when CONNECTION=UNIQUE because a connection is performed every time a table is opened.

Data source: Hadoop

See: CONNECTION= LIBNAME option

Syntax

DEFER=YES | NO

Syntax Description

YES
specifies that the connection to the DBMS occurs when a table in the DBMS is opened.

NO
specifies that the connection to the DBMS occurs when the libref is assigned by a LIBNAME statement.

DIRECT_SQL= LIBNAME Option

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

Valid in: SAS/ACCESS LIBNAME statement

Category: Data Set Control

Default: YES

Data source: Hadoop

See: SQL_FUNCTIONS= LIBNAME option

Syntax

DIRECT_SQL=YES | NO | NONE

DIRECT_SQL=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.

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
CONNECT statement

Category: Data Access

Default: 30

Data source: Hadoop

Syntax

LOGIN_TIMEOUT=*numeric-value*

Syntax Description

numeric-value
specifies a positive integer for the number of seconds to wait for the connection. A value of 0 indicates to wait indefinitely.

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

Category: Data Set Control

Default: none

Data source: Hadoop

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 you are 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 whether 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.

DIRECT_SQL= can affect 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.

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)

Category: Data Set Control

Alias: PRESERVE_NAMES= (see “Details”)

Default: NO

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.

Interaction: If you use the DS2 language, quoting and casing of names is different. For more information, see the identifiers topic in *SAS Viya: DS2 Language Reference*.

Data source: Hadoop

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 invalid 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 Viya 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.

The SAS/ACCESS engine automatically converts all schema, table, and column names to lowercase.

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 that are not valid SAS names in your SAS program, 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 hadoop user=myusr1 password=mypwd1
        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

Category: Data Set Control

Alias: PRESERVE_NAMES= [see “Details”]

Default: NO

Data source: Hadoop

See: PRESERVE_COL_NAMES= LIBNAME option, PRESERVE_TAB_NAMES= data set option, SAS/ACCESS naming, SCHEMA= LIBNAME option, naming conventions in the DBMS-specific reference section for your SAS/ACCESS interface

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, 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.
```

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

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 hadoop user=myusr1 password=mypwd1
        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 hadoop user=myusr1
        password=mypwd1 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.

PROPERTIES= LIBNAME Option

Specifies JDBC custom connection properties, which override the default JDBC connection properties.

Valid in: SAS/ACCESS LIBNAME and CONNECT statements

Category: JDBC connection options

Default: none

Restriction: If you are using Cloudera Hadoop, you can specify only one JDBC connection property when HiveServer2 is prior to Hive 0.11.

Data source: Hadoop

Syntax

PROPERTIES='*JDBC-connection-property-1*<;*JDBC-connection-property-2*…>

Syntax Description

JDBC-connection-property
specifies one or more JDBC connection options to override the default JDBC connection options.

Details

When you specify JDBC connection properties using the PROPERTIES= LIBNAME option, the properties are appended to the JDBC URL. This overrides the default properties. Site-wide Hive properties are specified in the hive-site.xml file in the Hive configuration directory. In the JDBC URL, custom properties are separated from the default properties by the question mark (?) character. The ? denotes the start of Hive configuration options. You do not need to add the ? character in the PROPERTIES= LIBNAME option. To specify Hive variables, add the character # before the Hive variable in the LIBNAME option.

Examples

Example 1: Set Strict Mode

```
libname h4 hadoop SUBPROTOCOL=HIVE2 schema=sample user=hdusr1 server="hdp2ga"
        properties='hive.mapred.mode=strict';
   jdbc:hive2://hdp2ga.unx.sas.com:10000?hive.mapred.mode=strict
```

Example 2: Set Strict Mode with a Second Option

```
libname h4 hadoop SUBPROTOCOL=HIVE2 schema=sample user=hdusr1 server="hdp2ga"
        properties='hive.mapred.mode=strict;hive.optimize.groupby=false';
   jdbc:hive2://hdp2ga.unx.sas.com:10000?hive.mapred.mode=strict;
   hive.optimize.groupby=false
```

Example 3: Set a Hive Variable

```
libname h4 hadoop SUBPROTOCOL=HIVE2 schema=sample user=hdusr1 server="hdp2ga"
        properties='#D_TBL=dummy_t';
   jdbc:hive2://hdp2ga.unx.sas.com:10000?#D_TBL=dummy_t
```

Example 4: Set Strict Mode and a Hive Variable

```
libname h4 hadoop SUBPROTOCOL=HIVE2 schema=sample user=hdusr1 server="hdp2ga"
       properties='hive.mapred.mode=strict#D_TBL=dummy_t';
   jdbc:hive2://hdp2ga.unx.sas.com:10000?hive.mapred.mode=strict#D_TBL=dummy_t
```

Example 5: Set Strict Mode and a Hive Principal

```
libname h4 hadoop SUBPROTOCOL=HIVE2 principal=hive/HiveServer2Host@YOUR-REALM.COM
        schema=sample user=hdusr1 server="hdp2ga"
        properties='hive.mapred.mode=strict#D_TBL=dummy_t';
   jdbc:hive2://hdp2ga.unx.sas.com:10000;
   principal=
```

```
hive/HiveServer2Host@YOUR-REALM.COM?hive.mapred.mode=strict#D_TBL=dummy_t
```

READ_METHOD= LIBNAME Option

Specifies how to read data.

Valid in: SAS/ACCESS LIBNAME statement

Category: Data Set Control

Default: none

Data source: Hadoop

See: ANALYZE= LIBNAME option, ANALYZE= data set option, READ_METHOD= data set option

Syntax

READ_METHOD=JDBC| | HDFS

Syntax Description

JDBC
specifies that data is to be read through the JDBC connection to the Hive service. You can use the ANALYZE= option to potentially improve performance when querying small tables.

HDFS
specifies that data is to be read through a connection to the Hadoop HDFS service.

Details

Although HDFS cannot alter the behavior of operations that always use JDBC, in general HDFS is a faster alternative to JDBC. To take advantage of potential performance benefits, set this option to HDFS. Use JDBC when you cannot access the HDFS service or JDBC Read offers some other advantage.

Example: Read Data Using JDBC

In this example, a partition of data from the sales Hive table is read using JDBC.

```
libname hdp hadoop server=mysrv1 user=myusr1 pwd=mypwd1;
data work.sales_subset; set hdp.sales(READ_METHOD=JDBC);
where year_month='2012-10'; run;
```

SCHEMA= LIBNAME Option

Allows reading of such database objects as tables and views in the specified schema.

Valid in: SAS/ACCESS LIBNAME statement

Category: Data Access

Default: the Hive schema named default

Data source: Hadoop

See: DBCONINIT= LIBNAME option, PRESERVE_TAB_NAMES= LIBNAME option, SCHEMA= data set 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 database for your DBMS.

SCRATCH_DB= LIBNAME Option

Specifies a Hive schema so that SAS can store temporary output.

Valid in: SAS/ACCESS LIBNAME statement

Category: Data Access

Default: none (the current schema is used)

Requirement: SAS must be able to write to the schema.

Data source: Hadoop

Syntax

SCRATCH_DB=*schema-name*

Syntax Description

schema-name
specifies the name that is assigned to a logical classification of objects in a relational database.

Details

This option lets the user specify a target schema that SAS can use for temporary output. It is needed when a user does not have permissions to perform DDL operations such as CREATE TABLE or DROP TABLE in the current schema.

Example

```
libname hdp hadoop server=hxpduped
   user=myusr1 password=mypwd1 scratch_db="tempdb";
```

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

Category: Data Set Control

Default: YES

Data source: Hadoop

See: CONNECTION= LIBNAME option

Syntax

SPOOL=YES | NO

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.

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.

SQL_FUNCTIONS= LIBNAME Option

Customizes the in-memory SQL dictionary function list for this particular LIBNAME statement.

Valid in: SAS/ACCESS LIBNAME statement

Category: Data Set Control

Default: none

Restrictions: You must specify a two-part data set name, such as <*libref.member*>. Otherwise, an error results.

<*libref.member*> must be a SAS data set. No check is performed to ensure that it is assigned to the default SAS engine.

Data source: Hadoop

Syntax

SQL_FUNCTIONS=ALL <*libref.member*> |
EXTERNAL_REPLACE=<*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*>
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*>
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 a DBMS-specific date format. So, be careful when you use this option.

See the reference section for Hadoop for the 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	•			Truncated to 32 characters if length is greater than 32
SASFUNCNAMELEN	•			Must correctly reflect the length of SASFUNCNAME
DBMSFUNCNAME	•			Truncated to 50 characters if length is greater than 50
DBMSFUNCNAMELEN	•			Must correctly reflect the length of DBMSFUNCNAME
FUNCTION_CATEGORY		•		AGGREGATE , CONSTANT, SCALAR

Variable	Required*	Optional**	Read-Only**	Valid Values
FUNC_USAGE_CONTEXT		●		SELECT_LIST, WHERE_ORDERBY
FUNCTION_RETURNTYP		●		BINARY, CHAR, DATE, DATETIME, DECIMAL, GRAPHIC, INTEGER, INTERVAL, NUMERIC, TIME, VARCHAR
FUNCTION_NUM_ARGS		●		0
CONVERT_ARGS			●	Must be set to 0 for a newly added function.
ENGINEINDEX			●	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.

TRANSCODE_FAIL= LIBNAME Option

Lets you specify how to handle processing and notification of transcoding errors.

Valid in: SAS/ACCESS LIBNAME statement

Category: Data Set Control

Default: ERROR

Restriction: This option is not available for use with SAS Embedded Process.

Data source: Hadoop

Tip: You can use TRANSCODE_FAIL= to determine whether you want to halt or continue processing when transcoding errors are encountered.

See: TRANSCODE_FAIL= data set option

Syntax

TRANSCODE_FAIL=<ERROR> | <WARNING> | <SILENT>

Optional Arguments

ERROR
stops processing data and provides an informative error message.

WARNING
continues processing of data but provides an informative error message.

SILENT
continues processing of data but suppresses messages.

Chapter 9

Data Set Options

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. 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 48.

Here is an example of how you can specify SAS/ACCESS data set options.

```
libname mydblib hadoop;
proc print mydblib.mytable(data-set-option=value);
```

You cannot use most data set options in a PROC SQL DROP (table or view) statement.

You can use the CNTLLEV=, DROP=, FIRSTOBS=, IN=, KEEP=, OBS=, 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 Viya 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 it 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 DBMS-specific reference section for your SAS/ACCESS interface.

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 31.

Dictionary

ANALYZE= Data Set Option

Lets SAS improve performance when a single Hive store table is queried.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Category: Data Set Control

Default: NO

Data source: Hadoop

See: ANALYZE= LIBNAME option, READ_METHOD= LIBNAME option, READ_METHOD= data set option

Syntax

ANALYZE=YES | NO

Syntax Description

YES
specifies that SAS might run an ANALYZE TABLE command to update table statistics. Current table statistics might improve SAS Read performance when a single table is queried. This operation is considered a hint, so if statistics are up-to-date, SAS might not perform the operation. The format of the ANALYZE TABLE command is subject to change in future releases of SAS as needed.

NO
specifies that SAS does not perform an additional operation to update table statistics.

Details

Performance improves when Hive statistics are up-to-date. When the Hive Boolean variable hive.stats.autogather is set to TRUE, in most cases Hive automatically gathers statistics. This option can be useful when hive.stats.autogather is set to FALSE or when statistics are not being computed. Specifically, Hive statistics are not generated when loading a target table with a file format of TEXT. Users can check the state of Hive statistics using the Hive DESCRIBE FORMATTED command.

BL_DELIMITER= Data Set Option

Specifies override of the default delimiter character for separating columns of data during data transfer or retrieval during bulk loading or unloading.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Categories: Bulk Loading
Data Set Control

Alias: DELIM, DELIMIT

Default: \001 (Ctrl-A)

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

Data source: Hadoop

See: BULKLOAD= data set option

Syntax

BL_DELIMITER='*<any-single-character>*'

Details

Overview

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 are transferred to or retrieved from the DBMS during bulk loading.
- if your character data contains the default delimiter character, to avoid any problems while parsing the data stream

Specifics

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. The value 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.

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_PORT= Data Set Option

Specifies the port number to use.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Categories: Bulk Loading
Data Set Control

Alias: BULKLOAD_PORT=

Default: 8020

Restriction: This option is required if Hadoop HDFS service is running on a port other than 8020.

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

Data source: Hadoop

See: BULKLOAD= data set option

Syntax

BL_PORT=*port*

Syntax Description

port
specifies the port number to use.

Details

Use this option to specify the port number that bulk loading uses to communicate with the server where the input data file resides.

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)

Categories: Bulk Loading
Data Set Control

Default: NO

Data source: Hadoop

Tip: Using BULKLOAD=YES is the fastest way to insert rows into a DBMS table.

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.

COLUMN_DELIMITER= Data Set Option

Specifies the single character to use as a column (field) delimiter.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Category: Data Set Control

Aliases: COL_DELIM=
COL_DELIMIT=
COL_DELIMITER=

Default: \001 (Ctrl-A)

Restriction: You must specify a single character or a three-digit decimal value. Other commonly used formats (for example, `'\t'`, 0x09, or `'09'x`) are invalid.

Requirement: Specify a single-character or three-digit decimal ASCII value between 001 and 127.

Data source: Hadoop

Examples: COLUMN_DELIMITER='#'
COLUMN_DELIMITER=009 (tab)

Syntax

COLUMN_DELIMITER=*single-character*

CONFIGDIR= Data Set Option

Specifies a directory path to search for the required XML configuration files to use for bulk-load options.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS)

Category: Data Set Control

Alias: CFGDIR=, HD_CONFIGDIR=

Default: LIBNAME option setting

Requirement: You must enclose the value in quotation marks.

Data source: Hadoop

See: BULKLOAD= LIBNAME option, CONFIGDIR= LIBNAME option

Syntax

CONFIGDIR='*config-dir*'

Required Argument

config-dir
specifies the directory path to search for the required configuration files (core-site.xml and hdfs-site.xml) for bulk-load options that contain the required host, port, and Kerberos principal for HDFS.

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)

Category: Data Set Control

Default: none

Restriction: DBCONDITION= is ignored if it specifies ORDER BY and you also use a BY statement.

Data source: Hadoop

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

When you use this option to specify selection criteria in the form of DBMS-specific SQL query clauses, performance is often enhanced because the SAS/ACCESS engine passes these clauses directly to the DBMS for processing. The DBMS checks the criteria for syntax errors when it receives the SQL query.

DBCREATE_TABLE_EXTERNAL= Data Set Option

Specifies the type of table to create and how associated data files are handled.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Category: Data Set Control

Aliases: 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
creates an *external* table—one that is stored outside of the Hive warehouse.

NO
creates a *managed* table—one that is managed within the Hive warehouse.

Details

When a managed table is dropped, its data is also deleted. When an external table is dropped, its data is preserved. Create an EXTERNAL table if you want to preserve table data if the table is dropped. SAS issues a DROP TABLE statement when PROC DELETE references a Hive table and also with the DROP TABLE statement in PROC SQL.

Example: Protect Data from DROP TABLE

In this example, DBCREATE_TABLE_LOCATION= stores the table data outside of the Hive warehouse. DBCREATE_TABLE_EXTERNAL=YES protects the data from being deleted if the table is dropped.

```
LIBNAME db HADOOP SERVER=mysrv1 USER=myusr1 DB=myschema1;
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)

Category: Data Set Control

Aliases: DBCREATE_LOCATION=
DBCREATE_LOC=
DBCREATE_PATH=

Defaults: /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=mysrv1 USER=myusr1 DB=myschema1;
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 end of the CREATE TABLE statement.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Category: Data Set Control

Alias: POST_STMT_OPTS=

Default: LIBNAME option setting

Data source: Hadoop

Tips: If you are already using DBTYPE= within an SQL CREATE TABLE statement, you can also use it to include column modifiers.

If you want all output tables to be in the default (non-TEXTFILE) format, use the LIBNAME option. (See the LIBNAME option for examples.)

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, POST_STMT_OPTS= data set option, POST_TABLE_OPTS= data set option, PRE_STMT_OPTS= data set option, PRE_TABLE_OPTS= data set option

Syntax

DBCREATE_TABLE_OPTS='*DBMS-SQL-clauses***'**

Required Argument

DBMS-SQL-clauses
specifies one or more DBMS-specific clauses that can be appended to 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 need to add an option in a location other than at the end of your CREATE TABLE statement, use one of these data set options: POST_TABLE_OPTS=, PRE_STMT_OPTS=, and PRE_TABLE_OPTS=.

Example: Partition a Hive Table

In this example, a Hive table PART is created with the value of the DBCREATE_TABLE_OPTS= option appended to the CREATE TABLE statement.

```
options sastrace=',,,d' sastraceloc=saslog;

libname x HADOOP server=XXXX user=XXXXX pwd=XXXXXX ;
data x.part (DBCREATE_TABLE_OPTS="PARTITIONED BY(I INT)");
```

```
i=1; output;
j=2; output;
run;
```

When you use this data set option to create this table, the Hadoop interface generates a CREATE TABLE statement similar to this one.

```
HADOOP_8: Executed: on connection 2
CREATE TABLE `PART` (`j` DOUBLE) PARTITIONED BY(I INT)
   TBLPROPERTIES ('SAS OS Name'='W32_7PRO', 'SAS Version'='9.04.01M3D04152015')
```

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)

Category: Data Set Control

Default: NO

Data source: Hadoop

See: DBTYPE= data set option

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 DBTYPE= data set option—namely, you must specify both DBFORCE= and DBTYPE=. 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.

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)

Category: Data Set Control

Default: DBMS

Data source: Hadoop

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.

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**.

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)

Category: Data Set Control

Default: none

Data source: Hadoop

See: MULTI_DATASRC_OPT= LIBNAME option

Syntax

DBMASTER=YES

Syntax Description

YES
designates which of two tables that are referenced in a join operation is the larger table.

Details

You can use this option 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.

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: 32767

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

Requirement: You must set the value to 4000 when you are using procedures that work with SAS High-Performance Analytics Server.

Data source: Hadoop

Note: If you set the value of DBMAX_TEXT= so that data in a table is truncated, the data load fails for that table. The number of bytes that are used to store characters might vary and is based on your session encoding.

See: DBMAX_TEXT= LIBNAME option

Syntax

DBMAX_TEXT=*integer*

Syntax Description

integer
is a number between 1 and 32,767.

Details

This option applies for the STRING data type.

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)

Category: Data Set Control

Default: DBMS-specific

Data source: Hadoop

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.

If you convert a long string value to the NUMERIC type, the numeric value that is stored in SAS might not exactly match the original character value. This happens with long strings that contain more than 15 significant digits. For example, if SAS reads in a character value of '12345678901234567890123456789 0' and converts that to type NUMERIC, then the numeric value that SAS stores is 12345678901234600000000000000. For more information, see "Choosing Your Degree of Numeric Precision" on page 7.

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 DBMS 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 SAS can accurately support, such as DECIMAL(20). If you use only PROC PRINT on the DBMS table, the data might be rounded or 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).

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)

Category: Data Set Control

Default: DBMS-specific

Data source: Hadoop

See: DBCREATE_TABLE_OPTS= data set option,

Syntax

DBTYPE=(*column-name-1*=<'> *DMBS-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 writes 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.

Example: 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;
```

POST_STMT_OPTS= Data Set Option

Allows additional database-specific options to be placed after the CREATE TABLE statement in generated SQL code.

Valid in: DATA and PROC steps (when creating DBMS tables using SAS/ACCESS software).

Category: Data Set Control

Alias: DBCREATE_TABLE_OPTS=

Default: none

Data source: Hadoop

See: DBCREATE_TABLE_OPTS= data set option, DBIDIRECTEXEC system option, POST_TABLE_OPTS= data set option, PRE_STMT_OPTS= data set option, PRE_TABLE_OPTS= data set option

Syntax

POST_STMT_OPTS='*DBMS-SQL-option(s)***'**

Required Argument

DBMS-SQL-option(s)
specifies database-specific options to be placed after the CREATE TABLE statement. Enclose the options that you specify within single or double quotation marks.

Details

You can use the POST_STMT_OPTS= data set option with these related data set options: PRE_STMT_OPTS, PRE_TABLE_OPTS=, and POST_TABLE_OPTS=. For example, you can supply database options according to this template:

```
proc sql;
  create table crtab1 ( POST_TABLE_OPTS= "/* post_table_hint */"
                        PRE_TABLE_OPTS=  "/* pre_table_hint */"
                        POST_STMT_OPTS=  "/* post_stmt_hint */"
                        PRE_STMT_OPTS=   "/* pre_stmt_hint */"
                       ) as
     select * from rdtab;
  <additional-clauses>
quit;
```

The resulting code varies depending on whether the DBIDIRECTEXEC system option is enabled. When DBIDIRECTEXEC is set, the code might result in this generated SQL code (assuming the columns C1, D1, and E1):

```
/* pre_stmt_hint */ CREATE  /* pre_table_hint */  TABLE DBID20.crtab1
    /* post_table_hint */  as ( select TXT_1."C1", TXT_1."D1", TXT_1."E1"
           from DBID20.RDTAB TXT_1 ) WITH NO DATA IN UDBID20.USERDATA
           /* post_stmt_hint */
```

When you specify NODBIDIRECTEXEC, the code might result in this generated SQL code (assuming the columns C1, D1, and E1):

```
/* pre_stmt_hint */ CREATE  /* pre_table_hint */  TABLE DBID20.CRTAB2
  /* post_table_hint */ (C1 FLOAT, D1 CHAR(1), E1 CHAR(1)) IN UDBID20.USERDATA
  /* post_stmt_hint */
```

Another usage template that does not use the SQL procedure is shown in this code:

```
data mylib.crtab1 ( POST_TABLE_OPTS= "/* post_table_hint */"
                    PRE_TABLE_OPTS= "/* pre_table_hint */"
                    POST_STMT_OPTS= "/* post_stmt_hint */"
                    PRE_STMT_OPTS= "/* pre_stmt_hint */"
                  );
set work.localtable;
run;
```

POST_TABLE_OPTS= Data Set Option

Allows additional database-specific options to be placed after the table name in a CREATE TABLE statement.

Valid in: DATA and PROC steps (when creating DBMS tables using SAS/ACCESS software).

Category: Data Set Control

Default: none

Data source: Hadoop

See: DBIDIRECTEXEC system option, POST_STMT_OPTS= data set option, PRE_STMT_OPTS= data set option, PRE_TABLE_OPTS= data set option

Syntax

POST_TABLE_OPTS=*DBMS-SQL-option(s)*

Required Argument

DBMS-SQL-option(s)
specifies additional database-specific options to be placed after the table name in a CREATE TABLE statement.

Details

You can use the POST_TABLE_OPTS= data set option with these related data set options: PRE_STMT_OPTS, POST_STMT_OPTS=, and PRE_TABLE_OPTS=. For example, you can supply database options according to this template:

```
proc sql;
  create table crtab1 ( POST_TABLE_OPTS= "/* post_table_hint */"
                        PRE_TABLE_OPTS=  "/* pre_table_hint */"
                        POST_STMT_OPTS=  "/* post_stmt_hint */"
                        PRE_STMT_OPTS=   "/* pre_stmt_hint */"
                       ) as
     select * from rdtab;
  <additional-clauses>
quit;
```

The resulting code varies depending on whether the DBIDIRECTEXEC system option is enabled. When DBIDIRECTEXEC is set, the code might result in this generated SQL code (assuming the columns C1, D1, and E1):

```
/* pre_stmt_hint */ CREATE  /* pre_table_hint */  TABLE DBID20.crtab1
    /* post_table_hint */  as ( select TXT_1."C1", TXT_1."D1", TXT_1."E1"
           from DBID20.RDTAB TXT_1 ) WITH NO DATA IN UDBID20.USERDATA
           /* post_stmt_hint */
```

When you specify NODBIDIRECTEXEC, the code might result in this generated SQL code (assuming the columns C1, D1, and E1):

```
/* pre_stmt_hint */ CREATE  /* pre_table_hint */  TABLE DBID20.CRTAB2
  /* post_table_hint */ (C1 FLOAT, D1 CHAR(1), E1 CHAR(1)) IN UDBID20.USERDATA
  /* post_stmt_hint */
```

Another usage template that does not use the SQL procedure is shown in this code:

```
data mylib.crtab1 ( POST_TABLE_OPTS= "/* post_table_hint */"
                    PRE_TABLE_OPTS= "/* pre_table_hint */"
                    POST_STMT_OPTS= "/* post_stmt_hint */"
                    PRE_STMT_OPTS= "/* pre_stmt_hint */"
                  );
set work.localtable;
run;
```

PRE_STMT_OPTS= Data Set Option

Allows additional database-specific options to be placed before a CREATE TABLE statement.

Valid in: DATA and PROC steps (when creating DBMS tables using SAS/ACCESS software).

Category: Data Set Control

Default: none

Data source: Hadoop

See: DBIDIRECTEXEC system option, POST_STMT_OPTS= data set option, POST_TABLE_OPTS= data set option, PRE_TABLE_OPTS= data set option

Syntax

PRE_STMT_OPTS=*DBMS-SQL-option(s)*

Required Argument

DBMS-SQL-option(s)
specifies additional database-specific options to be placed before the CREATE TABLE statement.

Details

You can use the PRE_STMT_OPTS= data set option with these related data set options: PRE_TABLE_OPTS, POST_STMT_OPTS=, and POST_TABLE_OPTS=. For example, you can supply database options according to this template:

```
proc sql;
  create table crtab1 ( POST_TABLE_OPTS= "/* post_table_hint */"
                        PRE_TABLE_OPTS=  "/* pre_table_hint */"
                        POST_STMT_OPTS=  "/* post_stmt_hint */"
                        PRE_STMT_OPTS=   "/* pre_stmt_hint */"
                       ) as
     select * from rdtab;
  <additional-clauses>
quit;
```

The resulting code varies depending on whether the DBIDIRECTEXEC system option is enabled. When DBIDIRECTEXEC is set, the code might result in this generated SQL code (assuming the columns C1, D1, and E1):

```
/* pre_stmt_hint */ CREATE  /* pre_table_hint */  TABLE DBID20.crtab1
    /* post_table_hint */  as ( select TXT_1."C1", TXT_1."D1", TXT_1."E1"
           from DBID20.RDTAB TXT_1 ) WITH NO DATA IN UDBID20.USERDATA
           /* post_stmt_hint */
```

When you specify NODBIDIRECTEXEC, the code might result in this generated SQL code (assuming the columns C1, D1, and E1):

```
/* pre_stmt_hint */ CREATE  /* pre_table_hint */  TABLE DBID20.CRTAB2
  /* post_table_hint */ (C1 FLOAT, D1 CHAR(1), E1 CHAR(1)) IN UDBID20.USERDATA
  /* post_stmt_hint */
```

Another usage template that does not use the SQL procedure is shown in this code:

```
data mylib.crtab1 ( POST_TABLE_OPTS= "/* post_table_hint */"
                    PRE_TABLE_OPTS= "/* pre_table_hint */"
                    POST_STMT_OPTS= "/* post_stmt_hint */"
                    PRE_STMT_OPTS= "/* pre_stmt_hint */"
                  );
set work.localtable;
run;
```

PRE_TABLE_OPTS= Data Set Option

allows additional database-specific options to be placed before the table name in a CREATE TABLE statement.

Valid in: DATA and PROC steps (when creating DBMS tables using SAS/ACCESS software).

Category: Data Set Control

Default: none

Data source: Hadoop

See: DBIDIRECTEXEC system option, POST_STMT_OPTS= data set option, POST_TABLE_OPTS= data set option, PRE_STMT_OPTS= data set option

Syntax

PRE_TABLE_OPTS=*DBMS-SQL-option(s)*

Required Argument

DBMS-SQL-option(s)
specifies additional database-specific options to be placed before the table name in a CREATE TABLE statement.

Details

You can use the PRE_TABLE_OPTS= data set option with these related data set options: PRE_STMT_OPTS, POST_STMT_OPTS=, and POST_TABLE_OPTS=. For example, you can supply database options according to this template:

```
proc sql;
  create table crtab1 ( POST_TABLE_OPTS= "/* post_table_hint */"
                        PRE_TABLE_OPTS=  "/* pre_table_hint */"
                        POST_STMT_OPTS=  "/* post_stmt_hint */"
                        PRE_STMT_OPTS=   "/* pre_stmt_hint */"
                       ) as
     select * from rdtab;
  <additional-clauses>
quit;
```

The resulting code varies depending on whether the DBIDIRECTEXEC system option is enabled. When DBIDIRECTEXEC is set, the code might result in this generated SQL code (assuming the columns C1, D1, and E1):

```
/* pre_stmt_hint */ CREATE  /* pre_table_hint */  TABLE DBID20.crtab1
    /* post_table_hint */  as ( select TXT_1."C1", TXT_1."D1", TXT_1."E1"
           from DBID20.RDTAB TXT_1 ) WITH NO DATA IN UDBID20.USERDATA
           /* post_stmt_hint */
```

When you specify NODBIDIRECTEXEC, the code might result in this generated SQL code (assuming the columns C1, D1, and E1):

```
/* pre_stmt_hint */ CREATE  /* pre_table_hint */  TABLE DBID20.CRTAB2
  /* post_table_hint */ (C1 FLOAT, D1 CHAR(1), E1 CHAR(1)) IN UDBID20.USERDATA
  /* post_stmt_hint */
```

Another usage template that does not use the SQL procedure is shown in this code:

```
data mylib.crtab1 ( POST_TABLE_OPTS= "/* post_table_hint */"
                    PRE_TABLE_OPTS= "/* pre_table_hint */"
                    POST_STMT_OPTS= "/* post_stmt_hint */"
                    PRE_STMT_OPTS= "/* pre_stmt_hint */"
                  );
set work.localtable;
run;
```

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).

Category: Data Set Control

Alias: PRESERVE_NAMES= (see "Details")

Default: LIBNAME option setting

Data source: Hadoop

See: PRESERVE_COL_NAMES= LIBNAME option,PRESERVE_TAB_NAMES= LIBNAME option, VALIDVARNAME= system option, SAS Names and Support for DBMS Names, and the DBMS-specific naming conventions sections for your SAS/ACCESS interface

Syntax

PRESERVE_COL_NAMES=YES | NO

Syntax Description

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 are preserved.

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 invalid when DBMS normalization rules are applied. To include nonstandard SAS symbols or database keywords, specify PRESERVE_COL_NAMES=YES.

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 Viya Data Set Options: Reference*.
- To enable the DBMS to normalize the column names according to its naming conventions, specify variables with any case and set PRESERVE_COL_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 PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= LIBNAME options.

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 hadoop user=myusr1 password=mypwd1 server=hadoopsvr;
   proc sql dquote=ansi;
     create table mydblib.mytable (preserve_col_names=yes) ("my$column" int);
```

READ_METHOD= Data Set Option

Specifies how to read data.

Valid in: DATA and PROC steps (when creating and appending to DBMS tables using SAS/ACCESS software)

Category: Data Set Control

Default: none

Data source: Hadoop

See: ANALYZE= LIBNAME option, ANALYZE= data set option, READ_METHOD= LIBNAME option

Syntax

READ_METHOD=JDBC | HDFS

Syntax Description

JDBC
specifies that data is to be read through the JDBC connection to the Hive service. You can use the ANALYZE= option to potentially improve performance when querying small tables.

HDFS
specifies that data is to be read through a connection to the Hadoop HDFS service.

Details

Although HDFS cannot alter the behavior of operations that always use JDBC, in general HDFS is a faster alternative to JDBC. To take advantage of potential performance benefits, set this option to HDFS. Use JDBC when you cannot access the HDFS service or JDBC Read offers some other advantage.

Example: Read Data Using JDBC

In this example, a partition of data from the sales Hive table is read using JDBC.

```
libname hdp hadoop server=mysrv1 user=myusr1 pwd=mypwd1;
data work.sales_subset; set hdp.sales(READ_METHOD=JDBC);
where year_month='2012-10'; 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)

Category: Data Access

Default: LIBNAME option setting

Data source: Hadoop

See: DBCONINIT= LIBNAME option, 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 access the schema that is specified.

SCRATCH_DB= Data Set Option

Specifies a Hive schema so that SAS can store temporary output.

Valid in: SAS/ACCESS LIBNAME statement, CONNECT statement

Category: Data Set Control

Default: none (the current schema is used)

Requirement: SAS must be able to write to the schema.

Data source: Hadoop

Syntax

SCRATCH_DB=*schema-name*

Syntax Description

schema-name
specifies the name that is assigned to a logical classification of objects in a relational database.

Details

This option lets the user specify a target schema that SAS can use for temporary output. It is needed when a user does not have permissions to perform DDL operations such as CREATE TABLE or DROP TABLE in the current schema.

TRANSCODE_FAIL= Data Set Option

Lets you specify how to handle processing and notification of transcoding errors.

Valid in: DATA and PROC steps (when accessing DBMS data using SAS/ACCESS software)

Category: Data Set Control

Default: LIBNAME option setting

Restriction: This option is not available for use with SAS Embedded Process.

Data source: Hadoop

Tip: You can use TRANSCODE_FAIL= to determine whether you want to halt or continue processing when transcoding errors are encountered.

See: TRANSCODE_FAIL= LIBNAME option

Syntax

TRANSCODE_FAIL=<ERROR> | <WARNING> | <SILENT>

Optional Arguments

ERROR
stops processing data and provides an informative error message.

WARNING
continues processing of data but provides an informative error message.

SILENT
continues processing of data but suppresses messages.

Chapter 10

Macro Variables and System Options

Introduction to Macro Variables and System Options

This section describes macro variables on page 95 and system options on page 96 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 Viya Macro Language: Reference*. For more information about SAS system options, see *SAS Viya 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.

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 104.

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 Viya System Options: Reference*.)

SAS System Options	Default
DBIDIRECTEXEC	NODBIDIRECTEXEC
REPLACE=	No SAS/ACCESS interface support is available. See *SAS Viya System Options: Reference*.
SASTRACE=	none
SASTRACELOC=	stdout
VALIDVARNAME=	Controls the type of SAS variable names that can be used or created during a SAS session.

Dictionary

DBIDIRECTEXEC SAS 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

Categories: Files: External Files
System Administration: Performance

Default: NODBIDIRECTEXEC

Data source: Hadoop

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 32 and "When Passing Joins to the DBMS Will Fail" on page 33.

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 a table by selecting from a SAS table is not sent to the database for execution. The reason is that the data sources are not compatible. The libref must also use the same database server for all compatible data sources.

SASTRACE= SAS System Option

Generates trace information from a DBMS engine.

Valid in: configuration file, SAS invocation, OPTIONS statement

Category: Log and procedure output control: SAS log

Default: none

Data source: Hadoop

Tip: You can also use more than one SASTRACE= option at a time (for example, **`SASTRACE=',,d,d'`**).

See: SASTRACELOC= system option,

Syntax

SASTRACE= ',,,d' | ' ,,d,' | ' d,' | ',,,db' | ' ,,,s' | ' ,,,sa'

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.

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.

- 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.

SASTRACELOC= SAS System Option

Specifies the location where SASTRACE= information should be printed.

Valid in: configuration file, SAS invocation, OPTIONS statement

Category: SAS log and procedure output: SAS log

Default: stdout

Data source: Hadoop

See: SASTRACE= system option

Syntax

SASTRACELOC=stdout | SASLOG | FILE '*path-and-filename*'

Required Arguments

stdout
specifies the default output location for your operating environment.

SASLOG
specifies that trace messages should be printed to the SAS log.

FILE='*path-and-filename*'
specifies that trace messages should be printed to a filename that you provide. If you do not provide a path, the file is generated in the current working directory.

Details

SASTRACELOC= lets you specify where to put the trace messages that SASTRACE= generates.

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';
```

VALIDVARNAME= SAS 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.

Details

The VALIDVARNAME= system option is supported for all DBMSs that support the SQL pass-through facility. You can set this option on start-up or in an OPTIONS statement, and the option value is used in the call to the SQL procedure. Alternatively, you can specify the VALIDVARNAME= option in the CONNECT statement.

Chapter 11

SQL Pass-Through Facility

About SQL Procedure Interactions

Overview of SQL Procedure Interactions with SAS/ACCESS

The SQL procedure implements Structured Query Language (SQL) for SAS software. For details about PROC SQL , see the *SAS Viya SQL Procedure User's 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 LIBNAME Statement for Relational Databases on page 44.)
- 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 47.)
- 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 for the SQL Pass-Through Facility for Relational Databases on page 104.)

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.

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.

For DBMS-specific details about the SQL pass-through facility, see the reference section for your SAS/ACCESS interface.

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 DBMS-specific reference section 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. For details about these return codes, see "Macro Variables for Relational Databases" on page 95.

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*>
<(*connect-statement-arguments*)>
<(*database-connection-arguments*)>;

CONNECT USING *libref* <AS *alias*>;

Required Arguments

dbms-name
specifies the DBMS to which you want to connect. You must specify the DBMS name for your SAS/ACCESS interface. See the SQL pass-through section in the DBMS-specific reference section for your SAS/ACCESS interface.

libref
specifies the libref for which a DBMS connection has already been established through the LIBNAME statement.

Optional Arguments

AS *alias*
specifies an alias for the connection that has 1 to 32 characters. The AS keyword must precede *alias*. If you do not specify an alias, the DBMS name is used as the name of the SQL pass-through connection. Some DBMSs allow more than one connection. You can use the AS clause to name connections so that you can refer to them later.

connect-statement-arguments
specifies values for arguments that indicate whether you can make multiple connections, shared or unique connections, and so on, to the database. With these arguments, the SQL pass-through facility can use some of the connection management features of the LIBNAME statement or of SAS system options. Although these arguments are optional, you must enclose them in parentheses if you include any:

- CONNECTION=
- DBCONINIT=
- DBGEN_NAME=
- DBMAX_TEXT=
- DEFER=
- VALIDVARNAME=

Note: In addition to the arguments that are listed here, several other LIBNAME or system options are available for use with the CONNECT statement. For options that are available for your SAS/ACCESS interface, see the SQL pass-through facility section for your interface in the DBMS-specific reference section. 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
indicates that 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=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=.

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.

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.

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.

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=V7
indicates that only SAS 7 variable names are considered valid. Specify this connection argument if you want the SQL pass-through facility to operate in SAS 7 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=V7 is specified, the SAS/ACCESS engine for the DBMS truncates column names to eight characters, as it does in SAS 7. 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.

  For this example, if you omit VALIDVARNAME=V7 as a connection argument, you must add it in an OPTIONS= statement in order for PROC CONTENTS to work.

  ```
  options validvarname=v7;
  proc contents data=budget2000;run;
  ```

  So using it as a connection argument saves you coding later.

database-connection-arguments
specifies values for DBMS-specific arguments that PROC SQL needs to connect to the DBMS. Though they are optional for most databases, you must enclose them in parentheses if you include any. For information about these arguments, see Chapter 12, "SAS/ACCESS Interface to Hadoop," on page 115.

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, 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.

A CONNECT USING statement uses the existing connection that is established in the LIBNAME statement for the specified libref.

Any return code or message that the DBMS generates is available in the SQLXRC and SQLXMSG macro variables after the statement executes. For information about these macro variables, see "Macro Variables for Relational Databases" on page 95.

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 DBMS to which you direct the DBMS-specific SQL statement. See the SQL pass-through section in the DBMS-specific reference section for your SAS/ACCESS interface.

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 CONNECTION statement on page 105.) If you omit the CONNECT statement, an implicit connection is performed by using default values for all database connection arguments. This automatic connection occurs at the first PROC SQL SELECT statement that contains a CONNECTION TO component. For details, see the SQL pass-through section in the DBMS-specific reference section for your SAS/ACCESS interface.

Because relational databases and SAS have different naming conventions, some DBMS column names might be changed when you retrieve DBMS data through the CONNECTION TO component. See Chapter 3, "SAS Names and Support for DBMS Names," on page 15 for more information.

DISCONNECT Statement

Terminates the connection to the DBMS

Valid in: PROC SQL steps (when accessing DBMS data using SAS/ACCESS software)

Syntax

DISCONNECT FROM *dbms-name* | *alias*

Required Arguments

dbms-name
specifies the DBMS from which you want to disconnect. You must either specify the DBMS name for your SAS/ACCESS interface or use an alias in this statement. See the LIBNAME section in the DBMS-specific reference section for your SAS/ACCESS interface. 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 95 for more information about these macro variables.

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 95 for more information about these macro variables.

dbms-name

specifies the DBMS to which you direct the DBMS-specific SQL statement. The keyword BY must appear before the *dbms-name* argument. You must either specify the DBMS name for your SAS/ACCESS interface or use an alias. See the SQL pass-through section in the DBMS-specific reference section for your SAS/ACCESS interface.

alias

specifies an alias that was defined in the CONNECT statement. (You cannot use an alias if the CONNECT statement was omitted.)

Details

Overview

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 105.) 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. For details, see the SQL pass-through section in the DBMS-specific reference section for your SAS/ACCESS interface.

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.
DROP	deletes a DBMS table, view, or other DBMS object, depending on how the statement is specified.

Statement	Description
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.

For more information and restrictions on these and other SQL statements, see the SQL documentation for your DBMS.

Part 3

DBMS-Specific Reference

Chapter 12

SAS/ACCESS Interface to Hadoop

Introduction to SAS/ACCESS Interface to Hadoop

Hadoop Concepts

Hadoop is general-purpose data storage and computing platform that includes database-like tools, such as Hive and HiveServer2. SAS/ACCESS Interface to Hadoop lets you work with your data using SQL constructs through Hive and HiveServer2. It also lets you access data directly from the underlying data storage layer, the Hadoop Distributed File System (HDFS). This differs from the traditional SAS/ACCESS engine behavior, which exclusively uses database SQL to read and write data.

With SAS/ACCESS Interface to Hadoop for Hive and HiveServer2, you can read and write data to and from Hadoop as if it were any other relational data source to which SAS can connect. This interface provides fast, efficient access to data stored in Hadoop through HiveQL. You can access Hive tables as if they were native SAS data sets and then analyze them using SAS.

For Hive operations like submitting HiveQL, the Hadoop engine requires access to the Hive service that runs on the Hadoop cluster, often port 10000. For HDFS operations, such as writing data to Hive tables, the Hadoop engine requires access to the HDFS service that runs on the Hadoop cluster, often port 8020. If the Hadoop engine cannot access the HDFS service, its full functionality is not available.

Without the need for Hive and HiveServer2, the Hadoop engine can look in an assigned HDFS directory for metadata that SAS creates: files with a SASHDMD file type. Specify the HDFS_METADIR= option if you are working with metadata that SAS creates. The Hadoop engine can create SASHDMD metadata when it writes output from SAS, or the HDMD procedure can create these metadata files.

For details about using the HDMD procedure to make your HDFS files available to the SAS system, see the HDMD Procedure in the *SAS Viya Visual Data Management and Utility Procedures Guide*.

Hadoop Configuration

Your Hadoop administrator configures the Hadoop cluster that you use. Your administrator has set the defaults for system parameters such as block size and replication factor that affect the Read and Write performance of your system. Replication factors greater than 1 help prevent data loss yet slow the writing of data. Consult with your administrator about how your particular cluster was configured.

To connect to a Hadoop server, you must complete these configuration steps.

- You must use a supported Hadoop distribution and configure a required set of Hadoop JAR files. The JAR files must be located in one location and must be available to the SAS client machine. The SAS environment variable

SAS_HADOOP_JAR_PATH must be defined and point to the folders that contain all Hadoop JAR files.

- Hadoop cluster configuration files include core-site.xml, hdfs-site.xml, hive-site.xml, mapred-site.xml, and, if applicable, yarn-site.xml. You must copy Hadoop configuration files from the Hadoop cluster to a physical location that the SAS client machine can access. You must also define and set the SAS environment variable SAS_HADOOP_CONFIG_PATH to the location of the Hadoop configuration files. If you are using MapR, no hdfs-site.xml file is required in the directory.
- To connect to the Hadoop server through WebHDFS, the SAS environment variable SAS_HADOOP_RESTFUL must be defined and set to the value 1. In addition, the Hadoop hdfs-site.xml configuration file must include the properties for the WebHDFS location.

For more information, see the *SAS Viya and SAS/ACCESS: Hadoop Configuration Guide*.

National Language Support Limitations

Keep these limitations in mind when working with Hadoop and Hive.

- Column names must use only WLATIN1 characters
- Table comments should also contain only WLATIN1 characters when the DBCREATE_TABLE_OPTS= data set option creates them using **`DBCREATE_TABLE_OPTS="COMMENT 'my table comment'"`**. Although this option accepts any NLS character, the NLS portion of the comment is not displayed properly later.

LIBNAME Statement Specifics for Hadoop

Overview

This section describes the LIBNAME statement that SAS/ACCESS Interface to Hadoop supports. For general information about this feature, see LIBNAME Statement for Relational Databases on page 44.

Here is the LIBNAME statement syntax for accessing Hadoop.

LIBNAME *libref* **Hadoop** *<connection-options> <LIBNAME-options>*

JDBC Read Security

SAS/ACCESS can access Hadoop data through a JDBC connection to a HiveServer or HiveServer2 service. Depending on what release of Hive you have, Hive might not implement Read security. If Hive does not implement Read security, a successful connection from SAS allows Read access to all data that is accessible to the Hive service. SAS/ACCESS can connect to a Hive or HiveServer2 service that is unsecured, user name and password secured, or secured by Kerberos. SAS/ACCESS supports Kerberos 5 Version 1.9 or later.

HDFS Write Security

SAS/ACCESS creates and appends to Hive tables using the HDFS service. SAS/ACCESS can connect to a Hive or HiveServer2 service that is unsecured, user name and password secured, or secured by Kerberos. Your HDFS connection needs Write access to the HDFS /tmp directory. After data is written to /tmp, a Hive LOAD command is issued on your JDBC connection to associate the data with a Hive table. Therefore, the JDBC Hive session also needs Write access to /tmp.

Note: If HDFS /tmp has enabled the sticky bit, the LOAD command can fail. To resolve this, either disable the /tmp sticky bit or use the HDFS_TEMPDIR option to specify an alternative HDFS directory for SAS/ACCESS to write data to.

HDFS Permission Requirements for Optimized Reads

To optimize big data Reads, SAS/ACCESS creates a temporary table in HDFS /tmp. This requires that the SAS JDBC connection have Write access to /tmp. The temporary table is read using HDFS, so the SAS HDFS connection needs Read access to the temporary table that is written to /tmp. Alternatively, use the HDFS_TEMPDIR option to specify an HDFS directory to use instead of /tmp.

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.

Note: All of the following connection options are also valid in the CONNECT statement when you use the SQL pass-through facility (SQL procedure) to connect to your DBMS.

USER=<'>*Hadoop-user-name*<'>
specifies the user name for Read (JDBC) and Write (HDFS) operations. Do not use the USER= argument if your Hadoop cluster is secured by Kerberos.

Alias: UID=

PASSWORD=<'>*Hadoop-password*<'>
specifies the Hadoop password that is associated with your user ID. If the password 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 *SAS Viya Visual Data Management and Utility Procedures Guide* for a method to encode it.

Alias: PASS=, PWD=, PW=

SERVER=<'>*Hadoop-server-name*<'>
specifies the Hadoop server name that runs the Hive service. 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 service.

Alias: SERVICE=

Default: 10000

SCHEMA=*Hive-schema*
specifies the Hive schema.

Alias: DATABASE=, DB=

Default: **`default`**

HDFS_DATADIR='*path*'
when not in Hive mode, specifies the path to the Hadoop directory where SAS reads and writes data (for example, *'/sas/hpa'*). Use this option only when you are not using Hive or HiveServer2. For details, see the "Accessing Data Independently from Hive" topic in the *SAS Viya Visual Data Management and Utility Procedures Guide*

Alias: HDFS_PERMDIR=

HDFS_METADIR='*path*'
specifies the path to an HDFS directory that contains XML-based table definitions, called SASHDMD descriptors. Through these descriptors, SAS then accesses the data using HDFS instead of Hive. If you want the Hadoop engine to connect to Hive and use HiveQL, do not set this option.

HDFS_PRINCIPAL=
specifies the HDFS principal string in an environment that uses Kerberos.

Default: none

This argument is not required if the core-site.xml and hdfs-site.xml files exist in the directory that is specified by the SAS_HADOOP_CONFIG_PATH environment variable. If you are using MapR, no hdfs-site.xml file is required in the directory.

HDFS_TEMPDIR='*path*'
specifies the path to the HDFS directory where SAS reads and writes temporary data.

Default: **`HDFS_TEMPDIR='/tmp'`**

HIVE_PRINCIPAL=
specifies the Hive principal string in an environment that uses Kerberos.

Default: none

This argument is not required if hive-site.xml exists in the directory that is specified by the SAS_HADOOP_CONFIG_PATH environment variable.

Alias: HIVE_KERBEROS_PRINCIPAL=

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. This table also identifies LIBNAME options that are valid in the CONNECT statement in the SQL procedure. For details, see LIBNAME Options for Relational Databases on page 48.

Table 12.1 *SAS/ACCESS LIBNAME Options for Hadoop*

Option	Default Value	Valid in CONNECT
ACCESS=	none	
ANALYZE= LIBNAME option	NO	
BL_PORT=	8020	
BULKLOAD=	none	
CONFIGDIR=	none (SAS_HADOOP_JAR_PATH is the default when you set only BULKLOAD=YES)	
CONNECTION=	UNIQUE	•
DBCONINIT=	none	•
DBCREATE_TABLE_ EXTERNAL=	NO	•
DBCREATE_TABLE_OPTS=	none	•
DBGEN_NAME=	DBMS	
DBMAX_TEXT=	none	•
DBMSTEMP=	none	
DEFER=	NO	•
DIRECT_SQL=	YES	
LOGIN_TIMEOUT=	0	•
MULTI_DATASRC_OPT=	none	
PRESERVE_COL_NAMES=	NO	
PRESERVE_TAB_NAMES=	NO	
PROPERTIES=	none	•
READ_METHOD=	none	•
SCHEMA=	Hive schema **`default`**	
SCRATCH_DB=	none	•
SPOOL=	YES	

Option	Default Value	Valid in CONNECT
SQL_FUNCTIONS=	none	•
TRANSCODE_FAIL=	ERROR	•

Hadoop LIBNAME Statement Examples

This example uses the default HiveServer2 port and schema.

```
libname hdp hadoop server=hxpduped
   user=myusr1 password=mypwd1;
```

This example explicitly specifies the default HiveServer2 port and schema.

```
libname hdp hadoop server=hxpduped port=10000 schema=default
   user=myusr1 password=mypwd1;
```

This example specifies the Hive Kerberos principal to connect to a Kerberos secured HiveServer2 instance. It assumes that a Kerberos **'kinit'** has been successfully performed.

```
libname hdp hadoop server=hxpduped
   hive_principal='hive/_HOST@HD.COMPANY.COM';
```

This example assumes a Kerberos secured HiveServer2 instance. It also assumes that SAS_HADOOP_CONFIG_PATH is set to a directory that contains the Hadoop hive-site.xml configuration file. Finally, this example assumes that a Kerberos **'kinit'** has been successfully performed.

```
libname hdp hadoop server=hxpduped;
```

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 details, see Data Set Options for Relational Databases on page 71.

Table 12.2 *SAS/ACCESS Data Set Options for Hadoop*

Option	Default Value
ANALYZE= data set option	LIBNAME option setting
COLUMN_DELIMITER=	\001 (Ctrl-A)
CONFIGDIR=	LIBNAME option setting
DBCONDITION=	none
DBCREATE_TABLE_EXTERNAL=	NO

Option	Default Value
DBCREATE_TABLE_LOCATION=	/user/hive/warehouse/*tabname* [with the default schema], /user/hive/warehouse/*schema*.db/*tabname* [with a nondefault schema]
DBCREATE_TABLE_OPTS=	LIBNAME option setting
DBFORCE=	NO
DBGEN_NAME=	DBMS
DBMASTER=	none
DBMAX_TEXT=	none
DBSASTYPE=	see Data Types for Hadoop on page 126
DBTYPE=	see Data Types for Hadoop on page 126
POST_STMT_OPTS=	none
POST_TABLE_OPTS=	none
PRE_STMT_OPTS=	none
PRE_TABLE_OPTS=	none
READ_METHOD=	none
SCHEMA=	LIBNAME option value
"SCRATCH_DB= Data Set Option"SCRATCH_DB=	none
TRANSCODE_FAIL=	ERROR

SQL Pass-Through Facility Specifics for Hadoop

Key Information

For general information about this feature, see "Overview of SQL Procedure Interactions with SAS/ACCESS" on page 103.

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 CONNECT statement *database-connection-arguments* are identical to its LIBNAME connection options.

CONNECT Statement Examples

This example uses the default HiveServer2 port and schema.

```
proc sql;
   connect to hadoop (user="myusr1" pw="mypwd1" server=hxpduped);
```

This example explicitly specifies the default HiveServer2 port and schema.

```
proc sql;
   connect to hadoop (user="myusr1" pw="mypwd1"
      server=hxpduped port=10000 schema=default);
```

This example specifies the Hive Kerberos principal to connect to a Kerberos secured HiveServer2 instance. It assumes that a Kerberos **'kinit'** has been successfully performed.

```
proc sql;
   connect to hadoop (hive_principal='hive/_HOST@HD.COMPANY.COM'
      server=hxpduped);
```

Temporary Table Support for Hadoop

SAS/ACCESS Interface to Hadoop supports temporary tables. For more information, see "Temporary Table Support for SAS/ACCESS" on page 30.

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 32.

ABS	LOWCASE (LOWER)
ARCOS (ACOS)	MAX
ARSIN (ASIN)	MIN
ATAN	MINUTE
AVG	MONTH
CEIL	SECOND
COALESCE	SIN
COMPRESS (REGEXP_REPLACE)*	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

* Only when you specify two arguments; for example, COMPRESS(string,' ').

SQL_FUNCTIONS=ALL allows for SAS functions that have slightly different behavior from corresponding Hadoop functions that are passed down to Hadoop. Only when SQL_FUNCTIONS=ALL can the SAS/ACCESS engine also pass these SAS SQL functions to Hadoop. Due to incompatibility in date and time functions between Hadoop and SAS, Hadoop might not process them correctly. Check your results to determine whether these functions are working as expected.

DATE (TO_DATE(UNIX_TIMESTAMP))	REPEAT
DATEPART (TO_DATE(UNIX_TIMESTAMP))	TODAY (TO_DATE(UNIX_TIMESTAMP)
DATETIME (FROM_UNIXTIME(UNIX_TIMESTAMP))	TRANSTRN (REGEXP_REPLACE)

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 to pass a cross-schema join to Hadoop.

For more information, see "Passing Joins to the DBMS" on page 32.

Bulk Loading for Hadoop

Loading

SAS/ACCESS Interface to Hadoop has no differentiation between bulk loading and a standard load process. Although BULKLOAD=YES syntax is supported, it does not change the underlying load process.

Here is how a text-based table (STORED AS TEXTFILE) is created.

1. SAS issues a CREATE TABLE Hive 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 uses HDFS to upload table data to the HDFS /tmp directory. The resulting file is a UTF-8-delimited text file that by default uses CTRL-A ('\001') as a field delimiter and newline ('\n') as a record separator.
3. SAS issues a LOAD DATA command to move the data file from the HDFS /tmp directory to the appropriate Hive warehouse location. The data file is now part of the Hive table.

Here is how a non-text table is created:

1. Specify the DBCREATE_TABLE_OPTS= data set option containing a Hive STORED AS clause to the new table reference. Here is an example:

   ```
   data hdp.new_hive_table(DBCREATE_TABLE_OPTS='STORED AS SEQUENCEFILE');
      set sas_table;
      run;
   ```

2. SAS issues two CREATE TABLE statements to the Hive server. One CREATE TABLE statement creates the target Hive table. The other CREATE TABLE statement creates a temporary table.
3. SAS uses HDFS to upload table data to the HDFS /tmp directory. The resulting file is a UTF-8-delimited text file.
4. SAS issues a LOAD DATA command to move the data file from the HDFS /tmp directory into the temporary table.
5. SAS issues an INSERT INTO statement that copies and transforms the temp table text data into the target (non-text) Hive table.
6. SAS deletes the temporary table.

Hive 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 data to the correct location.

When PROC APPEND is used to append to the Hive table, the Hadoop interface places data in a new HDFS file. The interface then issues either the LOAD DATA pattern or the LOAD DATA plus INSERT INTO pattern described earlier.

Examples

This example creates and loads the FLIGHTS98 HiveServer2 table from the SASFLT.FLT98 SAS data set.

```
libname sasflt 'SAS-library';
libname hdp_air hadoop user=myusr1 pwd=mypwd1 server='hdpcluster' schema=statsdiv;

proc sql;
create table hdp_air.flights98
        as select * from sasflt.flt98;
quit;
```

This example creates and loads the ALLFLIGHTS HiveServer2 table in SEQUENCEFILE format from the SASFLT.ALLFLIGHTS SAS data set.

```
data hdp_air.allflights (dbcreate_table_opts='stored as sequencefile');
   set sasflt.allflights;
run;
```

In this example, the SASFLT.FLT98 SAS data set is appended to the ALLFLIGHTS HiveServer2 table.

```
proc append base=hdp_air.allflights data=sasflt.flt98;
run;
```

Naming Conventions for SAS and Hive

For general information, see Chapter 3, “SAS Names and Support for DBMS Names,”.

Current versions of Hive do not preserve the case of identifiers, converting them to lowercase. By default, SAS converts them to uppercase. Users can set the PRESERVE_COL_NAMES= and PRESERVE_TAB_NAMES= options (shortcut PRESERVE_NAMES=) to preserve the case of identifiers. Doing this usually is not required unless the case must be preserved for display purposes.

Hive does not currently permit a table name to begin with the underscore (_) character.

SAS and Hadoop objects include tables, views, columns, and indexes. Here is how SAS handles Hive names:

- 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.
- For National Language Support, because of Hive limitations, column names must use only WLATIN1 characters. Table comments should also contain only WLATIN1 characters when the DBCREATE_TABLE_OPTS= data set option creates them using **`DBCREATE_TABLE_OPTS="COMMENT 'my table comment'"`**. Although this option accepts any NLS character, the NLS portion of the comment is not displayed properly later.

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 DBMSs. Hive tables are defined with a CREATE TABLE statement, so every column in a table has a name and a data type. 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 documents at https://cwiki.apache.org/confluence/display/Hive.

- *Hive Getting Started*

- *Hive Language Manual*
- *Hive Plug-in Interfaces — User-Defined Functions and SerDes*
- *Hive Tutorial*

Supported Hive Data Types

Here are the Hive data types that the Hadoop engine supports.

- Numeric data:

BIGINT	FLOAT
BOOLEAN	INT
DECIMAL	SMALLINT
DOUBLE	TINYINT

  Note: When performing calculations on numeric values and when storing numeric values, SAS maintains up to 15 digits of precision. When you read values that contain more than 15 decimal digits of precision from a database into SAS, the values that SAS reads are rounded to meet this condition. When you use a large numeric value in a WHERE clause, this rounding can cause unexpected results, such as not selecting desired rows. For noncomputational purposes, such as storing ID values or credit card numbers, you can read the data in as character data. For more information, see "Your Options When Choosing Your Needed Degree of Precision" on page 8.

- String data:

BINARY	STRING
CHAR*n*	VARCHAR*n*

- Date and time data:

DATE	TIMESTAMP

- Complex data:

ARRAY	STRUCT
MAP	

Note: Here are some important Hive-specific release details.

- Prior to Hive .12: Dates, times, and timestamps are typically stored in Hive STRING columns in ANSI format. For example, the last day of this millennium is stored as the string *'2999-12-31'*. SAS columns that are formatted as DATE, TIME, or DATETIME are stored in Hive STRING columns.
- With Hive .12: SAS/ACCESS supports the Hive DATE and TIMESTAMP data types.
- With Hive .12 and later: SAS/ACCESS generates output for Hive DATE and TIMESTAMP data types for SAS date and SAS datetime formatted columns. As a best practice, use the Hive DATE and TIMESTAMP data types and not the STRING data type.
- Hive .13 and higher: CHAR is supported. SAS issues CREATE TABLE syntax using VARCHAR instead of STRING.
- Hive does not yet support a TIME data type. SAS TIME data types are stored in Hive as STRING data types.

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.

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 12.3 *Hive to SAS: Default SAS Formats for Hive Data Types*

Hive Data Type	SAS Data Type	Default SAS Format
BINARY	character	$HEX32767
CHAR		$255.
STRING		$32767.
VARCHAR		$65355
BOOLEAN	numeric	1.
BIGINT		20.
DATE		DATE9.
DOUBLE		none
FLOAT		none
INT		11.
SMALLINT		6.
TIMESTAMP		DATETIME25.6
TINYINT		4.

Highlighting New Hive .12 Data Types

This chapter presents considerations for sharing data between Hive and SAS. For example, you need to know how to identify the Hive STRING columns that contain DATE, TIME, or TIMESTAMP information. You also need to know how to minimize the length of SAS character columns that are created from Hive STRING columns. Where you use the VARCHAR, DATE, and TIMESTAMP data types that are available

in Hive .12 and later, you optimize sharing data between Hive and SAS. For example, the SASFMT table properties that are described later are not needed for Hive columns that are declared VARCHAR, DATE, and TIMESTAMP.

Information in the rest of this chapter that concerns BIGINT and TIME values remains relevant regardless of whether you use Hive .12 and later.

Highlighting Support for Hive .13 Data Types

When you specify a SAS format for a DECIMAL data type with specified precision and scale, specify *w.d* format. *w* is the width of the variable (including the decimal portion, if any), the decimal point, and a potential negative sign. For example, given a column of type **`DECIMAL(10,0)`**, the format is **`11`**. Given a column of type **`DECIMAL(18,2)`**, the format is **`20.2`**.

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. To specify a limit for an individual STRING column, issue a Hive ALTER TABLE statement limiting the column length. Here is an example: **`ALTER TABLE weblogs SET TBLPROPERTIES ('SASFMT:webdata'='CHAR(1000)')`**. To specify a general limit for multiple STRING columns, use the DBMAX_TEXT= option.
- 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 135.
- using the LIBNAME statement. See "Address Issues When Converting Data from Hive to SAS with Table Properties" on page 133.

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=myusr1 pwd=mypwd1;
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 that STRING column **dt_stamp** contains an ANSI timestamp. SAS/ACCESS automatically converts and formats it as 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 130.

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 12.4 *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)
	$HEX32767	BINARY	none
	$n.	STRING	VARCHAR(n)

SAS Data Type	SAS Format	Hive Data Type	Hive Table Property
numeric	DATETIMEw.p	STRING	DATETIME(w.p)
	DATEw.	STRING	DATE(w.0)
	TIMEw.p.	STRING	TIME(w.p)
	1. to 4.	SMALLINT	none
	5. to 9.	INT	none
	other numeric formats	DOUBLE	none
	w.d	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. In this example, 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 hadoop server=mysrv1 user=myusr1 pwd=mypwd1;
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 web data 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 hadoop server=mysrv1 user=myusr1 pwd=mypwd1;
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 affect 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 122). 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 129.) 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 HiveQL ALTER TABLE statements. In the statements are these sample Hive columns: **d** contains ANSI date, **ts** contains ANSI timestamp, and **t** contains ANSI time.

Note: Keep in mind that you cannot run ALTER TABLE commands in SAS because they are not SAS syntax. You must execute these commands within Hive.

```
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 Hive **string_col** column 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. However, a BIGINT column can preserve up to 19 significant digits of precision, plus one character for the possible sign (+/-). You can address this issue by applying a CHAR(20) table property format. SAS then automatically reads a Hive BIGINT column into a SAS character string with $20. format. This format 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(20)')
```

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 130 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 assumes a SASFMT table property, the second one assumes no table property, and DBSASTYPE= is added to achieve the same functionality. (For details, see the DBSASTYPE= data set option.)

Here is the SAS LIBNAME statement for all of these SAS DATA steps.

Note: Remember that you cannot run ALTER TABLE commands in SAS because they are not SAS syntax. You must execute these commands within Hive.

```
libname hdp hadoop server=mysrv1 user=myusr1 pwd=mypwd1;

ALTER TABLE sample_table SET TBLPROPERTIES ('SASFMT:d'='DATE(9.0)')
[---assumes table property 'SASFMT:d'='DATE(9.0)' ---]
data work.local_sample; set hdp.sample_table( keep=d ); run;

[---assumes 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)')
[---assumes table property 'SASFMT:ts'='DATETIME(25.6)' ---]
data work.local_sample; set hdp.sample_table( keep=ts ); run;

[---assumes 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)')
[---assumes table property 'SASFMT:t'='TIME(15.6)' ---]
data work.local_sample; set hdp.sample_table( keep=t ); run;

[---assumes 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)')
[---assumes table property 'SASFMT:string_col'='CHAR(200)' ---]
data work.local_sample; set hdp.sample_table( keep=string_col ); run;
```

```
[---assumes no table property for column 'string_col'---]
data work.local_sample;
set hdp.sample_table( keep=string_col dbsastype=(string_col='CHAR(200)') );
run;
```

DBMAX_TEXT=[*n*]
You can use the DBMAX_TEXT= option to limit the SAS length of all STRING columns read from Hive. For example, if you set DBMAX_TEXT=100, then all SAS character variables that are created from Hive STRING columns are limited to width **$100**. Setting the DBMAX_TEXT= option likewise limits the length in SAS of Hive 12 and higher CHAR and VARCHAR columns.

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. For example, suppose 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 hadoop server=mysrv1 user=myusr1 pwd=mypwd1;
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(20)')
```

A LIBNAME-based table that is read to SAS recognizes 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 hadoop(server=mysrv1 user=myusr1 pwd=mypwd1);
create table work.local as select
bgint length 20 format $20. informat $20.,
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 hadoop( 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
```

If SAS detects that a column length for a numeric variable is 32767 and could be less, it writes a message:

```
WARNING: These columns could have a length in SAS of 32767. If so,
SAS performance is impacted. See SAS/ACCESS documentation for
details. The columns read from Hive followed by the maximum length
observed were: bgint:20, ts:26, d:9, t:15, string_col:20.
```

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 23.

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 user=myusr1 password=mypwd1;
create table work.a as
   select * from connection to hadoop (select * from newtab limit 10);
```

Create a Partitioned Table with a File Type of SEQUENCEFILE

Use the DBCREATE_TABLE_OPTS value PARTITIONED BY (*column data-type*) STORED AS SEQUENCEFILE.

```
libname hdp HADOOP server=hxpduped user=myusr1 password=mypwd1;

data hdp.part_tab (DBCREATE_TABLE_OPTS="PARTITIONED BY (s2 int, s3 string)
                   STORED AS SEQUENCEFILE");
   set work.part_tab;
run;
```

Using the SAS Data Connect Accelerator forHadoop

Before you can load data in parallel to SAS Cloud Analytic Services using the SAS Data Connect Accelerator for Hadoop, you must first install the SAS Embedded Process on your Hadoop cluster. For installation information, see *SAS Viya: Deployment Guide*.

Recommended Reading

Here is the recommended reading list for this title.

- *SAS Viya Visual Data Management and Utility Procedures Guide*
- *Learning SAS by Example: A Programmer's Guide*
- *The Little SAS Book: A Primer*
- *The Little SAS Book for Enterprise Guide*
- *PROC SQL: Beyond the Basics Using SAS*
- "Processing Multilingual Data with the SAS® 9.2 Unicode Server" [SAS Institute technical paper]
- *SAS Viya Component Objects: Reference*
- *SAS Viya Data Set Options: Reference*
- *SAS Encoding: Understanding the Details*
- *SAS Viya Formats and Informats: Reference*
- *SAS Viya Functions and CALL Routines: Reference*
- *SAS Viya and SAS/ACCESS: Hadoop Configuration Guide*
- *SAS Intelligence Platform: Security Administration Guide*
- *SAS Viya Macro Language: Reference*
- *SAS Viya National Language Support: Reference Guide*
- *SAS Viya SQL Procedure User's Guide*
- *SAS Viya Statements: Reference*
- *SAS Viya System Options: Reference*
- *SAS/ACCESS Interface to PC Files for SAS Viya: Reference*
- SAS offers instructor-led training and self-paced e-learning courses to help you get started with SAS/ACCESS and related SAS products. For more information about available courses, see support.sas.com/training.

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

SAS Books
SAS Campus Drive
Cary, NC 27513-2414
Phone: 1-800-727-0025

Fax: 1-919-677-4444
Email: sasbook@sas.com
Web address: sas.com/store/books

Glossary

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. *See also* view descriptor.

appliance
See data warehouse appliance.

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. *See also* rollback.

data entity (entity)
an Rdb or VMS item that can be defined and manipulated using VAX SQL. Rdb and VMS entities include databases, tables, views, columns, and indexes.

data set
See SAS data set.

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 (type)
an attribute of every column in a table or database, indicating the type of data in the column and how much physical storage it occupies.

data value
a unit of character, numeric, or alphanumeric information that is stored as a single item in a data record.

data view
See SAS data view.

data warehouse appliance (appliance)
an integrated set of servers, storage, one or more operating systems, DBMS, and software that is specifically pre-installed and pre-optimized for data warehousing.

database
an organized collection of related data. A database usually contains named files, named objects, or other named entities such as tables, views, and indexes.

database management system (DBMS)
a software application that enables you to create and manipulate data that is stored in the form of databases. *See also* relational database management system.

DBMS
See database management system.

engine (SAS 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.

entity
See data entity.

explicit pass-through
a form of the SQL pass-through facility that passes the user-written, DBMS-specific SQL query code directly to a particular DBMS for processing. *See also* implicit pass-through.

foreign key
a column or combination of columns in one table that references the corresponding primary key in another table. A foreign key must have the same attributes as the primary key that it references.

function
See SAS function.

Hadoop Distributed File System (HDFS)
a portable, scalable framework, written in Java, for managing large files as blocks of equal size. The files are replicated across multiple host machines in a Hadoop cluster in order 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 a particular DBMS for processing. *See also* explicit pass-through.

index
See SAS index.

informat
See SAS informat.

interactive line mode (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). *See also* engine.

library member
any of several types of SAS file in a SAS library. A library member can be a data set, a view, a catalog, a stored program, or an access descriptor.

library reference
See libref.

libref (library reference)
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. *See also* SAS library.

line mode
See interactive line mode.

member name
a name that is assigned to a SAS file in a SAS library. *See also* member type.

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. *See also* null value.

null value
a special value that indicates the absence of information. Null values are analogous to SAS missing values. *See also* missing value.

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
See SQL 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. *See also* SAS data view.

query
a set of instructions that requests particular information from one or more data sources.

RDBMS
See relational database management system.

referential integrity
a set of rules that a DBMS uses to ensure that whenever a data value in one table is changed, the appropriate change is also made to any related values in other tables or in the same table. Referential integrity is also used to ensure that related data is not deleted or changed accidentally.

relational database management system (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, SAP ASE, 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. *See also* commit.

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. *See also* SAS data set, SAS data view.

SAS data set (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 (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.

SAS engine
See engine.

SAS file
a specially structured file that is created, organized, and maintained by SAS. A SAS file can be a SAS data set, a catalog, a stored program, an access descriptor, a utility file, a multidimensional database file, a financial database file, a data mining database file, or an item store file.

SAS function (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 index (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.

SAS informat (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.

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 (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.

server
software that provides either resources or services to requesting clients, possibly over a network.

SQL
See Structured Query Language.

SQL pass-through facility (pass-through facility)
the technology that enables SQL query code to be passed to a particular DBMS for processing.

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.

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. *See also* referential integrity.

type
See data type.

variable
See SAS variable.

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. *See also* access descriptor.

windowing procedure
a SAS procedure that you can use by entering information in one or more windows or dialog boxes. For example, the FSVIEW procedure is a windowing procedure. Some procedures, such as ACCESS and DBLOAD, can be used either as windowing procedures or in batch mode.

Index

D

E

F

G

Q

R

S

T

U

V

W