# CHAPTER 1

# SAS/ACCESS for Teradata


*Introduction to the SAS/ACCESS Interface to Teradata* **2**
*The SAS/ACCESS Teradata Client* **2**
*LIBNAME Statement Specifics for Teradata* **3**
*Arguments* **3**
*Teradata LIBNAME Statement Example* **6**
*Data Set Options for Teradata* **6**
*Pass-Through Facility Specifics for Teradata* **7**
*Examples* **8**
*Autopartitioning Scheme for Teradata* **9**
*Overview* **10**
*FastExport and Case Sensitivity* **10**
*FastExport Setup* **11**
*FastExport Usage* **11**
*FastExport and Explicit SQL* **12**
*FastExport Usage Exceptions* **12**
*Threaded Reads with Partitioning WHERE Clauses* **12**
*FastExport Versus Partitioning WHERE Clauses* **13**
*Passing SAS Functions to Teradata* **13**
*Passing Joins to Teradata* **14**
*Temporary Table Support for Teradata* **14**
*Establishing a Temporary Table* **14**
*Terminating a Temporary Table* **15**
*Examples* **15**
*Maximizing Teradata Read Performance* **16**
*About the PreFetch Facility* **16**
*How PreFetch Works* **17**
*The PreFetch Option Arguments* **17**
*When and Why Use PreFetch* **17**
*Possible Unexpected Results* **18**
*PreFetch Processing of Unusual Conditions* **18**
*Using PreFetch as a LIBNAME Option* **19**
*Using Prefetch as a Global Option* **19**
*Maximizing Teradata Load Performance* **20**
*Maximizing Teradata Load Performance with FastLoad* **20**
*FastLoad Supported Features and Restrictions* **20**
*Invoking FastLoad* **20**
*FastLoad Data Set Options* **21**
*Maximizing Teradata Load Performance with MultiLoad* **21**
*MultiLoad Supported Features and Restrictions* **21**
*MultiLoad Setup* **22**
*MultiLoad Data Set Options* **22**

*Examples* **24**
*Teradata Processing Tips for SAS Users* **25**
*Reading from and Inserting to the Same Teradata Table* **25**
*Using a BY Clause to Order Query Results* **26**
*Using TIME and TIMESTAMP* **27**
*Replacing PROC SORT with a BY Clause* **28**
*Reducing Workload on Teradata by Sampling* **29**
*Locking in the Teradata Interface* **29**
*Understanding SAS/ACCESS Locking Options* **31**
*When to Use SAS/ACCESS Locking Options* **31**
*Examples* **32**
*Setting the Isolation Level to ACCESS for Teradata Tables* **32**
*Setting Isolation Level to WRITE to Update a Teradata Table* **33**
*Preventing a Hung SAS Session When Reading and Inserting to the Same Table* **33**
*Naming Conventions for Teradata* **34**
*Teradata Conventions* **34**
*SAS Naming Conventions* **34**
*Naming Objects to Meet Teradata and SAS Conventions* **34**
*Accessing Teradata Objects That Do Not Meet SAS Naming Conventions* **34**
*Example 1: Unusual Teradata Table Name* **34**
*Example 2: Unusual Teradata Column Names* **35**
*Data Types for Teradata* **35**
*Binary String Data* **35**
*Character String Data* **35**
*Date/Time Data* **36**
*Numeric Data* **36**
*Teradata Null Values* **37**
*LIBNAME Statement Data Conversions* **37**
*Data Returned as SAS Binary Data with Default Format $HEX* **39**

# Introduction to the SAS/ACCESS Interface to Teradata

This document includes details *only* about the SAS/ACCESS Interface to Teradata. It should be used as a supplement to the main SAS/ACCESS documentation, *SAS/ACCESS for Relational Databases: Reference*. For more detailed information on SAS/ACCESS to Teradata, please refer to the SAS/ACCESS to Teradata white paper (http://support.sas.com/rnd/warehousing/papers/teradataOct01.pdf).

*Note:* SAS/ACCESS for Teradata does not support the DBLOAD and ACCESS procedures. The LIBNAME engine technology enhances and replaces the functionality of these procedures. Consequently, you must revise SAS jobs that were written for a different SAS/ACCESS interface and that include DBLOAD or ACCESS procedures before you can run them with SAS/ACCESS for Teradata. △

## The SAS/ACCESS Teradata Client

Teradata is a massively parallel (MPP) RDBMS. A high-end Teradata server supports many users, simultaneously loading and extracting table data, and processing complex queries.

Because Teradata customers run many processors at the same time for queries of the database, users enjoy excellent DBMS *server* performance. The challenge to client software, such as SAS, is to leverage Teradata performance by rapidly extracting and loading table data. The SAS/ACCESS interface to Teradata meets the challenge by enabling you to optimize extracts and loads (reads and creates).

This documentation provides information throughout about how to optimize DBMS operations. SAS/ACCESS can create and update Teradata tables. It supports a FastLoad interface that rapidly creates new tables. It optimizes table reads, optionally using FastExport for the highest possible read performance.
SAS/ACCESS also supports MultiLoad, which loads both empty and existing Teradata tables and greatly accelerates the speed of insertion into Teradata tables.

# LIBNAME Statement Specifics for Teradata

This section describes the LIBNAME statement as supported in the SAS/ACCESS interface to Teradata. For a complete description of this feature, see the LIBNAME statement section in *SAS/ACCESS for Relational Databases: Reference*. The Teradata specific syntax for the LIBNAME statement is:

**LIBNAME** *libref* **teradata** <*connection-options*> <*LIBNAME-options*>;

## Arguments

*libref*
is any SAS name that serves as an alias to associate SAS with a database, schema, server, or group of tables and views.

teradata
is the SAS/ACCESS engine name for the interface to Teradata.

*connection-options*
provide connection information and control how SAS manages the timing and concurrence of the connection to the DBMS. The connection options for the interface to Teradata are as follows:

USER=<'>*Teradata-user-name*<'>
this is a required connection option that specifies a Teradata user name. If the name contains blanks or national characters, enclose it in quotation marks.

PASSWORD=<'>*Teradata-password*<'>
this is a required connection option that specifies a Teradata password. The password that you specify must be correct for your USER= value.

*Note:* If you do not wish to enter your Teradata password in clear text on this statement, see PROC PWENCODE in *Base SAS Procedures Guide* for a method to encode it. △

ACCOUNT=<'>*account_ID*<'>
this is an optional connection option that specifies the account number that you want to charge for the Teradata session.

DATABASE=<'>*database-name*<'>
this is an optional connection option that specifies the name of the Teradata database that you want to access, enabling you to view or modify a different user's Teradata DBMS tables or views, if you have the required privileges. (For example, to read another user's tables, you must have the Teradata privilege SELECT for that user's tables.) If you do not specify DATABASE=, the libref points to your default Teradata database, which is often named the same as your user name. If the database value that you specify contains

spaces or non-alphanumeric characters, you must enclose it in quotation marks.

SCHEMA= is an alias for this option. To use this option on an individual data set, see the SCHEMA= data set option. For more information about changing the default database, see the DATABASE statement in your Teradata documentation.

TDPID=<'>*dbcname*<'>

this is a required connection option if you run more than one Teradata server. TDPID= operates differently for network-attached and channel-attached systems, as described below.

*Note:* SERVER= can be substituted for TDPID= in all circumstances. △

- For NETWORK-ATTACHED systems (PC and UNIX), *dbcname* specifies an entry in your (client) HOSTS file that provides an IP address for a database server connection.

  By default, SAS/ACCESS connects to the Teradata server that corresponds to the *dbccop1* entry in your HOSTS file. When you run only one Teradata server, and your HOSTS file defines the *dbccop1* entry correctly, you do not need to specify TDPID=.

  However, if you run more than one Teradata server, you must use the TDPID= option to specifying a *dbcname* of eight characters or less. SAS/ACCESS adds the specified *dbcname* to the login string that it submits to Teradata. (Teradata documentation refers to this name as the tdpid component of the login string.)

  After SAS/ACCESS submits a *dbcname* to Teradata, Teradata searches your HOSTS file for all entries that begin with the same *dbcname*. In order for Teradata to recognize the HOSTS file entry, the *dbcname* suffix must be COP*x* (*x* is a number). If there is only one entry that matches the *dbcname*, *x* must be 1. If there are multiple entries for the *dbcname*, *x* must begin with 1 and increment sequentially for each related entry. (See the example HOSTS file entries below).

  When there are multiple, matching entries for a *dbcname* in your HOSTS file, Teradata does simple load balancing by selecting one of the Teradata servers specified for login. Teradata distributes your queries across these servers so that it can return your results as fast as possible.

  The TDPID= examples below assume that your HOSTS file contains the following *dbcname* entries and IP addresses:

  Example 1: TDPID= is not specified.

  **`dbccop1 10.25.20.34`**

  The TDPID= option is not specified, establishing a login to the Teradata server that runs at 10.25.20.34.

  Example 2: TDPID= myserver *or* SERVER=myserver

  **`myservercop1 130.96.8.207`**

  You specify a login to the Teradata server that runs at 130.96.8.207.

  Example 3: TDPID=xyz *or* SERVER=xyz

  **`xyzcop1 33.44.55.66`**

  or **`xyzcop2 11.22.33.44`**

  You specify a login to a Teradata server that runs at 11.22.33.44 or to a Teradata server that runs at 33.44.55.66.

- For CHANNEL-ATTACHED systems (z/OS), TDPID= specifies the subsystem name, which must be TDPx, where x can be 0-9, A-Z (not case-sensitive), or $, # or @. If there is only one Teradata server, and your z/OS System Administrator has set up the HSISPB and HSHSPB modules, you do not need to specify TDPID=. For further information, see your Teradata TDPID documentation for z/OS.

*LIBNAME-options*
define how DBMS objects are processed by SAS. Some LIBNAME options can enhance performance; others determine locking or naming behavior. The following table describes the LIBNAME options that are supported for Teradata, and presents default values where applicable. See the section about the SAS/ACCESS LIBNAME statement in *SAS/ACCESS for Relational Databases: Reference* for detailed information about these options.

**Table 1.1** SAS/ACCESS LIBNAME Options for Teradata

| Option | Default Value |
|---|---|
| ACCESS= | none |
| BULKLOAD= | NO |
| CAST= | none |
| CAST_OVERHEAD_MAXPERCENT= | 20 percent |
| CONNECTION= | for channel-attached systems (OS/390), the default is SHAREDREAD; for network attached systems (UNIX and PC platforms), the default is UNIQUE |
| CONNECTION_GROUP= | none |
| DBCOMMIT= | 1000 when inserting rows; 0 when updating rows |
| DATABASE= (see SCHEMA=) | none |
| DBCONINIT= | none |
| DBCONTERM= | none |
| DBCREATE_TABLE_OPTS= | none |
| DBGEN_NAME= | DBMS |
| DBINDEX= | NO |
| DBLIBINIT= | none |
| DBLIBTERM= | none |
| DBPROMPT= | NO |
| DBSASLABEL= | COMPAT |
| DBSLICEPARM= | THREADED_APPS,2 |
| DEFER= | NO |
| DIRECT_EXE= | |
| DIRECT_SQL= | YES |
| ERRLIMIT= | 1 million |
| LOGDB= | Default teradata database for the libref |
| MULTI_DATASRC_OPT= | IN_CLAUSE |
| PREFETCH= | not enabled |
| PRESERVE_COL_NAMES= | YES |

| Option | Default Value |
|---|---|
| PRESERVE_TAB_NAMES= | YES |
| READ_ISOLATION_LEVEL= | see “Locking in the Teradata Interface” on page 29 |
| READ_LOCK_TYPE= | none |
| READ_MODE_WAIT= | none |
| REREAD_EXPOSURE= | NO |
| SCHEMA= | your default Teradata database |
| SPOOL= | YES |
| UPDATE_ISOLATION_LEVEL= | see “Locking in the Teradata Interface” on page 29 |
| UPDATE_LOCK_TYPE= | none |
| UPDATE_MODE_WAIT= | none |
| UTILCONN_TRANSIENT= | NO |

## Teradata LIBNAME Statement Example

In the following example, the connection is made using the USER= and PASSWORD= connection options. These options are required for Teradata and must be used together.

```
libname myteralib TERADATA user=testuser password=testpass;
```

# Data Set Options for Teradata

The following table includes all of the SAS/ACCESS data set options that are supported for the Teradata interface. Default values are provided where applicable. See the section about data set options in *SAS/ACCESS for Relational Databases: Reference* for detailed information about these options.

**Table 1.2** SAS/ACCESS Data Set Options for Teradata

| Option | Default Value |
|---|---|
| BUFFERS= | 2 |
| BULKLOAD= | NO |
| CAST= | none |
| CAST_OVERHEAD_MAXPERCENT= | 20 percent |
| DATABASE= (see SCHEMA=) | none |
| DBCOMMIT= | the current LIBNAME option setting |
| DBCONDITION= | none |
| DBCREATE_TABLE_OPTS= | the current LIBNAME option setting |
| DBFORCE= | NO |
| DBGEN_NAME= | DBMS |
| DBINDEX= | the current LIBNAME option setting |
| DBKEY= | none |

| Option | Default Value |
|---|---|
| DBLABEL= | NO |
| DBMASTER= | none |
| DBNULL= | none |
| DBSASLABEL= | COMPAT |
| DBSLICE= | none |
| DBSLICEPARM= | THREADED_APPS,2 |
| DBTYPE= | see "Data Types for Teradata" on page 35 |
| ERRLIMIT= | 1 |
| MBUFFSIZE= | 0 |
| ML_CHECKPOINT= | 0 |
| ML_ERROR1= | none |
| ML_ERROR2= | none |
| ML_LOG= | none |
| ML_RESTART= | none |
| ML_WORK= | none |
| MULTILOAD= | NO |
| NULLCHAR= | SAS |
| NULLCHARVAL= | a blank character |
| PRESERVE_COL_NAMES= | YES |
| READ_ISOLATION_LEVEL= | the current LIBNAME option setting |
| READ_LOCK_TYPE= | the current LIBNAME option setting |
| READ_MODE_WAIT= | the current LIBNAME option setting |
| SASDATEFORMAT= | none |
| SCHEMA= | the current LIBNAME option setting |
| SLEEP= | 6 |
| TENACITY= | 4 |
| UPDATE_ISOLATION_LEVEL= | the current LIBNAME option setting |
| UPDATE_LOCK_TYPE= | the current LIBNAME option setting |
| UPDATE_MODE_WAIT= | the current LIBNAME option setting |

# Pass-Through Facility Specifics for Teradata

See the section about the Pass-Through Facility in *SAS/ACCESS for Relational Databases: Reference* for general information about this feature.

The Pass-Through Facility specifics for Teradata are as follows:

- The *dbms-name* is **`TERADATA`**.
- The CONNECT statement is required.

- The Teradata interface can connect to multiple Teradata servers and to multiple Teradata databases. However, if you use multiple, simultaneous connections, you must use an *alias* argument to identify each connection.
- The CONNECT statement *database-connection-arguments* are identical to the LIBNAME connection options.

  In addition, the MODE= option is available with the CONNECT statement. By default, SAS/ACCESS opens Teradata connections in ANSI mode. In contrast, most Teradata tools, such as BTEQ, run in Teradata mode. If you specify MODE=TERADATA, Pass-Through connections open in Teradata mode, forcing Teradata mode rules for all SQL requests that are passed to the Teradata DBMS. For example, MODE= impacts transaction behavior and case-sensitivity. See your Teradata SQL reference manual for a complete discussion of ANSI versus Teradata mode.
- By default, SAS/ACCESS opens Teradata in ANSI mode, so you must use one of the following techniques when you write PROC SQL steps that use the Pass-Through Facility:
  - Specify an explicit COMMIT statement to close a transaction. You must also specify an explicit COMMIT statement after any Data Definition Language (DDL) statement. The examples below demonstrate these rules. For further information about ANSI mode and DDL statements, see your Teradata SQL reference manual.
  - Specify MODE=TERADATA in your CONNECT statement. When MODE=TERADATA, you do not specify explicit COMMIT statements as described above. When MODE=TERADATA, data processing is case insensitive. This option is available only when you are using the Pass-Through Facility.

***CAUTION:***

Do not issue a Teradata DATABASE statement within the EXECUTE statement in PROC SQL. Add the SCHEMA= option to your CONNECT statement if you must change the default Teradata database. △

## Examples

In the following example, SAS/ACCESS connects to the Teradata DBMS using the alias **dbcon**.

```
proc sql;
   connect to teradata as dbcon (user=testuser pass=testpass);
quit;
```

In the following example, SAS/ACCESS connects to the Teradata DBMS using the alias **tera**, drops and then recreates the SALARY table, inserts two rows, and then disconnects from the Teradata DBMS. Notice that COMMIT must follow each DDL statement. DROP TABLE and CREATE TABLE are DDL statements. The COMMIT statement that follows the INSERT statement is required too; otherwise Teradata rolls back the inserted rows.

```
proc sql;
   connect to teradata as tera ( user=testuser password=testpass );
   execute (drop table salary) by tera;
   execute (commit) by tera;
   execute (create table salary (current_salary float, name char(10)))
         by tera;
```

```
   execute (commit) by tera;
   execute (insert into salary values (35335.00, 'Dan J.')) by tera;
   execute (insert into salary values (40300.00, 'Irma L.')) by tera;
   execute (commit) by tera;
   disconnect from tera;
quit;
```

In the following example, SAS/ACCESS connects to the Teradata DBMS using the alias **tera**, updates a row, and then disconnects from the Teradata DBMS. The COMMIT statement causes Teradata to commit the update request. Without the COMMIT statement, Teradata rolls back the update.

```
 proc sql;
   connect to teradata as tera ( user=testuser password=testpass );
   execute (update salary set current_salary=45000
            where (name='Irma L.')) by tera;
   execute (commit) by tera;
   disconnect from tera;
  quit;
```

In the following example, SAS/ACCESS connects to the Teradata database using the alias **tera2**, selects all rows in the SALARY table and displays them using PROC SQL, and then disconnects from the Teradata database. A COMMIT statement is not needed in this example, because the operations are only reading data; no changes are made to the database.

```
proc sql;
   connect to teradata as tera2 ( user=testuser password=testpass );
   select * from connection to tera2 (select * from salary);
   disconnect from tera2;
  quit;
```

In the following example, MODE=TERADATA is specified to obtain case insensitive behavior. Because Teradata Mode is used, SQL COMMIT statements are not required.

```
/* Create and populate table in Teradata Mode (case insensitive)*/
proc sql;
   connect to teradata (user=testuser pass=testpass mode=teradata);
   execute(create table casetest(x char(28)) ) by teradata;
   execute(insert into casetest values('Case Insensitivity Desired') ) by teradata;
quit;
/* Query table in Teradata Mode (for case insensitive match) */
proc sql;
   connect to teradata (user=testuser pass=testpass mode=teradata);
   select * from connection to teradata (select * from
   casetest where x='case insensitivity desired');
quit;
```

# Autopartitioning Scheme for Teradata

See the section about threaded reads in *SAS/ACCESS for Relational Databases: Reference* for general information about this feature.

## Overview

The FastExport Utility is the fastest means available for reading large Teradata tables. FastExport is NCR-provided software that delivers data over multiple Teradata connections, or sessions. If FastExport is available, SAS threaded reads use it. If unavailable, SAS threaded reads generate partitioning WHERE clauses. Use of the DBSLICE= option overrides FastExport. If you have FastExport available and want to use it, do not use DBSLICE=. To use FastExport everywhere possible, use DBSLICEPARM= with the ALL qualifier.

*Note:* On OS/390, z/OS, and Unix, only FastExporting is supported. Partitioning WHERE clauses, either automatically generated or created by using DBSLICE=, are not supported. △

## FastExport and Case Sensitivity

In certain situations, Teradata returns different row results to SAS when using FastExport, as compared to reading normally without FastExport. The difference arises only when all of the following conditions are met:

- A WHERE clause is asserted that compares a character column with a character literal.
- The column definition is NOT CASESPECIFIC.

  Unless you specify otherwise, most Teradata native utilities create NOT CASESPECIFIC character columns. The SAS/ACCESS interface to Teradata, on the other hand, creates CASESPECIFIC columns. In general, this means that you will not see result differences with SAS created tables but you might with tables created by Teradata utilities, which are frequently many of your tables. To determine how a table is created, look at your column declarations with Teradata's SHOW TABLE statement.
- A character literal matches to a column value that differs only in case.

  You will see differences in the rows returned if your character column has mixed-case data that is otherwise identical. For example, 'Top' and 'top' are identical except for case.

Case sensitivity is an issue when SAS generates SQL code that contains a WHERE clause with one or more character comparisons. It is also an issue when you supply the Teradata SQL yourself with the explicit SQL feature of PROC SQL. The following are examples of each scenario, using DBSLICEPARM=ALL to invoke FastExport instead of the normal SAS read:

```
/*SAS generates the SQL for you*/
libname trlib teradata user=username password=userpwd dbsliceparm=all;
proc print data=trlib.employees;
where lastname='lovell';
run;

/*Use explicit SQL with PROC SQL and supply the SQL yourself, also invoking FastExport*/
proc sql;
   connect to teradata(user=username password=userpwd dbsliceparm=all);
select * from connection to teradata
   (select * from sales where gender='f' and salesamt>1000);
quit;
```

For more information about case sensitivity, consult your Teradata documentation.

## FastExport Setup

There are three requirements for using FastExport with SAS:

- You must have the Teradata FastExport Utility present on your system. If you do not have FastExport and want to use it with SAS, contact NCR to obtain the Utility.
- SAS must be able to locate the FastExport Utility on your system.
- The FastExport Utility must be able to locate the SasAxsm access module, which is supplied with your SAS/ACCESS interface to Teradata product. SasAxsm is in the SAS directory tree, in the same location as the sasiotra component.

Assuming you have the Teradata FastExport Utility, perform the following setup, which varies by system:

- Windows

  As needed, modify your Path environment variable to include *both* the directories containing Fexp.exe (FastExport) and SasAxsm. Place these directory specifications last in your path.
- UNIX

  As needed, modify your library path environment variable to include the directory containing sasaxsm.sl (HP) or sasaxsm.so (Solaris and AIX). These shared objects are delivered in the $SASROOT/sasexe directory. You may copy these modules where you wish, but ensure that the directory you copy them into is in the appropriate shared library path environment variable. On Solaris, the library path variable is LD_LIBRARY_PATH. On HP-UX, it is SHLIB_PATH. On AIX, it is LIBPATH. Also, make sure that the directory containing the Teradata FastExport utility (fexp), is included in the PATH environment variable. FastExport is usually installed in the /usr/bin directory.
- z/OS

  No action is needed when invoking FastExport under TSO. When invoking FastExport with a batch JCL, the SAS source statements must be assigned to a DD name other than SYSIN. This can be done by passing a parameter such as SYSIN=SASIN in the JCL where all the SAS source statements are assigned to the DD name SASIN.

Keep in mind that future releases of SAS might require an updated version of SasAxsm. Therefore, when upgrading to a new SAS version, you should update the path for SAS on Windows and the library path for SAS on UNIX.

## FastExport Usage

To utilize FastExport, SAS writes a specialized script to a disk that is read by the FastExport Utility. SAS might also log FastExport log lines to another disk file. These files are created and deleted by SAS on your behalf, and require no intervention. Sockets deliver the data from FastExport to SAS, so aside from installing the SasAxsm access module that enables the data transfer, no action is needed on your part.

On Windows, when the FastExport Utility is active, a DOS window appears minimized as an icon on your toolbar. You can maximize the DOS window, but do not close it. After a FastExport operation is complete, SAS closes the window for you.

The following example demonstrates the creation of a SAS data set that is a subset of a Teradata table, using FastExport to transfer the data:

```
libname trlib teradata user=username password=userpwd;
data saslocal(keep=EMPID SALARY);
```

```
set trlib.employees(dbsliceparm=all);
run;
```

## FastExport and Explicit SQL

FastExport is also supported for the explicit SQL feature of PROC SQL.

The following example demonstrates the creation of a SAS data set that is a subset of a Teradata table, using explicit SQL and FastExport to transfer the data.

```
proc sql;
connect to teradata as pro1 (user=username password=userpwd dbsliceparm=all);
create table saslocal as select * from connection to pro1
   (select EMPID, SALARY from employees);
quit;
```

FastExport for explicit SQL is a Teradata extension only, for optimizing read operations, and is not covered in the threaded read documentation.

## FastExport Usage Exceptions

With the Teradata FastExport Utility and the SAS supplied SasAxsm module in place, FastExport works automatically for all SAS steps that have threaded reads enabled, except for one situation. FastExport does not handle single AMP queries. In this case, SAS/ACCESS simply reverts to a normal single connection read. For information about FastExport and single AMP queries, refer to your Teradata documentation.

To determine if FastExport worked, turn on SAS tracing in advance of the step that attempts to use FastExport. If FastExport is used, you will see the following message (English only) written to your SAS log:

```
sasiotra/tryottrm(): SELECT was processed with FastExport.
```

To turn on SAS tracing, issue the following statement:

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

## Threaded Reads with Partitioning WHERE Clauses

If FastExport is unavailable, threaded reads use partitioning WHERE clauses. You can create your own partitioning WHERE clauses using the DBSLICE= option. Otherwise, SAS/ACCESS to Teradata attempts to generate them on your behalf. Like other SAS/ACCESS interfaces, this partitioning is based on the MOD function. In order to generate partitioning WHERE clauses, SAS/ACCESS to Teradata must locate a table column suitable for applying MOD. The following types are eligible:

- BYTEINT
- SMALLINT
- INTEGER
- DATE
- DECIMAL (integral DECIMAL columns only)

A DECIMAL column is only eligible if the column definition restricts it to integer values. In other words, the DECIMAL column must be defined with a scale of zero.

If the table you are reading contains more than one column of the above mentioned types, SAS/ACCESS to Teradata applies some nominal intelligence to select a best choice. Top priority is given to the primary index, if it is MOD-eligible. Otherwise, preference is given to any column that is defined as NOT NULL. Since this is an unsophisticated set of selection rules, you might want to supply your own partitioning using the DBSLICE= option.

To view your table's column definitions, use Teradata's SHOW TABLE statement.

*Note:* Partitioning WHERE clauses, either automatically generated or created by using DBSLICE=, are not supported on z/OS. △

### FastExport Versus Partitioning WHERE Clauses

Partitioning WHERE clauses are innately less efficient than FastExport. The Teradata DBMS must process separate SQL statements that vary in the WHERE clause. In contrast, FastExport is optimal because only one SQL statement is transmitted to the Teradata DBMS. However, older editions of the Teradata DBMS place severe restrictions on the system-wide number of simultaneous FastExport operations allowed. Even with newer versions of Teradata, your database administrator might be concerned about large numbers of FastExport operations.

Threaded reads with partitioning WHERE clauses also place higher workload on Teradata and might not be appropriate on a widespread basis. Both technologies expedite throughput between SAS and the Teradata DBMS, but should be used judiciously. For this reason, only SAS threaded applications are eligible for threaded read by default. To enable more threaded reads or to turn them off entirely, use the DBSLICEPARM= option.

Even when FastExport is available, you can force SAS/ACCESS to Teradata to generate partitioning WHERE clauses on your behalf. This is accomplished with the DBI argument to the DBSLICEPARM= option (DBSLICEPARM=DBI). This feature is available primarily to enable comparisons of these techniques. In general, FastExport should be used if it is available.

FastExport is supported for the explicit SQL feature of PROC SQL. Partitioning of WHERE clauses is not supported for explicit SQL.

## Passing SAS Functions to Teradata

The interface to Teradata passes the following SAS functions to Teradata for processing. See the section about optimizing SQL usage in *SAS/ACCESS for Relational Databases: Reference* for information.

ABS

AVG

EXP

LOG

LOG10

LOWCASE

MAX

MIN

SQRT

STRIP (TRIM)

SUBSTR

TODAY

UPCASE

SUM

COUNT

# Passing Joins to Teradata

In order for a multiple libref join to pass to Teradata, all of the following components of the LIBNAME statements must match exactly:

- □ user ID
- □ password
- □ server
- □ account

See the section about performance considerations in *SAS/ACCESS for Relational Databases: Reference* for more information about when and how SAS/ACCESS passes joins to the DBMS.

# Temporary Table Support for Teradata

See the section on the temporary table support in *SAS/ACCESS for Relational Databases: Reference* for general information about this feature.

## Establishing a Temporary Table

When you specify CONNECTION=GLOBAL, you can reference a temporary table throughout a SAS session, in both DATA steps and procedures. Due to a limitation of Teradata, FastLoad and FastExport do *not* support the use of temporary tables at this time.

Teradata supports two types of temporary tables, global and volatile. With the use of global temporary tables, the rows are deleted after the connection is closed but the table definition itself remains. With volatile temporary tables, the table (and all rows) are dropped when the connection is closed.

When accessing a volatile table with a LIBNAME statement, it is recommended that you do *not* use these options:

- □ DATABASE= (as a LIBNAME option)
- □ SCHEMA= (as a LIBNAME option or a data set option)

If you use either DATABASE= or SCHEMA=, then the LIBNAME statement *must* specify the option DBMSTEMP=YES to denote that all tables accessed through it, and all tables that are created by it, will be volatile tables.

DBMSTEMP= will also cause all table names to be not fully qualified for either SCHEMA= or DATABASE=. In this case, the LIBNAME statement should only be used to access tables (either permanent or volatile) within the your default database/schema.

## Terminating a Temporary Table

You can drop a temporary table at any time, or allow it to be implicitly dropped when the connection is terminated. Temporary tables do not persist beyond the scope of a single connection.

## Examples

The following is an example of temporary table use:

```
/* Global connection for all tables */
libname x teradata user=test pw=test server=boom connection=global;

/* Create global temporary table and store in current database schema */
proc sql;
   connect to teradata(user=test pw=test server=boom connection=global);
   execute (CREATE GLOBAL TEMPORARY TABLE temp1 (col1 INT )
            ON COMMIT PRESERVE ROWS) by teradata;
   execute (COMMIT WORK) by teradata;
quit;

/* Insert one row into the temporary table, materializing the table */
proc sql;
   connect to teradata(user=test pw=test server=boom connection=global);
   execute (INSERT INTO temp1 VALUES(1)) by teradata;
   execute (COMMIT WORK) by teradata;
quit;

/* Access the temporary table through the global libref */
data work.new_temp1;
set x.temp1;
run;

/* Access the temporary table through the global connection */
proc sql;
   connect to teradata (user=test pw=test server=boom connection=global);
   select * from connection to teradata (select * from temp1);
quit;

/* Drop the temporary table */
proc sql;
   connect to teradata(user=prboni pw=prboni server=boom connection=global);
   execute (DROP TABLE temp1) by teradata;
   execute (COMMIT WORK) by teradata;
quit;
```

The following is an example of volatile table use:

```
/* Global connection for all tables */
libname x teradata user=test pw=test server=boom connection=global;

/* Create a volatile table */
proc sql;
   connect to teradata(user=test pw=test server=boom connection=global);
   execute (CREATE VOLATILE TABLE temp1 (col1 INT)
```

```
            ON COMMIT PRESERVE ROWS) by teradata;
   execute (COMMIT WORK) by teradata;
quit;

/* Insert one row into the volatile table */
proc sql;
   connect to teradata(user=test pw=test server=boom connection=global);
   execute (INSERT INTO temp1 VALUES(1)) by teradata;
   execute (COMMIT WORK) by teradata;
quit;

/* Access the temporary table through the global libref */
data _null_;
   set x.temp1;
   put _all_;
run;

/* Access the volatile table through the global connection*/
proc sql;
   connect to teradata (user=test pw=test server=boom connection=global);
   select * from connection to teradata (select * from temp1);
quit;

/* Drop the connection, the volatile table is automatically dropped */
libname x clear;

/* to convince yourself it's gone, try to access it */
libname x teradata user=test pw=test server=boom connection=global;

/* it's not there */
proc print data=x.temp1;
run;
```

# Maximizing Teradata Read Performance

A major objective of SAS/ACCESS when you are reading DBMS tables is to take advantage of Teradata's rate of data transfer. The DBINDEX=, SPOOL= and PREFETCH= options can help you achieve optimal read performance.

## About the PreFetch Facility

PreFetch is a SAS/ACCESS for Teradata facility that speeds up a SAS job by exploiting the parallel processing capability of Teradata. To obtain benefit from the facility, your SAS job must run more than once and have the following characteristics:

- use SAS/ACCESS to query Teradata DBMS tables
- should *not* contain SAS statements that create, update, or delete Teradata DBMS tables
- run SAS code that changes infrequently or not at all.

In brief, the ideal job is a stable read-only SAS job.

Use of PreFetch is optional. To use the facility, you must explicitly enable it with the LIBNAME option PREFETCH.

## How PreFetch Works

When reading DBMS tables, SAS/ACCESS submits SQL statements on your behalf to Teradata. Each SQL statement that is submitted has an execution cost: the amount of time Teradata spends processing the statement before it returns the requested data to SAS/ACCESS.

When PreFetch is enabled, the first time you run your SAS job, SAS/ACCESS identifies and selects statements with a high execution cost. SAS/ACCESS then stores (caches) the selected SQL statements to one or more Teradata macros that it creates.

On subsequent runs of the job, when PreFetch is enabled, SAS/ACCESS extracts statements from the cache and submits them to Teradata in advance. The rows selected by these SQL statements are immediately available to SAS/ACCESS because Teradata 'prefetches' them. Your SAS job runs faster because PreFetch reduces the wait for SQL statements with a high execution cost. However, PreFetch improves elapsed time only on subsequent runs of a SAS job. During the first run, SAS/ACCESS only creates the SQL cache and stores selected SQL statements; no prefetching is performed.

## The PreFetch Option Arguments

*unique_storename*

As mentioned, when PreFetch is enabled, SAS/ACCESS creates one or more Teradata macros to store the selected SQL statements that PreFetch caches. You can easily distinguish a PreFetch macro from other Teradata macros. The PreFetch Teradata macro contains a comment that is prefaced with the text,

```
"SAS/ACCESS PreFetch Cache"
```

The name that the PreFetch facility assigns for the macro is the value that you enter for the *unique_storename* argument. The *unique_storename* must be unique. Do not specify a name that exists in the Teradata DBMS already for a DBMS table, view or macro. Also, do not enter a name that exists already in another SAS job that employs the Prefetch facility.

*#sessions*

This argument specifies how many cached SQL statements SAS/ACCESS submits in parallel to Teradata. In general, your SAS job completes faster if you increase the number of statements that Teradata works on in advance. However, a large number (too many sessions) can strain client and server resources. A valid value is 1 through 9. If you do not specify a value for this argument, the default is 3.

In addition to the specified number of sessions, SAS/ACCESS adds an additional session for submitting SQL statements that are not stored in the PreFetch cache. Thus, if the default is 3, SAS/ACCESS actually opens up to 4 sessions on the Teradata server.

*algorithm*

This argument is present to handle future enhancements. Currently PreFetch only supports one algorithm, SEQUENTIAL.

## When and Why Use PreFetch

If you have a read-only SAS job that runs frequently, this is an ideal candidate for PreFetch; for example, a daily job that extracts data from Teradata tables. To help you decide when to use PreFetch, consider the following daily jobs:

- *Job 1*

  Reads and collects data from the Teradata DBMS.

- *Job 2*

  Contains a WHERE clause that reads in values from an external, variable data source. As a result, the SQL code that the job submits through a Teradata LIBNAME statement or through PROC SQL changes from run to run.

In these examples, Job 1 is an excellent candidate for the facility. In contrast, Job 2 is not. Using PreFetch with Job 2 does not return incorrect results, but can impose a performance penalty. PreFetch uses stored SQL statements. Thus, Job 2 is not a good candidate because the SQL statements that the job generates with the WHERE clause change each time the job is run. Consequently, the SQL statements that the job generates never match the statements that are stored.

The impact of Prefetch on processing performance varies by SAS job. Some jobs improve elapsed time 5% or less; others improve elapsed time 25% or more.

## Possible Unexpected Results

It is unlikely, but possible, to write a SAS job that delivers unexpected or incorrect results. This can occur if the job contains code that waits on some Teradata or system event before proceeding. For example, SAS code that pauses the SAS job until another user updates a given data item in a Teradata table. Or, SAS code that pauses the SAS job until a given time; for example, 5:00 p.m. In both cases, PreFetch would generate SQL statements in advance. But, table results from these SQL statements would not reflect data changes that are made by the scheduled Teradata or system event.

## PreFetch Processing of Unusual Conditions

PreFetch is designed to handle unusual conditions gracefully. Some of these unusual conditions include:

Condition: Your job contains SAS code that creates updates, or deletes Teradata tables.

PreFetch is designed only for read operations and is disabled when it encounters a non-read operation. The facility returns a performance benefit up to the point where the first non-read operation is encountered. After that, SAS/ACCESS disables the PreFetch facility and continues processing.

Condition: Your SQL cache name (*unique_storename* value) is identical to the name of a Teradata table.

PreFetch issues a warning message. SAS/ACCESS disables the PreFetch facility and continues processing.

Condition: You change your SAS code for a job that has PreFetch enabled.

PreFetch detects that the SQL statements for the job changed and deletes the cache. SAS/ACCESS disables Prefetch and continues processing. The next time that you run the job, PreFetch creates a fresh cache.

Condition: Your SAS job encounters a PreFetch cache that was created by a different SAS job.

PreFetch deletes the cache. SAS/ACCESS disables Prefetch and continues processing. The next time that you run the job, PreFetch creates a fresh cache.

Condition: You remove the PreFetch option from an existing job.

Prefetch is disabled. Even if the SQL cache (Teradata macro) still exists in your database, SAS/ACCESS ignores it.

Condition: You accidentally delete the SQL cache (the Teradata macro created by PreFetch) for a SAS job that has PreFetch enabled.

SAS/ACCESS simply rebuilds the cache on the next run of the job. In subsequent job runs, PreFetch continues to enhance performance.

## Using PreFetch as a LIBNAME Option

If you specify the PREFETCH= option in a LIBNAME statement, PreFetch applies the option to tables read by the libref.

*Note:* If you have more than one LIBNAME in your SAS job, and you specify PREFETCH= for each LIBNAME, remember to make the SQL cache name for each LIBNAME unique. △

This example applies PREFETCH= to one of two librefs. During the first job run, PreFetch stores SQL statements for tables referenced by the libref ONE in a Teradata macro named PF_STORE1 for reuse later.

```
libname one teradata user=testuser password=testpass
 prefetch='pf_store1';
libname two teradata user=larry password=riley;
```

This example applies PREFETCH= to multiple librefs. During the first job run, PreFetch stores SQL statements for tables referenced by the libref EMP to a Teradata macro named EMP_SAS_MACRO and SQL statements for tables referenced by the libref SALE to a Teradata macro named SALE_SAS_MACRO.

```
libname emp teradata user=testuser password=testpass
 prefetch='emp_sas_macro';
libname sale teradata user=larry password=riley
 prefetch='sale_sas_macro';
```

## Using Prefetch as a Global Option

Unlike other Teradata LIBNAME options, you can also invoke PreFetch globally for a SAS job. To do this, place the OPTION DEBUG= statement in your SAS program before all LIBNAME statements and PROC SQL steps. If your job contains multiple LIBNAME statements, the global PreFetch invocation creates a uniquely named SQL cache name for each of the librefs.

*Note:* Do not be confused by the DEBUG= option here. It is merely a mechanism to deliver the PreFetch capability globally. PreFetch is not for debugging; it is a supported feature of SAS/ACCESS for Teradata. △

In this example, the first time you run the job with PreFetch enabled, the facility creates three Teradata macros: UNIQUE_MAC1, UNIQUE_MAC2, and UNIQUE_MAC3. In subsequent runs of the job, PreFetch extracts SQL statements from these Teradata macros, enhancing the job performance across all three librefs referenced by the job.

```
option debug="PREFETCH(unique_mac,2,SEQUENTIAL)";
libname one teradata user=kamdar password=ellis;
libname two teradata user=kamdar password=ellis
                          database=larry;
libname three teradata user=kamdar password=ellis
                            database=wayne;
proc print data=one.kamdar_goods;
run;
proc print data=two.larry_services;
run;
proc print data=three.wayne_miscellaneous;
run;
```

In this example, PreFetch selects the algorithm, that is, the order of the SQL statements. (The OPTION DEBUG= statement must be the first statement in your SAS job.)

```
option debug='prefetch(pf_unique_sas,3)';
```

In this example, the user specifies for PreFetch to use the SEQUENTIAL algorithm. (The OPTION DEBUG= statement must be the first statement in your SAS job.)

```
option debug='prefetch(sas_pf_store,3,sequential)';
```

# Maximizing Teradata Load Performance

## Maximizing Teradata Load Performance with FastLoad

### FastLoad Supported Features and Restrictions

The SAS/ACCESS interface to Teradata supports a bulk-load capability, called FastLoad, which greatly accelerates insertion of data into empty Teradata tables. For general information about using FastLoad and error recovery, see Teradata's FastLoad documentation.

*Note:* Implementation of the SAS/ACCESS FastLoad facility will change in a future release of SAS. Consequently, you might need to change SAS programming statements and options that you specify to enable this feature in the future. △

The SAS/ACCESS FastLoad facility is similar to the native Teradata FastLoad Utility. They share the following limitations:

- FastLoad can load only empty tables; it cannot append to a table that already contains data. If you attempt to use FastLoad when appending to a table that contains rows, the append step fails.
- Both the Teradata FastLoad Utility and the SAS/ACCESS FastLoad facility log data errors to tables. Error recovery can be difficult. You must refer to Teradata's FastLoad documentation to find the error that corresponds to the code stored in the error table.
- FastLoad does not load duplicate rows (rows where all corresponding fields contain identical data) into a Teradata table. If your SAS data set contains duplicate rows, you can use the normal insert (load) process.

### Invoking FastLoad

If you do not specify FastLoad, your Teradata tables are loaded normally (slowly). To invoke FastLoad in the SAS/ACCESS interface, you can use one of the following:

- the data set option BULKLOAD=YES in a processing step that populates an empty Teradata table.
- the LIBNAME option BULKLOAD=YES on the destination libref (the Teradata DBMS library where the intended table(s) is to be created and loaded).
- the FASTLOAD= alias for either of these options.

## FastLoad Data Set Options

The following data set options are available for use with the FastLoad facility:

- BL_LOG=

  specifies the names of the error tables that are created when you are using the SAS/ACCESS FastLoad facility. By default, FastLoad errors are logged in Teradata tables named SAS_FASTLOAD_ERRS1_*randnum* and SAS_FASTLOAD_ERRS2_*randnum* where *randnum* is a randomly generated number.

  For example, if you specify **`BL_LOG=my_load_errors`**, errors are logged in tables *my_load_errors1* and *my_load_errors2*. If you specify **`BL_LOG=errtab`**, errors are logged in tables name *errtab1* and *errtab2*.

  *Note:* SAS/ACCESS automatically deletes the error tables if no errors are logged. If there are errors, the tables are retained, and SAS/ACCESS issues a warning message that includes the names of the error tables. △

- DBCOMMIT=*n*

  causes a Teradata "checkpoint" after each group of *n* rows are transmitted. The use of checkpoints slows performance but provides known synchronization points if there is a failure during the loading process.

  *Note:* If BULKLOAD=YES, and DBCOMMIT= is not explicitly set, then the default is that checkpoints are not used. △

  The Teradata alias for this option is CHECKPOINT=.

See the section about data set options in *SAS/ACCESS for Relational Databases: Reference*for additional information about these options.

---

# Maximizing Teradata Load Performance with MultiLoad

## MultiLoad Supported Features and Restrictions

The SAS/ACCESS interface to Teradata supports a bulk-load capability called MultiLoad that greatly accelerates insertion of data into Teradata tables. For general information about using MultiLoad with Teradata tables and for information about error recovery, see Teradata's MultiLoad documentation. SAS/ACCESS examples"Examples" on page 24 are available.

Unlike FastLoad, which only loads empty tables, MultiLoad loads both empty and existing Teradata tables. If you do not specify MultiLoad, your Teradata tables are loaded normally (inserts are sent one row at a time).

The SAS/ACCESS MultiLoad facility loads both empty and existing Teradata tables. SAS/ACCESS supports these features:

- Only one target table can be loaded at a time.
- Only insert operations are supported.

The SAS/ACCESS MultiLoad facility is similar to the native Teradata MultiLoad utility. They share the following limitations:

- Unique secondary indices on the target tables must be dropped prior to the load.

- Foreign key references on the target table must be dropped prior to the load.
- Join indices on the target table must be dropped prior to the load.
- Duplicate rows are not loaded.

Both the Teradata MultiLoad utility and the SAS/ACCESS MultiLoad facility log data errors to tables. Error recovery can be difficult, but the ability to restart from the last checkpoint is possible. You must refer to Teradata's MultiLoad documentation to find the error that corresponds to the code stored in the error table.

## MultiLoad Setup

The following are requirements for using the MultiLoad bulk-load capability in SAS:

- The native Teradata MultiLoad utility must be present on your system. If you do not have the Teradata MultiLoad utility and you want to use it with SAS, contact NCR to obtain the utility.
- SAS must be able to locate the Teradata MultiLoad utility on your system.
- The Teradata MultiLoad utility must be able to locate the SASMlam access module and the SasMlne exit routine. They are supplied with the SAS/ACCESS Interface to Teradata software.
- SAS MultiLoad requires Teradata client TTU 8.0 or later.

If it has not been done so already as part of the post-installation configuration process, refer to the SAS configuration documentation for your system for information about how to configure SAS to work with MultiLoad.

## MultiLoad Data Set Options

Invoke the SAS/ACCESS MultiLoad facility by specifying MULTILOAD=YES. See the data set option MULTILOAD= for detailed information and examples on loading data and recovering from errors during the load process.

The following data set options are available for use with the MultiLoad facility:

- ML_LOG= specifies a prefix for the temporary tables used by the Teradata MultiLoad utility during the load process. The MultiLoad utility uses a log table, two error tables, and a work table while loading data to the target table. These tables are named by default as SAS_ML_RS_*randnum*, SAS_ML_ET_*randnum*, SAS_ML_UT_*randnum*, and SAS_ML_WT_*randnum* where *randnum* is a randomly generated number.

  ML_LOG= is used to override the default names used. For example, if you specify **`ML_LOG=MY_LOAD`** the log table is named **`MY_LOAD_RS`**. Errors are logged in tables **`MY_LOAD_ET`** and **`MY_LOAD_UT`**. The work table is named **`MY_LOAD_WT`**.

  *Note:* Note: SAS/ACCESS automatically deletes the error tables if no errors are logged. If there are errors, the tables are retained, and SAS/ACCESS issues a warning message that includes the names of the tables in error. △

- ML_RESTART= allows the user to name the log table that MultiLoad will use for tracking checkpoint information. By default, the log table is named SAS_ML_RS_*randnum* where *randnum* is a random number. When restarting a failed MultiLoad job, you need to specify the same log table from the earlier run so that the MultiLoad job can restart correctly. Note that the same error tables and work table must also be specified upon restarting the job, using ML_ERROR1, ML_ERROR2, and ML_WORK data set options. ML_RESTART and ML_LOG are mutually exclusive and cannot be specified together.
- ML_ERROR1= allows the user to name the error table that MultiLoad will use for tracking errors from the acquisition phase. Please refer to Teradata's MultiLoad

reference for more information on what is stored in this table. By default, the acquisition error table is named SAS_ML_ET_*randnum* where *randnum* is a random number.

When restarting a failed MultiLoad job, you need to specify the same acquisition table from the earlier run so that the MultiLoad job can restart correctly. Note that the same log table, application error table, and work table must also be specified upon restarting, using ML_RESTART, ML_ERROR2, and ML_WORK data set options. ML_ERROR1 and ML_LOG are mutually exclusive and cannot be specified together.

- ML_ERROR2= allows the user to name the error table that MultiLoad will use for tracking errors from the application phase. Please refer to Teradata's MultiLoad reference for more information on what is stored in this table. By default, the acquisition error table is named SAS_ML_UT_*randnum* where *randnum* is a random number.

  When restarting a failed MultiLoad job, you need to specify the same application table from the earlier run so that MultiLoad can restart correctly. Note that the same log table, acquisition error table, and work table must also be specified upon restarting the job using ML_RESTART, ML_ERROR1, and ML_WORK data set options.

  ML_ERROR2 and ML_LOG are mutually exclusive and cannot be specified together.

- ML_WORK= allows the user to name the work table that MultiLoad will use for loading the target table. Please refer to Teradata's MultiLoad reference for more information on what is stored in this table. By default, the work table is named SAS_ML_WT_*randnum* where *randnum* is a random number.

  When restarting a failed MultiLoad job, you need to specify the same work table from the earlier run so that the MultiLoad job can restart correctly. Note that the same log table, acquisition error table and application error table must also be specified upon restarting the job using ML_RESTART, ML_ERROR1, and ML_ERROR2 data set options.

  ML_WORK and ML_LOG are mutually exclusive and cannot be specified together.

- ML_CHECKPOINT= specifies the checkpoint rate. ML_CHECKPOINT=0 is the default; no checkpoints are taken if the default is used. If the value of ML_CHECKPOINT= is between 1 and 59 inclusive, checkpoints are taken at the specified interval in minutes. If ML_CHECKPOINT= is greater than or equal to 60, then a checkpoint operation occurs after a multiple of the specified rows are loaded.

  ML_CHECKPOINT= functions very much like the CHECKPOINT in the native Teradata MultiLoad utility, but it functions very differently from the DBCOMMIT= data set option. Note that DBCOMMIT= is disabled for MultiLoad to prevent any conflict.

- SLEEP= specifies the number of minutes that MultiLoad waits before it retries a logon operation when the maximum number of utilities are already running on the Teradata database. The default value is 6. SLEEP= functions very much like the SLEEP run-time option of the native Teradata MultiLoad utility.
- TENACITY= specifies the number of hours that MultiLoad tries to log on when the maximum number of utilities are already running on the Teradata database. The default value is 4. TENACITY= functions very much like the TENACITY run-time option of the native Teradata MultiLoad utility.
- MBUFFSIZE= sets the size of the buffer used for data transfer. The default size of each buffer used for data transfer is 64K. This size can be increased up to 1MB using the MBUFSIZE=.

*Note:* Be aware that these options are disabled while you are using the SAS/ACCESS MultiLoad facility:

- The LIBNAME and data set options DBCOMMIT= are disabled because DBCOMMIT= functions very differently from CHECKPOINT of the native Teradata MultiLoad utility.
- The data set option ERRLIMIT= is disabled because the number of errors is not known until all the records have been sent to MultiLoad. The default value of ERRLIMIT=1 is not honored.

△

## Examples

The following example invokes the FastLoad facility.

```
libname fload teradata user=testuser password=testpass;
data fload.nffloat(bulkload=yes);
     do x=1 to 1000000;
        output;
     end;
run;
```

The following example uses FastLoad to append SAS data to an empty Teradata table and specifies the BL_LOG= option to name the error tables **`Append_Err1`** and **`Append_Err2`**. (In practice, applications typically append many rows.)

```
/* create the empty Teradata table */
proc sql;
     connect to teradata as tera(user=testuser password=testpass);
     execute (create table performers
             (userid int, salary decimal(10,2), job_desc char(50)))
              by tera;
     execute (commit) by tera;
quit;

/* create the SAS data to be loaded */
data local;
     input userid 5. salary 9. job_desc $50.;
           datalines;
             0433 35993.00 grounds keeper
             4432 44339.92 code groomer
             3288 59000.00 manager
             ;

/* append the SAS data and name the Teradata error tables */
libname tera teradata user=testuser password=testpass;

proc append data=local base=tera.performers
     (bulkload=yes bl_log=append_err);
run;
```

The following example invokes the MultiLoad facility.

```
libname trlib teradata user=testuser pw=testpass server=dbc;

/* MultiLoad a table with 2000 rows */
data trlib.mlfloat(MultiLoad=yes);
```

```
      do x=1 to 2000;
         output;
      end;
run;


/* Append another thousand rows */
data work.testdata;
   do x=2001 to 3000;
      output;
   end;
run;

/* Append the SAS data to the Teradata table */
proc append data=work.testdata base=trlib.mlfload
      (MultiLoad=yes);
run;
```

# Teradata Processing Tips for SAS Users

## Reading from and Inserting to the Same Teradata Table

If you use SAS/ACCESS to read rows from a Teradata table and then attempt to insert these rows into the same table, you will hang (suspend) your SAS session.

Behind the scenes, the following happens:

- a SAS/ACCESS connection requests a standard Teradata READ lock for the read operation.
- a SAS/ACCESS connection then requests a standard Teradata WRITE lock for the insert operation.
- the WRITE lock request suspends because the read connection already holds a READ lock on the table. Consequently, your SAS session hangs (is suspended).

In the following example, the following happens:

- SAS/ACCESS creates a read connection to Teradata to fetch the rows selected (select *) from TRA.SAMETABLE, requiring a standard Teradata READ lock; Teradata issues a READ lock.
- SAS/ACCESS creates an insert connection to Teradata to insert the rows into TRA.SAMETABLE, requiring a standard Teradata WRITE lock. But the WRITE lock request suspends because the table is locked already by the READ lock.
- Your SAS/ACCESS session hangs.

```
libname tra teradata user=testuser password=testpass;
proc sql;
insert into tra.sametable
   select * from tra.sametable;
```

To avoid the situation described, use the SAS/ACCESS locking options "Locking in the Teradata Interface" on page 29.

## Using a BY Clause to Order Query Results

SAS/ACCESS returns table results from a query in random order because Teradata returns the rows to SAS/ACCESS randomly. In contrast, traditional SAS processing returns SAS data set observations in the same order during every run of your job. If maintaining row order is important, then you should add a BY clause to your SAS statements. A BY clause ensures consistent ordering of the table results from Teradata.

In the following examples, a Teradata table, ORD, has columns NAME and NUMBER. The PROC PRINT statements illustrate consistent and inconsistent ordering in the display of the ORD table rows.

```
libname prt teradata user=testuser password=testpass;
```

```
proc print data=prt.ORD;
var name number;
run;
```

If this statement is run several times, it yields inconsistent ordering, meaning that the ORD rows are likely to be arranged differently each time. This happens because SAS/ACCESS displays the rows in the order that Teradata returns them; that is, randomly.

```
proc print data=prt.ORD;
var name number;
by name;
run;
```

This statement achieves more consistent ordering because it orders PROC PRINT output by the NAME value. However, on successive runs of the statement, display of rows with a different number and an identical name can vary, as depicted in the PROC PRINT displays below.

**Output 1.1** PROC PRINT Display 1

```
Rita Calvin 2222
Rita Calvin 199
```

**Output 1.2** PROC PRINT Display 2

```
Rita Calvin 199
Rita Calvin 2222
```

```
proc print data=prt.ORD;
var name number;
by name number;
run;
```

The above statement always yields identical ordering because every column is specified in the BY clause. Thus, your PROC PRINT output always looks the same.

## Using TIME and TIMESTAMP

This example creates a Teradata table and assigns the SAS TIME8. format to the TRXTIME0 column. Teradata creates the TRXTIME0 column as the equivalent Teradata data type, TIME(0), with the value of 12:30:55.

```
libname mylib teradata user=testuser password=testpass;

data mylib.trxtimes;
   format trxtime0 time8.;
   trxtime0 = '12:30:55't;
run;
```

The following example creates a Teradata column that specifies very precise time values. The format TIME(5) is specified for the TRXTIME5 column. When SAS reads this column, it assigns the equivalent SAS format TIME14.5.

```
libname mylib teradata user=testuser password=testpass;

proc sql noerrorstop;
   connect to teradata (user=testuser password=testpass);
   execute (create table trxtimes (trxtime5 time(5)
      )) by teradata;
   execute (commit) by teradata;
   execute (insert into trxtimes
            values (cast('12:12:12' as time(5))
      )) by teradata;
   execute (commit) by teradata;
quit;

/* you can print the value that is read SAS/ACCESS */
proc print data =mylib.trxtimes;
run;
```

*Note:* SAS might not preserve more than four digits of fractional precision for Teradata TIMESTAMP. △

The following example creates a Teradata table and specifies a simple timestamp column with no digits of precision. Teradata stores the value 2000–01–01 00:00:00. SAS assigns the default format DATETIME19. to the TRSTAMP0 column generating the corresponding SAS value of 01JAN2000:00:00:00.

```
proc sql noerrorstop;
   connect to teradata (user=testuser password=testpass);
   execute (create table stamps (tstamp0 timestamp(0)
      )) by teradata;
   execute (commit) by teradata;
   execute (insert into stamps
            values (cast('2000--01--01 00:00:00' as
                    timestamp(0))
      )) by teradata;
   execute (commit) by teradata;
quit;
```

The following example creates a Teradata table and assigns the SAS format DATETIME23.3 to the TSTAMP3 column, generating the value

13APR1961:12:30:55.123. Teradata creates the TSTAMP3 column as the equivalent data type TIMESTAMP(3) with the value 1961–04–13 12:30:55.123.

```
libname mylib teradata user=testuser password=testpass;

data mylib.stamps;
format tstamp3 datetime23.3;
tstamp3 = '13apr1961:12:30:55.123'dt;
run;
```

The following example illustrates how the SAS engine passes the literal value for TIMESTAMP in a WHERE statement to Teradata for processing. Note that the value is passed without being rounded or truncated so that Teradata can handle the rounding or truncation during processing. This example would also work in a DATA step.

```
proc sql ;
  select * from trlib.flytime where col1 = '22Aug1995 12:30:00.557'dt ;
  quit;
```

In SAS Version 8, the interface to Teradata did not create TIME and TIMESTAMP data types. Instead, the interface generated FLOAT values for SAS times and dates. The following example shows how to format a column that contains a FLOAT representation of a SAS datetime into a readable SAS datetime.

```
libname mylib teradata user=testuser password=testpass;

proc print data=mylib.stampv80;
format stamp080 datetime25.0;
run;
```

Here, the old Teradata table STAMPV80 contains the FLOAT column, STAMP080, which stores SAS datetime values. The FORMAT statement displays the FLOAT as a SAS datetime value.

## Replacing PROC SORT with a BY Clause

In general, PROC SORT steps are not useful to output a Teradata table. In traditional SAS processing, PROC SORT is used to order observations in a SAS data set. Subsequent SAS steps that use the sorted data set receive and process the observations in the sorted order. Teradata does not store output rows in the sorted order. Therefore, do not sort rows with PROC SORT if the destination sorted file is a Teradata table.

The following example illustrates a PROC SORT statement found in typical SAS processing. This statement cannot be used in SAS/ACCESS for Teradata.

```
libname sortprt '.';
proc sort data=sortprt.salaries;
by income;
proc print data=sortprt.salaries;
```

The following example removes the PROC SORT statement shown in the previous example. Instead, it uses a BY clause, along with a VAR clause, with PROC PRINT. The BY clause returns Teradata rows ordered by the INCOME column.

```
libname sortprt teradata user=testuser password=testpass;
proc print data=sortprt.salaries;
var income;
by income;
```

## Reducing Workload on Teradata by Sampling

The OBS= option triggers SAS/ACCESS to add a SAMPLE clause to generated SQL. In the following example, 10 rows are printed from dbc.ChildrenX:

```
Libname tra teradata user=sasdxs pass=****** database=dbc;
Proc print data=tra.ChildrenX (obs=10);
run;
```

The SQL passed to Teradata is:

```
SELECT "Child","Parent" FROM "ChildrenX"  SAMPLE 10
```

Especially against large Teradata tables, small values for OBS= reduce workload and spool space consumption on Teradata, and your queries complete much sooner. See SAMPLE in your Teradata documentation for further information.

# Locking in the Teradata Interface

The following LIBNAME and data set options enable you to control how the interface to Teradata handles locking. Use SAS/ACCESS locking options only when Teradata's standard locking is undesirable. See the section about the LIBNAME statement in *SAS/ACCESS for Relational Databases: Reference* for additional information about these options. See "Understanding SAS/ACCESS Locking Options" on page 31 and "When to Use SAS/ACCESS Locking Options" on page 31 for tips on using these options. Examples are available.

READ_LOCK_TYPE= TABLE | VIEW

UPDATE_LOCK_TYPE= TABLE | VIEW

READ_MODE_WAIT= YES | NO

UPDATE_MODE_WAIT= YES | NO

READ_ISOLATION_LEVEL= ACCESS | READ | WRITE
The valid values for this option, ACCESS, READ, and WRITE, are defined in the following table.

**Table 1.3** Read Isolation Levels for Teradata

| Isolation Level | Definition |
|---|---|
| ACCESS | Obtains an ACCESS lock by ignoring other users' ACCESS, READ, and WRITE locks. Permits other users to obtain a lock on the table or view.<br>Can return inconsistent or unusual results. |
| READ | Obtains a READ lock if no other user holds a WRITE or EXCLUSIVE lock. Does not prevent other users from reading the object.<br>Specify this isolation level whenever possible, it is usually adequate for most SAS/ACCESS processing. |
| WRITE | Obtains a WRITE lock on the table or view if no other user has a READ, WRITE, or EXCLUSIVE lock on the resource. You cannot explicitly release a WRITE lock. It is released only when the table is closed. Prevents other users from acquiring any lock but ACCESS.<br>This is unnecessarily restrictive, because it locks the entire table until the read operation is finished. |

UPDATE_ISOLATION_LEVEL= ACCESS | READ | WRITE

The valid values for this option, ACCESS, READ, and WRITE, are defined in the following table.

**Table 1.4** Update Isolation Levels for Teradata

| Isolation Level | Definition |
|---|---|
| ACCESS | Obtains an ACCESS lock by ignoring other users' ACCESS, READ, and WRITE locks. Avoids a potential deadlock but can cause data corruption if another user is updating the same data. |
| READ | Obtains a READ lock if no other user holds a WRITE or EXCLUSIVE lock. Prevents other users from being granted a WRITE or EXCLUSIVE lock.<br>Locks the entire table or view, allowing other users to acquire READ locks. Can lead to deadlock situations. |
| WRITE | Obtains a WRITE lock on the table or view if no other user has a READ, WRITE, or EXCLUSIVE lock on the resource. You cannot explicitly release a WRITE lock. It is released only when the table is closed. Prevents other users from acquiring any lock but ACCESS.<br>Prevents all users, except those with ACCESS locks, from accessing the table. Prevents the possibility of a deadlock, but limits concurrent use of the table. |

These locking options cause the LIBNAME engine to transmit a locking request to the DBMS; Teradata performs all the data locking. If you correctly specify a set of SAS/ACCESS read or update locking options, SAS/ACCESS generates locking modifiers that override Teradata's standard locking.

*Note:* If you specify an incomplete set of locking options, SAS/ACCESS returns an error message. If you do not use SAS/ACCESS locking options, Teradata's lock defaults are in effect. For a complete description of Teradata locking, see the LOCKING statement in your Teradata SQL Reference manual. △

## Understanding SAS/ACCESS Locking Options

SAS/ACCESS locking options modify Teradata's standard locking. Teradata usually locks at the row level; SAS/ACCESS lock options lock at the table or view level. The change in the scope of the lock from row to table affects concurrent access to DBMS objects. Specifically, READ and WRITE table locks increase the time that other users must wait to access the table and can decrease overall system performance. The following measures help minimize these negative effects:

- Apply READ or WRITE locks *only* when you must apply special locking on Teradata tables.

  SAS/ACCESS locking options can be appropriate for special situations, as described in "When to Use SAS/ACCESS Locking Options" on page 31. If SAS/ACCESS locking options do not meet your specialized needs, you can use additional Teradata locking features using views. See CREATE VIEW in your Teradata SQL Reference manual for details.
- Limit the span of the locks by using data set locking options instead of LIBNAME locking options whenever possible. (LIBNAME options affect all the tables referenced by your libref that you open, while data set options apply only to the table specified.)

If you specify the following read locking options, SAS/ACCESS generates and submits to Teradata locking modifiers that contain the values that you specify for the three read lock options:

- READ_ISOLATION_LEVEL= specifies the level of isolation from other table users that is required during SAS/ACCESS read operations.
- READ_LOCK_TYPE= specifies and changes the scope of the Teradata lock during SAS/ACCESS read operations.
- READ_MODE_WAIT= specifies during SAS/ACCESS read operations whether Teradata should wait to acquire a lock or fail your request when the DBMS resource is locked by a different user.

If you specify the following update lock options, SAS/ACCESS generates and submits to Teradata locking modifiers that contain the values that you specify for the three update lock options:

- UPDATE_ISOLATION_LEVEL= specifies the level of isolation from other table users that is required as SAS/ACCESS reads Teradata rows in preparation for updating the rows.
- UPDATE_LOCK_TYPE= specifies and changes the scope of the Teradata lock during SAS/ACCESS update operations.
- UPDATE_MODE_WAIT= specifies during SAS/ACCESS update operations whether Teradata should wait to acquire a lock or fail your request when the DBMS resource is locked by a different user.

## When to Use SAS/ACCESS Locking Options

This section describes situations that might require SAS/ACCESS lock options instead of the standard locking provided by Teradata.

- Use SAS/ACCESS locking options to reduce the isolation level for a read operation.

  When you READ lock a table, you can lock out both yourself and other users from updating or inserting into the table. Conversely, when other users update or insert into the table, they can lock you out from reading the table. In this situation, you want to reduce the isolation level during a read operation. To do this, you specify the following read SAS/ACCESS lock options and values:

READ_ISOLATION_LEVEL=ACCESS

READ_LOCK_TYPE=TABLE

READ_MODE_WAIT=YES

The effect of the options and settings in this situation is one of the following:

- Specify ACCESS locking, eliminating a lock out of yourself and other users. Since ACCESS can return inconsistent results to a table reader, specify ACCESS only if you are casually browsing data, not if you require precise data.
- Change the scope of the lock from row-level to the entire table.
- Request that Teradata wait if it attempts to secure your lock and finds the resource already locked.

- Use SAS/ACCESS lock options to avoid contention.

  When you read or update a table, contention can occur: the DBMS is waiting for other users to release their locks on the table that you want to access. This contention suspends your SAS/ACCESS session. In this situation, to avoid contention during a read operation, you specify the following SAS/ACCESS read lock options and values:

  READ_ISOLATION_LEVEL=READ

  READ_LOCK_TYPE=TABLE

  READ_MODE_WAIT=NO

The effect of the options and settings in this situation is one of the following:

- Specify a READ lock.
- Change the scope of the lock. Because SAS/ACCESS does not support row locking when you obtain the lock requested, you lock the entire table until your read operation finishes.
- Tell SAS/ACCESS to fail the job step if Teradata cannot immediately obtain the READ lock.

## Examples

### Setting the Isolation Level to ACCESS for Teradata Tables

```
  /*Generates a quick survey of unusual customer purchases.*/
libname cust teradata user=testuser password=testpass
               READ_ISOLATION_LEVEL=ACCESS
               READ_LOCK_TYPE=TABLE
               READ_MODE_WAIT=YES
               CONNECTION=UNIQUE;
proc print data=cust.purchases(where= (bill<2));
run;
data local;
 set cust.purchases (where= (quantity>1000));
run;
```

In this example, SAS/ACCESS does the following:

- □ Connects to the Teradata DBMS and specifies the three SAS/ACCESS LIBNAME read lock options.
- □ Opens the PURCHASES table and obtains an ACCESS lock if a different user does not hold an EXCLUSIVE lock on the table.
- □ Reads and displays table rows with a value less than 2 in the BILL column.
- □ Closes the PURCHASES table and releases the ACCESS lock.
- □ Opens the PURCHASES table again and obtains an ACCESS lock if a different user does not hold an EXCLUSIVE lock on the table.
- □ Reads table rows with a value greater than 1000 in the QUANTITY column.
- □ Closes the PURCHASES table and releases the ACCESS lock.

## Setting Isolation Level to WRITE to Update a Teradata Table

```
  /*Updates the critical Rebate row.*/
libname cust teradata user=testuser password=testpass;
proc sql;
 update cust.purchases(UPDATE_ISOLATION_LEVEL=WRITE
                       UPDATE_MODE_WAIT=YES
                       UPDATE_LOCK_TYPE=TABLE)
 set rebate=10 where bill>100;
quit;
```

In this example, SAS/ACCESS does the following:

- □ Connects to the Teradata DBMS and specifies the three SAS/ACCESS data set update lock options.
- □ Opens the PURCHASES table and obtains a WRITE lock if a different user does not hold a READ, WRITE or EXCLUSIVE lock on the table.
- □ Updates table rows with BILL greater than 100 and sets the REBATE column to 10.
- □ Closes the PURCHASES table and releases the WRITE lock.

## Preventing a Hung SAS Session When Reading and Inserting to the Same Table

```
  /* The SAS/ACCESS lock options prevent the session hang */
  /* that occurs when reading and inserting into the same table    */
libname tra teradata user=testuser password=testpass
                     connection=unique;
proc sql;
insert into tra.sametable
   select * from tra.sametable(read_isolation_level=access
                               read_mode_wait=yes
                               read_lock_type=table);
```

In this example, SAS/ACCESS does the following:

- □ Creates a read connection to fetch the rows selected (SELECT *) from TRA.SAMETABLE and specifies an ACCESS lock (READ_ISOLATION_LEVEL=ACCESS). Teradata grants the ACCESS lock.
- □ Creates an insert connection to Teradata to process the insert operation to TRA.SAMETABLE. Because the ACCESS lock that is already on the table permits access to the table, Teradata grants a WRITE lock.
- □ Performs the insert operation without hanging (suspending) your SAS session.

# Naming Conventions for Teradata

## Teradata Conventions

The data objects that you can name in Teradata include tables, views, columns, indexes and macros. When naming a Teradata object, use the following conventions:

- A name must start with a letter unless you enclose it in double quotation marks.
- A name must be from 1 to 30 characters long.
- A name can contain the letters A through Z, the digits 0 through 9, the underscore (_), $, and #. A name in double quotation marks can contain any characters except double quotation marks.
- A name, even when enclosed in double quotation marks, is not case-sensitive. For example, CUSTOMER is the same as customer.
- A name cannot be a Teradata reserved word.
- The name must be unique between objects. That is, a view and table in the same database cannot have the identical name.

## SAS Naming Conventions

When naming a SAS object, use the following conventions:

- A name must start with a letter or underscore.
- A name cannot be enclosed in double quotation marks.
- A name must be from 1 to 32 characters long.
- A name can contain the letters A through Z, the digits 0 through 9, and the underscore (_).
- A name is not case-sensitive. For example, CUSTOMER is the same as customer.
- A name does not need to be unique between object types.

## Naming Objects to Meet Teradata and SAS Conventions

To share objects easily between the DBMS and SAS, create names that meet both SAS and Teradata naming conventions. Make the name follow these conventions:

- start with a letter
- include only letters, digits, and underscores
- have a length of 1 to 30 characters.

## Accessing Teradata Objects That Do Not Meet SAS Naming Conventions

The following are SAS/ACCESS code examples to help you access Teradata objects (existing Teradata DBMS tables and columns) that have names that do not follow SAS naming conventions.

### Example 1: Unusual Teradata Table Name

```
libname unusual teradata user=testuser password=testpass;
proc sql dquote=ansi;
```

```
  create view myview as
  select * from unusual."More names";
proc print data=myview;run;
```

### Example 2: Unusual Teradata Column Names

SAS/ACCESS automatically converts Teradata column names that are invalid for SAS, mapping any invalid characters to underscores. It also appends numeric suffixes to identical names to ensure that column names are unique.

```
create table unusual_names( Name$ char(20), Name# char(20),
                             "Other strange name" char(20))
```

In this example, SAS/ACCESS converts the spaces found in the Teradata column name, OTHER STRANGE NAME, to Other_strange_name. After the automatic conversion, SAS programs can then reference the table as usual, for example:

```
libname unusual teradata user=testuser password=testpass;
proc print data=unusual.unusual_names; run;
```

**Output 1.3** PROC PRINT Display

```
         Name_          Name_0         Other_strange_name
```

# Data Types for Teradata

Every column in a table has a name and data type. The data type tells Teradata how much physical storage to set aside for the column, as well as the form in which to store the data.

*Note:* SAS/ACCESS 9 does not support the following Teradata data types: GRAPHIC, VARGRAPHIC, and LONG VARGRAPHIC. △

## Binary String Data

BYTE (*n*)
: specifies a fixed-length column of length *n* for binary string data. The maximum for *n* is 64,000.

VARBYTE (*n*)
: specifies a varying-length column of length *n* for binary string data. The maximum for *n* is 64,000.

## Character String Data

CHAR (*n*)
: specifies a fixed-length column of length *n* for character string data. The maximum for *n* is 64,000.

VARCHAR (*n*)
: specifies a varying-length column of length *n* for character string data. The maximum for *n* is 64,000. VARCHAR is also known as CHARACTER VARYING.

LONG VARCHAR
: specifies a varying-length column, of the maximum length, for character string data. LONG VARCHAR is equivalent to VARCHAR(32000) or VARCHAR(64000) depending on which Teradata version your server is running.

## Date/Time Data

The date type in Teradata is similar to the SAS date value. It is stored internally as a numeric value and is displayed in a site-defined format. Date type columns might contain Teradata values that are out of range for SAS, which handles dates from A.D. 1582 through A.D. 20,000. If SAS/ACCESS encounters an unsupported date, for example, a date earlier than A.D. 1582, it returns an error message and display the date as a missing value.

See "Using TIME and TIMESTAMP" on page 27 for examples.

The Teradata date/time types that SAS supports are listed here.

DATE
: specifies date values in the default format YYYY-MM-DD. For example, January 25, 1989, is input as 1989-01-25. Values for this type can range from 0001-01-01 through 9999-12-31.

TIME (*n*)
: specifies time values in the format HH:MM:SS.SS. In the time, SS.SS is the number of seconds ranging from 00 to 59 with the fraction of a second following the decimal point.

  *n* is a number from 0 to 6 that represents the number of digits (precision) of the fractional second. For example, TIME(5) is 11:37:58.12345 and TIME(0) is 11:37:58. This type is supported for Teradata Version 2, Release 3 and later.

TIMESTAMP (*n*)
: specifies date/time values in the format YYYY-MM-DD HH:MM:SS.SS. In the timestamp, SS.SS is the number of seconds ranging from 00 through 59 with the fraction of a section following the decimal point.

  *n* is a number from 0 to 6 that represents the number of digits (precision) of the fractional second. For example, TIMESTAMP(5) is 1999-01-01 23:59:59.99999 and TIMESTAMP(0) is 1999-01-01 23:59:59. This type is supported for Teradata Version 2, Release 3 and later.

***CAUTION:***

When processing WHERE statements (using PROC SQL or the DATA step) that contain *literal* values for TIME or TIMESTAMP, the SAS engine passes the values to Teradata exactly as they were entered, without being rounded or truncated. This is done so that Teradata can handle the rounding or truncation during processing. △

## Numeric Data

When reading Teradata data, SAS/ACCESS converts all Teradata numeric data types to the SAS internal format, floating-point.

BYTEINT
: specifies a single-byte signed binary integer. Values can range from -128 to +127.

DECIMAL(*n*,*m*)
: specifies a packed-decimal number. *n* is the total number of digits (precision). *m* is the number of digits to the right of the decimal point (scale). The range for precision is 1 through 18. The range for scale is 0 through *n*.

If *m* is omitted, 0 is assigned and *n* can also be omitted. Omitting both *n* and *m* results in the default DECIMAL(5,0). DECIMAL is also known as NUMERIC.

***CAUTION:***

Because SAS stores numbers in floating-point format, a Teradata DECIMAL number with very high precision can lose precision. For example, when SAS/ACCESS running on a UNIX MP-RAS client reads a Teradata column specified as DECIMAL (18,18), it maintains only 13 digits of precision. This can cause problems. A large DECIMAL number can cause the WHERE clause that SAS/ACCESS generates to perform improperly (fail to select the expected rows). There are other potential problems. For this reason, *use carefully* large precision DECIMAL data types for Teradata columns that SAS/ACCESS will access. △

FLOAT
: specifies a 64-bit Institute of Electrical and Electronics Engineers (IEEE) floating-point number in sign-and-magnitude form. Values can range from approximately 2.226 x $10^{-308}$ to 1.797 x $10^{308}$. FLOAT is also known as REAL or DOUBLE PRECISION.

*Note:* When the SAS/ACCESS client internal floating point format is IEEE, Teradata FLOAT numbers convert precisely to SAS numbers. Exact conversion applies to SAS/ACCESS for Teradata running under UNIX MP-RAS. However, if you are running SAS/ACCESS for Teradata under z/OS, there can be minor precision and magnitude discrepancies. △

INTEGER
: specifies a large integer. Values can range from –2,147,483,648 through +2,147,483,647.

SMALLINT
: specifies a small integer. Values can range from –32,768 through +32,767.

## Teradata Null Values

Teradata has a special value that is called NULL. A Teradata NULL value means an absence of information and is analogous to a SAS missing value. When SAS/ACCESS reads a Teradata NULL value, it interprets it as a SAS missing value.

By default, Teradata columns accept NULL values. However, you can define columns so that they do not contain NULL values. For example, when you create a SALES table, define the CUSTOMER column as NOT NULL, telling Teradata not to add a row to the table unless the CUSTOMER column for the row has a value. When creating a Teradata table with SAS/ACCESS, you can use the DBNULL= data set option to indicate whether NULL is a valid value for specified columns.

For more information about how SAS handles null values, see "Potential Result Set Differences When Processing Null Data" in *SAS/ACCESS for Relational Databases: Reference*.

*Note:* To control how SAS missing character values are handled by Teradata, use the NULLCHAR= and NULLCHARVAL= data set options. △

## LIBNAME Statement Data Conversions

When you *read* a Teradata table with the LIBNAME statement, SAS/ACCESS assigns default SAS data types and formats to the Teradata table columns. In assigning the defaults, SAS/ACCESS does not use Teradata's column format information. The following table shows the default SAS formats that SAS/ACCESS assigns to Teradata data types when you use the LIBNAME statement.

**Table 1.5** Default SAS Formats for Teradata

| Teradata Data Type | Default SAS Format |
|---|---|
| CHAR(*n* ) | $*n* (*n*<= 32,767) |
| CHAR(*n* ) | $32767.(*n*>32,767) [1] |
| VARCHAR(*n* ) | $*n* (*n*<= 32,767) |
| VARCHAR(*n* ) | $32767.(*n*> 32,767) [1] |
| LONG VARCHAR(*n* ) | $32767. [1] |
| BYTE(*n* ) | $HEX*n*. (*n*<= 32,767) |
| BYTE(*n* )[1] | $HEX32767.(*n*> 32,767) |
| VARBYTE(*n* ) | $HEX*n*. (*n*<= 32,767) |
| VARBYTE(*n* ) | $HEX32767.(*n*> 32,767) |
| INTEGER | 11.0 |
| SMALLINT | 6.0 |
| BYTEINT | 4.0 |
| DECIMAL(*n*, *m* )[2] | (*n*+2 ).(*m* ) |
| FLOAT | none |
| DATE[3] | DATE9. |
| TIME(*n*)[4] | for *n*=0, TIME8.<br>for *n*>0, TIME9+*n*.*n* |
| TIMESTAMP(*n*)[4] | for *n*=0, DATETIME19.<br>for *n*>0, DATETIME20+*n*.*n* |

1 When reading Teradata data into SAS, DBMS columns that exceed 32,767 bytes are truncated. The maximum size for a SAS character column is 32,767 bytes.
2 If the DECIMAL number is extremely large, SAS can lose precision. For details, see the topic “Numeric Data”.
3 See the topic “Date/Time Data” for how SAS/ACCESS handles dates that are outside the valid SAS date range.
4 TIME and TIMESTAMP are supported for Teradata Version 2, Release 3 and later. The TIME with TIMEZONE, TIMESTAMP with TIMEZONE, and INTERVAL types are presented as SAS character strings, and thus are harder to use.

When you *create* Teradata tables, the default Teradata columns that SAS/ACCESS creates are based on the type and format of the SAS column. The following table shows the default Teradata data types that SAS/ACCESS assigns to the SAS formats during output processing when you use the LIBNAME statement.

**Table 1.6** Default Output Teradata Data Types

| SAS Data Type | SAS Format | Teradata Data Type |
|---|---|---|
| Character | $*w*.<br>$CHAR*w*.<br>$VARYING*w*. | CHAR[*w*] |
| Character | $HEX*w*. | BYTE[*w*] |

| SAS Data Type | SAS Format | Teradata Data Type |
|---|---|---|
| Numeric | A date format | DATE |
| Numeric | TIME*w.d* | TIME(*d*)[1] |
| Numeric | DATETIME*w.d* | TIMESTAMP(*d*)[1] |
| Numeric | *w.*($w \leq 2$) | BYTEINT |
| Numeric | *w.*($3 \leq w \leq 4$) | SMALLINT |
| Numeric | *w.*($5 \leq w \leq 9$) | INTEGER |
| Numeric | *w.*($w \geq 10$) | FLOAT |
| Numeric | *w.d* | DECIMAL(*w*-1,*d*) |
| Numeric | All other numeric formats | FLOAT |

1 For Teradata Version 2, Release 2 and earlier, FLOAT is the default Teradata output type for SAS time and datetime values. To display Teradata columns that contain SAS times and datetimes properly, you must explicitly assign the appropriate SAS time or datetime display format to the column.

To override any default output type, use the data set option DBTYPE=.

## Data Returned as SAS Binary Data with Default Format $HEX

BYTE
VARBYTE
LONGVARBYTE
GRAPHIC
VARGRAPHIC
LONG VARGRAPHIC