Preface

Purpose

This book provides information about Teradata Parallel Transporter Application Programming Interface, a Teradata® Tools and Utilities product. Teradata Tools and Utilities is a group of products designed to work with Teradata Database.

Teradata Parallel Transporter Application Programming Interface is a set of application programming interfaces used to load into, and export data from, the Teradata Database. This book describes how to set up the interface, error reporting, checkpoint and restart, and other relevant topics and includes coding examples.

Audience

This book is intended for use by:

- Database administrators
- Relational database developers
- Field engineers
- Test engineers

Supported Releases

This book supports the following releases:

- Teradata Database V2R6.x, V12.0, V13.0, V13.10
- Teradata Tools and Utilities 13.10
- Teradata Parallel Transporter Application Programming Interface 13.10
- Teradata PT ICU Library

**Note:** To verify the Teradata Parallel Transporter Application Programming Interface driver version number review these events:

For the Load driver, see “TD_Evt_Version” on page 72 and “TD_TRACE_LEVEL” on page 69.

For the Update driver, see “TD_Evt_Version” on page 90 and “TD_TRACE_LEVEL” on page 87.

For the Stream driver, see “TD_Evt_Version” on page 117 and “TD_TRACE_LEVEL” on page 113.
For the Export driver, see “TDEvt_Version” on page 136 and “TD_TRACE_LEVEL” on page 135.

The Teradata Tools and Utilities 13.10 version of the following products are required for Teradata Parallel Transporter Application Programming Interface:

- Teradata International Components for Unicode (Teradata ICU)
- Teradata Generic Security Services (TeraGSS)
- Teradata Call Level Interface, Version 2 (Teradata CLI V2)
- Teradata Parallel Transporter Operator Support Library
- Teradata Parallel Transporter Export Operator (if used)
- Teradata Parallel Transporter Load Operator (if used)
- Teradata Parallel Transporter Update Operator (if used)
- Teradata Parallel Transporter Stream Operator (if used)

See Appendix A: “Platform Compilers” for the supported versions and releases of the platform compilers.

To locate detailed supported-release information:

3. Type 3119 in the Publication Product ID box.
4. Under Sort By, select Date.
5. Click Search.
6. Open the version of the Teradata Tools and Utilities ##.##.## Supported Platforms and Product Versions spreadsheet associated with this release.

The spreadsheet includes supported Teradata Database versions, platforms, and product release numbers.

Prerequisites

The following prerequisite knowledge is required for this product:

- Computer technology and terminology
- Teradata Database
- Relational database management systems
- Connectivity software, such as ODBC
- Microsoft® Windows® 2000, Windows XP®, Solaris SPARC®, Solaris Opteron®, IBM AIX®, LINUX®, HP-UX®, and IBM z/OS operating systems
- Utilities that load and export data
Changes to This Book

The following changes were made to this book in support of the current release. Changes are marked with change bars. For a complete list of changes to the product, see the *Teradata Tools and Utilities Release Definition* associated with this release.

<table>
<thead>
<tr>
<th>Date and Release</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>February 2010</td>
<td>• Support for Red Hat Linux Advanced Server 2.1 platform compilers is discontinued (was listed in appendix A)</td>
</tr>
<tr>
<td>13.10</td>
<td>• Support for library built with gcc 2.96 (32-bit TPT API on Linux) is discontinued.</td>
</tr>
<tr>
<td></td>
<td>• z/Linux support has been added.</td>
</tr>
<tr>
<td></td>
<td>• Appendix A has been updated with additional Windows 32-bit compiler versions.</td>
</tr>
<tr>
<td></td>
<td>• Appendix A has been updated with information on z/Linux compiler</td>
</tr>
<tr>
<td></td>
<td>• The Stream driver now supports the following optional attributes: TD_DROPMACRO specifies whether to keep or drop the macro created during the current Stream job.</td>
</tr>
<tr>
<td></td>
<td>• TD_PACKMAXIMUM triggers the Stream driver to dynamically determine the maximum possible pack factor for the current Stream job.</td>
</tr>
<tr>
<td></td>
<td>• A new GetEvent query has been added to the Stream driver. TD_Evt_PackFactor is a pointer to the value of the pack factor as an integer.</td>
</tr>
<tr>
<td></td>
<td>• The Export driver now supports the following optional attributes: TD_OUTLIMIT limits the number of rows that the Export driver exports.</td>
</tr>
<tr>
<td></td>
<td>• TD_SPOOLMODE specifies whether to use spool or not while running the current export job.</td>
</tr>
<tr>
<td></td>
<td>• For Load and Update drivers, the TD_Evt_ApplyCount GetEvent query for multi-instance jobs only queries the number in the master job instance. The master instance returns the total number of rows applied to the target table from all instances.</td>
</tr>
<tr>
<td>Date and Release</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| August 2008 13.00 | • IBM z/OS support has been added. Note the following:  
|                  |   • On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). In Chapter 2: “Interface Specifications” syntax examples, it has been noted to use #include “connecti.h” instead of “connection.h.”  
|                  |   • For the TD_CHARSET attribute on channel-attacted z/OS platforms, only EBCDIC encoding is supported and is automatically selected.  
|                  |   • Appendix A was updated to include compiler versions.  
|                  |   • In Table 34 on page 175, note that Teradata PT ICU Library and Teradata PT Operator Support software packages have been removed from the table; they are included in Teradata PT API.  
|                  |   • Appendix A was updated for Linux 32-bit compiler versions.  
|                  |   • MP-RAS in no longer supported. Information has been removed.  
|                  |   • Chapter 9: “Converting TIME, TIMESTAMP, and INTERVAL Data Types” has been added to address working with the TIME and DATE data types.  
|                  |   • If the restart log table name is not fully qualified in the TD_LOG_TABLE attribute, a new restart log table will be created. Additionally, if the TD_WORKINGDATABASE attribute is specified, the restart table name must be fully qualified. See Table 5 on page 54.  
|                  |   • For the Load driver, see “TD_LOG_TABLE” on page 62.  
|                  |   • For the Update driver, see “TD_LOG_TABLE” on page 78 and “TD_WORKINGDATABASE” on page 88  
|                  |   • For the Stream driver, see “TD_LOG_TABLE” on page 102 and “TD_WORKINGDATABASE” on page 114.  
|                  |   • The Stream driver now has the ability to use an existing error table across jobs. See the optional attributes “TD_APPENDERRORTABLE” on page 103 and “TD_DROPERRORTABLE” on page 105.  
|                  |   • The section “Stream Driver Macro Support” on page 121 was added to outline how to use pre-defined macros.  
|                  |   • A new DML option provides Stream applications with a way to protect against an update or delete affecting multiple rows, which can happen because the primary index can be non-unique. See Table 5 on page 54. |
Additional Information

Additional information that supports this product and Teradata Tools and Utilities is available at the web sites listed in the table that follows.

In the table, mmyx represents the publication date of a manual, where mm is the month, y is the last digit of the year, and x is an internal publication code.

Match the mmy of a related publication to the date on the cover of this book. This ensures that the publication selected supports the same release.

<table>
<thead>
<tr>
<th>Date and Release</th>
<th>Description</th>
</tr>
</thead>
</table>
| August 2008 13.00 (continued) | • Single-instance Teradata PT API applications do not have to store any checkpoint data when using the checkpoint and restart functionality. A new event has been defined to return the total number of rows that have been received and checkpointed in a single-instance job.  
  • For the Load driver, see “TD_EVT_RowsCheckpointed” on page 71.  
  • For the Update driver, see “TD_EVT_RowsCheckpointed” on page 90  
  • For the Stream driver, see “TD_EVT_RowsCheckpointed” on page 116.  
  • Teradata PT API can dynamically create the schema when using the Export driver and can share the schema with other instances of Teradata PT API using different drivers.  
  • New data types have been added in support of the Teradata Database PERIOD data type. See Table 6 on page 56. Table 3 on page 51 has information on the schema Add Column object.  
  • The Load driver now support the following optional attributes:  
    • TD_DROPERRORTABLE specifies whether or not error tables are dropped upon successful completion of the load job, even if the error tables are empty.  
    • TD_DROPLOGTABLE specifies whether or not restart log tables are dropped upon successful complete of the load job.  
  • The Update driver now supports the following optional attributes:  
    • TD_DROPERRORTABLE specifies whether or not error tables are dropped upon successful completion of the update job, even if the error tables are empty.  
    • TD_DROPLOGTABLE specifies whether or not restart log tables are dropped upon successful complete of the update job.  
    • TD_DROPWORKTABLE specifies whether or not the work tables are dropped upon successful completion of the update job. |
<table>
<thead>
<tr>
<th>Type of Information</th>
<th>Description</th>
<th>Access to Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release overview</td>
<td>Use the Release Definition for the following information:</td>
<td>1  Go to <a href="http://www.info.teradata.com/">http://www.info.teradata.com/</a>.</td>
</tr>
<tr>
<td>Late information</td>
<td>• Overview of all of the products in the release</td>
<td>2  Under <strong>Online Publications</strong>, click <strong>General Search</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Information received too late to be included in the manuals</td>
<td>3  Type 2029 in the <strong>Publication Product ID</strong> box.</td>
</tr>
<tr>
<td></td>
<td>• Operating systems and Teradata Database versions that are certified to work with each product</td>
<td>4  Click <strong>Search</strong>.</td>
</tr>
<tr>
<td></td>
<td>• Version numbers of each product and the documentation for each product</td>
<td>5  Select the appropriate Release Definition from the search results.</td>
</tr>
<tr>
<td></td>
<td>• Information about available training and the support center</td>
<td></td>
</tr>
</tbody>
</table>
### Type of Information

<table>
<thead>
<tr>
<th>Description</th>
<th>Access to Information</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2. Under the Online Publications subcategory, Browse by Category, click Data Warehousing.</td>
</tr>
<tr>
<td></td>
<td>3. Do one of the following:</td>
</tr>
<tr>
<td></td>
<td>• For a list of Teradata Tools and Utilities documents, click Teradata Tools and Utilities and then select an item under Releases or Products.</td>
</tr>
<tr>
<td></td>
<td>• Select a link to any of the data warehousing publications categories listed.</td>
</tr>
<tr>
<td></td>
<td>Specific books related to Teradata PT API are as follows:</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">International Character Set Support</a> B035-1125-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">SQL Data Types and Literals</a> B035-1143-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">SQL Stored Procedures and Embedded SQL</a> B035-1148-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">SQL External Routine Programming</a> B035-1147-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">Teradata FastExport Reference</a> B035-2410-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">Teradata FastLoad Reference</a> B035-2411-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">Messages</a> B035-1096-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">Teradata MultiLoad Reference</a> B035-2409-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">Teradata Parallel Data Pump Reference</a> B035-3021-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">Teradata Parallel Transporter Reference</a> B035-2435-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">Teradata Tools and Utilities Installation Guide for Microsoft Windows</a> B035-2407-mmyx</td>
</tr>
<tr>
<td></td>
<td>• <a href="http://www.info.teradata.com/">Teradata Tools and Utilities Installation Guide for UNIX and Linux</a> B035-2459-mmyx</td>
</tr>
<tr>
<td></td>
<td>• Utilities, Volume 1: A - F B035-1102-mmyx</td>
</tr>
<tr>
<td></td>
<td>2. Under the Online Publications subcategory, Browse by Category, click Data Warehousing.</td>
</tr>
<tr>
<td></td>
<td>3. Click CD-ROM List and Images.</td>
</tr>
<tr>
<td></td>
<td>4. Follow the ordering instructions.</td>
</tr>
</tbody>
</table>

Use the Teradata Information Products web site to view or download specific manuals that supply related or additional information to this manual.
## Type of Information

<table>
<thead>
<tr>
<th>Type of Information</th>
<th>Description</th>
<th>Access to Information</th>
</tr>
</thead>
</table>
| General information about Teradata    | The Teradata home page provides links to numerous sources of information about Teradata. Links include:  
  - Executive reports, case studies of customer experiences with Teradata, and thought leadership  
  - Technical information, solutions, and expert advice  
# Table of Contents

## Preface

- Purpose .......................................................................................................................... 3
- Audience ......................................................................................................................... 3
- Supported Releases ........................................................................................................ 3
- Prerequisites .................................................................................................................... 4
- Changes to This Book .................................................................................................... 5
- Additional Information ................................................................................................... 7

## Chapter 1: Coding a Teradata PT API Application ......................................................... 21

- Introduction to Teradata PT API .................................................................................... 21
- Coding Steps ................................................................................................................... 23
  - Build a Teradata PT API Database Connection Object .............................................. 23
  - Initiate a Teradata PT API Database Connection ....................................................... 27
  - Load Data into or Export Data from the Teradata Database ....................................... 27
  - Interpret Errors ........................................................................................................... 33
  - Add Checkpoint and Restart ....................................................................................... 33
  - Query Run-Time Statistics ........................................................................................ 34
  - Modify Job after Initiate ............................................................................................. 35
  - Utilize Serialization with the Stream Driver ............................................................... 35
  - Utilize Use Lists with DML Groups .......................................................................... 36
  - Terminate the Teradata PT API Connection .............................................................. 36
- Special Topics .................................................................................................................. 37
  - Reuse a Teradata PT API Connection Object ............................................................ 37
  - Query Banding Teradata Database Feature ............................................................... 37
  - Using the UTF16 Session Character Set .................................................................... 37
  - Using the Export Driver’s Dynamic Schema Feature ................................................. 39
  - Available Teradata Parallel Transporter Functions .................................................. 40
Chapter 2: Interface Specifications

Connection Class ........................................ 41
Schema Class ........................................... 51
DML Groups ............................................. 52
Teradata PT API Constants ......................... 54
  DML Option ........................................ 54
  TD_Attribute .................................... 56
  TD_DataType ................................... 56
  TD_Encoding .................................... 57
  TD_OperatorType ................................. 58
  TD_TRACE_LEVEL ................................ 58
Teradata PT API Status Messages ............. 59

Chapter 3: Load Driver ................................. 61

Attribute Definitions .................................. 61
  Required Attributes ................................ 61
  Optional Attributes ................................ 62
GetEvent Queries ..................................... 70
Programming Considerations ................... 72
  Error Tables and Error Reporting .............. 72
  Limiting Insertion Errors ......................... 72
  Dropping Tables During a Load .................. 73
  Required Privileges .............................. 73
  Session Limits ................................... 73
  Space Requirements and Limitations ............ 74
  Checkpoint and Restart Operations ............... 74
Code Example .......................................... 74

Chapter 4: Update Driver ............................. 77

Attribute Definitions ............................... 77
  Required Attributes .............................. 77
  Optional Attributes ................................ 78
GetEvent Queries ..................................... 88
Table of Contents

Chapter 5:
Stream Driver ................................................. 101

Attribute Definitions........................................ 101
  Required Attributes....................................... 102
  Optional Attributes..................................... 103
GetEvent Queries ............................................ 114
Programming Considerations............................... 117
  Error Tables............................................. 117
  Reusing Stream Driver Table Names.................... 117
  Limiting Errors in the Stream Driver.................. 117
  Dropping Tables During a Load......................... 118
  Required Privileges................................... 118
  Session Limits........................................ 118
  Obtaining the Row Count Using TD_EVT_ApplyCount..... 119
  Space Requirements and Limitations.................... 120
  Tuning the Pack Factor................................ 120
  Using DELETE in Import Tasks........................... 120
  Stream Driver Macro Support............................ 121
  Checkpoint and Restart Operations...................... 122
Code Example.................................................. 122

Chapter 6:
Export Driver .................................................. 127

Attribute Definitions........................................ 127
Table of Contents

Chapter 9: Converting TIME, TIMESTAMP, and INTERVAL Data Types. 165

ANSI/SQL DateTime Specifications ............................................. 165

Appendix A: Platform Compilers .................................................. 171

Appendix B: Code Samples ......................................................... 175

Windows ................................................................................. 176
Solaris SPARC, Solaris Opteron, HP-UX and AIX ......................... 177
Linux ...................................................................................... 178
IBM z/OS .................................................................................. 179

Appendix C: Compiling and Linking Options ............................. 181

Glossary .................................................................................... 183

Index ....................................................................................... 189
List of Figures

Figure 1: Typical Utility Integration Architecture for Customer Applications .............. 22
Figure 2: Teradata PT API Integration Architecture for Customer Applications .............. 22
Figure 3: Schema Definition for a Multiple Table Select Statement ......................... 25
Figure 4: Data Buffer Format ................................................................. 29
Figure 5: Bytes Illustrating Indicator-Mode Input Data Format ................................. 32
Figure 6: Layout of Data Buffer returned by the TD_Evt_ApplyCount Method ............ 120
Figure 7: Master and Slave Parallel Relationships during Load and Export ............... 143
List of Tables

Table 1: Function Call Guide for the Teradata PT API Drivers ........................................... 40
Table 2: Connection Class Objects ................................................................. 42
Table 3: Schema Class Objects ................................................................. 51
Table 4: DMLGroup Class Objects .............................................................. 52
Table 5: DML Group Options ................................................................. 54
Table 6: Schema Object AddColumn Constants ........................................... 56
Table 7: TD_Encoding Constants .............................................................. 57
Table 8: TD_SYSTEM-OPERATOR Attribute Constants ........................................ 58
Table 9: TD_TRACE_LEVEL Attribute Constants ........................................... 58
Table 10: Teradata PT API Status Messages .................................................. 59
Table 11: Load Driver Required Attributes .................................................. 61
Table 12: Load Driver Optional Attributes ................................................... 62
Table 13: Load Driver Events ................................................................. 70
Table 14: TD_ERRORLIMIT Values .............................................................. 72
Table 15: Update Driver Required Attributes .............................................. 77
Table 16: Update Driver Optional Attributes ............................................... 78
Table 17: Update Driver Events ............................................................... 88
Table 18: Update Driver Error Tables ........................................................... 90
Table 19: Update Driver ERRORLIMIT Values ............................................. 91
Table 20: Offline AMP Conditions and Effects on Update Driver Tasks ............... 94
Table 21: Nonparticipant AMP Conditions and Effects on Update Driver Tasks .... 95
Table 22: Stream Driver Required Attributes .............................................. 102
Table 23: Stream Driver Optional Attributes ............................................... 103
Table 24: Stream Driver Events ............................................................... 115
Table 25: Stream Driver ERRORLIMIT Values ............................................. 117
Table 26: Export Driver Required Attributes ............................................... 127
Table 27: Export Driver Optional Attributes ............................................... 129
Table 28: Export Driver Events ............................................................... 136
Table 29: PutEvent Modifier Expected Data .................................................. 137
Table 30: Export Driver SELECT Requests ................................................... 138
Table 31: Synchronization Return Code Combinations and Suggested Actions .... 145
Table 32: ANSI/SQL DateTime Specifications ................................................. 165
List of Tables

Table 33: Teradata PT API Platform Compilers ........................................171
Table 34: Required Software Dependencies for Teradata PT API ..................175
CHAPTER 1
Coding a Teradata PT API Application

The Teradata Parallel Transporter Application Programming Interface (Teradata PT API) is a set of application programming interfaces used to load into, and export data from, the Teradata Database.

Teradata PT API enables an application to access the Teradata Database using proprietary Teradata load and export protocols (Teradata FastLoad Reference, Teradata FastExport Reference, and Teradata MultiLoad Reference).

Unlike the Teradata utilities and Teradata Parallel Transporter that are driven by scripts, Teradata PT API is a functional library that is part of your applications. This allows the applications to have more control during the load and export processes.

The advantages for using Teradata PT API are:

- Open standard reduces research and development costs to integrate with the Teradata Database
- Robust API facilitates easier access to key Teradata Database functions
- Better control of the runtime environment simplifies the management processes

Introduction to Teradata PT API

Figure 1 and Figure 2 illustrate the difference between the environments from an application enterprise overview. In the existing script environment, the partner application and a Teradata utility must run as two separate jobs (Figure 1).

In the API environment, the transform, load, export, and other functions can be integrated into one application and run as one job (Figure 2).
Chapter 1: Coding a Teradata PT API Application
Introduction to Teradata PT API

Figure 1: Typical Utility Integration Architecture for Customer Applications

Figure 2: Teradata PT API Integration Architecture for Customer Applications
Coding Steps

To connect to the Teradata Database and perform related tasks, review the information in each of these steps:

1. Build a Teradata PT API Database Connection Object
2. Initiate a Teradata PT API Database Connection
3. Load Data into or Export Data from the Teradata Database
4. Interpret Errors
5. Add Checkpoint and Restart
6. Query Run-Time Statistics
7. Modify Job after Initiate
8. Utilize Serialization with the Stream Driver
9. Utilize Use Lists with DML Groups
10. Terminate the Teradata PT API Connection

Also review these special topics:

- Reuse a Teradata PT API Connection Object
- Query Banding Teradata Database Feature
- Using the UTF16 Session Character Set
- Using the Export Driver’s Dynamic Schema Feature
- Available Teradata Parallel Transporter Functions

Note: Two other special topics are discussed in Chapter 7: “Parallelism Enabling Protocol,” and Chapter 8: “Using Teradata PT API in an External Stored Procedure.”

Build a Teradata PT API Database Connection Object

In order to establish a connection with the Teradata Database, users must create an object that will manage all of that connection's important attributes. In Teradata PT API, this object is called a Connection object.

There are two steps involved in building a Connection object:

1. Create a Connection Object
   
   The Connection object is defined as a C++ class. A Connection object can be created by calling the Connection class constructor.
   
   ```c++
   using namespace teradata::client::API;
   Connection* conn = new Connection();
   ```

2. Add Parameters
   
   The Connection object is used to house parameters for a database connection. There are three types of parameters:
   
   a. attributes
There are various types of options that can be set using attributes. Each driver has a set of required attributes and optional attributes. Depending on the attribute, there can be multiple values for that attribute. These are called *array attributes*. The following is an example of adding attributes and array attributes:

```c
conn->AddAttribute(TD_MAX_SESSIONS, 4);
conn->AddArrayAttribute(TD_WORK_TABLE, 2, "testa_wt", "testb_wt", NULL);
```

Set the value(s) for an attribute in a single call to either the `AddAttribute` function or the `AddArrayAttribute` function. Once an attribute has been set, the value(s) of the attribute cannot be changed and additional values cannot be added to it.

Set all attributes prior to calling the `Initiate` function. After a `Connection` object has been initiated, no additional attributes may be set.

### Schemas

A schema object defines the format of the data being used in the load and export tasks. A schema consists of one or more columns, and the format of each column’s data. The schema lets the driver know what data to expect.

Unlike an attribute, a schema is defined as an external C++ class. When creating a schema object using the class constructor, the schema must be designated as an input or output schema. This means the task will involve either loading, updating data, or exporting data. The following is an example of creating a schema object:

```c
Schema * schema = new Schema("input");
```

Columns are added to the schema object one at a time using the `TD_DataType` constants to designate the data type of the column. The following are examples of adding a column to a schema object:

```c
schema->AddColumn("Associate_Name", TD_CHAR, 25);
schema->AddColumn("Salary", TD_FLOAT, 8);
```

*Figure 3* depicts the coding lines for a two-table select statement and how the data is brought together in the answer set.
When all columns have been added to the schema object, the schema object is then passed to the Connection class. The following is an example of adding a schema object to a Connection object:

```cpp
conn->AddSchema(schema);
```

**Working with Schemas**

When working with schemas, note the following:

- Every driver requires a schema except when using either the Update driver’s Delete Task with no variable substitution or the Export driver’s dynamic schema feature.
- A Connection object may have only one schema.
- The column names defined in the schema do not have to match the actual names of the columns in the target table(s) as they can be renamed in the SQL statements used for the job. The column names in the schema must match the names used in the job’s SQL statements.
- The column data types defined in the schema do not have to match the actual data types of the columns in the target table(s) as they can be cast into other data types in the SQL statements used. The column data type in the schema must however match the data types used in the job’s SQL statements.
• The order of the columns defined in the schema does not have to match the actual order of the columns in the target table(s) as the order can be changed in the SQL statements used. The order of the columns in the schema must however match the order of the columns in the job’s SQL statements.

• When loading multiple tables using the Update or Stream driver, the input schema defined for the job is a superset of the columns in all of the tables being loaded. One or more of the columns in the schema must correlate to the columns being loaded in each of the target tables.

**Examples of Schema Definitions**

If the Export driver is being used to export data from Table 1 which has columns A, B, and C, and the SELECT statement is:

```
SELECT A FROM Table 1
```

then the output schema is defined as one column with a data type matching column A’s definition in Table 1. If column A is defined as a variable character with a length of 50, then the column in the output schema is defined as TD_VARCHAR with a length of 50.

```cpp
Schema* schema = new Schema("output");
schema->AddColumn("A", TD_VARCHAR, 50);
```

Here’s another example using the same table. If the select statement is changed to:

```
SELECT A(CHAR(100)) FROM Table 1
```

then the column in the output schema is defined as TD_CHAR with a length of 100.

```cpp
Schema* schema = new Schema("output");
schema->AddColumn("A", TD_CHAR, 100);
```

**DML Groups**

DMLGroup objects are used to store sets of DML statements and DML options which can later be applied when loading or updating data. A DMLGroup is defined as an external C++ class.

The following is an example of creating a DMLGroup object:

```cpp
DMLGroup* dmlGr = new DMLGroup();
```

Each DML statement is stored as a character string within the DMLGroup object. DML statements can be added one at a time to the DMLGroup object using the AddStatement function. More than one DML statements can be added to the same DMLGroup object but each DML statement must be added in a separate call to the AddStatement function. The following is an example of adding a statement to a DMLGroup object:

```cpp
dmlGr->AddStatement("INSERT INTO test1( :Associate_Id, :Salary );");
```

Along with DML statements, a series of DML options can be set for a DMLGroup object using the TD_DMLOption parameters. The following is an example of setting DML options for a DMLGroup object:

```cpp
dmlGr->AddDMLOption(MARK_DUPLICATE_ROWS);
```

A DMLgroup is added to the Connection class by using the AddDMLGroup function. The AddDMLGroup function takes two parameters:

• A pointer to the DML group being added
• a pointer to a TD_Index object through which the function will return the index reference for the DML group. This index is used later with the UseDMLGroups function.

The following is an example of adding a DML group to the Connection class:

```c++
TD_Index index;
TD_StatusCode returnValue = conn->AddDMLGroup(dmlGr, &index);
```

**Initiate a Teradata PT API Database Connection**

After defining a Teradata PT API Connection object with parameters, initiate a connection to the Teradata Database by using the Connection object’s Initiate method:

```c++
/* Initiate a connection with the database */
TD_StatusCode returnValue = conn->Initiate();
```

**Load Data into or Export Data from the Teradata Database**

After establishing a connection, load or export data from the Teradata Database. Data can be loaded on a row by row basis using the PutRow function or as a block by using the PutBuffer function which is available to the Load, Update, and Stream drivers. Likewise, data can be exported one row at a time using the Export driver’s GetRow function or multiple rows can be exported together in a buffer using the Export driver’s GetBuffer function.

Data can be loaded into or exported from all of the columns in a table or from a subset of the columns in a table. When loading data into a subset of the columns in a table, default values must be specified for the columns not being loaded. In each case, only the columns being loaded or exported from should be defined in the schema.

For example, Table1 has columns A, B, and C. Columns A and B are defined as integers and column C is a variable character column with a length of 25. To load data into column A only, a valid insert statement would be

```sql
INSERT INTO Table1 (A, B, C) VALUES (:A, 0, NULL);
```
where zero is the default value for B and NULL is the default value for column C. The input schema for this example would be defined as only having column A.

**Row by Row Loading into the Teradata Database**

There are three parts to loading data into the Teradata Database using Teradata PT API:

- Load rows
- Inform Teradata PT API that acquisition is complete
- Apply rows to the Teradata Database

Data is loaded using the Connection object’s PutRow function. The PutRow function accepts as its arguments a pointer to a single row of data in null-indicator mode format, and the length of the data row (including the length of the null-indicator bytes). See Figure 5 for an illustration of the indicator-mode input data format.

The PutRow length is the total length of all the column values (including the length bytes for variable-length values) plus the number of indicator bytes. See the Data Format section for information on how to determine the number of indicator bytes in a row.
while (!endOfData) {
    /* Assemble a row in null-indicator format */
    ...
    /* Pass a row to the Teradata Database */
    conn->PutRow(buffer, length);
}

Although the application passes a row at a time, the Load, Update, and Stream drivers accumulate the rows in the CLlv2 buffer and then send them to the DBS by the buffer load.

Once all data has been loaded, the acquisition method is completed by calling the Connection object’s EndAcquisition function. This must be done before the data can be applied.

    conn->EndAcquisition( );

When the acquisition method has been completed, the data is applied in the Teradata Database by using the Connection object’s ApplyRows method. This method should only be called when using the Load or Update driver. The Stream driver automatically applies the changes which means the ApplyRows method should not be called when using the Stream driver.

    conn->ApplyRows( );

**Load Rows into a Teradata Database using PutBuffer**

The Teradata PT API PutBuffer function is available for the Load, Update and Stream drivers. Using PutBuffer for the Update and Stream drivers will only provide performance improvement if the function call overhead to PutRow is costly. If that is not the case, continue to use PutRow for the Update and Stream drivers.

There are five steps to using this method:

1. Set the buffer mode attribute
2. Query the buffer layout
3. Load the buffers
4. Inform Teradata PT API that acquisition is complete
5. Apply rows to the database (except when using the Stream driver)

Before Initiate is called, set the buffer mode attribute to Yes in the Connection object as follows:

    conn->AddAttribute(TD_BUFFER_MODE,"Yes");

In order to buffer the data for block loading, query the Connection object using the event method to obtain the layout for the data buffer. The event method should be queried using the TD_EVT_BufferLayout parameter as follows:

    returnValue = conn->GetEvent(TD_EVT_BufferLayout, &dataptr, &datalen);

The layout returned by the event method for the buffer event contains four 4-byte unsigned integers corresponding to the maximum buffer size, the row header size, the row length size, and the buffer trailer size.

**Note:** These layout values will change depending on the user environment and may be defined differently in future releases. Always obtain these values from the event method before buffering data.
**Figure 4** is a diagram representing the layout of the Load driver data buffer.

When constructing the data buffer, the user is responsible for filling in the row length for each row and the row data. From TTU 8.2, the row length is two bytes for Load and Update operator and it is four bytes for Stream operator. The beginning of the row header and the buffer trailer should be left blank as they will be filled in by Teradata PT API. The total size of the buffer cannot exceed the maximum buffer size returned by the buffer layout event.

Buffers of data are loaded using the Connection object’s PutBuffer method. The PutBuffer method accepts as its arguments a pointer to a buffer of data, the length of the data buffer, and an indicator informing Teradata PT API whether the data in the buffer is in indicator mode (a value of one) or in non-indicator mode (a value of zero). Data for the Update and Stream drivers must be in indicator mode (a value of one).

```c
while (!endOfData) {
    /* Assemble a buffer of rows in null-indicator format */
    ...
    /* Pass a buffer to the Teradata Database */
    conn->PutBuffer(buffer, length, 1);
}
```

Once all data has been loaded, the acquisition method is completed by calling the Connection object’s EndAcquisition method. This must be done before the data can be applied.

```c
conn->EndAcquisition( );
```

When the acquisition method has been completed, the data is applied in the Teradata Database by using the Connection object’s ApplyRows method. This is not needed when using the Stream driver.

```c
conn->ApplyRows( );
```

**Note:** When using the Load, Stream, or Update drivers, the PutBuffer and PutRow features can not be used in the same job.

**Export Data from a Teradata Database using GetRow**

Data can be exported one row at a time using the Connection object’s GetRow function. The GetRow function accepts as its arguments a pointer to an allocated buffer with enough room for one row of data and a pointer to a TD_Length object for storing the length of the data exported.
while ( returnValue != TD_END_Method ) {
    /* Retrieve row from the Teradata Database */
    returnValue = conn->GetRow(&buffer, &length);
}

The GetRow function will return the TD_Success value when a row of data has been successfully retrieved. The TD_End_Method value will be returned when all rows have been retrieved and the data acquisition process is complete.

Export Data from a Teradata Database using GetBuffer

Multiple rows can be exported together in a buffer using the Connection object’s GetBuffer function. Exporting an entire buffer full of rows using the GetBuffer function rather than one row at a time using the GetRow function improves performance by reducing the amount of data movement required. The GetBuffer function accepts as its arguments a pointer to an unallocated character buffer and a pointer to a TD_Length variable. The GetBuffer function uses these two arguments to return a pointer to a data buffer and to return the total length of that data buffer.

conn->GetBuffer(&buffer,&length);

The GetBuffer function returns the TD_Success value when a buffer of data is successfully retrieved. The buffer of data returned by the GetBuffer function must be immediately either copied or processed. The buffer used to return the data is overwritten on the next GetBuffer function call, and the buffer is destroyed when the Connection object is deleted. The GetBuffer function returns the TD_END_Method value when all rows have been retrieved and the data acquisition process is complete.

In order to use the GetBuffer feature, set the TD_BUFFER_MODE attribute to Yes (or Y) before initiating the Connection object. The GetRow function cannot be called if the GetBuffer feature is enabled.

Specify the size and format of the data buffer returned by the GetBuffer function by setting one or more of the following buffer layout attributes:

- TD_BUFFER_MAX_SIZE
- TD_BUFFER_HEADER_SIZE
- TD_BUFFER_LENGTH_SIZE
- TD_BUFFER_TRAILER_SIZE

These attributes correspond to the total maximum size of the data buffer, the row header size allocated for each row in the data buffer, the row length size used for each row in the data buffer, and the trailer size allocated for the data buffer. See Figure 4 on page 29 for a description of the format of the data buffer and see Table 26 for a detailed description of each of the four buffer layout attributes and their default values.

The layout of the data buffer returned by GetBuffer can alternatively be set by using the Export driver’s TD_EVT_BufferLayout modifier along with the PutEvent function. The Export driver’s TD_EVT_BufferLayout modifier can be used to set the data buffer layout any time after the Export driver has been initialized up until the first call to the Export driver’s GetBuffer function. The modifier returns TD_Unavailable if called after the first call to the GetBuffer function.
function. See “PutEvent Modifiers” on page 136 for more information on how to use the Export driver’s TD_Evt_BufferLayout modifier.

**Transfer Data Using GetBuffer and PutBuffer**

Data exported using the Connection object’s GetBuffer function can be directly loaded using the Connection object’s PutBuffer function. This feature improves performance when data transfer is needed and no data transformation is required.

To directly transfer data from the Export driver’s GetBuffer function to the Load, Update, or Stream driver’s PutBuffer function, the Export driver’s buffer layout attributes must be set using the buffer layout values returned by the event method for the Load, Update, or Stream drivers. This causes the Export driver to format the data returned by the GetBuffer function so that it will match the buffer layout required by the Load, Update, or Stream driver’s PutBuffer function. The following steps describe how to accomplish this transfer.

---

**To transfer data using GetBuffer and PutBuffer functions (non-dynamic schema transfer)**

1. Create a Load, Update, or Stream driver and enable the PutBuffer feature by setting the TD_BUFFER_MODE attribute to Yes (or Y).
2. Initiate the Load, Update or Stream driver.
3. Create an Export driver and enable the GetBuffer feature by setting the TD_BUFFER_MODE attribute to Yes (or Y).
4. Use the GetEvent function to query the TD_Evt_BufferLayout event for the Load, Update, or Stream driver and use the buffer layout values returned by this event to set the following attributes for the Export driver:
   - TD_BUFFER_MAX_SIZE
   - TD_BUFFER_HEADER_SIZE
   - TD_BUFFER_LENGTH_SIZE
   - TD_BUFFER_TRAILER_SIZE

After completing these steps, the Export driver can be initiated and the data retrieved by the Export driver’s GetBuffer function can be directly loaded using the Load, Update, or Stream driver’s PutBuffer function.

When using the Export driver’s dynamic feature and sharing the dynamic schema between drivers, the Export driver must be initiated first instead of last. In this situation, use the following alternative procedure.

**Transferring Data using the Dynamic Schema Feature with GetBuffer and PutBuffer Functions**

To implement the Export driver’s dynamic schema feature while using GetBuffer and PutBuffer to transfer data, follow these steps:
Chapter 1: Coding a Teradata PT API Application

Coding Steps

1. Create an Export driver and enable the GetBuffer feature by setting the TD_BUFFER_MODE attribute to Yes (Y).
2. Create a Load, Update, or Stream driver and enable the PutBuffer feature by setting the TD_BUFFER_MODE to Yes (Y).
3. Initiate the Export driver.
4. Initiate the Load, Update, or Stream driver.
5. Use the GetEvent function to query the TD_Evt_BufferLayout event for the Load, Update, or Stream driver.
6. Use the PutEvent function to pass the event data returned by the Load, Update, or Stream driver’s TD_Evt_BufferLayout event directly into the Export driver’s TD_Evt_BufferLayout modifier.

At this point, data can be retrieved from the Export driver using the GetBuffer function and loaded directly using the Load, Update, or Stream driver’s PutBuffer function.

**Note:** The buffer layout values for the Load, Update, or Stream drivers change depending on the user environment and may be defined differently in future releases. Always obtain these values from the event method when transferring data between the Export driver’s GetBuffer function and the Load, Update, and Stream drivers’ PutBuffer function.

**Data Format**

The Teradata PT API PutRow and GetRow methods currently only support the indicator-mode input data format:

- A variable-length indicator bytes field
- A variable-length input data field

The positions of the indicator bits correspond to the fields of the loading and exporting row. There is one bit for each column in the table. The first bit in the first byte is the indicator for the first field in the record. If an indicator bit is set to one, the Teradata Database nulls the corresponding field when the record is loaded. If the indicator bit is set to zero, the Teradata Database loads the data specified for that field.

For example, when there are less than 9 columns, there is a 1-byte indicator-byte length. Between 9 and 16 columns, there is a 2-byte indicator-byte length. There is a 3-byte indicator-byte length between 17 and 24 columns, etc.

**Figure 5: Bytes Illustrating Indicator-Mode Input Data Format**

Find more information about the indicator byte in *Teradata FastLoad Reference*. See “Additional Information” on page 7.
When using the Teradata PT API PutBuffer function with the Load driver, the input data can be in indicator mode or in non-indicator mode format. When using the PutBuffer function with the Update or Stream driver, the input data must be in indicator mode format.

**Interpret Errors**

When the user application receives an error, find more information about the error by using the Connection object’s GetErrorInfo function. The GetErrorInfo function takes two arguments: a pointer to a character buffer for returning the error message string associated with the error and a pointer to a TD_ErrorType variable for returning the type of error.

```c
conn->GetErrorInfo(&errorStringBuffer, &errorType );
```

If GetErrorInfo is called and no error has occurred, then the message string will be set to NULL and the error type will be a value of −1.

Some errors which occur during the acquisition method may require the user application to end the acquisition method before error information can be returned to the user application. In these cases, the status code TD_Call_EndAcq will be returned and the user application must proceed to call the EndAcquisition function. The EndAcquisition function will return an error code for the problem which caused the acquisition method to be aborted. At this time, additional error information relating to this error code will be available through the use of the GetErrorInfo function.

**Add Checkpoint and Restart**

To prevent the loss of work done due to an execution error, Teradata PT API has Checkpoint and Restart functions which establish locations in the data to return to if an error occurs. Both functions are part of the Connection object and can be called at any point after the connection has been initialized. The Checkpoint and Restart functions are available for the Load, Update, and Stream drivers. If an Export driver job fails, the job must start over from the beginning.

**Set Checkpoints**

The Checkpoint function takes two arguments which are used to return checkpoint data to the user application: a pointer to an unallocated data buffer (the checkpoint data storage area) and a pointer to a TD_Length variable that stores the length of the checkpoint data.

```c
conn->Checkpoint(&buffer,&length);
```

The Checkpoint function returns the TD_END Method return value when the end of the Checkpoint method has been reached and a successful checkpoint has been taken. At this point, the Checkpoint function also passes checkpoint data to the user application. This is the only point at which valid checkpoint data is returned. For all other return values, no valid checkpoint data will be returned.

Once the Checkpoint function has successfully completed and has returned valid checkpoint data to the user application, the user application must make a copy of the checkpoint data and store it for later use. The character buffer used to pass the checkpoint data to the user application will be overwritten the next time the Checkpoint function is called and will be destroyed when the Connection object is deleted. It is recommended that an external file or table be used to store the checkpoint data to guard against the user application failing.
Chapter 1: Coding a Teradata PT API Application
Coding Steps

The user application determines at what interval to call the Checkpoint function. If the Checkpoint function returns an error then the data was not successfully checkpointed and no valid checkpoint data will be returned.

**Note:** No checkpoint data is returned by the Checkpoint function for the Load driver and no valid checkpoint data is required by the Restart function for the Load driver. Still, it is highly recommended that user applications using the Load driver be prepared to store and retrieve valid checkpoint data when using the Checkpoint and Restart functions in case this feature is required in future releases.

### Perform a Restart

Here are the main tasks to restarting a Teradata PT API operation:

1. Terminate the connection in which the error occurred.
2. Resolve the issue which caused the error to occur.
3. Create a new connection instance with the same parameters as the original connection which failed.
4. Set the TD_RESTARTMODE attribute with a value of one.
5. Call the Initiate function.
6. Call the Restart function using the checkpoint data from the last successful checkpoint.

The Restart function takes two arguments: a pointer to a character buffer containing the latest checkpoint data and a pointer to a TD_Length variable containing the length of the checkpoint data.

```c
conn->Restart(buffer,length);
```

The Restart function requires that valid checkpoint data be specified. The checkpoint data used to restart must be a copy of the checkpoint data returned by the last successful checkpoint in order to avoid losing any data. If invalid data is given to the Restart function, error code 21044 will be returned, indicating that invalid arguments have been specified.

If no successful checkpoint was taken prior to the job failing, the job cannot be restarted. In this case all error tables, log tables, and work tables must be dropped and the job must start over from the beginning.

### Resubmit the Data

If the job failed during the acquisition phase, then resubmit rows starting from the row after the last successful checkpoint. The user application keeps track of when the last successful checkpoint was taken.

### Query Run-Time Statistics

You can use the GetEvent function to query the driver for run-time statistics. The GetEvent function takes four arguments:

- Type of event
- A pointer for the output data buffer
- A pointer for the output data buffer length
• A target table index. The target table index argument is only used by the Update and Stream drivers.

conn->GetEvent(eventType, eventData, &eventDataLen, tableIndex);

Call the GetEvent method after the driver initiates and before it terminates. Some events require the job to reach a certain point before their data becomes available. If the job has not yet reached this point, the GetEvent method returns TD_Unavailable.

**Modify Job after Initiate**

The PutEvent function can be used to modify certain aspects of a job after the driver has already been initiated. The PutEvent function takes four arguments:

• Type of modifier
• A pointer to the buffer containing the input data for the modifier
• The length of the buffer containing the input data for the modifier
• A target table index. The target table index argument is not currently used by any modifier.

conn->PutEvent(modifierType, ModifierData, modifierDataLen, tableIndex);

The PutEvent function can be called after the driver initiates and before it terminates. Some modifiers require that the job reach a certain point before they can be used to modify a job while others are available right after the driver initiates but become unavailable later in the job. TD_Unavailable is returned when a modifier cannot be used.

**Utilize Serialization with the Stream Driver**

In certain uses of the Stream driver it is possible to have multiple changes to one row in the same job. For instance, the row may be inserted and then updated during the job or it may be updated and then deleted. In any case, the correct ordering of these operations is very important. By using the serialization feature, the Stream driver can guarantee that this ordering of operations is maintained correctly.

The serialization feature works by hashing each data record based upon a set of columns to determine which session transmits the record to the Teradata Database. Thus there is extra overhead in the application derived from the mathematical operation of hashing and from the extra amount of buffering necessary to save data rows when a request is already pending on the session chosen for transmission.

The serialization feature greatly reduces the potential frequency of Teradata Database deadlock. Deadlocks can occur when requests for the application happen to effect row(s) that use the same hash code within the Teradata Database. Although deadlocks are handled by the Teradata Database and by Teradata PT API correctly, the resolution process is time consuming and adds additional overhead to the application because it must re-execute requests which roll back due to deadlock.

In Teradata PT API, serialization is enabled through the DMLGroup object. The DMLGroup object has an AddSerializeOn function which takes two arguments: the number of columns in the column set and a list of column names terminated by a NULL value.

```
dmlGr->AddSerializeOn(2, "Associate_Id", "Associate_Name", NULL);
```
The intent of the column set parameter is to allow the user to specify the column(s) corresponding to the primary index of the target table. If there is more than one target table specified in the DML statements of the DMLGroup, then it is up to the user to make sure the primary indexes of all the tables match when using the serialize feature.

Each DMLGroup object has a single list of columns to use for serialization. Columns can be added to this list in one call or through a series of multiple calls to the AddSerializeOn function. For example, two columns can be added to a serialization list one at a time using separate calls to the AddSerializeOn function:

```c
  dmlGr->AddSerializeOn(1, "Associate_Id", NULL);
  dmlGr->AddSerializeOn(1, "Associate_Name", NULL);
```

Or the two columns can be added in the same call:

```c
  dmlGr->AddSerializeOn(2, "Associate_Id", "Associate_Name", NULL);
```

### Utilize Use Lists with DML Groups

When the SQL statement(s) in a DML group are executed during the application process, there are cases when not all columns of data are needed. In such cases, it is wasteful to use all columns.

In Teradata PT API, use lists allow the user to define which columns will be needed by the SQL statements in a DML group. The DMLGroup object has an AddUseList method which takes two arguments: the number of columns in the column set and a list of column names followed by a NULL value.

```c
  dmlGr->AddUseList(3, "Associate_Id", "Associate_Name", "Salary", NULL);
```

A DMLGroup object has a maximum of one use list. Columns can be added to this use list in one call or through a series of multiple calls to the AddUseList function. For example, three columns can be added to a use list in separate calls to the AddUseList function:

```c
  dmlGr->AddUseList(1, "Associate_Id", NULL);
  dmlGr->AddUseList(2, "Associate_Name", "Salary", NULL);
```

Or three columns can be added in the same call:

```c
  dmlGr->AddUseList(3, "Associate_Id", "Associate_Name", "Salary", NULL);
```

### Terminate the Teradata PT API Connection

Once a Teradata PT API driver task completes, the connection must be terminated. Do this with the Connection object’s Terminate method.

```c
  conn->Terminate();
```

The Terminate method closes all sessions associated with the connection and ends the driver’s processing environment.

Your program should always call Terminate whenever any program fails prematurely, so that Teradata PT API can do cleanup tasks such as releasing storage and logging off all DBS sessions. If your program cannot call Terminate and has stopped, you may be able to submit a restart job. If you submit a restart job and get errors, wait until the DBS logs off all orphan sessions (the default time is 20 minutes) and then submit the restart job again.
Note: If you are making direct calls to CLIv2, do not call the DBCHCLN function until all initiated Teradata PT API drivers within the same process have been terminated. In addition to freeing shared CLIv2 structures, the DBCHCLN method disconnects all currently connected CLIv2 sessions within the same address space. If you call DBCHCLN while a Teradata PT API driver in the same process is still active, the driver's session will prematurely disconnect and the job will fail.

Special Topics

Reuse a Teradata PT API Connection Object

An instance of the Connection object can be reused multiple times to perform the same task. The following restrictions apply:

- The Connection object must be terminated before re-initiating.
- No new attributes, schemas, or DML groups can be added after the first time the Connection object is initiated.
- Existing attributes, schemas, or DML groups cannot be changed.

Query Banding Teradata Database Feature

Available with the V12.0 Teradata Database, the query banding feature provides a user-defined query band expression that is set for every SQL session connected by the Teradata PT API driver. A query band expression is a set of name-value pairs that identify a query's originating source. In the expression, each name-value pair is separated by a semicolon and the expression ends with a semicolon. The following is an example of a valid query band expression:

```
a=1;b=2;c=3;d=4;
```

If the TD_QUERY_BAND_SESS_INFO is set, the following request will be sent by every SQL session connected by the Teradata PT API driver:

```
SET QUERY_BAND = '<User-Defined Query Band Expression>' FOR SESSION;
```

Setting the TD_QUERY_BAND_SESS_INFO attribute in jobs running against non-supported versions of the Teradata Database causes a non-fatal error. No error code is returned to the user during initiation and the job is allowed to proceed. The log table will not be dropped at the end of the job and the TD_Evt_ExitCode event returns a warning value of four instead of the normal success value of zero if queried. In this case, error information can be found in the trace file.

Using the UTF16 Session Character Set

Setting the TD_CHARSET optional attribute to UTF16 allows users to load and export UTF-16 data to and from Teradata systems. The TD_CHARSET attribute is available for each driver. Teradata PT API applications that use the UTF16 session character set must be aware of the differences in the expected sizes for CHAR and VARCHAR columns in the Schema object. The Schema object’s AddColumn method expects different sizes for CHAR and VARCHAR
columns when the session character set is UTF16 or UTF8 than it does when the session character set is ASCII.

**Note:** On channel-attached z/OS platforms, only the UTF8 and UTF16 session character sets are not supported.

For example, a table is created with a column defined for data in unicode format:

```sql
create table utf16_tbl (c1 VARCHAR(100) character set unicode);
```

If a Teradata PT API application wanted to load or export ASCII data into this table then column c1 would be added to the Schema object in the following way:

```csharp
schema>AddColumn(“c1”, TD_VARCHAR, 100);
```

However, if a Teradata PT API application wanted to load or export UTF-16 data into this table then column c1 would have to be added to the Schema object in this way:

```csharp
schema>AddColumn(“c1”, TD_VARCHAR, 200);
```

The size parameter in the AddColumn method refers to the size of the column in bytes. Valid UTF-16 data in a Teradata Database contains two bytes per character, which is why 200 bytes is specified instead of 100 bytes. The same principle is true when the session character set is UTF8 because UTF-8 characters can be up to three bytes in length. Therefore, if a Teradata PT API application wanted to load or export UTF-8 data into the table defined above then column c1 would be added to the Schema object in the following way:

```csharp
schema>AddColumn(“c1”, TD_VARCHAR, 300);
```

### Using the UTF16 Encoding for Teradata PT API Objects and Messages

User attributes, DML statements, and column names can be in the UTF-16 encoding when the session character set is UTF16. Teradata PT API applications have the choice of passing in these values in UTF-16 or in UTF-8 when the session character set is UTF16. To accommodate this option, the Connection, Schema, and DML Group objects can be instantiated with an encoding parameter instead of the default void parameter. UTF-8, which is an ASCII-based encoding for Unicode characters, is the default encoding for all Connection, Schema, and DML Group objects.

**Note:** On channel-attached z/OS platforms, the use of UTF-16 or UTF-8 encoding is not supported because EBCDIC encoding is always used.

If users specify all Connection attributes in the UTF-16 encoding then their Connection objects must be instantiated in the following way:

```csharp
Connection*conn = new Connection(TD_UTF16_ENCODING);
```

If users specify all column names passed to the Schema object methods in the UTF-16 encoding then their Schema objects must be instantiated in the following way:

```csharp
Schema*schema = new Schema(TD_UTF16_ENCODING);
```

If users specify all column names and DML statements passed to their DMLGroup object methods in the UTF-16 encoding then their DMLGroup objects must be instantiated in the following way:

```csharp
DMLGroup*dmlgr = new DMLGroup(TD_UTF16_ENCODING);
```
If an object is instantiated with TD_UTF16_ENCODING then Teradata PT API will treat all character strings passed in to any of that object’s methods as being in the UTF-16 encoding. Also, all UTF-16 character strings passed to the Connection, Schema, or DMLGroup object methods must be NULL-terminated. This means that the last two bytes of the UTF-16 string must be zero. This is important so Teradata PT API can accurately determine the length of the UTF-16 string passed to any of the Connection, Schema, or DMLGroup object methods.

An option also exists for the Teradata PT API application to receive messages between itself and Teradata PT API in UTF-8 or UTF-16 when the session character set is UTF16. The optional attribute TD_MSG_ENCODING will dictate the behavior of Teradata PT API in this regard. Messages that are passed between Teradata PT API and the calling application include Teradata CLIv2 errors, Teradata Database errors, and table or macro or database names found in the buffer for the Stream driver’s ApplyCount event. The value for the TD_MSG_ENCODING attribute can only be TD_UTF8_ENCODING (the default) or TD_UTF16_ENCODING. This optional attribute should only be used when the session character set attribute, TD_CHARSET, is set to UTF16.

The following cases will result in errors reported by Teradata PT API:

- If the TD_MSG_ENCODING attribute is used and has a value of TD_UTF16_ENCODING but the TD_CHARSET attribute is not set to UTF16.
- If a Connection object is instantiated with TD_UTF16_ENCODING but the TD_CHARSET attribute is not set to UTF16.
- If a Schema object is instantiated with TD_UTF16_ENCODING but the TD_CHARSET attribute is not set to UTF16.
- If a DMLGroup object is instantiated with TD_UTF16_ENCODING but the TD_CHARSET attribute is not set to UTF16.

Additionally, all invalid character errors detected in any UTF-16 attribute, column name, or DML statement will be treated as terminating errors.

### Using the Export Driver’s Dynamic Schema Feature

The Export driver’s dynamic schema feature provides the option of having the Export driver dynamically generate the output schema based on the SELECT statement. This feature frees the user application from having to provide a pre-defined output schema.

Enable this feature by not adding a schema to the Export driver prior to initialization. When the Export driver detects that no user-defined output schema has been added, the Export driver will create an output schema based on the user-provided SELECT statement and the job will continue as normal.

### Sharing the Dynamic Schema with Other Drivers

Follow these steps to share the Export driver’s dynamically-generated output schema with other drivers:

1. Create an instance of the Export driver and add all necessary attributes. Enable the dynamic schema feature by not adding a schema to the Export driver.
2. Initialize the Export driver.
3 Retrieve the dynamically generated schema from the Export driver through the use of the GetSchema function:

```c
Schema* dynamicSchema;
returnValue = exportConnection->GetSchema(&dynamicSchema);
```

4 Pass the dynamically generated schema to the other drivers by using the AddSchema function:

```c
loadConnection->AddSchema(dynamicSchema);
```

5 Initialize the other drivers.

**Note:** The Export driver’s dynamic schema feature is available only in single instance Export jobs. Attempting to use the feature in multiple instance Export jobs will result in an error for both the master Export instance and the slave Export instances.

### Available Teradata Parallel Transporter Functions

Table 1 gives an overall view of which functions are available to each Teradata PT API driver.

<table>
<thead>
<tr>
<th>Load</th>
<th>Update</th>
<th>Stream</th>
<th>Export</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>with Pause</td>
<td>with Delete</td>
<td>Standard</td>
</tr>
<tr>
<td>Pause Acquisition</td>
<td>Task</td>
<td>Acquisition</td>
<td>Standard</td>
</tr>
<tr>
<td>Initiate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PutRow</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>PutBuffer</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>GetRow</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>GetBuffer</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>EndAcquisition</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>ApplyRows</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Terminate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The following tables list objects and their functions within each class. The syntax of the function and all possible return codes are also listed. This chapter’s main sections cover:

- Connection Class
- Schema Class
- DML Groups
- Teradata PT API Constants
- DML Option
- TD_Attribute
- TD_DataType
- TD_OperatorType
- TD_TRACE_LEVEL
- Teradata PT API Status Messages

Connection Class

The Connection class manages all of the attributes required to connect to the Teradata Database. These attributes store attribute, schema and DML Group objects.

Note: On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Notes have been made for the following syntax examples to use #include “connecti.h” instead of “connection.h.”
## Table 2: Connection Class Objects

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Syntax and Notes</th>
<th>Return Codes</th>
</tr>
</thead>
</table>
| **AddArrayAttribute**    | Adds a connection attribute with more than one value. | #include “connection.h”  
Void AddArrayAttribute(  
    TD_Attribute attribute,  
    TD_Count count,  
    Char* value, ...,  
    NULL);  
Void AddArrayAttribute(  
    TD_Attribute attribute,  
    TD_Count count,  
    TD_IntValue value, ...,  
    NULL );  
|  
| Where these parameters specify: |  
| • attribute Input for the type of attribute being added. |  
| • count Input for the number of values. |  
| • value Input for the values of attributes terminated by NULL. |  
| • On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.” |  
| **AddAttribute**        | Adds a connection attribute. | #include “connection.h”  
Void AddAttribute(  
    TD_Attribute attribute,  
    Char* value);  
Void AddAttribute(  
    TD_Attribute attribute,  
    TD_IntValue value);  
|  
| Where these parameters specify: |  
| • attribute Input for the type of attribute being added. |  
| • value Input for the value of the attribute. |  
| • On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.” |  
| **null**                | None            | None        |
Table 2: Connection Class Objects (continued)

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Syntax and Notes</th>
<th>Return Codes</th>
</tr>
</thead>
</table>
| **AddDMLGroup**          | Adds a DMLGroup object to the connection. | #include “connection.h”  
TD_StatusCode  
AddDMLGroup(  
DMLgroup* dmlgroup,  
TD_Index* index);  
- Where these parameters specify:  
  - dmlgroup Input pointer to the DmlGroup object.  
  - index Output for the index number of the DMLGroup.  
- On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.” | TD_Error, TD_Success  
See “Teradata PT API Status Messages” on page 59 for more information. |
| **AddSchema**            | Adds a schema object to the connection. A schema is the definition of the columns in a table or data source. An input schema defines the fields in a record of input data. | #include “connection.h”  
void  
AddSchema(  
Schema* schema);  
- Where the schema parameter is input for the pointer to the schema object.  
- On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.” | None |
| **ApplyRows**            | Applies the data loaded in the acquisition phase. (Load and Update only). | #include “connection.h”  
TD_StatusCode  
ApplyRows( );  
On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.” | TD_Error,  
TD_END_Method  
TD_SYNC_Barrier,  
TD_SYNC_TELINFO  
See “Teradata PT API Status Messages” on page 59 for more information. |
Table 2: Connection Class Objects (continued)

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Syntax and Notes</th>
<th>Return Codes</th>
</tr>
</thead>
</table>
| **Checkpoint**           | **Tell the driver to perform a checkpoint.**
   
   ```
   #include "connection.h"
   TD_StatusCode
   Checkpoint(
       char **data,
       TD_Length* length);
   ```
   
   - Where these parameters specify:
     - **Data** Output buffer for the checkpoint data.
     - **Length** Output for the length of the checkpoint buffer.
   
   - The checkpoint method currently returns NULL with a data length of zero. These Checkpoint parameters are reserved for future expansion.
   
   Refer to “Add Checkpoint and Restart” on page 33 for more information.
   
   - On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECT). Use #include “connecti.h” instead of “connection.h.”

| **Class Constructor**  | **Returns the Connection class instance.**
   
   ```
   #include "connection.h"
   using namespace teradata::client::API;
   Connection* Connection( );
   ```
   
   On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECT). Use #include “connecti.h” instead of “connection.h.”

| **Class Constructor**  | **Specifies encoding for the Connection class and returns the Connection class instance.**
   
   ```
   #include "connection.h"
   using namespace teradata::client::API;
   Connection* Connection(TD_Encoding);
   ```
   
   - Valid values are TD_UTF8_ENCODING (the default) and TD_UTF16_ENCODING.
   
   - This Class Constructor is not supported on channel-attached z/OS platforms.

| **EndAcquisition**     | **Informs the driver that the data acquisition phase is completed. This function must be called before the data can be applied.**
   
   ```
   #include "connection.h"
   TD_StatusCode
   EndAcquisition( );
   ```
   
   On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECT). Use #include “connecti.h” instead of “connection.h.”

   - TD_Call_EndAcq, TD_END_Method, TD_Error, TD_SYNC_Barrier, TD_SYNC_TELINFO
   
   See “Teradata PT API Status Messages” on page 59 for more information.
### Table 2: Connection Class Objects (continued)

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Syntax and Notes</th>
<th>Return Codes</th>
</tr>
</thead>
</table>
| GetErrorInfo             | #include “connection.h”  
Void GetErrorInfo(  
    Char* errorMsg,  
    TD_ErrorType* errorType);  
| Where these parameters specify:  
  - `errorMsg` Output error string message associated with the last received error. Set to NULL if no error has occurred.  
  - `errorType` Output error type associated with the last received error (A value of zero for Teradata PT API errors, a value of one for DBS errors, a value of two for CLIv2 errors and a value of minus one when no errors have occurred.)  
  - On channel-attached z/OS platforms, `connection.h` is located in `TWB.H(CONNECTI)`. Use `#include “connecti.h”` instead of “connection.h.” | None |
| GetBuffer                | #include “connection.h”  
TD_StatusCode GetBuffer(  
    char** data, TD_Length* length);  
| Where these parameters specify:  
  - `Data` Output for the buffer of data.  
  - `Length` Output parameter that contains the length (in bytes) of the data buffer.  
  - On channel-attached z/OS platforms, `connection.h` is located in `TWB.H(CONNECTI)`. Use `#include “connecti.h”` instead of “connection.h.” | TD_END_Method,  
TD_Error,  
TD_SYNC_Barrier,  
TD_SYNC_TELINFO,  
TD_Success  
See “Teradata PT API Status Messages” on page 59 for more information. |
### Table 2: Connection Class Objects (continued)

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Syntax and Notes</th>
<th>Return Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>GetEvent</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Retrieves run-time statistics from the driver.  
```
#include "connection.h"
TD_StatusCode
GetEvent(
    TD_EventType eventType,
    char** eventData,
    TD_Length* eventDataLen,
    TD_Index eventIndex = 0);
```
- Where these parameters specify:
  - `eventType` Input for the name of the event to retrieve statistics for.
  - `eventData` Output pointer to the appropriate expected return data for the event.
  - `eventDataLen` Output parameter that contains the length (in bytes) of the data for the event.
  - `eventIndex` Input Target table index for the desired event data. This optional parameter only applies to a few events with the default value being 0.
- On channel-attached z/OS platforms, `connection.h` is located in `TWB.H(CONNECTI)`. Use `#include "connecti.h"` instead of `"connection.h"`.

| **GetRow**                | 
Retrieves a row of data from the Teradata Database. Data is in null indicator mode.  
```
#include "connection.h"
TD_StatusCode
GetRow(  
    Byte** data,  
    TD_Length* length);
```
- Where these parameters specify:
  - `Data` Output for the buffer of data.
  - `Length` Output for the length of the data buffer.
- On channel-attached z/OS platforms, `connection.h` is located in `TWB.H(CONNECTI)`. Use `#include "connecti.h"` instead of `"connection.h"`.

| **GetSchema**             | 
Retrieves the schema.  
```
#include "connection.h"
TD_StatusCode
GetSchema(Schema** schema);
```
- Where `schema` is the output schema pointer.
- On channel-attached z/OS platforms, `connection.h` is located in `TWB.H(CONNECTI)`. Use `#include "connecti.h"` instead of `"connection.h"`.

See “Teradata PT API Status Messages” on page 59 for more information.
### GetTELINFO

Retrieves the TELINFO area from the master instance in a multi-instance environment.

```c
#include "connection.h"
TD_StatusCode
GetTELINFO(Char** TELINFO, TD_Length* length);
```

Where these parameters specify:
- **TELINFO** Output pointer to the buffer containing the TELINFO area.
- **Length** Output for the length of the TELINFO area.
- On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include "connecti.h" instead of "connection.h."

### Initiate

Processes the Connection class attributes, schemas, and DML groups, initializes the driver and SQL Teradata FastLoad Reference, Teradata FastExport Reference, and Teradata MultiLoad Reference sessions are connected by the driver.

```c
#include "connection.h"
TD_StatusCode
Initiate();
```

On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include "connecti.h" instead of "connection.h."

### PutBuffer

Sends a buffer of data to the Teradata Database. It eliminates the extra data movement to the CLlv2 Data or IndicData. See Load Data into or Export Data from the Teradata Database for more information.

```c
#include "connection.h"
TD_StatusCode
PutBuffer(Char* data, TD_Length length, TD_Boolean indicator);
```

Where these parameters specify:
- **Data** Input for the buffer of data.
- **Length** Input for the length of the data buffer.
- **Indicator** Input parameter that equals one if the data is in indicator mode or equals zero if in non-indicator mode. Must be set to one when used with the Update and Stream drivers.
- On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include "connecti.h" instead of "connection.h."
### Table 2: Connection Class Objects (continued)

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Syntax and Notes</th>
<th>Return Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PutEvent</strong></td>
<td><code>#include &quot;connection.h&quot;</code>&lt;br&gt;<code>TD_StatusCode PutEvent(</code>&lt;br&gt;<code>   TD_EventType modifierType,</code>&lt;br&gt;<code>   char * modifierData,</code>br&gt;<code>   TD_Length modifierDataLen,</code>&lt;br&gt;<code>   TD_Index modifierIndex = 0);</code></td>
<td><code>TD_EndMethod,</code>&lt;br&gt;<code>TD_Error,</code>&lt;br&gt;<code>TD_SYNC_Barrier,</code>&lt;br&gt;<code>TD_SYNC_TELINFO,</code>&lt;br&gt;<code>TD_Unavailable</code>&lt;br&gt;<code>See “Teradata PT API Status Messages” on page 59 for more information.</code></td>
</tr>
<tr>
<td><strong>PutRow</strong></td>
<td><code>#include &quot;connection.h&quot;</code>&lt;br&gt;<code>TD_StatusCode PuRow(</code>&lt;br&gt;<code>   Char* data,</code>&lt;br&gt;<code>   TD_Length length );</code></td>
<td><code>TD_Call_EndAcq,</code>&lt;br&gt;<code>TD_Error, TD_Success</code>&lt;br&gt;<code>See “Teradata PT API Status Messages” on page 59 for more information.</code></td>
</tr>
</tbody>
</table>

- **PutEvent**
  - Modifies the driver at runtime.
  - Where these parameters specify:
    - `modifierType` Input used for the modifier name. Specifies which modifier receives information.
    - `modifierData` Input pointer for the appropriate expected modifier data
    - `modifierDataLen` Input parameter specifying the length (in bytes) for the modifier data.
    - `modifierIndex` Input Target table index for the desired event. This optional parameter is not currently used by any modifier.
    - On channel-attached z/OS platforms, `connection.h` is located in `TWB.H(CONNECTI)`. Use `#include "connecti.h"` instead of “connection.h.”

- **PutRow**
  - Sends a row of data to the Teradata Database. Data must be in null indicator mode format.
  - Where these parameters specify:
    - `Data` Input for the buffer containing a row of data.
    - `Length` Input for the length of the data buffer. The length is the total length of all the column values (including the length bytes for variable-length values) plus the number of indicator bytes. See “Data Format” on page 32 for information on how to determine the number of indicator bytes.
    - On channel-attached z/OS platforms, `connection.h` is located in `TWB.H(CONNECTI)`. Use `#include "connecti.h"` instead of “connection.h.”
Table 2: Connection Class Objects (continued)

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Syntax and Notes</th>
<th>Return Codes</th>
</tr>
</thead>
</table>
| PutTELINFO               | #include “connection.h”
                          | TD_StatusCode
                          | PutTELINFO( Char* TELINFO, TD_Length length);
                          | • Where these parameters specify:
                          | • TELINFO Input to the buffer containing the TELINFO area.
                          | • Length Input for the length of the TELINFO buffer.
                          | • On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.”
                          | TD_Error, TD_Success
                          | See “Teradata PT API Status Messages” on page 59 for more information. |
| Terminate                | #include “connection.h”
                          | TD_StatusCode
                          | Terminate();
                          | On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.”
                          | TD_END_Method, TD_Error,
                          | TD_SYNC_Barrier,
                          | TD_SYNC_TELINFO
                          | See “Teradata PT API Status Messages” on page 59 for more information. |
| Restart                  | #include “connection.h”
                          | TD_StatusCode
                          | ReStart( char *data,
                          | TD_length length );
                          | • Where these parameters specify:
                          | • Data Input buffer for the checkpoint data.
                          | • Length Input for the length of the checkpoint buffer.
                          | • On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.”
                          | TD_END_Method, TD_Error,
                          | TD_SYNC_Barrier,
                          | TD_SYNC_TELINFO
                          | See “Teradata PT API Status Messages” on page 59 for more information. |
### Table 2: Connection Class Objects (continued)

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Syntax and Notes</th>
<th>Return Codes</th>
</tr>
</thead>
</table>
| UseDMLGroups             | Designates which DML group to use. | #include “connection.h”  
TD_StatusCode  
UseDMLGroups(index, count); | TD_Error, TD_Success  
See “Teradata PT API Status Messages” on page 59 for more information. |

- Where these parameters specify:
  - *Index* Input pointer to an array of indexes. Each index refers to a DML group added using the AddDMLGroup function.
  - *Count* Input for the number of indexes in the index array.
- On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.”
Schema Class

The schema class of attributes contains information about the format of the data used in the task. This information is used by the drivers to process the data being loaded and exported.

When loading data with the Load, Update, or Stream drivers, ensure that the row data is consistent with the layout defined in the input schema. Discrepancies in the length of the row data could result in data corruption. If the row data is longer than defined in the input schema during PutRow or PutBuffer, Teradata PT API will terminate with an error.

Note: On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Notes have been made for the following syntax examples to use #include “connecti.h” instead of “connection.h.”

Table 3: Schema Class Objects

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Structure and Notes</th>
<th>Return Codes</th>
</tr>
</thead>
</table>
| AddColumn                | #include “connection.h”
                            Void
                            AddColumn(
                                Char* colName,
                                TD_DataType datatype,
                                TD_ColumnSize size = 0,
                                TD_ColumnPrecision precision = 0,
                                TD_ColumnScale scale = 0);
                            • Where these parameters specify:
                              • colName Input for column name.
                              • dataType Input for the column's SQL data type.
                              • size Input for column size.
                              • precision Input for Column precision (Only applicable to decimal and period types).
                              • scale Input for Column scale (Only applicable to decimal type).
                              • On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.” |
| Class Constructor        | #include “connection.h”
                            using namespace teradata::client::API;
                            Schema*
                            Schema(Char* stype);
                            • Where stype is the input parameter for the type of schema (input or output).
                            • On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.” | None |
DML Groups

Each DMLGroup has one or more DML Statements along with a set of DML Options which is used to load or update data.

Note: On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Notes have been made for the following syntax examples to use #include “connecti.h” instead of “connection.h.”

Table 4: DMLGroup Class Objects

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Syntax</th>
<th>Return Codes</th>
</tr>
</thead>
</table>
| AddArraySupport          | #include “connection.h”
                          | Void
                          | AddArraySupport($char* arraysupport); |
                          | • Where arraysupport is on or off. |
                          | • On channel-attached z/OS platforms, connection.h is located in TWB.H(CONNECTI). Use #include “connecti.h” instead of “connection.h.” | None |
AddDMLOption
Sets a DML option to be used with the DMLGroup.

```c
#include "connection.h"
Void AddDMLOption(DMLOption option);
```

- Where `option` is an input parameter for the DML Option constant.
- On channel-attached z/OS platforms, `connection.h` is located in TWB.H(CONNECTI). Use `#include "connecti.h"` instead of "connection.h."

AddSerializeOn
Designates which column(s) to serialize when using the Stream driver.

```c
#include "connection.h"
Void AddSerializeOn(Int argc, ..., NULL);
```

- Where these parameters specify:
  - `argc` Input for the number of values.
  - `column(s) (...)` Input for the name(s) of column(s) as defined in the table.
- On channel-attached z/OS platforms, `connection.h` is located in TWB.H(CONNECTI). Use `#include "connecti.h"` instead of "connection.h."

AddStatement
Adds a DML statement to the DMLGroup.

```c
#include "connection.h"
Void AddStatement(Char* statement);
```

- Where `statement` is an input parameter for the DML statement.
- On channel-attached z/OS platforms, `connection.h` is located in TWB.H(CONNECTI). Use `#include "connecti.h"` instead of "connection.h."

AddUseList
Designates which column(s) to use when executing the SQL statements in a DML group during the application process.

```c
#include "connection.h"
Void AddUseList(Int argc, ..., NULL);
```

- Where these parameters specify:
  - `argc` Input for the number of values.
  - `column(s) (...)` Input for the name(s) of column(s) as defined in the table.
- On channel-attached z/OS platforms, `connection.h` is located in TWB.H(CONNECTI). Use `#include "connecti.h"` instead of "connection.h."

Class Constructor
Returns DMLGroup class instance.

```c
#include "DMLGroup.h"
using namespace teradata::client::API;
DMLGroup* DMLGroup();
```

None
Table 4: DMLGroup Class Objects (continued)

<table>
<thead>
<tr>
<th>Object Name and Function</th>
<th>Syntax</th>
<th>Return Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class Constructor</strong></td>
<td></td>
<td>None</td>
</tr>
</tbody>
</table>
| Specifies encoding for the DMLGroup class and returns the DMLGroup class instance. | ```
#include "DMLGroup.h"
using namespace teradata::client::API;
DMLGroup* DMLGroup(TD_Encoding encoding);
``` | |
| ```
- Where encoding is the input parameter for the type of encoding for this class. Valid values are TD_UTF8_ENCODING (the default) and TD_UTF16_ENCODING.
- This Class Constructor is not supported on channel-attached z/OS platforms.
``` | |

**Teradata PT API Constants**

For use with the AddDMLOption function in the DMLGroup object. This section describes options to use with the DML statements in the DML group. Default settings are indicated in the description column. These options apply only to the Update and Stream drivers.

**DML Option**

Table 5: DML Group Options

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IGNORE_DUPLICATE.Rows</td>
<td>Duplicate rows for both insert and update operations are ignored.</td>
</tr>
<tr>
<td>IGNORE_DUPLICATE_INSERT.Rows</td>
<td>Duplicate rows for insert operations are ignored.</td>
</tr>
<tr>
<td>IGNORE_DUPLICATE_UPDATE.Rows</td>
<td>Duplicate rows for update operations are ignored.</td>
</tr>
<tr>
<td>IGNORE_EXTRA_DELETE.Rows</td>
<td>Rows that effect more than one row in the target table for delete operations are ignored. This option is supported only by the Stream driver.</td>
</tr>
<tr>
<td>IGNORE_EXTRA_ROWS</td>
<td>Rows that effect more than one row in the target table for delete or update operations are ignored. This option is supported only by the Stream driver.</td>
</tr>
<tr>
<td>IGNORE_EXTRA_UPDATE.Rows</td>
<td>Rows that effect more than one row in the target table for update operations are ignored. This option is supported only by the Stream driver.</td>
</tr>
<tr>
<td>IGNORE_MISSING_ROWS</td>
<td>Missing rows for update and delete operations are ignored.</td>
</tr>
<tr>
<td>IGNORE_MISSING_DELETE.Rows</td>
<td>Missing rows for delete operations are ignored.</td>
</tr>
<tr>
<td>IGNORE_MISSING_UPDATE.Rows</td>
<td>Missing rows for update operations are ignored.</td>
</tr>
</tbody>
</table>
## Table 5: DML Group Options (continued)

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT_FOR_MISSING_UPDATE_ROWS</td>
<td>This option applies only when the DML group consists of a single update statement, followed by a single insert statement.</td>
</tr>
<tr>
<td>MARK_DUPLICATE_ROWS</td>
<td>Duplicate rows for both insert and update operations are placed in error table. This is the default setting.</td>
</tr>
<tr>
<td>MARK_DUPLICATE_INSERT_ROWS</td>
<td>Duplicate rows for insert operations are placed in error table. This is the default setting.</td>
</tr>
<tr>
<td>MARK_EXTRA_DELETE_ROWS</td>
<td>Rows that effect more than one row in the target table for delete operations are placed in the error table. This is the default setting. This option is supported only by the Stream driver.</td>
</tr>
<tr>
<td>MARK_EXTRA_ROWS</td>
<td>Rows that effect more than one row in the target table for delete or update operations are placed in the error table. This is the default setting. This option is supported only by the Stream driver.</td>
</tr>
<tr>
<td>MARK_EXTRA_UPDATE_ROWS</td>
<td>Rows that effect more than one row in the target table for update operations are placed in the error table. This is the default setting. This option is supported only by the Stream driver.</td>
</tr>
<tr>
<td>MARK_MISSING_DELETE_ROWS</td>
<td>Missing rows for delete operations are placed in error table. This is the default setting.</td>
</tr>
<tr>
<td>MARK_MISSING_ROWS</td>
<td>Missing rows for update and delete operations are placed in error table. This is the default setting.</td>
</tr>
<tr>
<td>MARK_MISSING_UPDATE_ROWS</td>
<td>Missing rows for update operations are placed in error table. This is the default setting.</td>
</tr>
<tr>
<td>MARK_MISSING_UPDATE_ROWS</td>
<td>Missing rows for update operations are placed in error table. This is the default setting.</td>
</tr>
</tbody>
</table>
TD_Attribute

For use with the AddAttribute and AddArrayAttribute functions in the Connection object. Once an attribute is set in the connection object, it cannot be reset to a different value in the same connection.

- For the Load driver, see “Attribute Definitions” on page 61.
- For the Update driver, see “Attribute Definitions” on page 77.
- For the Stream driver, see “Attribute Definitions” on page 101.
- For the Export driver, see “Attribute Definitions” on page 127.

TD_DataType

For use with the AddColumn function in the schema object. The following table defines data type for each column in the schema. Refer to SQL Data Types and Literals for detailed information on each data type including an explanation of precision and scale for the decimal type and length and precision for the period data type. Check http://www.info.teradata.com/ for the latest version.

Table 6: Schema Object AddColumn Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_BIGINT</td>
<td>big integer</td>
</tr>
<tr>
<td>TD_BYTE</td>
<td>byte</td>
</tr>
<tr>
<td>TD_BYTEINT</td>
<td>byte integer</td>
</tr>
<tr>
<td>TD_CHAR</td>
<td>character</td>
</tr>
<tr>
<td>TD_DATE</td>
<td>date</td>
</tr>
<tr>
<td>TD_DATE_ANSI</td>
<td>ANSI date</td>
</tr>
<tr>
<td>TD_DECIMAL</td>
<td>decimal</td>
</tr>
<tr>
<td>TD_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>TD_GRAPHIC</td>
<td>graphic</td>
</tr>
<tr>
<td>TD_INTEGER</td>
<td>integer</td>
</tr>
<tr>
<td>TD_LONGVARCHAR</td>
<td>long variable length string</td>
</tr>
<tr>
<td>TD_LONGVARGRAPHIC</td>
<td>long variable length graphic</td>
</tr>
<tr>
<td>TD_NONE</td>
<td>no type specified</td>
</tr>
<tr>
<td>TD_PERIOD_DATE</td>
<td>period(date)</td>
</tr>
<tr>
<td>TD_PERIOD_TIME</td>
<td>period(time)</td>
</tr>
<tr>
<td>TD_PERIOD_TIME_TZ</td>
<td>period(time with time zone)</td>
</tr>
<tr>
<td>TD_PERIOD_TS</td>
<td>period(time stamp)</td>
</tr>
</tbody>
</table>
On Specifying Data Types

Note the following when specifying data types:

- The length of any CHAR or VARCHAR column should be an even number when using the UTF16 session character set. If the length for either one of these data types is an odd number when using the UTF16 session character set then an error will be returned by the Initiate method.
- Do not use the TD_LONGVARCHAR data type when the server storage character set or the client session character set is a multi-byte character set. KANJISJIS_OS, UTF8, and UTF16 are examples of multi-byte character sets.
- The PERIOD data types that support the date and time are in the form:
  - PERIOD(DATE)
  - PERIOD(TIME(n))
  - PERIOD(TIME(n) WITH TIME ZONE)
  - PERIOD(TIMESTAMP(n))
  - PERIOD(TIMESTAMP(n) WITH TIME ZONE)

The (n) corresponds to the precision of the column. The precision of these columns can be an integer from zero to six. Six is the default if no precision is specified. The PERIOD(DATE) data type has no precision.

- PERIOD(DATE), PERIOD(TIME(n)) and PERIOD(TIME(n)) WITH TIME ZONE are fixed length.
- PERIOD(TIMESTAMP(n)) and PERIOD(TIMESTAMP(n)) WITH TIME ZONE are variable length.
- When using the TIME, TIMESTAMP, and INTERVAL data types, refer to the Teradata Parallel Transporter Reference. Check http://www.info.teradata.com/ for the latest version.

TD_Encoding

Table 7: TD_Encoding Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_UTF8_ENCODING</td>
<td>UTF-8 encoding</td>
</tr>
</tbody>
</table>
Chapter 2: Interface Specifications
Teradata PT API Constants

Note: The default setting for all drivers is TD_UTF8_ENCODING. On channel-attached z/OS platforms, only EBCDIC encoding is supported and is automatically selected.

**TD_OperatorType**

For use with the AddAttribute function as values for the TD_SYSTEM_OPERATOR attribute. The following tables designate which driver to use with the connection.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_EXPORT</td>
<td>Export driver</td>
</tr>
<tr>
<td>TD_LOAD</td>
<td>Load driver</td>
</tr>
<tr>
<td>TD_NO_OPERATOR</td>
<td>No driver specified</td>
</tr>
<tr>
<td>TD_STREAM</td>
<td>Stream driver</td>
</tr>
<tr>
<td>TD_UPDATE</td>
<td>Update driver</td>
</tr>
</tbody>
</table>

**TD_TRACE_LEVEL**

For use with the AddArrayAttribute as values for the TD_TRACE_LEVEL attribute. Defines levels of tracing for both the driver level and infrastructure level. This option is for Teradata developers and Customer Support associates use only. The customers should have tracing disabled.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_OPER</td>
<td>Enables tracing for driver specific activities.</td>
</tr>
<tr>
<td>TD_OFF</td>
<td>Tracing is disabled.</td>
</tr>
<tr>
<td>TD_OPER_ALL</td>
<td>Enables all driver level tracing.</td>
</tr>
<tr>
<td>TD_OPER_CLI</td>
<td>Enables tracing for activities involving CLTv2.</td>
</tr>
<tr>
<td>TD_OPER_NOTIFY</td>
<td>Enables tracing for activities involving the Notify feature.</td>
</tr>
<tr>
<td>TD.Operator.OPCOMMON</td>
<td>Enables tracing for activities involving the operator common library.</td>
</tr>
</tbody>
</table>
Teradata PT API Status Messages

Table 10 contains all the possible status messages that can be returned by the Teradata PT API. The set of possible status messages returned by a specific Teradata PT API function varies. See the Return Code column in Table 2, Table 3, and Table 4 for the possible status messages returned by each function.

<table>
<thead>
<tr>
<th>Message</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLI Error</td>
<td>A CLIv2 error has occurred. Values range from 300 to 500.</td>
</tr>
<tr>
<td>DBS Error</td>
<td>A Teradata Database error has occurred. Values range from 2,000 to 8,000.</td>
</tr>
<tr>
<td>TPTAPI Error</td>
<td>A Teradata PT API specific error has occurred. Values range from 10000 and up.</td>
</tr>
<tr>
<td>TD_Call_EndAcq</td>
<td>The Teradata PT API driver is signaling to the user application that the acquisition method needs to be aborted due to an error. The application must immediately call the EndAcquisition function. In a multiple instance environment, if the master or one or more slave instances receives this status code, all instances must immediately proceed to call the EndAcquisition function.</td>
</tr>
</tbody>
</table>
| TD_END_Method         | The calling application may move on to the next method. In export jobs, this status also signifies end of data when returned by the GetRow or GetBuffer functions. In a multiple instance environment, the following must be done: 
  • Each instance waits until all instances have returned the end method code. 
  • Each instance enters the next method. Refer to the Parallel Processing Return Codes section in Chapter 7 for more information. |
| TD_Error              | An error has occurred. When an error occurs, the return value is equal to or greater than TD_Error. TD_Error is a generic error code. See error types included in this table for specific numeric error code ranges. |
| TD_Success            | The function has completed successfully. |
### TD_SYNC_Barrier

- Used in a multiple instance environment.
- Signifies that the following must be done:
  - Each instance waits until all instances return the barrier code.
  - Each instance then recalls the method that returned the barrier code.
- Refer to the Parallel Processing Return Codes section in Chapter 7 for more information.

### TD_SYNC_TELINFO

- Used in a multiple instance environment.
- Signifies that the following must be done:
  - Each instance waits until all instances return the barrier code.
  - Get the TELINFO area from the master instance using the GetTELINFO function.
  - Pass a copy of the master’s TELINFO area to each of the slave instances.
  - Store the master’s TELINFO area within each slave instance using the PutTELINFO function.
  - Each instance then recalls the method that returned the barrier code.
- Refer to the Parallel Processing Return Codes section in Chapter 7 for more information.

### TD_Unavailable

- The data for an event is currently unavailable.
The Load driver uses Teradata FastLoad Reference protocols to load a large volume of data at high speed into an empty table on the Teradata Database. Besides being empty, the target table on the Teradata Database can not have defined secondary indexes. Main topics on the Load driver include:

- Attribute Definitions
- Required Attributes
- Optional Attributes
- GetEvent Queries
- Programming Considerations
- Error Tables and Error Reporting
- Limiting Insertion Errors
- Duplicate Records
- Reusing Table Names
- Dropping Tables During a Load
- Required Privileges
- Session Limits
- Space Requirements and Limitations
- Checkpoint and Restart Operations
- Code Example

**Attribute Definitions**

Table 11 and Table 12 define the required and optional attributes needed to code an application.

**Required Attributes**

Table 11: Load Driver Required Attributes

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_BUFFER_MODE</td>
<td>varchar</td>
<td>Indicates which type of Load method is used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Valid Settings:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Default value is No.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Must be set to Yes when using the PutBuffer feature.</td>
</tr>
</tbody>
</table>
Chapter 3: Load Driver
Attribute Definitions

Table 11: Load Driver Required Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_INSTANCE_NUM</td>
<td>integer</td>
<td>Provides the instance number of the current instance. Required only when using multiple instances of the same driver in a master-slave environment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the current instance is the master instance, then the instance number is one.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the current instance is a slave instance, then the instance number should be a value greater than one.</td>
</tr>
<tr>
<td>TD_LOG_TABLE</td>
<td>varchar</td>
<td>Provides the name of the restart log table for restart information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> If the restart log table name is not fully qualified, it is created under the user’s default (logon) database. Alternately, a working database can be specified using the TD_WORKINGDATABASE attribute. If the TD_WORKINGDATABASE attribute is used, the restart log table name must be fully qualified, even if the restart log table is going to reside in the default (logon) database.</td>
</tr>
<tr>
<td>TD_MAX_INSTANCES</td>
<td>integer</td>
<td>Required only if using multiple instances of the same driver in a master-slave environment. Provides the total number of instances (master and slaves).</td>
</tr>
<tr>
<td>TD_RESTARTMODE</td>
<td>integer</td>
<td>Required only before restarting. Must be set to 1 before performing a restart.</td>
</tr>
<tr>
<td>TD_SYSTEM_OPERATOR</td>
<td>varchar</td>
<td>Provides the type of driver being used (in this case TD_LOAD).</td>
</tr>
<tr>
<td>TD_TARGET_TABLE</td>
<td>varchar</td>
<td>Provides the name of the Load target table.</td>
</tr>
<tr>
<td>TD_USER_NAME</td>
<td>varchar</td>
<td>Provides the name of the user for the Load driver logon sessions.</td>
</tr>
<tr>
<td>TD_USER_PASSWORD</td>
<td>varchar</td>
<td>Provides the password associated with the user name.</td>
</tr>
</tbody>
</table>

Optional Attributes

Table 12: Load Driver Optional Attributes

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_ACCOUNT_ID</td>
<td>varchar</td>
<td>Specifies the account associated with the specified user name. When omitted, this attribute defaults to the account identifier of the immediate owner database.</td>
</tr>
</tbody>
</table>
Chapter 3: Load Driver

Attribute Definitions

### TD_BUFFER_SIZE

- **Type:** integer
- **Description:** Specifies the output buffer size, in kilobytes, used for sending Load parcels to the Teradata Database.
  - The output buffer size and the size of the rows in the Load table determine the maximum number of rows included in each parcel to the Teradata Database. A larger buffer size reduces processing overhead by including more data in each parcel.
  - The allowable values are 1 through 64. However, if you specify a value of 64, the actual buffer size is set to 64260.
  - The default buffer size is the maximum size allowed, which depends on the Teradata Database and CLI version. The maximum buffer size on V2R6.0 and later is 64K bytes.
  - If you specify a value less than one, the Load driver issues an error message and terminates the job. Any other value specified is evaluated when the connection to the Teradata Database is made. Because some Teradata Database versions support buffer sizes of 32K only, specifying a value of 64K would be invalid, but the driver does not know this until it connects to the Teradata Database and queries its version.
  - If the supplied buffer size is too large, the Load driver scales it back to the maximum allowable size.

### TD_CHARSET

- **Type:** varchar
- **Description:** Character Set is the name of the session character set used for the job. On channel-attached z/OS platforms, only EBCDIC encoding is supported and is automatically selected. For the list of supported session character sets, see the Extended Character Sets section in the Teradata Parallel Transporter Reference. Check [http://www.info.teradata.com/](http://www.info.teradata.com/) for the latest version of this publication.

In a multi-instance environment the master and all the slave instances should have the same session character set. Also, the data for each instance should be in the same character set as the session character set. If the master instances does not use the TD_CHARSET attribute then the slave instance(s) should not use the attribute. If the master instance specifies a different session character set than any of its slave instances then the Load, Export, and Update drivers will use the master instance’s session character set for the entire job. The Stream driver, however, will let each instance use its own session character set for the job.

Also, note that the session character set for all instances should not change if a restart occurs. If any instance specified a value for TD_CHARSET before the restart then that instance needs to specify the same value for TD_CHARSET when the connection is initiated again and then restarted.

### TD_DATA_ENCRYPTION

- **Type:** varchar
- **Description:** Makes full security encryption of SQL requests, responses, and data available.
  - **Off** = No encryption occurs. This is the default setting.
  - **On** = All SQL requests, responses, and data are encrypted.

---

Table 12: Load Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TD_BUFFER_SIZE             | integer | Specifies the output buffer size, in kilobytes, used for sending Load parcels to the Teradata Database.  
  - The output buffer size and the size of the rows in the Load table determine the maximum number of rows included in each parcel to the Teradata Database. A larger buffer size reduces processing overhead by including more data in each parcel.  
  - The allowable values are 1 through 64. However, if you specify a value of 64, the actual buffer size is set to 64260.  
  - The default buffer size is the maximum size allowed, which depends on the Teradata Database and CLI version. The maximum buffer size on V2R6.0 and later is 64K bytes.  
  - If you specify a value less than one, the Load driver issues an error message and terminates the job. Any other value specified is evaluated when the connection to the Teradata Database is made. Because some Teradata Database versions support buffer sizes of 32K only, specifying a value of 64K would be invalid, but the driver does not know this until it connects to the Teradata Database and queries its version.  
  - If the supplied buffer size is too large, the Load driver scales it back to the maximum allowable size. |
| TD_CHARSET                 | varchar | Character Set is the name of the session character set used for the job. On channel-attached z/OS platforms, only EBCDIC encoding is supported and is automatically selected. For the list of supported session character sets, see the Extended Character Sets section in the Teradata Parallel Transporter Reference. Check [http://www.info.teradata.com/](http://www.info.teradata.com/) for the latest version of this publication.  
  In a multi-instance environment the master and all the slave instances should have the same session character set. Also, the data for each instance should be in the same character set as the session character set. If the master instances does not use the TD_CHARSET attribute then the slave instance(s) should not use the attribute. If the master instance specifies a different session character set than any of its slave instances then the Load, Export, and Update drivers will use the master instance’s session character set for the entire job. The Stream driver, however, will let each instance use its own session character set for the job.  
  Also, note that the session character set for all instances should not change if a restart occurs. If any instance specified a value for TD_CHARSET before the restart then that instance needs to specify the same value for TD_CHARSET when the connection is initiated again and then restarted. |
| TD_DATA_ENCRYPTION         | varchar | Makes full security encryption of SQL requests, responses, and data available.  
  - **Off** = No encryption occurs. This is the default setting.  
  - **On** = All SQL requests, responses, and data are encrypted. |
### TD_DATE_FORM
- **Type**: VARCHAR
- **Description**: Specifies the DATE data type for the Load driver job.
  - *IntegerDate* = Integer DATE data type. This is the default setting.
  - *AnsiDate* = ANSI fixed-length CHAR(10) DATE data type.

### TD_DROPERRORTABLE
- **Type**: VARCHAR
- **Description**: Directs the Load driver to drop the existing error tables at the end of the job. By default, the Load driver drops the error tables at the end of a job if the error tables are empty.
  
  If the error tables are not dropped at the end of a successfully terminating job and the same error table names are used in a subsequent Load job then the Teradata Database will return an error on those subsequent Load jobs, even if those error tables are empty.

  Valid values are:
  - *Yes* (‘Y’) = Drop the error table if it is empty at the end of the job. This is the default setting.
  - *No* (‘N’) = Do not drop the existing error table.

### TD_DROPLOGTABLE
- **Type**: VARCHAR
- **Description**: Directs the Load driver to drop the existing restart log table at the end of the job. By default, the Load driver drops the restart log table at the end of a job only if the job completes successfully.

  If the restart log table is not dropped at the completion of a successful job and the same restart log table name is provided in a subsequent Load job then the results will be unpredictable. This unpredictability is due to the nature of the Teradata FastLoad protocol, where the existence of a restart log table implies the job is a restart and the Load driver may attempt to restart the job at a point in time as dictated by the contents of the restart log table.

  The Load driver will try to detect whether this situation has occurred and will attempt to terminate the job with a meaningful error message but this attempt is dependent upon the contents of the restart log table.

  Valid values are:
  - *Yes* (‘Y’) = Drop the restart log table if the job completed successfully. This is the default setting.
  - *No* (‘N’) = Do not drop the existing restart log table.

### TD_ERROR_LIMIT
- **Type**: INTEGER
- **Description**: Specifies the maximum number of records stored in one of the error tables before the Load driver job is terminated. The ErrorLimit specification applies to each instance of the Load driver.

  - Specifying an invalid value causes the Load driver to terminate. By default, ErrorLimit value is unlimited.
  - The ErrorLimit specification must be greater than zero.
### Table 12: Load Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| **TD_ERROR_TABLE_1**          | varchar  | Specifies the name of the first error table. This must be a new table name. You cannot use a name that duplicates the name of an existing table unless you are restarting a paused Load driver job.  
  - ErrorTable1 is for containing records rejected during the acquisition phase of the Load driver job because of:  
    - Data conversion errors  
    - Constraint violations  
    - AMP configuration changes  
  - The default name for ErrorTable1 is `ttname_ET`.  
  For more information on the error table format and the procedure to correct errors, refer to the Teradata FastLoad Reference's Handling FastLoad Errors chapter. |
| **TD_ERROR_TABLE_2**          | varchar  | Specifies the name of the second error table. This must be a new table name. You cannot use a name that duplicates the name of an existing table unless you are restarting a paused Load driver job.  
  - ErrorTable2 contains records that violated the unique primary index constraint.  
  - These types of errors occur during the acquisition phase of the Load driver job.  
  - The default name for ErrorTable2 is `ttname_UV`.  
  For more information on the error table format and the procedure to correct errors, refer to the Teradata FastLoad Reference's Handling FastLoad Errors chapter. |
| **TD_LOGSQL**                 | varchar  | Directs the Load driver to output the full Teradata SQL request in the trace output file with the driver’s trace is enabled. By default, when the driver’s trace is enabled, the Load driver outputs the Teradata SQL request, up to 32 kilobytes, in the trace output file.  
  Valid values:  
  - Yes (‘Y’) = Output the full Teradata SQL in the trace output file when the driver’s trace is enabled. The maximum length of the Teradata SQL is 1 megabyte.  
  - No (‘N’) = Do not output the Teradata SQL in the trace output file.  
  - **Note:** When the driver’s trace is disabled, TD_LOGSQL has no effect. |
| **TD_LOGON_MECH**             | varchar  | Specifies which logon mechanism to use.  
  - See your site security administrator for specific mechanism names.  
  - For a list of available mechanisms, see Security Administration.  
  - The job terminates if the attribute exceeds eight bytes. |
| **TD_LOGON_MECH_DATA**        | varchar  | Passes additional logon mechanism data.  
  See your site security administrator for specific mechanism data. For more information, see Security Administration. |
Chapter 3: Load Driver
Attribute Definitions

Table 12: Load Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_MAX_SESSIONS</td>
<td>integer</td>
<td>Specifies the maximum number of sessions to log on.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The default is one session per available AMP. The maximum value cannot be more than the number of AMPS available.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The MaxSessions value must be greater than one. Specifying a value less than one terminates the job.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The MaxSessions value must be greater than or equal to the value of TD_MAX_INSTANCES.</td>
</tr>
<tr>
<td>TD_MIN_SESSIONS</td>
<td>integer</td>
<td>Specifies the minimum number of sessions required for the Load driver job to continue.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The default is one session.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The MinSessions value must be greater than zero and less than or equal to the maximum number of Load driver sessions. Specifying a value less than one terminates the Load driver.</td>
</tr>
<tr>
<td>TD_MSG_ENCODING</td>
<td>TD_Encoding</td>
<td>Specifies the encoding for the messages passed between Teradata PT API and a Teradata PT API application.</td>
</tr>
<tr>
<td>TD_NOTIFY_EXIT</td>
<td>varchar</td>
<td>Specifies the name of the user-defined notify exit routine with an entry point named _dynann.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If no name is supplied, the following default names are used:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• libnotfyext.dll for Windows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• libnotfyext.sl for HP-UX platforms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• libnotfyext.so for Linux and all other UNIX platforms</td>
</tr>
<tr>
<td>TD_NOTIFY_LEVEL</td>
<td>varchar</td>
<td>Indicates the level at which certain events are reported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The valid settings are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Off = No notification of events is provided. This is the default setting.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Low = Notification is provided for these events:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Initialize</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CLIv2/DBS error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Exit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Medium = Notification is provided for all the events except for Error Table 1 and Error Table 2.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• High = Notification is provided for all events.</td>
</tr>
</tbody>
</table>
### TD_NOTIFY_METHOD
- **Type**: `varchar`
- **Description**: Specifies the method for reporting events.
  - `None`: No event logging is done. This is the default method.
  - `Msg`: This method sends the events to a log.
  - `Exit`: This method sends the events to a user-defined notify exit routine.

  On Windows, the events are sent to the EventLog that can be viewed using the Event Viewer. The messages are sent to the application log.

  On AIX, HP-UX, Linux, and Solaris platforms, the destination of the events is specified in the `/etc/syslog.conf` file.

### TD_NOTIFY_STRING
- **Type**: `varchar`
- **Description**: Provides a user-defined string that precedes all messages sent to the system log. This string is also sent to the user-defined notify exit routine.

  The maximum length of the string is:
  - 80 bytes, if the NotifyMethod is `Exit`.
  - 16 bytes, if the NotifyMethod is `Msg`.

### TD_PAUSE_ACQ
- **Type**: `varchar`
- **Description**: Specifies whether to pause the Load job after the acquisition phase or enter the application phase.

  Valid values are:
  - `No` (or `N`) for normal Load driver jobs, to distribute all of the rows sent to the Teradata Database during the acquisition phase to their final destination on the AMPs. This is the default value.
  - `Yes` (or `Y`) to pause after the completion of the acquisition phase and skip the application phase. Specifying any other value terminates the job.

  The absence of any value means that the Load driver job executes both the acquisition phase and the application phase without pausing. This distributes all of the rows sent to the Teradata Database during the acquisition phase to their final destination on the AMPs.
### Chapter 3: Load Driver

**Attribute Definitions**

**TD_QUERY_BAND_SESS_INFO**
- **Type:** varchar
- **Description:** Provides a user-defined query band expression that is set for every SQL session connected by the Teradata PT API driver. The following is an example of a valid query band expression:

  \[
  \text{a}=1; \text{b}=2; \text{c}=3; \text{d}=4;
  \]

  If the TD_QUERY_BAND_SESS_INFO is set, the following request will be sent by every SQL session connected by the Teradata PT API Load driver:

  ```sql
  SET QUERY_BAND = '<User-Defined Query Band Expression>' FOR SESSION;
  ```

  Setting the TD_QUERY_BAND_SESS_INFO attribute in jobs running against non-supported versions of the Teradata Database causes a non-fatal error. No error code is returned to the user during initiation and the job is allowed to proceed. The log table will not be dropped at the end of the job and the TD_EVT_ExitCode event returns a warning value of four instead of the normal success value of zero if queried. In this case, error information can be found in the trace file.

**TD_TDP_ID**
- **Type:** varchar
- **Description:** Specifies the name of the Teradata Database machine.
  - The `dbname` can be up to 256 characters and can be a domain server name.
  - TDP stands for Teradata Director Program and is specified for mainframe z/OS platforms.
  - If you do not specify the value for the TdpId attribute, the driver uses the default TdpId established for the user by the system administrator.

**TD_TENACITY_HOURS**
- **Type:** integer
- **Description:** Specifies the number of hours that the Load driver continues trying to log on when the maximum number of Load and export operations are already running on the Teradata Database.
  - The default value is four hours. To make the tenacity feature available, the `hours` value must be greater than zero.
  - Specifying a value of zero will disable the tenacity feature.
  - Specifying a value of less than zero terminates the Load driver.

**TD_TENACITY_SLEEPS**
- **Type:** integer
- **Description:** Specifies the number of minutes the Load driver pauses before retrying to log on when the maximum number of Load or Export operations are already running on the Teradata Database.
  - The minutes value must be greater than zero. If you specify a value less than one, the Load driver responds with an error message and terminates the job.
  - The default is six minutes.

---

**Table 12: Load Driver Optional Attributes (continued)**

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TD_QUERY_BAND_SESS_INFO | varchar | Provides a user-defined query band expression that is set for every SQL session connected by the Teradata PT API driver. The following is an example of a valid query band expression:  

\[
\text{a}=1; \text{b}=2; \text{c}=3; \text{d}=4;
\]

If the TD_QUERY_BAND_SESS_INFO is set, the following request will be sent by every SQL session connected by the Teradata PT API Load driver:

```sql
SET QUERY_BAND = '<User-Defined Query Band Expression>' FOR SESSION;
```  
Setting the TD_QUERY_BAND_SESS_INFO attribute in jobs running against non-supported versions of the Teradata Database causes a non-fatal error. No error code is returned to the user during initiation and the job is allowed to proceed. The log table will not be dropped at the end of the job and the TD_EVT_ExitCode event returns a warning value of four instead of the normal success value of zero if queried. In this case, error information can be found in the trace file. |
| TD_TDP_ID | varchar | Specifies the name of the Teradata Database machine.  
- The `dbname` can be up to 256 characters and can be a domain server name.  
- TDP stands for Teradata Director Program and is specified for mainframe z/OS platforms.  
- If you do not specify the value for the TdpId attribute, the driver uses the default TdpId established for the user by the system administrator. |
| TD_TENACITY_HOURS | integer | Specifies the number of hours that the Load driver continues trying to log on when the maximum number of Load and export operations are already running on the Teradata Database.  
- The default value is four hours. To make the tenacity feature available, the `hours` value must be greater than zero.  
- Specifying a value of zero will disable the tenacity feature.  
- Specifying a value of less than zero terminates the Load driver. |
| TD_TENACITY_SLEEPS | integer | Specifies the number of minutes the Load driver pauses before retrying to log on when the maximum number of Load or Export operations are already running on the Teradata Database.  
- The minutes value must be greater than zero. If you specify a value less than one, the Load driver responds with an error message and terminates the job.  
- The default is six minutes. |
Chapter 3: Load Driver

Attribute Definitions

Table 12: Load Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_TRACE_LEVEL</td>
<td>integer</td>
<td>Specifies the types of diagnostic messages written by each instance of the driver to an external log file. The diagnostic trace function provides</td>
</tr>
<tr>
<td></td>
<td></td>
<td>more detailed information in the log file (including the version number) to aid in problem tracking and diagnosis. Use the AddArray attribute method to specify the two types of tracing levels: driver tracing and infrastructure tracing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TD_OFF is the default setting for both driver tracing and infrastructure tracing. No external log file is produced unless this default is changed. Specifying TD_OFF for both driver tracing and infrastructure tracing is the same as disabling tracing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the TraceLevel is set to any value other than TD_OFF, an external log file is created for each instance of the driver. The trace levels for driver tracing are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TD_OFF = Disables driver tracing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TD_OPER = Activates the tracing function for driver specific activities. The absence of any value for the PauseAcq attribute means that the Load driver job will execute both the acquisition phase and the application phase without pausing. This will distribute all of the rows that were sent to the Teradata Database during the acquisition phase to their final destination on the AMPs. Table 1 on page 40 lists which drivers have the Pause Acquisition attribute.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TD_OPER_CLI = Activates the tracing function for CLIv2-related activities (interaction with the Teradata Database).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TD_OPER_NOTIFY = Activates the tracing function for activities related to the Notify feature.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TD_OPER_OPCOMMON = Activates the tracing function for activities involving the opcommon library.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TD_OPER_ALL = Activates tracing for all of the above activities. The trace levels for infrastructure tracing should only be used when you are directed to by Teradata support. TD_OFF, which disables infrastructure tracing, should always be specified.</td>
</tr>
<tr>
<td>TD_TRACE_OUTPUT</td>
<td>varchar</td>
<td>Specifies the name of the external file used for trace messages. The default setting creates a new file name using the name of the driver followed by a time stamp. Note: If a file with the specified name already exists, then the file is overwritten.</td>
</tr>
</tbody>
</table>
All events must be queried after the driver is initiated and before it terminates. Events queried before their data is available return TD_Unavailable. The following table lists events used with the Connection object’s GetEvent function to retrieve runtime statistics from the Load driver.

Table 13: Load Driver Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_Evt_CLIError</td>
<td>One 4-byte integer for the CLIv2 error number and a 255-byte character buffer for the CLIv2 error text. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_DBSError</td>
<td>One 4-byte integer for the DBS error number and a 255-character buffer for the Teradata Database error text. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_BufferLayout</td>
<td>Four 4-byte unsigned integers corresponding to the maximum buffer size, the row header size, the row length size, and the buffer trailer size. Use this information to format the buffer required for block loading. <strong>Note:</strong> These layout values will change depending on the user environment and may be defined differently in future releases. Always obtain these values from the event method before buffering data. Query this event at any time.</td>
</tr>
</tbody>
</table>
Table 13: Load Driver Events (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_Evt_ConnectStatus</td>
<td>Three 4-byte integers corresponding to the number of sessions requested, the number of sessions connected, and the number for any CLIv2 error number that occurred during the connect process. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_CPUTime</td>
<td>One 8-byte double for the CPU time of the instance in seconds. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_RowCounts</td>
<td>Three 4-byte unsigned integers corresponding to the number of rows received, sent, and applied. This event can be queried any time before EndAcquisition to get the current number of rows received. The rest of the data for this event is available after ApplyRows. If rows are being loaded using PutRow, the data for this event will not be available until enough rows have been loaded to fill one internal buffer or until the Checkpoint function has been called. The rows-applied count returned by this event does not take into consideration rows that are rejected by the database during the application phase. Query the TD_Evt_ApplyCount event to get the final count of rows that have been applied to the target table.</td>
</tr>
<tr>
<td>TD_Evt_RowsCheckpointed</td>
<td>One 4-byte unsigned integer corresponding to the total number of rows that have been received and checkpointed for the entire job. Query this event any time after initiating a Teradata PT API database connection in a single-instance job. This event will return TD_Unavailable when queried in a job containing multiple instances; this event is invalid if multiple instances are used. Note that the event data returned by this event can be used as a replacement for the checkpoint data that must be passed into the Restart method during a restart job. It is recommended to always save the checkpoint data from the last successful call to the CheckPoint method in a potential restart job. However, if the checkpoint data is not saved then the event data returned by this event can be passed into the Restart method instead. This alternate method of performing checkpoint/restarts will only work with single-instance jobs.</td>
</tr>
<tr>
<td>TD_Evt_ApplyCount</td>
<td>One 4-byte unsigned integer for the number of rows inserted into the DBS. Query this event after ApplyRows. For a multi-instance job, only query this number in the master instance. The master instance returns the total number of rows applied to the target table from all instances.</td>
</tr>
</tbody>
</table>
Chapter 3: Load Driver
Programming Considerations

The following sections describe the items you should consider when coding a Load application.

Error Tables and Error Reporting

Duplicate Records
The Teradata Database ignores duplicate records and they are not inserted in either error table.

Reusing Table Names
Error tables with one or more rows are not dropped from the Teradata Database at the end of a Load driver job. To reuse the names specified for the error tables, use the DROP TABLE statement from BTEQ to remove the tables from the Teradata Database.

Limiting Insertion Errors
Use the TD_ERROR_LIMIT attribute to limit the number of insertion errors captured in ErrorTable1 during the acquisition phase of a Load job.

<table>
<thead>
<tr>
<th>Event</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_Evt_ErrorTable1</td>
<td>One 4-byte unsigned integer for the number of rows in error table 1. Query this event after the first call to PutRow or PutBuffer.</td>
</tr>
<tr>
<td>TD_Evt_ErrorTable2</td>
<td>One 4-byte unsigned integer for the number of rows in error table 2. Query this event after ApplyRows.</td>
</tr>
<tr>
<td>TD_Evt_ExitCode</td>
<td>One 2-byte integer for the driver exit code. Query this event right before the driver terminates.</td>
</tr>
<tr>
<td>TD_Evt_Version</td>
<td>A pointer to a character string containing the Teradata PT API version followed by a pointer to a character string containing the operator version. Query this event at any time. The Teradata PT API version is available any time. The operator version is available only after the driver has been initiated.</td>
</tr>
</tbody>
</table>

Table 13: Load Driver Events (continued)

<table>
<thead>
<tr>
<th>Load Driver Job Quality</th>
<th>ERRORLIMIT Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No errors or very few errors</td>
<td>low</td>
</tr>
<tr>
<td>Many errors that are considered allowable</td>
<td>high</td>
</tr>
</tbody>
</table>

Table 14: TD_ERRORLIMIT Values
The ErrorLimit specification applies to each instance of the Load driver, not to all instances combined. For example, you set the limit to 10,000 rows. A single instance must detect that 10,000 rows were inserted into the first error table to terminate the job, and those 10,000 rows must be controlled by the sessions managed by that instance.

**Dropping Tables During a Load**

Some tables are created during the execution of a Load job, and others must be created before the job begins. Target tables must exist and be empty when a Load driver job executes, unless the job is attempting to continue a paused or restarted job. A log table is created automatically when you run the Load job script. Error tables are created by the Teradata Database. Error tables are dropped by the Load driver during the cleanup phase if no errors were detected during the acquisition phase or the application phase. The log table is dropped by the Load driver after the job completes successfully. If a Load job terminates abnormally, then the log and error tables are not dropped. If you want to restart the Load job from scratch, you need to manually drop these tables.

Processing terminates when the number of errors encountered reaches the error limit.

If, for example, you expect no errors in the input data, set the error limit value to one. In this case, the job terminates when any record causes an error. Note, however, that when the specified error limit is reached, the Load driver continues processing until each session completes its current data block. This continued processing can cause the total number of error rows captured in the first error table to exceed the ErrorLimit specification.

**Required Privileges**

The user ID used by a Load application must have:

- SELECT and INSERT privileges on the Load target table
- SELECT and INSERT privileges on the error tables, and DROP privileges on the database containing the error tables.
- SELECT, INSERT, and DELETE privileges on the restart log table, and DROP privileges on the database containing the restart log table.

**Session Limits**

The values you specify using the TD_MIN_SESSIONS and TD_MAX_SESSIONS attributes are not the only factors limiting the number of sessions the Load driver establishes with the Teradata Database. When the Load driver executes, the actual session limit is determined by whichever limiting factor is encountered first.

The other limiting factors are:

- The Teradata Database limit of one session per AMP.
- The platform limit on the maximum number of sessions per application. This value is defined in the COP Interface software file, CLISPB.DAT, under the max_num_sess variable. You can use the TDP SET MAXSESSIONS command to specify a platform limit. The default TDP MAXSESS value is 1024 sessions.
The limit of the network protocol software on network-attached systems.

**Space Requirements and Limitations**

Always estimate the final size of the Load target table, and make sure that the destination database on the Teradata Database has enough space to accommodate the Load job. If the destination database runs out of space, the Teradata Database returns an error message and the Load driver pauses the Load job. When this happens, you must allocate more space to the database before you can restart the job.

**Checkpoint and Restart Operations**

The Load driver is fully checkpoint restartable. Refer to the checkpoint and restart discussion in “Add Checkpoint and Restart” on page 33.

**Code Example**

```cpp
#include "connection.h"
#include "schema.h"
#include "DMLGroup.h"

using namespace teradata::client::API;

int returnValue = 0;
char* errorMsg = NULL;
TD_ErrorType errorType;

cout << "*** Load Driver Example ***" << endl;
Connection *conn = new Connection();
	conexao->AddAttribute(TD_SYSTEM_OPERATOR,TD_LOAD);
	conn->AddAttribute(TD_TRACE_OUTPUT,"load.txt");
	conn->AddArrayAttribute(TD_TRACE_LEVEL,2,TD_OPER,TD_OFF,NULL);

* Set Operator Type and Trace/Log Levels

* Add Attributes

* Add Schema

* Add DMLGroups

TD_Index dmlGroupIndex = 0;
DMLGroup* dmlGr = new DMLGroup();
DMLGr->AddStatement("INSERT INTO tdload (Associate_Id, Associate_Name, Salary, DOJ, Designation, Loan_Amount, Martial_Status, No_Of_Dependents);");
conn->AddDMLGroup(dmlGr, &dmlGroupIndex);
```

74 Teradata Parallel Transporter Application Programming Interface Programmer Guide
* Initiate
*******************************************************************************/
returnValue = conn->Initiate();

cout << "Driver Initiated with status " << returnValue << endl;
if ( returnValue < TD_Error )
{
    /******************************************************************************/
    * Acquisition
    *******************************************************************************/

    char rowBuffer[78]; // must include the EOL byte */
    int loadStatus = 0;
    TD_Length bytes;
    int count = 0;

    while( loadStatus != -1 )
    {
        // user function - reads in a
        // row of data from a file
        // returns -1 when end of file
        // reached
        loadStatus = GetRowData(rowBuffer, 78);
        if( loadStatus != -1 )
        {
            /* pick up the first 2 bytes as the row length of the indicator mode record */
            bytes = *((unsigned short *)rowBuffer);
            returnValue = conn->PutRow( rowBuffer + 2, bytes);
            if ( returnValue >= TD_Error )
            {
                cout << "PutRow failed on row " << count+1;
                cout << " with status " << returnValue << endl;
                loadStatus = GetRowData(NULL, 0); //user function - closes file
            }else{
                count++;
            }
        }
    }

    if ( returnValue < TD_Error )
    {
        cout << "Sent " << count << " rows" << endl;
    }
    /******************************************************************************/
    * End Acquisition
    *******************************************************************************/

    returnValue = conn->EndAcquisition();
    cout << "Acquisition completed with status " << returnValue << endl;
    if ( returnValue < TD_Error )
    {
        /******************************************************************************/
        * Application
        *******************************************************************************/

        returnValue = conn->ApplyRows();
        cout << "Rows Applied with status " << returnValue << endl;
        if ( returnValue < TD_Error )
        {
            cout << "Load completed successfully" << endl;
        }else{
            //Get Error Information
            cout << "Error occurred during Application" << endl;
            conn->GetErrorInfo(&errorMsg, &errorType);
            if ( errorMsg != NULL )
            {
                cout << errorMsg << endl;
                cout << "Type: " << errorType << endl;
            }else{
                cout << "No Error Info Available" << endl;
            }
        }
    }else{
        //Get Error Information
        cout << "Error occurred during Application" << endl;
        conn->GetErrorInfo(&errorMsg, &errorType);
        if ( errorMsg != NULL )
        {
            cout << errorMsg << endl;
            cout << "Type: " << errorType << endl;
        }else{
            cout << "No Error Info Available" << endl;
        }
    }
}
}]
} else{
  //Get Error Information
  cout << "Error occurred during EndAcquisition" << endl;
  conn->GetErrorInfo(&errorMsg,&errorType);
  if ( errorMsg != NULL ){
    cout << errorMsg << endl;
    cout << "Type: " << errorType << endl;
  } else{
    cout << "No Error Info Available" << endl;
  }
}
} else{
  //Get Error Information
  cout << "Error occurred during Acquisition" << endl;
  conn->GetErrorInfo(&errorMsg,&errorType);
  if ( errorMsg != NULL ){
    cout << errorMsg << endl;
    cout << "Type: " << errorType << endl;
  } else{
    cout << "No Error Info Available" << endl;
  }
}
} else{
  //Get Error Information
  cout << "Error occurred during Initiate" << endl;
  conn->GetErrorInfo(&errorMsg,&errorType);
  if ( errorMsg != NULL ){
    cout << errorMsg << endl;
    cout << "Type: " << errorType << endl;
  } else{
    cout << "No Error Info Available" << endl;
  }
}

/**********************************************
* Terminate
***********************************************/
returnValue = conn->Terminate();

cout << "Driver Terminated with status " << returnValue << endl;
if ( returnValue >= TD_Error ){
  //Get Error Information
  cout << "Error occurred during Terminate" << endl;
  conn->GetErrorInfo(&errorMsg,&errorType);
  if ( errorMsg != NULL ){
    cout << errorMsg << endl;
    cout << "Type: " << errorType << endl;
  } else{
    cout << "No Error Info Available" << endl;
  }
} else{
  //Get Error Information
  cout << "Error occurred during Initiate" << endl;
  conn->GetErrorInfo(&errorMsg,&errorType);
  if ( errorMsg != NULL ){
    cout << errorMsg << endl;
    cout << "Type: " << errorType << endl;
  } else{
    cout << "No Error Info Available" << endl;
  }
} else{
  //Get Error Information
  cout << "Error occurred during Acquisition" << endl;
  conn->GetErrorInfo(&errorMsg,&errorType);
  if ( errorMsg != NULL ){
    cout << errorMsg << endl;
    cout << "Type: " << errorType << endl;
  } else{
    cout << "No Error Info Available" << endl;
  }
} else{
  //Get Error Information
  cout << "Error occurred during EndAcquisition" << endl;
  conn->GetErrorInfo(&errorMsg,&errorType);
  if ( errorMsg != NULL ){
    cout << errorMsg << endl;
    cout << "Type: " << errorType << endl;
  } else{
    cout << "No Error Info Available" << endl;
  }
} else{
  //Get Error Information
  cout << "Error occurred during Acquisition" << endl;
  conn->GetErrorInfo(&errorMsg,&errorType);
  if ( errorMsg != NULL ){
    cout << errorMsg << endl;
    cout << "Type: " << errorType << endl;
  } else{
    cout << "No Error Info Available" << endl;
  }
} else{
  //Get Error Information
  cout << "Error occurred during Initiate" << endl;
  conn->GetErrorInfo(&errorMsg,&errorType);
  if ( errorMsg != NULL ){
    cout << errorMsg << endl;
    cout << "Type: " << errorType << endl;
  } else{
    cout << "No Error Info Available" << endl;
  }
}
/**********************************************
* Clean Up
***********************************************/
cout << "Deleting objects" << endl;
delete dmlGr;
delete schemas;
delete conn;
cout << "*** Load Complete ***" << endl;
CHAPTER 4

Update Driver

The Update driver uses Teradata MultiLoad Reference protocols to load a large volume of data at high speed on the Teradata Database, using multiple sessions to perform highly scalable and parallel inserts, updates, deletes, and upserts into up to five new or existing tables in a single pass. Main topics on the Update driver include:

- Attribute Definitions
- Required Attributes
- Optional Attributes
- GetEvent Queries
- Programming Considerations
- Error Tables
- Using DELETE in Import Tasks
- Using Delete Task
- Dropping Tables During a Load
- Required Privileges
- Session Limits
- Offline AMPs
- Nonparticipant AMPs
- Single-AMP Systems
- Space Requirements and Limitations
- Checkpoint and Restart Operations

Attribute Definitions

Table 15 defines the Update driver’s required attributes, and Table 16 the optional attributes. This information will help you code your application.

Required Attributes

Table 15: Update Driver Required Attributes

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_BUFFER_MODE</td>
<td>varchar</td>
<td>Indicates which type of Update method is used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Valid Settings:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Default value is No.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Must be set to Yes when using the block Update feature.</td>
</tr>
<tr>
<td>TD_INSTANCE_NUM</td>
<td>integer</td>
<td>Specifies the instance number of the current instance. Required only when using multiple instances of the same driver in a master-slave environment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the current instance is the master instance, then the instance number should be one.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the current instance is a slave instance, then the instance number should be a value greater than one.</td>
</tr>
</tbody>
</table>
Chapter 4: Update Driver
Attribute Definitions

### Optional Attributes

#### Table 15: Update Driver Required Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_LOG_TABLE</td>
<td>varchar</td>
<td>Provides the name of the restart log table for restart information.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> If the restart log table name is not fully qualified, it is created under the user’s default (logon) database. Alternately, a working database can be specified using the TD_WORKINGDATABASE attribute. If the TD_WORKINGDATABASE attribute is used, the restart log table name must be fully qualified, even if the restart log table is going to reside in the default (logon) database.</td>
</tr>
<tr>
<td>TD_MAX_INSTANCES</td>
<td>integer</td>
<td>Specifies the total number of instances (master and slaves). Required only when using multiple instances of the same driver in a master-slave environment.</td>
</tr>
<tr>
<td>TD_RESTARTMODE</td>
<td>integer</td>
<td>Required only before restarting and must be set to one.</td>
</tr>
<tr>
<td>TD_SYSTEM_OPERATOR</td>
<td>varchar</td>
<td>Specifies the type of driver being used (in this case TD_UPDATE).</td>
</tr>
<tr>
<td>TD_TARGET_TABLE</td>
<td>varchar</td>
<td>Provides the name of the update target table.</td>
</tr>
<tr>
<td>TD_USER_NAME</td>
<td>varchar</td>
<td>Provides the name of the user for the update driver logon sessions.</td>
</tr>
<tr>
<td>TD_USER_PASSWORD</td>
<td>varchar</td>
<td>Specifies the password associated with the user name.</td>
</tr>
</tbody>
</table>

#### Table 16: Update Driver Optional Attributes

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_ACCOUNT_ID</td>
<td>varchar</td>
<td>Specifies the account associated with the specified user name. When omitted, this attribute defaults to the account identifier of the immediate owner database.</td>
</tr>
<tr>
<td>TD_AMP_CHECK</td>
<td>varchar</td>
<td>Specifies the update driver response to an offline AMP condition. Valid settings are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <em>Apply</em> = Inhibits the Update driver job from entering or exiting the application phase when an AMP is offline. This is the default setting.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <em>All</em> = Pauses the Update driver job when an AMP is offline.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <em>None</em> = Allows the Update job to start, restart, or continue as long as no more than one AMP is offline in a cluster.</td>
</tr>
</tbody>
</table>
**Chapter 4: Update Driver**

**Attribute Definitions**

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| **TD_BUFFER_SIZE**       | integer| Specifies the output buffer size, in kilobytes, used for sending Update parcels to the Teradata Database.  
  - The output buffer size and the size of the rows in the Update table determine the maximum number of rows included in each parcel to the Teradata Database. A larger buffer size reduces processing overhead by including more data in each parcel.  
  - The allowable values are 1 through 64. However, if you specify a value of 64, the actual buffer size is set to 64260.  
  - The default buffer size is the maximum size allowed, which depends on the Teradata Database and CLI version. The maximum buffer size on V2R6.0 and later is 64K bytes.  
  - If you specify a value less than one, the Update driver issues an error message and terminates the job. Any other value specified is evaluated when the connection to the Teradata Database is made. Because some Teradata Database versions support buffer sizes of 32K only, specifying a value of 64K would be invalid, but the driver does not know this until it connects to the Teradata Database and queries its version.  
  - If the supplied buffer size is too large, the Update driver scales it back to the maximum allowable size. |
| **TD_CHARSET**           | varchar| Specifies the name or code of the character set to be used for the job. For the list of supported character sets, see the Extended Character Sets section in the Teradata Parallel Transporter Reference. Check [http://www.info.teradata.com](http://www.info.teradata.com) for the latest version of this publication. On channel-attached z/OS platforms, only EBCDIC encoding is supported and is automatically selected. |
| **TD_DATA_ENCRYPTION**   | varchar| Activates full security encryption of SQL requests, responses and data.  
  Valid values are:  
  - **Off** = No encryption occurs. This setting is the default.  
  - **On** = All SQL requests, responses, and data are encrypted. |
| **TD_DATE_FORM**         | varchar| Specifies the DATE data type for the Update driver job.  
  Valid settings are:  
  - **IntegerDate** = Integer DATE data type. This is the default setting.  
  - **ANSIDate** = ANSI fixed-length CHAR(10) DATE data type. |
Attribute Definitions

Table 16: Update Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TD_DELETE_TASK              | varchar  | Specifies whether to perform the delete task to delete data from a single Teradata Database table. The Delete Task removes rows much more quickly than a DELETE SQL statement. You cannot use a delete task on a view. Valid settings for option are:
  - Yes (or Y) = Perform the delete task.
  - No (or N) = Do not perform the delete task. Specifying any other value results in an error. The absence of any value is the same as a No value; the Update driver executes an IMPORT task, and none of the above rules apply. If the Delete Task attribute processing is enabled, other relevant optional attributes are:
    - TD_TENACITY_HOURS
    - TD_TENACITY_SLEEP
    - TD_AMP_CHECK |
| TD_DROPERRORTABLE           | varchar  | Directs the Update driver to drop the existing error tables at the end of the job. By default, the Update driver drops the error tables at the end of a job if the error tables are empty. If the error tables are not dropped at the end of a successfully-terminating job and the same error table names are used in a subsequent Update job then the Teradata Database will return an error on those subsequent Update jobs, even if those error tables are empty. Valid values are:
  - Yes (‘Y’) = Drop the error tables if they are empty at the end of the job. This is the default setting.
  - No (‘N’) = Do not drop the existing error tables. |
### TD_DROPLOGTABLE

**Type**: varchar

Directs the Update driver to drop the existing restart log table at the end of the job. By default, the Update driver drops the restart log table at the end of a job only if the job completes successfully.

If the restart log table is not dropped at the completion of a successful job and the same restart log table name is provided in a subsequent Update job then the results will be unpredictable. This unpredictability is due to the nature of the Teradata MultiLoad protocol, where the existence of a restart log table implies the job is a restart and the Update driver may attempt to restart the job at a point in time as dictated by the contents of the restart log table.

The Update driver will try to detect whether this situation has occurred and will attempt to terminate the job with a meaningful error message but this attempt is dependent upon the contents of the restart log table.

Valid values are:
- **Yes** ('Y') = Drop the restart log table if the job completed successfully. This is the default setting.
- **No** ('N') = Do not drop the existing restart log table.

### TD_DROPWORKTABLE

**Type**: varchar

Directs the Update driver to drop the existing work tables at the end of the job. By default, the Update driver drops the work tables at the end of a job if the job completed successfully.

If the work tables are not dropped at the end of a successfully-terminating job and the same work table names are used in a subsequent Update job then the Teradata Database will return an error on those subsequent Update jobs.

Valid values are:
- **Yes** ('Y') = Drop the work tables if the job completes successfully. This is the default setting.
- **No** ('N') = Do not drop the existing work tables.

### TD_ERROR_LIMIT

**Type**: integer

Specifies the maximum number of records that can be stored in an error table before the Update driver job is terminated. The ErrorLimit specification applies to each instance of the Update driver.

The ErrorLimit specification must be greater than zero. Specifying an invalid value terminates the Update driver. By default, ErrorLimit value is unlimited.
### Chapter 4: Update Driver

## Attribute Definitions

### TD_ERROR_TABLE_1
- **Type**: varchar
- **Description**: Specifies the name of the first error table. This must be a new table name. You cannot use the name of an existing table unless you are restarting a paused Update driver job.

ErrorTable1 contains records that were rejected during the acquisition phase of the Update driver job because of:
- Data conversion errors
- Constraint violations
- AMP configuration changes

The default name for ErrorTable1 is `ttname_ET`.

For more information on the error table format and the procedure to correct errors, refer to the Teradata FastLoad Reference’s Handling FastLoad Errors chapter.

### TD_ERROR_TABLE_2
- **Type**: varchar
- **Description**: Specifies the name of the second error table. This must be a new table name. You cannot use the name of an existing table unless you are restarting a paused Update driver job.

ErrorTable2 contains records that violated the unique primary index constraint. This type of error occurs during the application phase of the Update driver job.

The default name for ErrorTable2 is `ttname_UV`.

For more information on the error table format and the procedure to correct errors, refer to the Teradata FastLoad Reference’s Handling FastLoad Errors chapter.

### TD_LOGON_MECH
- **Type**: varchar
- **Description**: Specifies which logon mechanism is used. The job terminates if the attribute exceeds eight bytes.

See your site security administrator for specific mechanism names. For a list of available mechanisms, see Security Administration.

### TD_LOGON_MECH_DATA
- **Type**: varchar
- **Description**: Passes additional logon mechanism data. See your site security administrator for specific mechanism data. For more information, see Security Administration.

### TD_LOGSQL
- **Type**: varchar
- **Description**: Directs the Update driver to output the full Teradata SQL request in the trace output file with the driver’s trace is enabled. By default, when the driver’s trace is enabled, the Load driver outputs the Teradata SQL request, up to 32 kilobytes, in the trace output file.

Valid values:
- Yes (‘Y’) = Output the full Teradata SQL in the trace output file when the driver’s trace is enabled. The maximum length of the Teradata SQL is 1 megabyte.
- No (‘N’) = Do not output the Teradata SQL in the trace output file.

**Note**: When the driver’s trace is disabled, TD_LOGSQL has no effect.

---

### Table 16: Update Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TD_ERROR_TABLE_1         | varchar | Specifies the name of the first error table. This must be a new table name. You cannot use the name of an existing table unless you are restarting a paused Update driver job. ErrorTable1 contains records that were rejected during the acquisition phase of the Update driver job because of:  
- Data conversion errors  
- Constraint violations  
- AMP configuration changes  
The default name for ErrorTable1 is `ttname_ET`. For more information on the error table format and the procedure to correct errors, refer to the Teradata FastLoad Reference’s Handling FastLoad Errors chapter. |
| TD_ERROR_TABLE_2         | varchar | Specifies the name of the second error table. This must be a new table name. You cannot use the name of an existing table unless you are restarting a paused Update driver job. ErrorTable2 contains records that violated the unique primary index constraint. This type of error occurs during the application phase of the Update driver job. The default name for ErrorTable2 is `ttname_UV`. For more information on the error table format and the procedure to correct errors, refer to the Teradata FastLoad Reference’s Handling FastLoad Errors chapter. |
| TD_LOGON_MECH            | varchar | Specifies which logon mechanism is used. The job terminates if the attribute exceeds eight bytes. See your site security administrator for specific mechanism names. For a list of available mechanisms, see Security Administration. |
| TD_LOGON_MECH_DATA       | varchar | Passes additional logon mechanism data. See your site security administrator for specific mechanism data. For more information, see Security Administration. |
| TD_LOGSQL                | varchar | Directs the Update driver to output the full Teradata SQL request in the trace output file with the driver’s trace is enabled. By default, when the driver’s trace is enabled, the Load driver outputs the Teradata SQL request, up to 32 kilobytes, in the trace output file. Valid values:  
- Yes (‘Y’) = Output the full Teradata SQL in the trace output file when the driver’s trace is enabled. The maximum length of the Teradata SQL is 1 megabyte.  
- No (‘N’) = Do not output the Teradata SQL in the trace output file.  
  **Note**: When the driver’s trace is disabled, TD_LOGSQL has no effect. |

---
### Table 16: Update Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_MAX_SESSIONS</td>
<td>integer</td>
<td>Specifies the maximum number of sessions to log on. The default is one session per available AMP. The maximum value cannot be more than the number of AMPS available.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The MaxSessions value must be greater than zero. Specifying a value less than one causes the job to terminate.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The MaxSessions value must be greater than or equal to the value of TD_MAX_INSTANCES.</td>
</tr>
<tr>
<td>TD_MIN_SESSIONS</td>
<td>integer</td>
<td>Specifies the minimum number of sessions required for the Update driver job to continue. The default is one session.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The MinSessions value must be greater than zero and less than or equal to the maximum number of Update driver sessions. Specifying a value less than 1 will cause the Update driver to terminate.</td>
</tr>
<tr>
<td>TD_MSG_ENCODING</td>
<td>TD_Encoding</td>
<td>Specifies the encoding for the messages passed between Teradata PT API and a Teradata PT API application.</td>
</tr>
<tr>
<td>TD_NOTIFY_EXIT</td>
<td>varchar</td>
<td>Specifies the name of the user-defined notify exit routine with an entry point named _dynamn.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If no name is supplied, the following default names are used:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• libnotifyext.dll for Windows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• libnotifyext.sl for HP-UX platforms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• libnotifyext.so for Linux and all other UNIX platforms</td>
</tr>
</tbody>
</table>
Chapter 4: Update Driver
Attribute Definitions

Table 16: Update Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_NOTIFY_LEVEL</td>
<td>varchar</td>
<td>Indicates the level at which certain events are reported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The valid settings are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Off = No notification of events is provided. This is the default setting.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Low = Notification is provided for these events:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Initialize</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CLIv2/DBS error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Exit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Delete Init</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Delete Exit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Medium = Notification is provided for all the events except:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Checkpoint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Error Table 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Error Table 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• AMPS offline</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Import Begin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Import End</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• High = Notification is to be provided for all events.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For detailed information on the Notify feature, see the Update operator chapter in the Teradata Parallel Transporter Reference.</td>
</tr>
<tr>
<td>TD_NOTIFY_METHOD</td>
<td>varchar</td>
<td>Specifies the method for reporting events.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The methods are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• None = No event logging is done. This is the default.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Msg = This method sends the events to a log.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Exit = This method sends the events to a user-defined notify exit routine and to the system log.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>On Windows, the events are sent to the EventLog that can be viewed using the Event Viewer. The messages are sent to the application log.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>On AIX, HP-UX, Linux, and Solaris, platforms the destination of the events is specified in the /etc/syslog.conf file.</td>
</tr>
<tr>
<td>TD_NOTIFY_STRING</td>
<td>varchar</td>
<td>Provides a user-defined string to precede all messages sent to the system log. This string is also sent to the user-defined notify exit routine.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The maximum length of the string is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 80 bytes, if NotifyMethod is Exit.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 16 bytes, if NotifyMethod is Msg.</td>
</tr>
</tbody>
</table>
### Table 16: Update Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TD_PAUSE_ACQ                | varchar    | Specifies whether to pause the Update driver job after the acquisition phase or enter the application phase.  
Valid values are:  
- *No* (or *N*) = Do not pause. This is the default setting.  
- *Yes* (or *Y*) = Pause the Update driver job after the acquisition phase.  
Specifying any other value terminates the job. The absence of any value causes the Update driver job to execute both the acquisition phase and the application phase without pausing. This distributes all rows sent to the Teradata Database during the acquisition phase to their final destination on the AMPs. |
| TD_QUERY_BAND_SESS_INFO     | varchar    | Provides a user-defined query band expression that is set for every SQL session connected by the Teradata PT API driver.  
The following is an example of a valid query band expression:  
\[ \text{a}=1;\text{b}=2;\text{c}=3;\text{d}=4; \]  
If the TD_QUERY_BAND_SESS_INFO is set, the following request will be sent by every SQL session connected by the Teradata PT API Update driver:  
\[
\text{SET QUERY_BAND =}'<\text{User-Defined Query Band Expression}>' \text{ FOR SESSION;}
\]  
Setting the TD_QUERY_BAND_SESS_INFO attribute in jobs running against non-supported versions of the Teradata Database causes a non-fatal error. No error code is returned to the user during initiation and the job is allowed to proceed. The log table will not be dropped at the end of the job and the TD_Evt_ExitCode event returns a warning value of four instead of the normal success value of zero if queried. In this case, error information can be found in the trace file. |
| TD_REPLICATION_OVERRIDE     | varchar    | Overrides the normal replication services controls. The default is not to send any SET SESSION OVERRIDE REPLICATION statement to the database.  
The following valid values are not case sensitive:  
- *On* = Sends \text{SET SESSION OVERRIDE REPLICATION} ON to the database. Normal replication services controls are overridden.  
- *Off* = Sends \text{SET SESSION OVERRIDE REPLICATION OFF} to the database. Normal replication services controls are not overridden.  
For detailed information on the replication services feature, refer to the Teradata Replication Services Using GoldenGate and the SQL Data Definition Language. Check [http://www.info.teradata.com/](http://www.info.teradata.com/) for the latest versions. |
### Attribute Definitions

**TD_TDP_ID**
- **Type**: varchar
- **Description**: Specifies the name of the Teradata Database machine.
  - The `dbname` can be up to 256 characters and can be a domain server name.
  - TDP stands for Teradata Director Program and is specified for mainframe z/OS platforms.
  - If you do not specify the value for the TdpId attribute, the driver uses the default TdpId established for the user by the system administrator.

**TD_TENACITY_HOURS**
- **Type**: integer
- **Description**: Specifies the number of hours that the Update driver attempts to log on when the maximum number of load and export operations are already running on the Teradata Database.
  - The default value is four hours. To enable the tenacity feature, the `hours` value must be greater than zero.
  - Specifying a value of zero will disable the tenacity feature.
  - Specifying a value of less than zero terminates the Update driver.

**TD_TENACITY_SLEEP**
- **Type**: integer
- **Description**: Specifies the number of minutes the Update driver pauses before retrying to log on when the maximum number of load and export operations are already running on the Teradata Database.
  - The default is six minutes.
  - The minutes value must be greater than zero. If you specify a value less than one, the Update driver responds with an error message and terminates the job.
### Chapter 4: Update Driver

#### Attribute Definitions

**TD_TRACE_LEVEL**

**Note:** The TraceLevel attribute is an internal diagnostic aid. Use only if instructed to by Teradata support. TD_OFF should always be specified.

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TD_TRACE_LEVEL             | integer   | Specifies the types of diagnostic messages written by each instance of the driver to an external log file. The diagnostic trace function provides more detailed information (including the version number) in the log file to aid in problem tracking and diagnosis. Use the AddArray attribute method to specify the two types of tracing levels: driver tracing and infrastructure tracing. TD_OFF is the default setting for both driver tracing and infrastructure tracing. No external log file is produced unless this default is changed. Specifying TD_OFF for both driver tracing and infrastructure tracing is the same as disabling tracing. If the TraceLevel is set to any value other than TD_OFF, an external log file is created for each instance of the driver. The trace levels for driver tracing are:

- **TD_OFF** = Disables driver tracing.
- **TD_OPER** = Activates the tracing function for driver specific activities. The absence of any value for the PauseAcq attribute means that the Update driver job will execute both the acquisition phase and the application phase without pausing. This will distribute all of the rows that were sent to the Teradata Database during the acquisition phase to their final destination on the AMPs. Table 1 on page 40 lists which drivers have the Pause Acquisition attribute.
- **TD_OPER_CLI** = Activates the tracing function for CLIv2-related activities (interaction with the Teradata Database).
- **TD_OPER_NOTIFY** = Activates the tracing function for activities related to the Notify feature.
- **TD_OPER_OPCOMMON** = Activates the tracing function for activities involving the opcommon library.
- **TD_OPER_ALL** = Activates tracing for all of the above activities. The trace levels for infrastructure tracing should only be used when you are directed to by Teradata support. TD_OFF, which disables infrastructure tracing, should always be specified. |

**TD_TRACE_OUTPUT**

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_TRACE_OUTPUT</td>
<td>varchar</td>
<td>Specifies the name of the external file used for tracing messages. The default setting creates a new file name with the name of the driver followed by a time stamp. <strong>Note:</strong> If a file with the specified name already exists, the file is overwritten.</td>
</tr>
</tbody>
</table>

**TD_WORK_TABLE**

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_WORK_TABLE</td>
<td>varchar</td>
<td>Specifies the name of the work table. This must be a new table name. You cannot use an existing table name unless you are restarting a paused Update driver job. If the name is not supplied, it is created by the Update driver. The name of the created table is appended with an identifying tname_WT, ensuring uniqueness.</td>
</tr>
</tbody>
</table>
Table 16: Update Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_WORKINGDATABASE</td>
<td>varchar</td>
<td>Specifies the name of the database used in a Teradata SQL DATABASE statement that the Update driver sends to the Teradata Database immediately after connecting the two SQL sessions. Use this attribute to specify a default database other than the logon database.</td>
</tr>
</tbody>
</table>

GetEvent Queries

All events must be queried after the driver is initiated and before it is terminated. Events queried before their data is available return TD_Unavailable. The following table lists events used with the Connection object’s GetEvent function to retrieve run time statistics from the Update driver.

Table 17: Update Driver Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_Evt_CLIError</td>
<td>One 4-byte integer for the CLlv2 error number and a 255 byte character buffer for the CLlv2 error text. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_BufferLayout</td>
<td>Four 4-byte unsigned integers corresponding to the maximum buffer size, the row header size, the row length size, and the buffer trailer size. Use this information to format the buffer required for block updating. Note: These layout values will change depending on the user environment and may be defined differently in future releases. Always obtain these values from the event method before buffering data. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_DBSError</td>
<td>One 4-byte integer for the Teradata Database error number and a 255 byte character buffer containing the error text. This event can be queried at any time.</td>
</tr>
<tr>
<td>TD_Evt_ConnectStatus</td>
<td>Three 4-byte integers corresponding to the number of sessions requested, the number of sessions connected, and any CLlv2 error number that occurred during the connect process. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_CPUTime</td>
<td>One 8-byte double for the CPU time of the instance in seconds. Query this event at any time.</td>
</tr>
</tbody>
</table>
Table 17: Update Driver Events (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_Evt_RowCounts</td>
<td>Three 4-byte unsigned integers corresponding to the number of rows received, sent, and applied. Query this event before EndAcquisition to get the current number of rows received. However, the rest of the data for this event is available only after ApplyRows. The rows-applied count returned by this event does not take into consideration rows that are rejected by the database during the application phase. Query the TD_Evt_ApplyCount event to get the final count of rows that have been applied to the target table(s).</td>
</tr>
<tr>
<td>TD_Evt_ApplyCount</td>
<td>Three 4-byte unsigned integers corresponding to the number of rows inserted, updated, and deleted. This event must be called with a table index greater than or equal to one to signify which target table the desired data is for. Query this event after ApplyRows. For a multi-instance job, only query this number in the master instance. The master instance returns the total number of rows applied to the target table from all instances.</td>
</tr>
<tr>
<td>TD_Evt_ErrorTable1</td>
<td>One 4-byte unsigned integer for the number of rows in error table 1. This event must be called with a table index greater than or equal to one to signify the desired data's target table. Query this event after ApplyRows.</td>
</tr>
<tr>
<td>TD_Evt_ErrorTable2</td>
<td>One 4-byte unsigned integer for the number of rows in error table 2. This event must be called with a table index greater than or equal to one to signify the desired data's target table. Query this event after ApplyRows.</td>
</tr>
<tr>
<td>TD_Evt_ExitCode</td>
<td>One 2-byte integer for the exit code of the driver. Query this event right before the driver terminates.</td>
</tr>
</tbody>
</table>

Chapter 4: Update Driver

GetEvent Queries
Chapter 4: Update Driver
Programming Considerations

Table 17: Update Driver Events (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_Evt_RowsCheckpointed</td>
<td>One 4-byte unsigned integer corresponding to the total number of rows that have been received and checkpointed for the entire job. Query this event any time after initiating a Teradata PT API database connection in a single-instance job. This event will return TD_Unavailable when queried in a job containing multiple instances; this event is invalid if multiple instances are used. Note that the event data returned by this event can be used as a replacement for the checkpoint data that must be passed into the Restart method during a restart job. It is recommended to always save the checkpoint data from the last successful call to the CheckPoint method in a potential restart job. However, if the checkpoint data is not saved then the event data returned by this event can be passed into the Restart method instead. This alternate method of performing checkpoint/restarts will only work with single-instance jobs.</td>
</tr>
<tr>
<td>TD_Evt_Version</td>
<td>A pointer to a character string containing the Teradata PT API version followed by a pointer to a character string containing the operator version. Query this event at any time. The operator version is available only after the driver has been initiated.</td>
</tr>
</tbody>
</table>

Table 18: Update Driver Error Tables

<table>
<thead>
<tr>
<th>Error Table</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorTable1 = et1name</td>
<td>Records rejected because of:</td>
</tr>
<tr>
<td></td>
<td>• Data conversion errors</td>
</tr>
<tr>
<td></td>
<td>• Constraint violations</td>
</tr>
<tr>
<td></td>
<td>• AMP configuration changes</td>
</tr>
<tr>
<td>ErrorTable2 = et2name</td>
<td>Records that violate the unique primary index constraint.</td>
</tr>
</tbody>
</table>
Reusing Table Names

When an error table has one or more rows, it is not dropped from the Teradata Database at the end of an Update driver job. Use the DROP TABLE statement from BTEQ to remove the tables from the Teradata Database so that you can reuse the names specified for the error tables.

Limiting Insertion Errors

Use the TD_ERROR_LIMIT attribute to restrict the number of insertion errors captured in the ErrorTable1 during the acquisition phase of an Update job.

<table>
<thead>
<tr>
<th>IF you expect your Update job to encounter...</th>
<th>THEN specify an ERRORLIMIT value that is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>No errors or very few errors</td>
<td>low</td>
</tr>
<tr>
<td>Many errors that are considered allowable</td>
<td>high</td>
</tr>
</tbody>
</table>

The ErrorLimit specification applies to each instance of the Update driver, not to all instances combined. For example, you set the limit to 10,000 rows. A single instance must detect that 10,000 rows were inserted into the first error table to terminate the job, and those 10,000 rows must be controlled by the sessions managed by that instance.

If you expect no errors in the input data, set the error limit value to one. In this case, the job terminates when any record causes an error. However, when the specified error limit is reached, the Update driver continues processing until each session completes its current data block. This continued processing can cause the total number of error rows captured in the first error table to exceed the ErrorLimit specification.

Using DELETE in Import Tasks

DELETE is a Teradata SQL statement that removes rows from a table or view that was previously identified as a target table through the use of the TD_TARGET_TABLE attribute.

The rules for using the DELETE statement in an Update import tasks are:

- Apply the DELETE statements to either a table or a view, provided that the view does not specify a join
- The number of input data records is unlimited
- The equality values must be specified for all the primary index columns in the WHERE clause of a DELETE statement. Greater than and Less than operators cannot be used to specify a range of rows.
- Do not use the OR construct in the WHERE clause of a DELETE statement. Instead, use two separate DELETE statements.

The procedure for using the DELETE statement in an Update import task is:

1. Add a DELETE statement to a DML group
2. Add the DML group to the Connection object
3. Initiate the Connection object
4 Set the DML group containing the DELETE statement using the UseDMLGroups function.

   Note: If the DML group only contains a DELETE statement, then no data will be loaded as a result of the PutRow function call.

5 Make at least one call to the PutRow function.

6 Call the ApplyRows function.

7 Terminate the Connection object.

   Note: If no row in the target table matches the DELETE statement, then the row of data sent with the DELETE statement will be put into the application error table.

Using Delete Task

Use the Update driver’s Delete Task with or without variable substitution. If variable substitution is used, the Update driver requires a schema. The following sections give an overview of the steps needed to use the Update Delete Task.

Using DELETE in Delete Tasks

Deleting rows in a Delete Task is faster than deleting rows in an import task. The rules for using the DELETE statement in Delete tasks are:

- Only one special session can be connected
- Only one instance may be specified
- Only one DML group may be specified
- Only one DML statement in the DML group may be specified
- Only one target table may be specified
- The first error table is not used and is ignored
- Only one data record is provided if using variable substitution in the WHERE clause.
- There can be no calls to the EndAcquisition function.

The procedure for using the Update Delete Task without variable substitution is:

1 Set the TD_DELETE_TASK attribute to “Yes”
2 Add one DELETE statement to a DML group
3 Add the DML group to the Connection object
4 Initiate the Connection object
5 call the ApplyRows function
6 Terminate the Connection object

   Note: Do not add a schema to the Connection object.

The procedure for using the Update Delete Task with variable substitution is:

1 Set the TD_DELETE_TASK attribute to “Yes”
2 Add the schema corresponding to the variable substitution to the Connection object.
3 Add one DELETE statement to a DML group
4 Add the DML group to the Connection object
Chapter 4: Update Driver
Programming Considerations

5 Initiate the Connection object
6 Use PutRow to pass in variable substitution data (one call only)
7 call the ApplyRows function
8 Terminate the Connection object

Dropping Tables During a Load

Some tables are created during the execution of a job, but others must be created by you before the job begins. Target tables must exist on the Teradata Database when an Update driver job is executed. The log table is created automatically when you run the Update job script. Error tables and work tables are created by the Teradata Database. Error tables are dropped by the Update driver during the cleanup phase if no errors were detected during the acquisition phase or the application phase. The work table is dropped by the Update driver during the cleanup phase if no errors were detected during the acquisition or the application phase. The log table is dropped by the Update driver after the job completes successfully.

When a job terminates abnormally, the log, error, and work tables may not be dropped depending on where in the job the termination occurred. If you want to restart the job from the beginning, you need to manually drop these tables by running a BTEQ script.

Caution: Care must be taken when you manually drop the target tables using a BTEQ script. If something goes wrong with an Update job and you drop the target tables manually, when you try to rerun the job, you may lose the original data. Because all rows are held in work tables until the Update job reaches the application phase (when they are copied to the target tables), you risk losing the original data.

Required Privileges

The user ID used by the Update driver to connect to the Teradata Database must have:

- SELECT and INSERT privileges on the Update target table
- SELECT and INSERT privileges on the error tables
- DROP privileges on the database that contains the error tables.
- SELECT, INSERT, and DELETE privileges on the restart log table
- DROP privileges on the database that contains the restart log table.

Session Limits

The values you specify with the Update driver’s TD_MIN_SESSIONS and TD_MAX_SESSIONS attributes are not the only factors limiting the number of sessions the Update driver can establish with the Teradata Database. When the Update driver executes, the actual session limit is determined by whichever limiting factor is encountered first.

The other limiting factors are:

- The Teradata Database limit of one session per AMP. This value is defined in the COP Interface software file, CLISPB.DAT, under the max_num_sess variable. You can use the TDP SET MAXSESSIONS command to specify a platform limit. The default TDP MAXSESS value is 1024 sessions.
• The limit of the network protocol software on network-attached systems.

**Offline AMPs**

The impact of offline AMPs on the Update driver depends on:

• The number of offline AMPs in a cluster, either logically or physically
• The operational phase of the Update tasks when the offline AMP condition occurs
• Whether the target tables are fallback or nonfallback

The table below describes the impact of offline AMPs on Update driver tasks on fallback and nonfallback tables.

<table>
<thead>
<tr>
<th>If...</th>
<th>And...</th>
<th>Then...</th>
</tr>
</thead>
</table>
| All of the target tables are fallback | Not more than one AMP is offline | Update driver tasks continue to run as long as there is not more than one AMP offline in a cluster, either logically or physically. The offline AMP does not participate in the application phase if:
- The AMP goes offline before the Update tasks enter the application phase, and the AmpCheck attribute is set to None.
- Certain I/O errors occur during the application phase. |
| Two or more AMPs are offline | Update driver tasks do not run or terminate if two or more AMPs are offline in a cluster, either logically or physically. **Note:** In the application phase, if AMPs are offline to the extent that data on the disk is corrupted, then you must restore the affected tables. |
| One or more of the target tables is nonfallback | One or more AMPs are offline | Update driver tasks terminate and you cannot restart until all of the AMPs are online. **Note:** The Update driver also terminates if I/O errors corrupt the target tables in the application phase. |

**Nonparticipant AMPs**

There are three ways for an AMP to become nonparticipant for an Update driver task:
Chapter 4: Update Driver
Programming Considerations

The Update driver treats a nonparticipant AMP as if it is an offline AMP. The Update driver does not execute if a cluster has any combination of more than one AMP that is:

- Offline
- Nonparticipant

If more than one AMP in a cluster becomes a nonparticipant during the application phase, the Update driver tasks do not continue; the target tables are considered unusable, and must be recovered from archives.

### Single-AMP Systems

The Update driver cannot run on a single-AMP Teradata Database system or on a multi-AMP system configured with single-AMP clusters. Any attempt to run the Update driver in this environment causes the Teradata Database to immediately reject the request, abort the job, and issue a diagnostic message stating that a single-AMP system is not supported.

### Space Requirements and Limitations

Always estimate the final size of the Update target table, and make sure the destination database on the Teradata Database has enough space to accommodate the Update job. If the database that owns the Update target table, log table, error tables, or work table runs out of space, the Teradata Database returns an error message and the Update driver pauses the Update job. When this happens, you must allocate more space to the database before you can restart the job.

---

**Table 21: Nonparticipant AMP Conditions and Effects on Update Driver Tasks**

<table>
<thead>
<tr>
<th>When...</th>
<th>Then the associated AMP becomes a nonparticipant...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any AMP is down at the end of the acquisition phase or the beginning of</td>
<td>If the TD_amp_check None option is specified. Because the Update driver does not run after the acquisition phase if an AMP is offline and the target table is non fallback, an AMP can become a nonparticipant only if the target table is defined as having fallback protection. The TD_amp_check Apply and All options would prevent the occurrence of nonparticipant AMPs in this situation.</td>
</tr>
<tr>
<td>the application phase</td>
<td></td>
</tr>
<tr>
<td>I/O errors occur in certain Update driver tables during the application</td>
<td>When the I/O recovery operation stops the Update task.</td>
</tr>
<tr>
<td>phase, a head/disk assembly (HDA) fails during the application phase</td>
<td></td>
</tr>
<tr>
<td>A head/disk assembly (HDA) fails during the application phase</td>
<td>But it returns after the disk is replaced and the Disk Copy and Table Rebuild utilities are run.</td>
</tr>
</tbody>
</table>
Checkpoint and Restart Operations

The Update driver is fully checkpoint restartable. Refer to the checkpoint and restart discussion in “Add Checkpoint and Restart” on page 33.

Code Example

```cpp
#include "connection.h"
#include "schema.h"
#include "DMLGroup.h"

using namespace teradata::client::API;

int returnValue = 0;
char* errorMsg = NULL;
TD_ErrorType errorType;

cout << "*** Update Driver Example ***" << endl;
Connection *conn = new Connection();

/**********************************************
* Set Operator Type and Trace/Log Levels
***********************************************/
conn->AddAttribute(TD_SYSTEM_OPERATOR,TD_UPDATE);
//conn->AddAttribute(TD_TRACE_OUTPUT,"update.txt");
//conn->AddAttribute(TD_TRACE_LEVEL,2,TD_OPER_ALL,TD_OFF,NULL);

/**********************************************
* Add Attributes
***********************************************/
conn->AddAttribute(TD_USER_NAME,"user");
conn->AddAttribute(TD_USER_PASSWORD,"password");
conn->AddAttribute(TD_TDP_ID,"database");
conn->AddAttribute(TD_LOG_TABLE,"tdupdate_log");

conn->AddArrayAttribute(TD_TARGET_TABLE, 2,"tdupdateA", "tdupdateB", NULL);
conn->AddArrayAttribute(TD_WORK_TABLE  , 2,"tdupdateA_wt","tdupdateB_wt", NULL);
conn->AddArrayAttribute(TD_ERROR_TABLE_1,2,"tdupdateA_e1","tdupdateB_e1", NULL);
conn->AddArrayAttribute(TD_ERROR_TABLE_2,2,"tdupdateA_e2","tdupdateB_e2", NULL);

/**********************************************
* Add Schema
***********************************************/
Schema *schema = new Schema("input");
schema->AddColumn("Associate_Id",TD_INTEGER,4);
schema->AddColumn("Associate_Name",TD_CHAR,25);
schema->AddColumn("Salary",TD_FLOAT,8);
schema->AddColumn("DOJ",TD_DATE,4);
schema->AddColumn("Designation",TD_VARCHAR,25);
schema->AddColumn("Loan_Amount",TD_DECIMAL,4,5,2);
schema->AddColumn("Martial_Status",TD_CHAR,1);
schema->AddColumn("No_Of_Dependents",TD_BYTEINT,1);
conn->AddSchema(schema);

/**********************************************
* Set DMLGroups
***********************************************/
DMLGroup *dmlGr = new DMLGroup();
dmlGr->AddStatement("INSERT INTO tdupdateA( :Associate_Id, :Associate_Name, :Salary, :DOJ, :Designation, :Loan_Amount, :Martial_Status, :No_Of_Dependents);" );
dmlGr->AddDMLOption(MARK_DUPLICATE_ROWS);
returnValue = conn->AddDMLGroup(dmlGr,&dmlGroupIndex[0]);
delete dmlGr;

dmlGr = new DMLGroup();
dmlGr->AddStatement("INSERT INTO tdupdateB( :Associate_Id, :Associate_Name, :Salary, :DOJ, :Designation, :Loan_Amount, :Martial_Status, :No_Of_Dependents);" );
dmlGr->AddDMLOption(IGNORE_DUPLICATE_ROWS);
returnValue = conn->AddDMLGroup(dmlGr,&dmlGroupIndex[1]);
delete dmlGr;

cout << "DMLGroups added with status " << returnValue << endl;
if (returnValue < TD_Error )
{
```
/**********************************************
* Initiate
***********************************************/

returnValue = conn->Initiate();
cout << "Driver Initiated with status " << returnValue << endl;
if( returnValue < TD_Error )
{
    /**********************************************
    * Acquisition
    ***********************************************/
    char rowBuffer[78]; /* must include the EOL byte */
    TD_Length bytes;
    int loadStatus = 0;
    loadStatus = GetRowData(rowBuffer,78); // user function - reads in a
    // row of data from a file
    // returns -1 when end of file
    // reached
    if( loadStatus != -1 ){
        bytes = *((unsigned short *)rowBuffer);
        for( int i = 0; i < 2; i++ ){
            returnValue = conn->UseDMLGroups(&dmlGroupIndex[i],1);
            if( returnValue >= TD_Error)
                break;
            cout << "Sending First Row" << endl;
            returnValue = conn->PutRow( rowBuffer + 2 , bytes);
            if( returnValue >= TD_Error)
                break;
            cout << "Sending Duplicate Row" << endl;
            returnValue = conn->PutRow( rowBuffer + 2 , bytes);
            if( returnValue >= TD_Error)
                break;
        }
        cout << "Rows sent with status " << returnValue << endl;
    loadStatus = GetRowData(NULL,0); //user function - close file
    if( returnValue < TD_Error )
    {
        /**********************************************
        * End Acquisition
        ***********************************************/
        returnValue = conn->EndAcquisition();
        cout << "Acquisition completed with status " << returnValue << endl;
        if( returnValue < TD_Error )
        {
            /**********************************************
            * Application
            ***********************************************/
            returnValue = conn->ApplyRows();
            cout << "Application completed with status " << returnValue << endl;
            if ( returnValue < TD_Error )
            {
                cout << "Update completed successfully" << endl;
} else {
    // Get Error Information
    cout << "Error occurred during Application" << endl;
    conn->GetErrorInfo(&errorMsg, &errorType);
    if (errorMsg != NULL) {
        cout << errorMsg << endl;
        cout << "Type: " << errorType << endl;
    } else {
        cout << "No Error Info Available" << endl;
    }
}
} else {
    // Get Error Information
    cout << "Error occurred during EndAcquisition" << endl;
    conn->GetErrorInfo(&errorMsg, &errorType);
    if (errorMsg != NULL) {
        cout << errorMsg << endl;
        cout << "Type: " << errorType << endl;
    } else {
        cout << "No Error Info Available" << endl;
    }
}
} else {
    // Get Error Information
    cout << "Error occurred during Acquisition" << endl;
    conn->GetErrorInfo(&errorMsg, &errorType);
    if (errorMsg != NULL) {
        cout << errorMsg << endl;
        cout << "Type: " << errorType << endl;
    } else {
        cout << "No Error Info Available" << endl;
    }
}
} else {
    // Get Error Information
    cout << "Error occurred during Initiate" << endl;
    conn->GetErrorInfo(&errorMsg, &errorType);
    if (errorMsg != NULL) {
        cout << errorMsg << endl;
        cout << "Type: " << errorType << endl;
    } else {
        cout << "No Error Info Available" << endl;
    }
}
} else {
    // Get Error Information
    cout << "Error occurred during AddDMLGroups" << endl;
    conn->GetErrorInfo(&errorMsg, &errorType);
    if (errorMsg != NULL) {
        cout << errorMsg << endl;
        cout << "Type: " << errorType << endl;
    } else {
        cout << "No Error Info Available" << endl;
    }
}
}/* **********************************************
 Terminate
 ***********************************************/
returnValue = conn->Terminate();
cout << "Driver Terminated with status " << returnValue << endl;
if (returnValue >= TD_Error) {
    // Get Error Information
    cout << "Error occurred during Terminate" << endl;
    conn->GetErrorInfo(&errorMsg, &errorType);
    if (errorMsg != NULL) {
        cout << errorMsg << endl;
        cout << "Type: " << errorType << endl;
    } else {
        cout << "No Error Info Available" << endl;
    }
}
Chapter 4: Update Driver

Code Example

/**********************************************
* Clean Up
**********************************************/

```
cout << "Deleting Objects" << endl;
delete schema;
delete conn;

cout << "*** Update Complete ***" << endl;
```
The Stream driver uses Teradata TPump protocol to perform high-speed DML transactions in a near-real-time mode on tables while the tables are being queried.

Using multiple sessions to load data, the Stream driver allows parallel inserts, updates, deletes, and upserts to empty or preexisting Teradata Database tables.

The Stream driver provides an alternative to the Update driver for the maintenance of large databases under control of a Teradata Database. Unlike the Update driver, the Stream driver allows access to the database and does not lock the target tables being updated so that interactive read and write activities can be performed concurrently.

The Stream driver also uses standard SQL/DML to maintain data in up to 128 tables at a time. Row level locking, as used in the Stream driver SQL transactions, allows constant load operations in the background during normal system use without locking target tables. It has the same restart ability, portability, and scalability as the Update driver. Learn more about the Stream driver in this chapter’s main sections:

- Attribute Definitions
- Required Attributes
- Optional Attributes
- GetEvent Queries
- Programming Considerations
- Error Tables
- Reusing Stream Driver Table Names
- Limiting Errors in the Stream Driver
- Dropping Tables During a Load
- Required Privileges
- Session Limits
- Obtaining the Row Count Using TD_Evt_ApplyCount
- Space Requirements and Limitations
- Tuning the Pack Factor
- Using DELETE in Import Tasks
- Stream Driver Macro Support
- Checkpoint and Restart Operations
- Code Example

Attribute Definitions

Required attributes are listed in Table 22, and optional attributes are defined in Table 23.
# Required Attributes

Table 22: Stream Driver Required Attributes

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_BUFFER_MODE</td>
<td>varchar</td>
<td>Indicates which type of Stream method is used. Valid Settings:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Default value is No.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Must be set to Yes when using the block Stream feature.</td>
</tr>
<tr>
<td>TD_INSTANCE_NUM</td>
<td>integer</td>
<td>Provides the instance number of the current instance. Required when using multiple instances of the same driver in a master-slave environment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the current instance is the master instance, then the instance number should be one.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the current instance is a slave instance, then the instance number should be a value greater than one.</td>
</tr>
<tr>
<td>TD_LOG_TABLE</td>
<td>varchar</td>
<td>Provides the name of the restart log table for restart information. Each Stream driver job needs to use its own restart log table if multiple Stream driver jobs are executed concurrently.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> If the restart log table name is not fully qualified, it is created under the user’s default (logon) database. Alternately, a working database can be specified using the TD_WORKINGDATABASE attribute. If the TD_WORKINGDATABASE attribute is used, the restart log table name must be fully qualified, even if the restart log table is going to reside in the default (logon) database.</td>
</tr>
<tr>
<td>TD_MAX_INSTANCES</td>
<td>integer</td>
<td>Provides the total number of instances (master and slaves). Required when using multiple instances of the same driver in a master-slave environment.</td>
</tr>
<tr>
<td>TD_RESTARTMODE</td>
<td>integer</td>
<td>Required only before restarting and must be set to one before performing a restart.</td>
</tr>
<tr>
<td>TD_SYSTEM_OPERATOR</td>
<td>varchar</td>
<td>Specifies the type of driver being used (in this case TD_STREAM).</td>
</tr>
<tr>
<td>TD_USER_NAME</td>
<td>varchar</td>
<td>Provides the name of the user for the Stream driver logon sessions.</td>
</tr>
<tr>
<td>TD_USER_PASSWORD</td>
<td>varchar</td>
<td>Specifies the password associated with the user name.</td>
</tr>
</tbody>
</table>
### Optional Attributes

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_ACCOUNT_ID</td>
<td>varchar</td>
<td>Specifies the account associated with the specified user name. When omitted, this attribute defaults to the account identifier of the immediate owner database.</td>
</tr>
</tbody>
</table>
| TD_APPENDERRORTABLE | varchar | Directs the Stream driver to use the existing error table that is specified in the TD_ERROR_TABLE attribute. By default, if the specified error table already exists, the Stream driver terminates the job with an error message.  
**Note:** Set the Stream driver’s TD_DROPERRORTABLE attribute to align the Stream driver default setting.  
Valid values are:  
- *No* (‘N’) = Do not use the existing error table. This is the default setting.  
- *Yes* (‘Y’) = Use the existing error table. The Stream driver will create the error table if the error table does not exist. If the error table does exist, the Stream driver will display the number of rows already in the error table in the trace file for the job if one exists (see “TD_TRACE_LEVEL” on page 113 and “TD_TRACE_OUTPUT” on page 114). If the structure of the existing error table is not compatible with the error table the Stream driver creates, the job terminates with an error message at the first attempt to insert or update to the existing error table. |
| TD_ARRAYSUPPORT     | varchar | Specifies default array support option for all DMLGroups. Each DMLGroup can modify this setting through the AddArraySupport function.  
Valid values are:  
- *On* = Array support is turned on. This is the default setting if the Teradata Database supports the array support functionality. An error will be returned and the job terminated if Array support is turned on for a Teradata Database that does not support this feature.  
- *Off* = Array support is turned off. This is the default setting if the Teradata Database does not support the array support functionality.  
**Note:** Performance will improve when Array Support is enabled. |
### Table 23: Stream Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TD_BUFFERS</strong></td>
<td>integer</td>
<td>Specifies whether to increase the number of request buffers. The range of values has a lower limit of two and no upper limit. The default value is three. The maximum number of request buffers you can allocate is the number of buffers multiplied by the number of connected sessions. Because request buffers are a global resource, buffers are assigned to any session as needed and then returned to a free pool. At any point in time, the number of request buffers assigned to a session can vary from zero to the maximum number you allocate.</td>
</tr>
<tr>
<td><strong>TD_CHARSET</strong></td>
<td>varchar</td>
<td>Specifies the name or code of the character set to be used for the job. On channel-attached z/OS platforms, only EBCDIC encoding is supported and is automatically selected. For the list of supported character sets, see the Extended Character Sets section in the <em>Teradata Parallel Transporter Reference</em>. Check <a href="http://www.info.teradata.com/">http://www.info.teradata.com/</a> for the latest version of this publication.</td>
</tr>
<tr>
<td><strong>TD_DATA_ENCRYPTION</strong></td>
<td>varchar</td>
<td>Provides full security encryption of SQL requests, responses and data. Valid values are: • <em>Off</em> = No encryption occurs. This is the default setting. • <em>On</em> = All SQL requests, responses, and data are encrypted.</td>
</tr>
<tr>
<td><strong>TD_DATE_FORM</strong></td>
<td>varchar</td>
<td>Specifies the DATE data type for the Stream driver job. Valid settings are: • <em>IntegerDate</em> = Integer DATE data type. This is the default setting. • <em>ANSIDate</em> = ANSI fixed-length CHAR(10) DATE data type.</td>
</tr>
</tbody>
</table>
TD_DROPERRORTABLE  varchar  Directs the Stream driver to drop the existing error table at the end of the job. By default, the Stream driver drops the error table at the end of a job if the error table is empty.

**Note:** Use the `TD_APPENDERRORTABLE` attribute to direct the Stream driver to continue using the existing error table.

Valid values are:
- **Yes (‘Y’) =** Drop the error table if it is empty at the end of the job. This is the default setting.
- **No (‘N’) =** Do not drop the existing error table.

---

TD_DROPMACRO  varchar  Specifies whether to keep or drop the macro created during the current Stream job.

Valid values are:
- **Yes (‘Y’) =** Drop the macro. This is the default setting.
- **No (‘N’) =** Keep the macro.

---

TD_ERROR_LIMIT  integer  Specifies the maximum number of records stored in one of the error tables before the Stream driver job is terminated. By default, the ErrorLimit value is unlimited.

The ErrorLimit specification must be greater than zero.

Specifying an invalid value terminates the Stream driver.

The ErrorLimit specification applies to each instance of the Stream driver.

---

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_DROPERRORTABLE</td>
<td>varchar</td>
<td>Directs the Stream driver to drop the existing error table at the end of the job. By default, the Stream driver drops the error table at the end of a job if the error table is empty. <strong>Note:</strong> Use the <code>TD_APPENDERRORTABLE</code> attribute to direct the Stream driver to continue using the existing error table. Valid values are: • <strong>Yes (‘Y’) =</strong> Drop the error table if it is empty at the end of the job. This is the default setting. • <strong>No (‘N’) =</strong> Do not drop the existing error table.</td>
</tr>
</tbody>
</table>
Table 23: Stream Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TD_ERROR_TABLE          | varchar  | Specifies the name of the error table. ErrorTable must be a new table name. You cannot use the name of an existing table unless you are restarting a Stream driver job. If the name is not supplied, it will be created by the Stream driver. The Error Table contains information concerning:  
  - Data conversion errors  
  - Constraint violations  
  - and other error conditions  
  It is acceptable, and even good practice, to prefix the error table with a database name as a qualifier. This means that because the database may have a lot of PERM space, which space will not have to be increased for all databases with tables involved in the load.  
  If the database for the error table is not specified, the table is placed in the database associated with the user logon.  
  Refer to the Teradata Parallel Data Pump Reference's Reading TPump Error Tables chapter for information on the error table format and the procedure to correct errors. |
| TD_LOGON_MECH           | varchar  | Specifies which logon mechanism is used.  
  - See your site security administrator for specific mechanism names. For a list of available mechanisms, see Security Administration.  
  - The job terminates if the attribute exceeds eight bytes.                                                                                                                                                                                                                                                                                                                                                         |
| TD_LOGON_MECH_DATA      | varchar  | Passes along additional logon mechanism data.  
  See your site security administrator for specific mechanism data. For more information, see Security Administration.                                                                                                                                                                                                                                                                                                                                          |
### Table 23: Stream Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TD_LOGSQL              | varchar  | Directs the Stream driver to output the full Teradata SQL request in the trace output file with the driver’s trace is enabled. By default, when the driver’s trace is enabled, the Load driver outputs the Teradata SQL request, up to 32 kilobytes, in the trace output file. Valid values:  
|                         |          | • Yes ('Y') = Output the full Teradata SQL in the trace output file when the driver’s trace is enabled. The maximum length of the Teradata SQL is 1 megabyte.  
|                         |          | • No ('N') = Do not output the Teradata SQL in the trace output file.  
|                         |          | **Note:** When the driver’s trace is disabled, TD_LOGSQL has no effect.  |
| TD_MACRODATABASE       | varchar  | Specifies the database that contains any macros used by the Stream driver. The default macro database is the restart log table database.          |
| TD_MAX_SESSIONS        | integer  | Specifies the maximum number of sessions to log on. The default is one session per available AMP. The maximum value cannot be more than the number of AMPS available. The MaxSessions value must be greater than zero. Specifying a value less than one causes the job to terminate. The MaxSessions value must be greater than or equal to the value of TD_MAX_INSTANCES. |
| TD_MIN_SESSIONS        | integer  | Specifies the minimum number of sessions required for the Stream driver job to continue. The default is one session. The MinSessions value must be greater than zero and less than or equal to the maximum number of Stream driver sessions. Specifying a value less than one terminates the Stream driver. |
| TD_MSG_ENCODING        | TD_Encoding | Specifies the encoding for the messages passed between Teradata PT API and a Teradata PT API application. |
Table 23: Stream Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_NOTIFY_EXIT</td>
<td>varchar</td>
<td>Specifies the name of the user-defined notify exit routine with an entry point named _dynamn.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If no name is supplied, the following default names are used:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• libnotfyext.dll for Windows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• libnotfyext.sl for HP-UX platforms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• libnotfyext.so for Linux and all other UNIX platforms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For detailed information on the Notify feature, see the Stream driver chapter in the Teradata Parallel Transporter Reference. Check <a href="http://www.info.teradata.com/">http://www.info.teradata.com/</a> for the latest version.</td>
</tr>
<tr>
<td>TD_NOTIFY_LEVEL</td>
<td>varchar</td>
<td>Indicates the level at which certain events are reported.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The valid settings are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Off = No notification of events will be provided. This is the default setting.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Low = Notification is provided for these events:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Initialize</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CLlv2/DBS error</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Exit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Medium = Notification is provided for all the events except:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Checkpoint Begin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Import Begin</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Import End</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Error Table</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Checkpoint End</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Interim Run Statistics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• High = Notification is provided for all events.</td>
</tr>
</tbody>
</table>
Chapter 5: Stream Driver
Attribute Definitions

**TD_NOTIFY_METHOD**
- **Type**: varchar
- Indicates the method used for reporting events. The methods are:
  - **None**: No event logging is done. This is the default method.
  - **Msg**: This method sends the events to a log.
  - **Exit**: This method sends the events to a user-defined notify exit routine and to the system log.
  - On Windows, the events are sent to the EventLog that can be viewed using the Event Viewer. The messages are sent to the application log.
  - On AIX, HP-UX, Linux, and Solaris platforms, the destination of the events is dependent upon the setting specified in the file called `/etc/syslog.conf`.

**TD_NOTIFY_STRING**
- **Type**: varchar
- Provides a user-defined string to precede all messages sent to the system log. This string is also sent to the user-defined notify exit routine.
- The maximum length of the string is:
  - 80 bytes, if NotifyMethod is `Exit`.
  - 16 bytes, if NotifyMethod is `Msg`.

**TD_PACK**
- **Type**: integer
- Specifies the number of statements to pack into a multiple-statement request. The default value is 20. The maximum value is 600.

**TD_PACKMAXIMUM**
- **Type**: varchar
- Triggers the Stream driver to dynamically determine the maximum possible pack factor for the current Stream job.
- Valid values are:
  - **No** (or `N`): Use the default pack factor. This is the default setting.
  - **Yes** (or `Y`): Determine maximum possible pack factor.
Table 23: Stream Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| **TD_QUERY_BAND_SESS_INFO** | varchar   | Provides a user-defined query band expression that is set for every SQL session connected by the Teradata PT API driver. The following is an example of a valid query band expression:  
  
  \[a=1;b=2;c=3;d=4;\]  
  
  If the TD_QUERY_BAND_SESS_INFO is set, the following request will be sent by every SQL session connected by the Teradata PT API Stream driver:  
  
  \[SET \text{ QUERY\_BAND} = \langle \text{User-Defined Query Band Expression} \rangle \text{ FOR SESSION};\]  
  
  Setting the TD_QUERY_BAND_SESS_INFO attribute in jobs running against non-supported versions of the Teradata Database causes a non-fatal error. No error code is returned to the user during initiation and the job is allowed to proceed. The log table will not be dropped at the end of the job and the TD_Evt ExitCode event returns a warning value of four instead of the normal success value of zero if queried. In this case, error information can be found in the trace file. |
| **TD_REPLICATION_OVERRIDE** | varchar   | Overrides the normal replication services controls. The default is not to send any SET SESSION OVERRIDE REPLICATION statement to the database. The following valid values are not case sensitive:  
  
  - **On** = Sends SET SESSION OVERRIDE REPLICATION ON to the database. Normal replication services controls are overridden.  
  - **Off** = Sends SET SESSION OVERRIDE REPLICATION OFF to the database. Normal replication services controls are not overridden.  
  
  For detailed information on the replication services feature, refer to Teradata Replication Services Using GoldenGate and SQL Data Definition Language. Check [http://www.info.teradata.com/](http://www.info.teradata.com/) for the latest versions. |
Chapter 5: Stream Driver

Attribute Definitions

Table 23: Stream Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_ROBUST</td>
<td>varchar</td>
<td>Specifies whether or not to use robust restart logic for recovery and restart operations. In robust mode, one database row is written in the log restart table for every request issued. These rows are the request log. Because the Teradata Database completely finishes or rolls back, the request log always accurately reflects the completion status of an import. Valid values are: • Yes (or Y) = Use robust restart logic. This is the default setting. In the robust mode, a number of partial checkpoint rows are written to the request log between checkpoints for each packed request. The rows are deleted each time a checkpoint is written. In Robust recovery mode, the Stream driver must next ascertain how much processing has been completed since the last logged checkpoint. This is accomplished by reading back a set of Partial Checkpoints from the Teradata Database, sorting them and then reprocessing all transactions that were left incomplete when the job was interrupted. • No (or N) = Use simple restart logic. Restarts begin where the last checkpoint occurred in the job. Any processing completed after that checkpoint is redone, which could lead to duplicate rows being sent to the target table. This method eliminates the extra overhead of the additional database writes used in the robust mode, and it is adequate for DML statements that can be repeated without changing the results of the operation. If you are not sure about using robust restart logic, it is always safe to set the robust attribute to Yes.</td>
</tr>
<tr>
<td>TD_TDP_ID</td>
<td>varchar</td>
<td>Specifies the name of the Teradata Database machine. • The dbname can be up to 256 characters and can be a domain server name. • TDP stands for Teradata Director Program and is specified for mainframe z/OS platforms. • If you do not specify a value for the TD_TDP_ID attribute, the Stream driver uses the default TDPID established for the user by the system administrator.</td>
</tr>
</tbody>
</table>
Table 23: Stream Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_TENACITY_HOURS</td>
<td>integer</td>
<td>Specifies the number of hours that the Stream driver continues trying to log on when the maximum number of load and export operations are already running on the Teradata Database.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The default value is four hours. To enable the tenacity feature, the “hours” value must be greater than zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Specifying a value of zero will disable the tenacity feature.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Specifying a value of less than zero will cause the Stream driver to terminate.</td>
</tr>
<tr>
<td>TD_TENACITY_SLEEP</td>
<td>integer</td>
<td>Specifies the number of minutes that the Stream driver pauses before retrying to log on when the maximum number of load and export operations are already running on the Teradata Database.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The default is six minutes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The minutes value must be greater than zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If you specify a value less than one, the Stream driver responds with an error message and terminates the job.</td>
</tr>
</tbody>
</table>
### TD_TRACE_LEVEL

**Note:** The TraceLevel attribute is an internal diagnostic aid. Use only if instructed to by Teradata support. TD_OFF should always be specified.

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TD_TRACE_LEVEL     | integer | Specifies the types of diagnostic messages written by each instance of the driver to an external log file. The diagnostic trace function provides more detailed information in the log file (including the version number) to aid in problem tracking and diagnosis. Use the AddArray attribute method to specify the two types of tracing levels: driver tracing and infrastructure tracing. **TD_OFF** is the default setting for both driver tracing and infrastructure tracing. No external log file is produced unless this default is changed. Specifying **TD_OFF** for both driver tracing and infrastructure tracing is the same as disabling tracing. If the TraceLevel is set to any value other than **TD_OFF**, an external log file is created for each instance of the driver. The trace levels for driver tracing are:  
  - **TD_OFF** = Disables driver tracing.  
  - **TD_OPER** = Activates the tracing function for driver specific activities. The absence of any value for the PauseAcq attribute means that the Stream driver job will execute both the acquisition phase and the application phase without pausing. This will distribute all of the rows that were sent to the Teradata Database during the acquisition phase to their final destination on the AMPs. *Table 1 on page 40* lists which drivers have the Pause Acquisition attribute.  
    - **TD_OPER_CLI** = Activates the tracing function for CLIv2-related activities (interaction with the Teradata Database).  
    - **TD_OPER_NOTIFY** = Activates the tracing function for activities related to the Notify feature.  
    - **TD_OPER_OPCOMMON** = Activates the tracing function for activities involving the opcmon library.  
    - **TD_OPER_ALL** = Activates tracing for all of the above activities.  
  The trace levels for infrastructure tracing should only be used when you are directed to by Teradata support. **TD_OFF**, which disables infrastructure tracing, should always be specified. |
GetEvent Queries

All events must be queried after the driver has initiated and before it has terminated. Events queried before their data is available return TD_Unavailable.

The following table lists events that may be used with the Connection object’s GetEvent function to retrieve run time statistics from the Stream driver.

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_TRACE_OUTPUT</td>
<td>varchar</td>
<td>Specifies the name of the external file to use for tracing messages. The default setting creates a new file name with the name of the driver followed by a time stamp. Note: If a file with the specified name already exists then the file will be overwritten.</td>
</tr>
<tr>
<td>TD_WORKINGDATABASE</td>
<td>varchar</td>
<td>Specifies the name of the database used in a Teradata SQL DATABASE statement that the Stream driver sends to the Teradata Database immediately after connecting the two SQL sessions. Use this attribute to specify a default database other than the logon database.</td>
</tr>
</tbody>
</table>
## Table 24: Stream Driver Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Returned Value</th>
</tr>
</thead>
</table>
| TD_Evt_ApplyCount | A buffer containing the concatenation of these fields in the following order:  
1 a 4-byte unsigned integer that represents the statement number in the DML Group.  
2 a 1-byte character that represents the type of DML statement. Valid values are I (insert), U (update), or D (delete).  
3 a 4-byte unsigned integer representing the length of the database name in bytes.  
4 $n$-byte character string for the database name (where $n$ is the length of the database name in bytes).  
5 a 4-byte unsigned integer representing the length of the table or macro name.  
6 $n$-byte character string for the table or macro name (where $n$ is the length of the table or macro name in bytes).  
7 a 4-byte unsigned integer for the number of rows inserted, updated, or deleted. If the type was I (see field 2) then this number is the number of rows inserted.  
For more information on using TD_Evt_ApplyCount to get the number of rows inserted, updated, or deleted for each DML Group in the job, see "Obtaining the Row Count Using TD_Evt_ApplyCount" on page 119. |
| TD_Evt_CLIError   | One 4-byte integer for the CLIv2 error number and a 255 byte character buffer for the CLIv2 error text. Query this event at any time.                                                                                                                                                                                                                                                                                                                                                                   |
| TD_Evt_BufferLayout | Four 4-byte unsigned integers corresponding to the maximum buffer size, the row header size, the row length size, and the buffer trailer size. Use this information to format the buffer required for block streaming.  
**Note:** These layout values will change depending on the user environment and may be defined differently in future releases. Always obtain these values from the event method before buffering data.  
Query this event at any time. |
| TD_Evt_DBSError   | One 4-byte integer for the DBS error number and a 255 byte character buffer for the DBS error text. Query this event at any time.                                                                                                                                                                                                                                                                                                                                                                   |
| TD_Evt_ConnectStatus | Three 4-byte integers corresponding to the number of sessions requested, the number of sessions connected, and any CLIv2 error number that may have occurred during the connect process. Query this event at any time.                                                                                                                                                                                                                                                                                               |
Table 24: Stream Driver Events (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_Evt_CPUPTime</td>
<td>One 8-byte double for the CPU time of the instance in seconds. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_PackFactor</td>
<td>A pointer to an integer containing the value of the pack factor. Query this event only after initiating connection and before terminating a connection to return the pack factor for the Stream driver.</td>
</tr>
<tr>
<td>TD_Evt_RowCounts</td>
<td>Three 4-byte unsigned integers corresponding to the number of rows received, rows sent, and rows that caused DBS errors. Query this event before EndAcquisition to get the current number of rows received. The rest of the data for this event, however, will only be available after EndAcquisition.</td>
</tr>
<tr>
<td>TD_Evt_Runstats</td>
<td>Five 4-byte unsigned integers corresponding to the number of SQL statements sent to the DBS, requests sent to the DBS, rows received, rows sent, and rows that caused DBS errors. Query this event before EndAcquisition to get the current number of rows received. The rest of the data for this event, however, will only be available after EndAcquisition.</td>
</tr>
<tr>
<td>TD_Evt_ErrorTable1</td>
<td>One 4-byte unsigned integer for the number of rows in error table 1. Query this event after the first call to PutRow or PutBuffer.</td>
</tr>
<tr>
<td>TD_Evt_ExitCode</td>
<td>One 2-byte integer for the exit code of the driver. Query this event right before the driver terminates.</td>
</tr>
<tr>
<td>TD_Evt_RowsCheckpointed</td>
<td>One 4-byte unsigned integer corresponding to the total number of rows that have been received and checkpointed for the entire job. Query this event any time after initiating a Teradata PT API database connection in a single-instance job. This event will return TD_Unavailable when queried in a job containing multiple instances; this event is invalid if multiple instances are used. Note that the event data returned by this event can be used as a replacement for the checkpoint data that must be passed into the Restart method during a restart job. It is recommended to save the checkpoint data from the last successful call to the CheckPoint method in a potential restart job. However, if the checkpoint data is not saved then the event data returned by this event can be passed into the Restart method instead. This alternate method of performing checkpoint/restarts will only work with single-instance jobs.</td>
</tr>
</tbody>
</table>
Programming Considerations

The following information discusses special topics to consider when coding a Stream application.

Error Tables

The Stream driver uses a single error table that contains records rejected because of data conversion, constraint, or other errors. The APPLY statement for a load operation provides DML attributes and error option attributes that tell the Stream driver what it should do with errors. These attributes define a label and error treatment options for one or more immediately following INSERT, UPDATE, or DELETE statements. These options allow you to mark or ignore error conditions, such as duplicate rows and missing rows, etc. Marked error conditions can be directed to the error table.

These MARK/IGNORE options are:

- DUPLICATE INSERT ROWS
- DUPLICATE UPDATE ROWS
- MISSING UPDATE ROWS
- MISSING DELETE ROWS

Reusing Stream Driver Table Names

If an error table has one or more rows, it is not dropped from the Teradata Database at the end of a Stream driver job. To reuse the names specified for the error tables, use the BTEQ utility's DROP TABLE statement to remove the tables from the Teradata Database.

Limiting Errors in the Stream Driver

Use the ErrorLimit attribute to limit the number of errors captured in the error table during the Stream job.

<table>
<thead>
<tr>
<th>IF you expect your Stream job to encounter...</th>
<th>THEN specify an ERRORLIMIT value that is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>No errors or very few errors</td>
<td>low</td>
</tr>
</tbody>
</table>
Note: The ErrorLimit specification applies to each instance of the Stream application, not to all instances combined. If, for example, you set the limit to 10,000, then a single instance would have to detect that 10,000 rows were inserted into the first error table in order to terminate the job, and those 10,000 rows would have to be controlled by the sessions managed by that instance.

Processing terminates when the number of errors encountered reaches the error limit. If, for example, you expect no errors in the input data, set the error limit value to one. In this case, the job terminates when any record causes an error.

Dropping Tables During a Load

Some tables are created by the Teradata Database during the execution of a job, and others must be created by the user before the job begins. When a job is run, the log table and error table are created automatically. The error table is dropped during the Cleanup phase if no errors were detected during the job. The log table is dropped after the job completes successfully.

If a job terminates abnormally, then the log and error tables are not dropped. If you want to restart the job from scratch, you need to manually drop these tables by running a BTEQ script to drop and recreate the tables.

Caution: Care must be taken dropping the target tables manually using a BTEQ script. If something goes wrong with a Stream job, and you drop the target table manually and then try to rerun the job, you may lose the original data.

Required Privileges

The user ID that is logged in by the Stream driver must have:

- SELECT, INSERT, UPDATE, and DELETE privileges on the Stream target tables.
- SELECT and INSERT privileges on the error tables, and CREATE and DROP privileges on the database that contains the error tables.
- SELECT, INSERT, and DELETE privileges on the restart log table, and CREATE and DROP privileges on the database that contains the restart log table.

Session Limits

The values that you specify with the Stream driver TD_MIN_SESSIONS and TD_MAX_SESSIONS attributes are not the only factors that limit the number of sessions that the Stream driver establishes with the Teradata Database.

The other limiting factors are:

<table>
<thead>
<tr>
<th>IF you expect your Stream job to encounter...</th>
<th>THEN specify an ERRORLIMIT value that is...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Many errors that are considered allowable</td>
<td>high</td>
</tr>
</tbody>
</table>
• The platform limit on the maximum number of sessions per application. This value is defined in the COP Interface software file, CLISPB.DAT, under the `max_num_sess` variable.

You can use the TDP SET MAXSESSIONS command to specify a platform limit. The default TDP MAXSESS value is 1024 sessions.

• The limit of the network protocol software on network-attached systems. When the Stream driver executes, the actual session limit is determined by whichever limiting factor is encountered first.

**Obtaining the Row Count Using TD_Evt_ApplyCount**

The event TD_Evt_ApplyCount can be used to query the Stream driver for the number of inserts, updates, and deletes performed by each DML Group in the job. The index parameter in the GetEvent method is used to indicate the data’s DML Group. The index that is passed to the GetEvent method is the same index that is returned by the AddDMLGroup method for the desired DML Group. This is different from the way the Update driver returns the number of rows inserted, updated, and deleted. The Update driver uses the index parameter in the GetEvent method to signify the data’s target table. However, just like the data returned by the Update driver’s TD_Evt_ApplyCount event, the data for this event only applies to the instance making the GetEvent call.

The TD_Evt_ApplyCount event can be queried at any time during the job after Initiate and before Terminate. The data returned will correspond to the number of rows inserted, updated or deleted since the last checkpoint was taken. If no checkpoint was taken during the job then the data returned will not be meaningful until after EndAcquisition is called. The final statistics for the number of rows inserted, updated, or deleted will always be available after the EndAcquisition method returns with a TD_END_Method. This event returns a block of data for each statement in the DML Group.

The `eventData` parameter provided to the GetEvent method (See “GetEvent” on page 46) points to a concatenation of these data blocks if the method returns with TD_END_Method. If the method does not return with TD_END_Method then the `eventData` pointer is not correct. Each DML statement only has one data block filled with the information described above as long as the statement is not an upsert or merge statement, in which case there would be two data blocks filled with information. For example, any insert statement only has a data block returned with information about the number of rows inserted. This is true for update and delete statements as well. Upsert and merge DML statements have two data blocks filled with information: one for the number of updates and one for the number of inserts.

Figure 6 depicts the buffer returned by the TD_Evt_ApplyCount event. For a description of the format of the fields, see the Returned Value column in “TD_Evt_ApplyCount” on page 115.
Figure 6: Layout of Data Buffer returned by the TD_Evt_ApplyCount Method

Based on this information and the length (in bytes) provided in `eventDataLen` after the GetEvent method returns, the number of rows inserted, updated, and deleted can be obtained for each DML statement in the DML Group.

**Space Requirements and Limitations**

Always estimate the final size of the Stream target table, and make sure that the destination database on the Teradata Database has enough space to accommodate the Stream job. If the database that owns the Stream target table, log table, or error table runs out of space, the Teradata Database returns an error message and the Stream driver terminates the Stream job. When this happens, more space must be allocated to the database before the job can be restarted.

**Tuning the Pack Factor**

Packing multiple-statement requests improves network/channel efficiency by reducing the number of sends and receives between the application and the Teradata Database.

To determine the ideal pack factor to specify in the Pack attribute, first use the PackMaximum attribute by setting it to Yes. Setting this attribute to Yes on the first job run sets up iterative interactions with the Teradata Database to heuristically determine the maximum possible pack factor.

At the end of the run, this value is displayed in the Stream driver’s logged output. The value determined should then be specified in the Pack attribute on subsequent runs and PackMaximum should be set to No.

Alternatively, query the TD_Evt_PackFactor event after initiating a connection and before terminating it to retrieve the pack factor currently in use.

**Using DELETE in Import Tasks**

DELETE is a Teradata SQL statement that removes rows from a table or view that was previously identified as a target table through the use of the TD_TARGET_TABLE attribute.

The rules for using the DELETE statement in Stream import tasks are:

- You can apply DELETE statements to either a table or a view, provided that the view does not specify a join
- The number of input data records is unlimited
- You must specify equality values for all the primary index columns in the WHERE clause
- You cannot use the OR construct in the WHERE clause of a DELETE statement. Instead, use two separate DELETE statement.
The procedure for using the DELETE statement in a Stream import task is:

1. Add a DELETE statement to a DML group
2. Add the DML group to the Connection object
3. Initiate the Connection object
4. Set the DML group containing the DELETE statement using the UseDMLGroups function
5. Make at least one call to the PutRow function
   - If the DML group only contains a DELETE statement, then no data will be loaded as a result of the PutRow function call.
6. Call the EndAcquisition function
7. Terminate the Connection object

**Note:** If no row in the target table matches the DELETE statement, then the row of data sent with the DELETE statement will be put into the error table.

**Stream Driver Macro Support**

Teradata PT API supports executing macros using the Stream driver. Adhering to Teradata Parallel Data Pump protocols that are documented in the *Teradata Parallel Data Pump Reference*, note the following guidelines:

- The macro specified in the EXECUTE statement must reside on the RDBMS prior to the start of the job. For more information on creating a macro, see *SQL Data Manipulation Language*.
- The database user defined for the Stream driver job must have the EXECUTE privilege for the macro specified in the EXECUTE statement.
- The EXECUTE statement used in the DML group must follow the format documented in the *Teradata Parallel Data Pump Reference*. This is the EXECUTE or EXEC command followed by the name of the macros (the database can be specified using the dot notation database.macro) followed by one of the key words, depending on what kind of statement is executed by the macro.

The key words are:

- INSERT
- UPDATE
- DELETE
- UPSERT

- The EXECUTE statement cannot include a parameter list. The parameter list is automatically generated by the Stream driver based on the provided schema.
- The macro must have one or more parameters. The Stream driver does not support executing macros that do not have parameters.
Executing a macro in a Stream driver job

1. Add the EXECUTE statement to a DML group. See “Stream Driver Macro Support” on page 121 for guidelines.
2. Define the schema based on the macro's parameter list.
3. Add the DML group and Schema to the Connection object.
4. Set the DML group containing the EXECUTE statement using the UseDMLGroups function.
5. Call the PutRow function. The row data provided to PutRow will be used to fill out the macro's parameter list and the macro will be executed.
6. Call the End Acquisition function.
7. Terminate the Connection object.

Code Example
The following macro was defined prior to execution:

```c
create macro ins_row
{
    p1 integer,
    p2 varchar(32000)
} as(
    insert into table1
    (col1,col2)
    values(:p1,:p2);
);
```

For the Stream driver, add this EXECUTE statement to the DML group:

```c
EXECUTE ins_row INSERT;
```

And this schema is built:

```c
Schema * strSchema=new Schema("input");
strSchema->AddColumn("col1",TD_INTEGER,4);
strSchema->AddColumn("col2",TD_VARCHAR,32000);
strConn->AddSchema(strSchema);
```

Checkpoint and Restart Operations

The Stream driver is fully checkpoint restartable. Refer to the checkpoint and restart discussion in “Add Checkpoint and Restart” on page 33.

Code Example

```c
#include "connection.h"
#include "schema.h"
#include "DMLGroup.h"

using namespace teradata::client::API;

int returnValue = 0;
char* errorMsg = NULL;
```
TD_ErrorType errorType;

cout << "*** Stream Driver Example ***" << endl;

Connection *conn = new Connection();

/**********************************************
* Set Operator Type and Trace/Log Levels
**********************************************/
conn->AddAttribute(TD_SYSTEM_OPERATOR, TD_STREAM);
conn->AddAttribute(TD_TRACE_OUTPUT, "stream.txt");
conn->AddArrayAttribute(TD_TRACE_LEVEL, 2, TD_OPER_ALL, TD_OFF, NULL);

/**********************************************
* Add Attributes
**********************************************/
conn->AddAttribute(TD_USER_NAME, "user");
conn->AddAttribute(TD_USER_PASSWORD, "password");
conn->AddAttribute(TD_TDP_ID, "database");
conn->AddAttribute(TD_LOG_TABLE, "tdload_log");
conn->AddAttribute(TD_ERROR_TABLE, "tdload_e1");
conn->AddAttribute(TD_TARGET_TABLE, "tdload");
conn->AddAttribute(TD_MAX_SESSIONS, 4);
conn->AddAttribute(TD_MIN_SESSIONS, 1);
conn->AddAttribute(TD_TENACITY_HOURS, 2);
conn->AddAttribute(TD_TENACITY_SLEEP, 3);
conn->AddAttribute(TD_ROBUST, "No");

/**********************************************
* Add Schema
**********************************************/
Schema * schema = new Schema("input");
schema->AddColumn("Associate_Id", TD_INTEGER, 4);
schema->AddColumn("Associate_Name", TD_CHAR, 25);
schema->AddColumn("Salary", TD_FLOAT, 8);
schema->AddColumn("DOJ", TD_DATE, 4);
schema->AddColumn("Designation", TD_VARCHAR, 25);
schema->AddColumn("Loan_Amount", TD_DECIMAL, 4, 5, 2);
schema->AddColumn("Martial_Status", TD_TINYINT, 1);
schema->AddColumn("No_Of_Dependents", TD_BYTEINT, 1);
conn->AddSchema(schema);

/**********************************************
* Add DMLGroups
**********************************************/
TD_Index dmlGroupIndex = 0;
DMLGroup * dmlGr = new DMLGroup();
dmlGr->AddStatement("INSERT INTO tdload( :Associate_Id, :Associate_Name, :Salary, :
DOJ, :Designation, :Loan_Amount, :Martial_Status, :No_Of_Dependents);");
dmlGr->AddSerializeOn(8, "Associate_Id", "Associate_Name", "Salary", "DOJ", :
"Designation", "Loan_Amount", "Martial_Status", "No_Of_Dependents", NULL);
dmlGr->AddUseList(8, "Associate_Id", "Associate_Name", "Salary", "DOJ", "Designation",
"Loan_Amount", "Martial_Status", "No_Of_Dependents", NULL);
returnValue = conn->AddDMLGroup(dmlGr, &dmlGroupIndex);

/**********************************************
* Initiate
**********************************************/
returnValue = conn->Initiate();

if ( returnValue < TD_Error )
{
    char rowBuffer[78]; /* must include the EOL byte */
    int loadStatus = 0;
    TD_Length bytes;
    int count = 0;
    while ( loadStatus != -1 )
    {
        loadStatus = GetRowData(rowBuffer, 78); // user function - reads in a row of data from a file
        // returns -1 when end of file reached
        if ( loadStatus != -1 )
        {
            /* pick up the first 2 bytes as the row length of the indicator mode record */
            bytes = *((unsigned short *)rowBuffer);
        }
    }
}
```c
returnValue = conn->PutRow( rowBuffer + 2 , bytes);
if ( returnValue >=TD_Error )
{
cout << "PutRow failed on row " << count+1;
cout << " with status " << returnValue << endl;
loadStatus = GetRowData(NULL,0); //user function - close file
} else{
count++;
}
}
if ( returnValue < TD_Error )
{
cout << "Sent " << count << " rows" << endl;
/*******************************
* End Acquisition
*****************************/
returnValue = conn->EndAcquisition();
cout << "Acquisition completed with status " << returnValue << endl;
} else{
//Get Error Information
cout << "Error occurred during EndAcquisition" << endl;
conn->GetErrorInfo(&errorMsg,&errorType);
if ( errorMsg != NULL ){
cout << errorMsg << endl;
cout << "Type: " << errorType << endl;
} else{
cout << "No Error Info Available" << endl;
}
} else{
//Get Error Information
cout << "Error occurred during Acquisition" << endl;
conn->GetErrorInfo(&errorMsg,&errorType);
if ( errorMsg != NULL ){
cout << errorMsg << endl;
cout << "Type: " << errorType << endl;
} else{
cout << "No Error Info Available" << endl;
}
} else{
//Get Error Information
cout << "Error occurred during Initiate" << endl;
conn->GetErrorInfo(&errorMsg,&errorType);
if ( errorMsg != NULL ){
cout << errorMsg << endl;
cout << "Type: " << errorType << endl;
} else{
cout << "No Error Info Available" << endl;
}
} 
@Transactional
******************************************************************************
returnValue = conn->Terminate();
cout << "Driver Terminated with status " << returnValue << endl;
if ( returnValue >= TD_Error )
{
//Get Error Information
cout << "Error occurred during Terminate" << endl;
conn->GetErrorInfo(&errorMsg,&errorType);
if ( errorMsg != NULL ){
cout << errorMsg << endl;
cout << "Type: " << errorType << endl;
} else{
cout << "No Error Info Available" << endl;
}
```
}  
*/
/* Clean Up
 **********************************************/
 cout << "Deleting Objects" << endl;
 delete dmlGr;
 delete schema;
 delete conn;
 cout << "*** Stream Complete ***" << endl;
CHAPTER 6
Export Driver

Using Teradata FastExport Reference protocols, the Export driver is designed to use multiple Export sessions to export large volumes of data at high speed from the Teradata Database to a client application.

For a sorted answer set, redistribution of the rows occurs over the BYNET. This allows for easy recombination of the rows and data blocks when they are returned to the client in sorted order. The main topics of this chapter are:

- Attribute Definitions
- Required Attributes
- Optional Attributes
- GetEvent Queries
- Programming Considerations
- SELECT REQUESTS
- Session Limits
- Performance Factors
- Checkpoint and Restart Operations
- Code Example

Attribute Definitions

Table 26 and Table 27 define the Export driver’s required and optional attributes. This information is helpful when coding your application.

Required Attributes

Table 26: Export Driver Required Attributes

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_BUFFER_MODE</td>
<td>varchar</td>
<td>Indicates which type of export method is used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No (‘N’) = Exporting will be on a row by row basis. This is the default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Yes (‘Y’) = The GetBuffer feature will be used to export. Please see</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Export Data from a Teradata Database using GetBuffer” on page 30 for more</td>
</tr>
<tr>
<td></td>
<td></td>
<td>information.</td>
</tr>
</tbody>
</table>
Table 26: Export Driver Required Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_INSTANCE_NUM</td>
<td>integer</td>
<td>Specifies the instance number of the current instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the current instance is the master instance, then the instance number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>should be one. Required only when using multiple instances of the same</td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver in a master-slave environment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If the current instance is a slave instance, then the instance number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>should be a value greater than one.</td>
</tr>
<tr>
<td>TD_LOGSQL</td>
<td>varchar</td>
<td>Directs the Export driver to output the full Teradata SQL request in the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>trace output file with the driver’s trace is enabled. By default, when the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>driver’s trace is enabled, the Load driver outputs the Teradata SQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>request, up to 32 kilobytes, in the trace output file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Valid values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Yes ('Y') = Output the full Teradata SQL in the trace output file when</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the driver’s trace is enabled. The maximum length of the Teradata SQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is 1 megabyte.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No ('N') = Do not output the Teradata SQL in the trace output file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: When the driver’s trace is disabled, TD_LOGSQL has no effect.</td>
</tr>
<tr>
<td>TD_MAX_INSTANCES</td>
<td>integer</td>
<td>Specifies the total number of instances (master and slaves). Required only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>when using multiple instances of the same driver in a master-slave</td>
</tr>
<tr>
<td></td>
<td></td>
<td>environment.</td>
</tr>
<tr>
<td>TD_RESTARTMODE</td>
<td>integer</td>
<td>Required only before restarting and must be set to one.</td>
</tr>
<tr>
<td>TD_SELECT_STMT</td>
<td>varchar</td>
<td>Specifies a Teradata SQL SELECT statement or statements.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Refer to the Teradata SQL reference documentation for your operating</td>
</tr>
<tr>
<td></td>
<td></td>
<td>system, and enter one or more valid Teradata SQL SELECT statements for the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SelectStmt attribute.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Each statement or group of statements must be enclosed in single quotes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and must be terminated with a semicolon. Multiple statements cannot be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>added using the AddArray Attribute function. If you have multiple SQL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SELECT statements, put them together in a single string with semicolons</td>
</tr>
<tr>
<td></td>
<td></td>
<td>separating the statements and use this string as the value in the Add</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Attribute function.</td>
</tr>
<tr>
<td>TD_SYSTEM_OPERATOR</td>
<td>varchar</td>
<td>Specifies the type of driver being used (in this case TD_EXPORT).</td>
</tr>
<tr>
<td>TD_USER_NAME</td>
<td>varchar</td>
<td>Provides the user name for the Export driver logon sessions.</td>
</tr>
<tr>
<td>TD_USER_PASSWORD</td>
<td>varchar</td>
<td>Specifies the password associated with the user name.</td>
</tr>
</tbody>
</table>
## Optional Attributes

Table 27: Export Driver Optional Attributes

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_ACCOUNT_ID</td>
<td>varchar</td>
<td>Specifies the account associated with the specified user name. When omitted, this attribute defaults to the account identifier of the immediate owner database.</td>
</tr>
<tr>
<td>TD_BLOCK_SIZE</td>
<td>integer</td>
<td>Specifies the block size in bytes used when returning data to the client. The minimum is 256 bytes. The default and maximum are 64330 bytes for Teradata Database V2R6.0 and later. <strong>Note:</strong> This value cannot be larger than the row size supported by the Teradata Database.</td>
</tr>
<tr>
<td>TD_BUFFER_HEADER_SIZE</td>
<td>integer</td>
<td>Specifies the row header size allocated for each row in the data buffer returned by the GetBuffer function. The row header size must be greater than or equal to the value for the TD_BUFFER_LENGTH_SIZE attribute. The default row header size is 2. For more information on using the GetBuffer function, see “Export Data from a Teradata Database using GetBuffer” on page 30.</td>
</tr>
</tbody>
</table>
| TD_BUFFER_LENGTH_SIZE    | integer    | Specifies the row length size allocated for each row in the data buffer returned by the GetBuffer function. Valid row length size values are:  
  - 4 = Each row’s length value to be returned as an integer value.  
  - 2 = Each row’s length value to be returned as a short value. This is the default value.  
  - 0 = Causes no row length value to be returned for each row. For more information on using the GetBuffer function, see “Export Data from a Teradata Database using GetBuffer” on page 30.                                                                                       |
| TD_BUFFER_MAX_SIZE       | integer    | Specifies the total maximum size for the data buffer returned by the GetBuffer function. The total maximum size of the data buffer must be large enough to hold at least one row of data including the row header, the actual row of data, and the buffer trailer. When exporting variable length data, the largest row size possible is used when calculating the minimum total maximum size of the data buffer. The default total maximum size of the data buffer is 64,260 (roughly 64k). For more information on using the GetBuffer function, see “Export Data from a Teradata Database using GetBuffer” on page 30. |
Table 27: Export Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TD_BUFFER_TRAILER_SIZE</strong></td>
<td>integer</td>
<td>Specifies the size of the buffer trailer allocated for the data buffer returned by the GetBuffer function. The default buffer trailer size is zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For more information on using the GetBuffer function, see “Export Data from a Teradata Database using GetBuffer” on page 30.</td>
</tr>
<tr>
<td><strong>TD_CHARSET</strong></td>
<td>varchar</td>
<td>Specifies the name or code of the character set to be used for the job. On channel-attached z/OS platforms, only EBCDIC encoding is supported and is automatically selected. For the list of supported character sets, see the Extended Character Sets section in the Teradata Parallel Transporter Reference. Check <a href="http://www.info.teradata.com/">http://www.info.teradata.com/</a> for the latest version of this publication.</td>
</tr>
<tr>
<td><strong>TD_DATA_ENCRYPTION</strong></td>
<td>varchar</td>
<td>Activates full security encryption of SQL requests, responses and data. Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Off = No encryption occurs. This is the default value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• On = All SQL requests, responses, and data are encrypted.</td>
</tr>
<tr>
<td><strong>TD_DATE_FORM</strong></td>
<td>varchar</td>
<td>Specifies the DATE data type for the Export driver job. Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• IntegerDate = Integer DATE data type. This is the default value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ANSIDate = ANSI fixed-length CHAR(10) DATE data type.</td>
</tr>
<tr>
<td><strong>TD_IGNORE_MAX_DECIMAL_DIGITS</strong></td>
<td>varchar</td>
<td>Signals to the Export driver whether to ignore the <strong>TD_MAX_DECIMAL_DIGITS</strong> attribute and continue the job if the Large Decimal feature is not supported by the Teradata Database or Teradata CLlV2. This attribute has no impact when used with V2R6.2 or later Teradata Database. Valid values are:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No (or N) = The Export Driver will use the <strong>TD_MAX_DECIMAL_DIGITS</strong> attribute. The Export job terminates with an error if the <strong>TD_MAX_DECIMAL_DIGITS</strong> attribute is used and the Teradata Database level is V2R6.1 or older. This is the default value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Yes (or Y) = The Export driver will ignore the <strong>TD_MAX_DECIMAL_DIGITS</strong> attribute if the Large Decimal feature is not supported by the Teradata CLlV2 or the Teradata Database. If the Large Decimal feature is not supported then the Export driver continues processing and returns an exit code of four instead of zero to indicate a warning when the TD_EVT_EXITCODE event is queried.</td>
</tr>
</tbody>
</table>
Table 27: Export Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_LOGON_MECH</td>
<td>varchar</td>
<td>Specifies which logon mechanism to use. See your site security administrator for specific mechanism names. For a list of available mechanisms, see Security Administration. The job terminates if the attribute exceeds eight bytes.</td>
</tr>
<tr>
<td>TD_LOGON_MECH_DATA</td>
<td>varchar</td>
<td>Passes additional logon mechanism data. See your site security administrator for specific mechanism data. For more information, see Security Administration.</td>
</tr>
<tr>
<td>TD_MAX_DECIMAL_DIGITS</td>
<td>integer</td>
<td>Specifies the maximum number of decimal digits (a value for the maximum precision) to be returned by the database. Valid values range from 1 to 38 inclusive. The default value is 18.</td>
</tr>
<tr>
<td>TD_MAX_SESSIONS</td>
<td>integer</td>
<td>Specifies the maximum number of sessions to log on. This MaxSessions value must be greater than one. Specifying a value less than one terminates the job. The default is one session per available AMP. The maximum value cannot be more than the number of AMPS available. The MaxSessions value must be greater than or equal to the value of TD_MAX_INSTANCES.</td>
</tr>
<tr>
<td>TD_MIN_SESSIONS</td>
<td>integer</td>
<td>Specifies the minimum number of sessions required for the Export driver job to continue. The default is one session. The MinSessions value must be greater than one and less than or equal to the maximum number of Export driver sessions. Specifying a value less than one terminates the Export driver.</td>
</tr>
<tr>
<td>TD_MSG.Encoding</td>
<td>TD_Encoding</td>
<td>Specifies the encoding for the messages passed between Teradata PT API and a Teradata PT API application.</td>
</tr>
<tr>
<td>TD_NOTIFY_EXIT</td>
<td>varchar</td>
<td>Specifies the name of the user-defined notify exit routine. When no name is supplied, the following default names are used: libnotfyext.dll for Windows, libnotfyext.sl for HP-UX platforms, libnotfyext.so for Linux and all other UNIX platforms For detailed information on the Notify feature, see the Export operator chapter in the Teradata Parallel Transporter Reference. Check <a href="http://www.info.teradata.com/">http://www.info.teradata.com/</a> for the latest version.</td>
</tr>
</tbody>
</table>
### TD_NOTIFY_LEVEL

**Type**: varchar

Indicates the level at which certain events are reported.

The valid settings are:

- **Off**: No notification of events is provided. This is the default setting.
- **Low**: Notification is provided for these events:
  - Initialize
  - CLIv2/DBS Error
  - Exit
  - File or OUTMODE Open
  - Statement Fetch Begin and End
- **Medium**: Notification is provided for all the events except for:
  - File or OUTMODE Open
  - Statement Fetch Begin and End
- **High**: Notification is provided for all events.

### TD_NOTIFY_METHOD

**Type**: varchar

Specifies the method for reporting events.

Valid methods are:

- **None**: No event logging is done. This is the default method.
- **Msg**: This method sends the events to a log.
- **Exit**: This method sends the events to a user-defined notify exit routine and to the system log.

On Windows, the events are sent to an EventLog that can be viewed using the Event Viewer. The messages are sent to the application log.

On AIX, HP-UX, Linux, and Solaris platforms, the destination of the events is specified in the `/etc/syslog.conf` file.

### TD_NOTIFY_STRING

**Type**: varchar

Specifies a user-defined string that precedes all messages sent to the system log. This string is also sent to the user-defined notify exit routine.

The maximum length of the string is:

- 80 bytes, if NotifyMethod is Exit.
- 16 bytes, if NotifyMethod is Msg.
Table 27: Export Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TD_OUTLIMIT</strong></td>
<td>integer</td>
<td>Limits the number of rows that the Export driver exports. Default setting is: No limit. Valid values are: Any non-negative integer value.</td>
</tr>
<tr>
<td><strong>Note:</strong> In a multiple instance environment, this limit applies only to the particular instance on which it is set, and not to the overall job. For example, in an Export job with one master instance and one slave instance, if the outlimit for both the master and the slave is set to 10 rows, then the master instance stops after exporting 10 rows and the slave instance stops after exporting 10 rows.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TD_QUERY_BAND_SESS_INFO</strong></td>
<td>varchar</td>
<td>Provides a user-defined query band expression that is set for every SQL session connected by the Teradata PT API driver. The following is an example of a valid query band expression: ( a=1; b=2; c=3; d=4; ) If the TD_QUERY_BAND_SESS_INFO is set, the following request will be sent by every SQL session connected by the Teradata PT API Export driver: ( \text{SET QUERY_BAND =} ' \text{&lt;User-Defined Query Band Expression&gt;} \text{'} \text{ FOR SESSION;} ) Setting the TD_QUERY_BAND_SESS_INFO attribute in jobs running against non-supported versions of the Teradata Database causes a non-fatal error. No error code is returned to the user during initiation and the job is allowed to proceed. The log table will not be dropped at the end of the job and the TD_Evt_ExitCode event returns a warning value of four instead of the normal success value of zero if queried. In this case, error information can be found in the trace file.</td>
</tr>
<tr>
<td><strong>TD_SPOOLMODE</strong></td>
<td>varchar</td>
<td>Specifies whether to use spool or not while running the current Export job. Valid values are: &quot;Spool&quot; = Use Spool. This is the default setting. &quot;NoSpool&quot; = Do not use Spool. <strong>Note:</strong> This value is valid only if DBS supports NoSpool. If DBS does not support NoSpool, it uses Spool instead. &quot;NoSpoolOnly&quot; = Do not use Spool in any case. <strong>Note:</strong> If the DBS does not support NoSpool, it terminates the job with an error.</td>
</tr>
</tbody>
</table>
### Table 27: Export Driver Optional Attributes (continued)

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_TDP_ID</td>
<td>varchar</td>
<td>Specifies the name of the Teradata Database machine.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The dbname can be up to 256 characters and can be a domain server name.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• TDP stands for Teradata Director Program and is specified for mainframe z/OS platforms.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If you do not specify the value for the TdpId attribute, the driver uses the default TdpId established for the user by the system administrator.</td>
</tr>
<tr>
<td>TD_TENACITY_HOURS</td>
<td>integer</td>
<td>Specifies the number of hours the Export driver attempts to log on when the maximum number of load and export operations are already running on the Teradata Database.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The default value is four hours. To enable the tenacity feature, this value must be greater than zero.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Specifying a value of zero will disables the tenacity feature.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Specifying a value less than zero terminates the Export driver.</td>
</tr>
<tr>
<td>TD_TENACITY_SLEEP</td>
<td>integer</td>
<td>Specifies the number of minutes the Export driver pauses before attempting to log on when the maximum number of load and export operations are already running on the Teradata Database.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The default is six minutes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• This value must be greater than one. If you specify a value less than one, the Export driver responds with an error message and terminates the job.</td>
</tr>
</tbody>
</table>
Chapter 6: Export Driver

Attribute Definitions

**TD_TRACE_LEVEL**

*Note:* The TraceLevel attribute is an internal diagnostic aid. Use only if instructed to by Teradata support. TD_OFF should always be specified.

<table>
<thead>
<tr>
<th>Attribute and Type</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| TD_TRACE_LEVEL       | integer | Specifies the types of diagnostic messages written by each instance of the driver to an external log file. The diagnostic trace function provides more detailed information in the log file (including the version number) to aid in problem tracking and diagnosis. Use the AddArray attribute method to specify the two types of tracing levels: driver tracing and infrastructure tracing.  
  *TD_OFF* is the default setting for both driver tracing and infrastructure tracing. No external log file is produced unless this default is changed. Specifying *TD_OFF* for both driver tracing and infrastructure tracing is the same as disabling tracing. If the TraceLevel is set to any value other than *TD_OFF*, an external log file is created for each instance of the driver.  
  The trace levels for driver tracing are:  
  - *TD_OFF* = Disables driver tracing.  
  - *TD_OPER* = Activates the tracing function for driver specific activities. The absence of any value for the PauseAcquisition attribute means that the Export driver job will execute both the acquisition phase and the application phase without pausing. This will distribute all of the rows that were sent to the Teradata Database during the acquisition phase to their final destination on the AMPs. Table 1 lists which drivers have the Pause Acquisition attribute.  
  - *TD_OPER_CLI* = Activates the tracing function for CLIv2-related activities (interaction with the Teradata Database).  
  - *TD_OPER_NOTIFY* = Activates the tracing function for activities related to the Notify feature.  
  - *TD_OPER_OPCOMMON* = Activates the tracing function for activities involving the opcommon library.  
  - *TD_OPER_ALL* = Activates tracing for all of the above activities.  
  The trace levels for infrastructure tracing should only be used when you are directed to by Teradata support. *TD_OFF*, which disables infrastructure tracing, should always be specified. |
| TD_TRACE_OUTPUT      | varchar | Specifies the name of the external file used for tracing messages. The default setting creates a new file name with the name of the driver followed by a time stamp.  
  *Note:* If a file with the specified name already exists, that file will be overwritten. |
| TD_WORKINGDATABASE   | varchar | Specifies the name of the database used in the Teradata SQL DATABASE statement sent by the Export driver to the Teradata Database immediately after connecting the two SQL sessions. Use this attribute to specify a default database other than the logon database. |
GetEvent Queries

All events must be queried after the driver has initiated and before it has terminated. Events queried before their data is available yet will return TD_Unavailable. The following table lists events that may be used with the Connection object’s GetEvent function to retrieve run time statistics from the Export driver.

<table>
<thead>
<tr>
<th>Event</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_Evt_BlockCount</td>
<td>One 4-byte integer for the total number of blocks returned from all of the SELECT requests. Query this event after the first call to GetRow has returned TD_Success.</td>
</tr>
<tr>
<td>TD_Evt_CLIError</td>
<td>One 4-byte integer for the CLIv2 error number and a 255 byte character buffer for the CLIv2 error text. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_ConnectStatus</td>
<td>Three 4-byte integers corresponding to the number of sessions requested, the number of sessions connected, and any CLIv2 error number that may have occurred during the connect process. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_CPUTime</td>
<td>One 8-byte double for the CPU time of the instance in seconds. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_DBSError</td>
<td>One 4-byte integer for the DBS error number and a 255 byte character buffer for the DBS error text. Query this event at any time.</td>
</tr>
<tr>
<td>TD_Evt_ExitCode</td>
<td>One 2-byte integer for the driver exit code. Query this event right before the driver terminates.</td>
</tr>
<tr>
<td>TD_Evt_ExportCount</td>
<td>One 4-byte unsigned integer for the total number of rows exported so far. Query this event any time after the first call to GetRow has returned TD_Success. If this event is queried after GetRow has returned TD_END_Method then the integer represents the total number of rows exported for the job.</td>
</tr>
<tr>
<td>TD_Evt_Version</td>
<td>A pointer to a character string containing the Teradata PT API version followed by a pointer to a character string containing the operator version. Query this event at any time. The operator version is available only after the driver has been initiated.</td>
</tr>
</tbody>
</table>

PutEvent Modifiers

All PutEvent modifiers must be used after the driver has initiated and before it has terminated. TD_Unavailable is returned when a modifier cannot be used. The following table lists
modifies that are used with the Connection object’s PutEvent function to modify the Export driver.

Table 29: PutEvent Modifier Expected Data

<table>
<thead>
<tr>
<th>PutEvent Modifier</th>
<th>Expected Input Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_Evt_BufferLayout</td>
<td>Buffer containing four 4-byte unsigned integers corresponding to the maximum buffer size, the row header size, the row length size, and the buffer trainer size. Values are used to format the buffer of data returned by the Export driver’s GetBuffer function. This modifier can be used anytime up until the first call to the GetBuffer function. <strong>Note:</strong> Event data returned by the Load, Update, and Stream driver’s TD_Evt_BufferLayout event can be directly passed in for this modifier.</td>
</tr>
</tbody>
</table>

**Programming Considerations**

These sections discuss the items you should consider when coding an Export application.

- **SELECT REQUESTS**
- **Session Limits**
- **Performance Factors**
- **Checkpoint and Restart Operations**

A Code Example follows these topics.

**SELECT REQUESTS**

A SELECT request is one or more Teradata SQL SELECT statements that may be optionally preceded by a LOCKING modifier. A SELECT request can have multiple SELECT statements.

When creating SELECT requests for an Export job:
SELECT Request Restrictions
Export SELECT requests cannot:

- Specify a USING modifier
• Access non-data tables, such as SELECT DATE or SELECT USER.
• Be satisfied by a single AMP, such as SELECT statement with a constraint containing an equality condition on the primary index or unique secondary index columns of a table.
• Contain character large object (CLOB) or binary large object (BLOB) data type.
• Use a WITH option to generate total or subtotal response rows. This option is used for report generation, and it produces response rows for the aggregations that are not in the same format as the detail rows.
• Utilize variable substitution

Other than these restrictions, the Teradata Database parses and processes SELECT statements from the Export driver as it would from any other data access facility. For a complete description of the SELECT statement, refer to the Teradata SQL reference documentation for your operating system.

Session Limits

The values that are specified with the Export driver TD_MAX_SESSIONS and TD_MIN_SESSIONS attributes are not the only factors limiting the number of sessions that the Export driver establishes with the Teradata Database.

The other limiting factors are:
• The Teradata Database limit of one session per AMP.
• The platform limit on the maximum number of sessions per application. This value is defined in the COP Interface software file, clispb.dat, under the max_num_sess variable. You can use the TDP SET MAXSESSIONS command to specify a platform limit. The default TDP MAXSESS value is 1024 sessions.
• The limit of the network protocol software on network-attached systems. When the Export driver executes, the actual session limit is determined by whichever limiting factor is encountered first.

Note: The Export driver will terminate with an error message if one or more instance cannot log on.

Performance Factors

Performance can vary depending on the number of sessions that the Export driver is using. Using too few sessions can limit the data throughput; using too many sessions involves increased session management overhead which can slow down the job. The latter can occur when running against a large Teradata Database configuration as the Export driver’s default setting is to have one session per AMP.

There is no general method to determine the optimal number of sessions to use since there are several factors to consider such as the performance and workload of the Teradata Database, the network performance, and the volume of data being processed. In many cases, using two or four sessions will result in the best performance, but users are encouraged to experiment with several different session configurations to determine the best setting for each Teradata Database configuration.
Checkpoint and Restart Operations

The checkpoint and restart operations are not useful for the Export applications. Refer to the checkpoint and restart discussion in Add Checkpoint and Restart.

Code Example

```c
#include "connection.h"
#include "schema.h"
#include "DMLGroup.h"

using namespace teradata::client::API;

int returnValue = 0;
char* errorMsg = NULL;
TD_ErrorType errorType;

cout << "*** Export Driver Example ***" << endl;
Connection *conn = new Connection();

//************ Set Operator Type and Trace/Log Levels ************
conn->AddAttribute(TD_SYSTEM_OPERATOR,TD_EXPORT);
conn->AddArrayAttribute(TD_TRACE_LEVEL,2,TD_OPER_ALL,TD_OFF,NULL);

//************ Add Attributes ************
conn->AddAttribute(TD_USER_NAME,"user");
conn->AddAttribute(TD_USER_PASSWORD,"password");
conn->AddAttribute(TD_SELECT_STMT,"sel Associate_Name from tdload;");
conn->AddAttribute(TD_TDP_ID,"database");

//************ Add Schema ************
Schema * schema = new Schema("output");
conn->AddColumn("Associate_Name",TD_CHAR,25);
conn->AddSchema(schema);

//************ Initiate ************
returnValue = conn->Initiate();

cout << "Driver Initiated with status " << returnValue << endl;

if ( returnValue < TD_Error )
{
    //************ Acquisition ************
    int count = 0;
    char* data;
    TD_Length dataLen = 0;
    int exportStatus = 0;

    while( exportStatus != -1 )
    {
        returnValue = conn->GetRow(&data, &dataLen);
        if ( returnValue >= TD_Error )
        {
            cout << "GetRow failed on row " << count+1;
        }
        cout << " with status " << returnValue << endl;
        exportStatus = WriteRowData(NULL,0); //user function - close file
        if (exportStatus <= 0)
            break;
        count++;
    }
}
```
```cpp
} else if (returnValue == TD_END_Method) {
    cout << "End of Data Reached" << endl;
    exportStatus = WriteRowData(NULL, 0); // user function - close file
    // returns -1 when closed
    count++;
} else {
    exportStatus = WriteRowData(data, dataLen); // user function - writes
    // data to external file
    // returns 0 if successful
    count++;
}
}

cout << "Export completed with status " << returnValue << endl;
if (returnValue < TD_Error) {
    cout << "Export completed successfully" << endl;
} else {
    // Get Error Information
    cout << "Error occurred during Export" << endl;
    conn->GetErrorInfo(&errorMsg, &errorType);
    if (errorMsg != NULL) {
        cout << errorMsg << endl;
        cout << "Type: " << errorType << endl;
    } else {
        cout << "No Error Info Available" << endl;
    }
}

} else {
    // Get Error Information
    cout << "Error occurred during Initiate" << endl;
    conn->GetErrorInfo(&errorMsg, &errorType);
    if (errorMsg != NULL) {
        cout << errorMsg << endl;
        cout << "Type: " << errorType << endl;
    } else {
        cout << "No Error Info Available" << endl;
    }
}

/**********************************************
* Terminate
**********************************************

returnValue = conn->Terminate();
cout << "Driver Terminated with status " << returnValue << endl;
if (returnValue >= TD_Error) {
    // Get Error Information
    cout << "Error occurred during Terminate" << endl;
    conn->GetErrorInfo(&errorMsg, &errorType);
    if (errorMsg != NULL) {
        cout << errorMsg << endl;
        cout << "Type: " << errorType << endl;
    } else {
        cout << "No Error Info Available" << endl;
    }
}

/**********************************************
* Clean Up
**********************************************

cout << "Deleting Objects" << endl;
delete schema;
delete conn;

cout << "*** Export Complete ***" << endl;
```
CHAPTER 7
Parallelism Enabling Protocol

The Teradata PT API parallelism-enabling protocol can be used in multi-node, multi-process, and multi-threaded applications. This chapter describes a Teradata PT API Load parallel application in these main sections:

- Overview
- Application
- Parallel Communication Area
- Parallel Processing Return Codes
- Special Parallel Parameters
- Parallel Checkpoint and Restart
- Parallel Errors
- Parallel Run-Time Statistics
- Code Example

Overview

Teradata PT API allocates the sessions as evenly as possible among the master and the slave instances. Each instance is connected with the number of sessions equaling the max-session value divided by the number of instances. Multiple parallel instances are used to load and export rows to and from the Teradata Database target tables. For example, Teradata PT API load parallel applications can load data into the same table on different nodes, as shown in the following illustration:

Figure 7: Master and Slave Parallel Relationships during Load and Export
**Note:** Multiple threads should not share the same Teradata PT API Connection object.

The same parallelism-enabling protocol is supported in all four Teradata PT API drivers to provide a consistent application interface. However, since the Stream driver does not require the Teradata Database to lock the target tables that are being updated, the choice can be made to run multiple parallel Stream application jobs instead of multiple instances in multi-node, multi-process, and multi-threaded applications.

The master and the slaves must run on the client machines that have the same endian mode (big-endian mode or little-endian mode). For channel-attached z/OS platforms, the master and slaves must all run on channel-attached z/OS platforms.

## Application

All master and slave applications must be synchronized:

- After receiving a TD_SYNC_Barrier or TD_SYNC_TELINFO status code.
- All instances should enter and exit the same following methods together:
  - Initiate
  - Checkpoint/Restart
  - EndAcquisition
  - ApplyRows
  - Terminate

Because of the synchronization, the instance on the slower machine will affect the instance on the faster machine. It is strongly recommended that you run all of your instances on machines that have the same power throughput.

## Parallel Communication Area

The Teradata PT API communication area, referred to as TELINFO, should be used for communication across nodes, among processes, and threads. It contains session information for linking the parallel instances into one job.

The master application also needs to distribute the Teradata PT API communication area (TELINFO) to the slave applications or copy it to the slave memory.

## Parallel Processing Return Codes

Teradata PT API supplies the following return codes for two types of synchronization:

- **TD_SYNC_TELINFO** - Synchronize the TELINFO communication area from the master to the slave instances, and do not proceed until all instances reach this barrier.
- **TD_SYNC_Barrier** - Do not proceed until all instances reach this barrier.

The Teradata PT API parallel synchronization is master driven. All slave instances will receive the same synchronization return code in the same phase. The master, however, may receive a different synchronization return code. In these cases, the action taken should follow the
synchronization code returned by the master. Table 31 lists possible synchronization return code combinations and the proper action to take.

Table 31: Synchronization Return Code Combinations and Suggested Actions

<table>
<thead>
<tr>
<th>Received by Master</th>
<th>Received by Slave</th>
<th>Suggested Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD_SYNC_Barrier</td>
<td>TD_SYNC_Barrier</td>
<td>Call the method again which returned the barrier code after all instances reach this barrier.</td>
</tr>
<tr>
<td>TD_SYNC_Barrier</td>
<td>TD_END_Method</td>
<td>Master calls the method again which returned the barrier code. Slave(s) wait to call the next method until Master receives TD_END_Method.</td>
</tr>
<tr>
<td>TD_SYNC_TELINFO</td>
<td>TD_SYNC_Barrier</td>
<td>Master calls the GetTELINFO function to get the TELINFO area and passes a copy of the TELINFO area to each of the slaves. Slave(s) call the PutTELINFO function to process the master’s TELINFO area. Master and slave(s) call the method again which returned the barrier code after all instances reach this barrier and after TELINFO area has been synchronized.</td>
</tr>
<tr>
<td>TD_SYNC_TELINFO</td>
<td>TD_SYNC_TELINFO</td>
<td>Master calls the GetTELINFO function to get the TELINFO area and passes a copy of the TELINFO area to each of the slaves. Slave(s) call the PutTELINFO function to process the master’s TELINFO area. Master and slave(s) call the method again which returned the barrier code after all instances reach this barrier and after TELINFO area has been synchronized.</td>
</tr>
<tr>
<td>TD_END_Method</td>
<td>TD_END_Method</td>
<td>Proceed to next method after all instances receive TD_END_Method.</td>
</tr>
</tbody>
</table>

**Usage Notes**

- The master instance job must be started before starting the slave instance jobs
- The CLIv2 environment variable \textit{THREADONOFF} must be set to one for running the Teradata PT API multi-threaded application. Refer to the \textit{Teradata Call-Level Interface Version 2 Reference for Network-Attached Systems} for details about this variable.

**Special Parallel Parameters**

The following attributes are required for both the master and slave instances in addition to the normally required attributes:

- \texttt{TD_INSTANCE_NUM}
- \texttt{TD_MAX_INSTANCES}
The master’s value for TD_INSTANCE_NUM is one and the slaves’ values for TD_INSTANCE_NUM starts at two and increments with each new slave. An instance’s number is used to access its designated area within the TELINFO area. No two instances can have the same value for TD_INSTANCE_NUM. The value for TD_INSTANCE_NUM cannot be greater than the value for TD_MAX_INSTANCES.

In a multiple instance environment, only the master uses the TD_MAX_SESSIONS and TD_MIN_SESSIONS attributes. The master instance will distribute the sessions amongst itself and the slave instances. The number of sessions needs to be greater than or equal to the number of instances. If no session preferences are specified, the default will be one session per instance.

Beyond these attributes, all other attributes, schemas, and DML groups must be the same for both the master and the slave instances.

**Parallel Checkpoint and Restart**

All instances must call the Checkpoint function in unison. While only the master instance will issue the actual checkpoint to the database, all the slave instances must call the Checkpoint function at the same time to ensure that all rows loaded using the PutRow function or the PutBuffer function are sent to the database prior to the Checkpoint.

Checkpoint data for each instance must be copied and stored separately. Each instance must use its own checkpoint data when restarting. See “Add Checkpoint and Restart” on page 33 for more information.

**Parallel Errors**

Errors can be reported by both master and slave instances. If an error occurs, the instance which discovered the error is responsible for calling the GetErrorInfo function to retrieve the detailed error information. If one or more instances fail, then all instances need to terminate immediately.

If any of the instances, master or slave, receive the TD_Call_EndAcq status code, all instances should immediately proceed to call the EndAcquisition function. The TD_Call_EndAcq status code indicates that there is an error which requires that the acquisition method be aborted. Upon completing the calls to the EndAcquisition function, the instance(s) which received the TD_Call_EndAcq status code will be returned an error code for the problem which caused the acquisition method to abort. At this time, these instances will be able to retrieve additional error information relating to this error code through the use of the GetErrorInfo function. After calling the EndAcquisition function and retrieving error information, all instances should proceed to call the Terminate function and end the job.

**Parallel Run-Time Statistics**

The run-time statistics returned by the GetEvent function pertain only to that particular instance and not to the overall job. The master and slave instances may call the GetEvent method asynchronously.
The following is a simple example of how to utilize Teradata PT API in a parallel environment. The code is not intended for actual use but rather to highlight the keys to the Teradata PT API parallel process. The architecture of the example could be implemented in either a multi-threaded or multi-process environment.

This is a multi-instance example using the Load driver. The structure of the example involves two components: a master component and a slave component.

The master component has two responsibilities: handling its designated portion of the workload (loading a subset of the total rows) and maintaining the synchronization between all of the instances.

The following is the main execution method for the master instance component:

```cpp
#include "connection.h"
#include "schema.h"
#include "DMLGroup.h"
using teradata::client::API;

int returnValue = 0;
Connection* mConnection = new Connection();
/******************************
* Add Connection Parameters
*******************************/
mConnection->AddAttribute(TD_SYSTEM_OPERATOR,TD_LOAD);
/* Add Attributes */

// Master Instance Specific Attributes
// TD_INSTANCE_NUM: Master is always "1"
// TD_MAX_INSTANCES: Total number of instances including master
// TD_MAX_SESSIONS: Number of sessions are divided between master
// and slaves
// TD_MIN_SESSIONS
/* Add Schema */
/* Add DMLGroups */

// Note: Except for the special attributes listed above, both master
// and slave instances should have exactly the same Connection
// Parameters.
/******************************/
* Initiate
******************************/

while ( returnValue != TD_END_Method ){
    returnValue = mConnection->Initiate();
    switch ( returnValue ){
        case TD_SYNC_Barrier:
            // 1) Wait for all slaves to signal barrier
            // 2) Tell slaves to proceed
            // 3) Continue
            break;
        case TD_SYNC_TELINFO:
            // 1) Wait for all slaves to signal barrier
            // 2) Use GetTELINFO to retrieve TELINFO area
            // 3) Pass copy of TELINFO area to slaves
            // 4) Tell slaves to proceed
            // 5) Continue
            break;
    }
}
```
case TD_END_Method:
  //Method is Complete
  break;
default:
  //When errors occur:
  // 1) Tell slaves to call Terminate - all instances
  //     must Terminate synchronously. See Terminate
  //     setup below.
  // 2) After Terminate, get error info
  // 3) Quit or restart
  return TD_ERROR;
}

//Wait for slaves to finish, then continue
/**********************************************************
* Acquisition
**********************************************************/
char rowBuffer[256];
unsigned short rowLength = 0;
bool doneExporting = false;
rcGetrow = 0;
returnValue = 0;
while (!doneExporting){
  //Get Row For Load
  rcGetrow = getRow( rowBuffer );
  if (rcGetrow != TD_END_Method){
    //Load Row
    rowLength = *((unsigned short *)rowBuffer);
    returnValue = PutRow( rowBuffer+2, rowLength );
    switch ( returnValue ){
      case TD_Success:
        //Continue to Next Row
        break;
      default:
        //When errors occur:
        // 1) Tell slaves to call Terminate - all instances
        //     must Terminate synchronously. See Terminate
        //     setup below.
        // 2) After Terminate, get error info
        // 3) Quit or restart
        return TD_ERROR;
    }
  }else {
    /* End Acquisition Phase */
    doneExporting = true;
    returnValue = mConnection->EndAcquisition();
  }
  switch ( returnValue ){ 
    case TD_SYNC_Barrier:
      // 1) Wait for all slaves to signal barrier
      // 2) Tell slaves to proceed
      // 3) Continue
      break;
    case TD_SYNC_TELINFO:
      // 1) Wait for all slaves to signal barrier
      // 2) Use GetTELINFO to retrieve TELINFO area
      // 3) Pass copy of TELINFO area to slaves
      // 4) Tell slaves to proceed
      // 5) Continue
      break;
    case TD_END_Method:
      //Acquisition Complete
      break;
  }
default:
  // When errors occur:
  // 1) Tell slaves to call Terminate – all instances
  //    must Terminate synchronously. See Terminate
  //    setup below.
  // 2) After Terminate, get error info
  // 3) Quit or restart
  return TD_ERROR;
}
}

// Wait for slaves to finish, then continue
******************************************************************************
* Application
******************************************************************************
returnValue = 0;
while ( returnValue != TD_END_Method ) {
  returnValue = mConnection->ApplyRows();
  switch ( returnValue ) {
    case TD_SYNC_Barrier:
      // 1) Wait for all slaves to signal barrier
      // 2) Tell slaves to proceed
      // 3) Continue
      break;
    case TD_SYNC_TELINFO:
      // 1) Wait for all slaves to signal barrier
      // 2) Use GetTELINFO to retrieve TELINFO area
      // 3) Pass copy of TELINFO area to slaves
      // 4) Tell slaves to proceed
      // 5) Continue
      break;
    case TD_END_Method:
      // Method is Complete
      break;
    default:
      // When errors occur:
      // 1) Tell slaves to call Terminate – all instances
      //    must Terminate synchronously. See Terminate
      //    setup below.
      // 2) After Terminate, get error info
      // 3) Quit or restart
      return TD_ERROR;
  }
}
// Wait for slaves to finish, then continue
******************************************************************************
* Terminate
******************************************************************************
returnValue = 0;
while ( returnValue != TD_END_Method ) {
  returnValue = mConnection->Terminate();
  switch ( returnValue ) {
    case TD_SYNC_Barrier:
      // 1) Wait for all slaves to signal barrier
      // 2) Tell slaves to proceed
      // 3) Continue
      break;
    case TD_SYNC_TELINFO:
      // 1) Wait for all slaves to signal barrier
      // 2) Use GetTELINFO to retrieve TELINFO area
      // 3) Pass copy of TELINFO area to slaves
      // 4) Tell slaves to proceed
      // 5) Continue
      break;
    case TD_END_Method:
      // Method is Complete
      break;
The slave instance has two responsibilities: handling its designated workload (loading a subset of the total rows) and reporting synchronization codes to the master and following the master’s instructions.

The following is the main execution method of the slave component:

```cpp
#include "connection.h"
#include "schema.h"
#include "DMLGroup.h"

using teradata::client::API;

int returnValue = 0;
Connection* mConnection = new Connection();
/** Add Connection Parameters
 * **********************************************
 * mConnection->AddAttribute(TD_SYSTEM_OPERATOR, TD_LOAD);
 *
 * Slave Instance Specific Attributes
 * TD_INSTANCE_NUM: Slave number must be greater than 1 and
 * no slave can have same instance number as another instance.
 * TD_MAX_INSTANCES: Total number of instances including master
 * This must be set correctly as the slave will
 * use this when accessing TELINFO area.
 */

/** Add Attributes */

/** Add Schema */

/** Add DMLGroups */

 /** Note: Except for the special attributes listed above, both master
 // and slave instances should have exactly the same Connection
 // Parameters. */

/** Initiate
 * **********************************************

while ( returnValue != TD_END_Method ){
    returnValue = mConnection->Initiate();
    switch ( returnValue ){
        case TD_SYNC_Barrier:
            // 1) Signal barrier to master
            // 2) Wait for master signal
            // 3) Continue
            break;
        case TD_SYNC_TELINFO:
            // 1) Signal barrier to master
            // 2) Use PutTELINFO to set the TELINFO
            // area passed by the master.
            // 5) Continue
            break;
```

```
case TD_END_Method:
    // Method is Complete
    break;
default:
    // When errors occur:
    // 1) Signal error to master - all instances
    //    must Terminate synchronously. See Terminate
    //    setup below.
    // 2) After Terminate, get error info
    // 3) Quit or restart
    return TD_ERROR;
}
/* Wait for master to finish, then continue */
/**********************************************
* Acquisition
***********************************************/
char rowBuffer[256];
unsigned short rowLength = 0;
bool doneExporting = false;
rcgetrow = 0;
returnValue = 0;
while (!doneExporting){
    // Get Row For Load
    rcGetrow = getRow( rowBuffer );
    if (rcGetrow != TD_END_Method){
        // Load Row
        rowLength = *((unsigned short *)rowBuffer);
        returnValue = PutRow( rowBuffer+2, rowLength );
        switch ( returnValue ){
            case TD_Success:
                // Continue to Next Row
                break;
            default:
                // When errors occur:
                // 1) Signal error to master - all instances
                //    must Terminate synchronously. See Terminate
                //    setup below.
                // 2) After Terminate, get error info
                // 3) Quit or restart
                return TD_ERROR;
        }
    }else {
        /* End Acquisition Phase */
        doneExporting = true;
        returnValue = mConnection->EndAcquisition();
        switch ( returnValue ){
            case TD_SYNC_Barrier:
                // 1) Signal barrier to master
                // 2) Wait for master signal
                // 3) Continue
                break;
            case TD_SYNC_TELINFO:
                // 1) Signal barrier to master
                // 2) Use PutTELINFO to set the TELINFO
                // area passed by the master.
                // 5) Continue
                break;
            case TD_END_Method:
                // Acquisition Complete
                break;
        }
    }
}
default:
    //When errors occur:
    // 1) Signal error to master - all instances
    // must Terminate synchronously. See Terminate
    // setup below.
    // 2) After Terminate, get error info
    // 3) Quit or restart

    return TD_ERROR;
}
}

//Wait for master to finish, then continue
/**********************************************
* Application
**********************************************/
returnValue = 0;
while ( returnValue != TD_END_Method ) {
    returnValue = mConnection->ApplyRows();
    switch ( returnValue ){
        case TD_SYNC_Barrier:
            // 1) Signal barrier to master
            // 2) Wait for master signal
            // 3) Continue
            break;
        case TD_SYNC_TELINFO:
            // 1) Signal barrier to master
            // 2) Use PutTELINFO to set the TELINFO
            // area passed by the master.
            // 5) Continue
            break;
        case TD_END_Method:
            //Method is Complete
            break;
        default:
            //When errors occur:
            // 1) Signal error to master - all instances
            // must Terminate synchronously. See Terminate
            // setup below.
            // 2) After Terminate, get error info
            // 3) Quit or restart

            return TD_ERROR;
    }
}

//Wait for master to finish, then continue
/**********************************************
* Terminate
**********************************************/
returnValue = 0;
while ( returnValue != TD_END_Method ) {
    returnValue = mConnection->Terminate();
    switch ( returnValue ){
        case TD_SYNC_Barrier:
            // 1) Signal barrier to master
            // 2) Wait for master signal
            // 3) Continue
            break;
        case TD_SYNC_TELINFO:
            // 1) Signal barrier to master
            // 2) Use PutTELINFO to set the TELINFO
            // area passed by the master.
            // 5) Continue
            break;
        case TD_END_Method:
            //Method is Complete
            break;
default:
    // When errors occur:
    // 1) Signal error to master - all instances
    //    must Terminate synchronously.
    // 2) After Terminate, get error info
    // 3) Quit or restart
    return TD_ERROR;
}

/***********************************************/
/* Clean Up */
/***********************************************/
//delete Schema objects
//delete DMLGroup objects
delete mConnection;
return returnValue;
Chapter 7: Parallelism Enabling Protocol
Code Example
Teradata PT API can be called from within an external stored procedure (XSP). An XSP is a code module, written in C or C++, which resides on the Teradata Database and is executed using the SQL CALL statement. Using Teradata PT API in an XSP provides a potential performance improvement by minimizing data transfer between the client and the database server.

Calling the Teradata PT API from within an XSP is only supported when using a 12.0 or later version of the Teradata Database. Using the Teradata PT API in a user-defined function (UDF) is not currently supported.

This chapter contains these main sections:

- Coding a Teradata PT API External Stored Procedure
- Building a Teradata PT API External Stored Procedure
- Installing a Teradata PT API External Stored Procedure
- Executing a Teradata PT API External Stored Procedure
- Getting Started with the External Stored Procedure Example

**Coding a Teradata PT API External Stored Procedure**

There are no special coding steps for calling the Teradata PT API from an XSP. As in all applications currently using Teradata PT API, the XSP must include the Teradata PT API header files and follow the standard Teradata PT API coding conventions described in this programmer guide.

Detailed instructions on how to code an XSP are in the External Stored Procedure chapter of *SQL External Routine Programming*. See “Additional Information” on page 7 for more information on obtaining this Teradata publication.
Building a Teradata PT API External Stored Procedure

The dynamic library containing the Teradata PT API XSP should be built before installing the Teradata PT API XSP on the Teradata Database. The Teradata PT API package must be installed prior to building an XSP which uses the Teradata PT API.

Installing a Teradata PT API External Stored Procedure

The following sections provide an overview and make reference to other documentation to help with the tasks for installing a Teradata PT API XSP onto a Teradata Database.

- Installing the Teradata PT API onto the Teradata Database
- Configuring the Teradata Database’s Environment
- Using the CREATE PROCEDURE SQL Statement

Installing the Teradata PT API onto the Teradata Database

Using the Teradata PT API in an XSP requires that the Teradata PT API product and all its dependencies be installed on the machine running the Teradata Database. Refer to the Teradata Tools and Utilities installation guides for more details.

Configuring the Teradata Database’s Environment

To call the Teradata PT API from an XSP, configure the Teradata Database’s environment to register the necessary Teradata PT API libraries. Depending on the operating system that the Teradata Database is running on, this may involve relocating specific Teradata PT API libraries as well as setting runtime environment variables using the `Cufconfig` utility. For more information on using the `Cufconfig` utility, see the Utilities publication.

Configuring the Teradata Database on Linux

Before using the CREATE PROCEDURE SQL statement to create a Teradata PT API XSP, the following libraries from the Teradata PT API package must be moved into the `/usr/tdbms/lib` directory:

- `libtelapi.so`
- `libpxicu.so`

If the default Teradata PT API installation is used, these libraries can be found in the `/opt/teradata/client/tbuild/13.10/lib64` directory.

After using the CREATE PROCEDURE SQL statement to create a Teradata PT API XSP, configure the Teradata Database’s environment to allow the XSP to run. If it is possible to
restart the Teradata Database, use the \textit{Cufconfig} utility to set the CLIEnvFile variable to register a file with the following environment variable settings:

\begin{verbatim}
NLS_PATH=/OPT/TERADATA/CLIENT/TBUILD/13.10/MSG64/%N
/* Default location of opermsgs.cat */
COPERR=/usr/lib64
COPLIB=/usr/lib64
LD_LIBRARY_PATH=/opt/teradata/client/tbuild/13.10/lib64
/* Default location of the driver libraries */
\end{verbatim}

Add the location of the dynamic library containing the user created XSP to the LD\_LIBRARY\_PATH in the environment variable settings file. After setting the CLIEnvFile variable using the \textit{Cufconfig} utility, the Teradata Database must be restarted to allow the database to register the new runtime environment settings.

If it is not possible to restart the Teradata Database, move the following Teradata PT API libraries into the /usr/tdbms/lib directory:

- \textit{libopcommon.so}
- \textit{libexportop.so} (If using the Export driver)
- \textit{libloadop.so} (If using the Load driver)
- \textit{libstreamop.so} (If using the Stream driver)
- \textit{libupdateop.so} (If using the Update driver)

If the default Teradata PT API installation is used, these libraries can be found in the following directory:

```
/opt/teradata/client/tbuild/13.10/lib64
```

Move the \textit{opermsgs.cat} message catalog into the / (root) directory. If the default Teradata PT API installation is used, find this catalog in the following directory:

```
/opt/teradata/client/tbuild/13.10/msg64
```

Lastly, place the dynamic library containing the user created XSP in the /usr/tdbms/lib directory.

**Configuring the Teradata Database on Windows**

After installing the Teradata PT API package and its dependencies, restart the Teradata Database to register the changes to the system path. Do this before using the CREATE PROCEDURE SQL statement to create a Teradata PT API XSP. Add the location of the dynamic library containing the XSP to the system path before restarting the Teradata Database or place the library in a directory currently included in the Teradata Database’s search path.

If restarting the Teradata Database is not possible, move the following Teradata PT API libraries to a directory which is currently included in the Teradata Database’s search path:

- \textit{telapi.lib}
- \textit{telapi.dll}
- \textit{pxicu.dll}
- \textit{opcommon.dll}
Chapter 8: Using Teradata PT API in an External Stored Procedure
Installing a Teradata PT API External Stored Procedure

- `exportop.dll` (If using the Export driver)
- `loadop.dll` (If using the Load driver)
- `streamop.dll` (If using the Stream driver)
- `updateop.dll` (If using the Update driver)

If the default Teradata PT API installation is used, find these libraries in the following directory on 32-bit systems:

C:\Program Files\Teradata\Client\13.10\Teradata Parallel Transporter\bin

and in the following directory on 64-bit systems:

C:\Program Files\Teradata\Client\13.10\Teradata Parallel Transporter\bin64

Move the `opermsgs.cat` message catalog into a directory which is currently included in the Teradata Database’s search path. If the default Teradata PT API installation is used, find this catalog in the following directory on 32-bit systems:

C:\Program Files\Teradata\Client\13.10\Teradata Parallel Transporter\msg

and in the following directory on 64-bit systems:

C:\Program Files\Teradata\Client\13.10\Teradata Parallel Transporter\msg64

Lastly, place both the `.dll` and `.lib` files containing the user created XSP in a directory currently included in the Teradata Database’s search path.

**Using the CREATE PROCEDURE SQL Statement**

The CREATE PROCEDURE SQL statement is used to link the dynamic library containing the user created XSP to the Teradata Database. The Teradata Database user must have the CREATE EXTERNAL PROCEDURE permission to be able to issue the CREATE PROCEDURE SQL statement.

The following is an example of a CREATE PROCEDURE SQL statement for an XSP which uses the Teradata PT API:
Chapter 8: Using Teradata PT API in an External Stored Procedure

Executing a Teradata PT API External Stored Procedure

XSPs are executed using the SQL CALL statement. For more information on using the SQL CALL statement to execute an XSP, see the External Stored Procedure chapter of *SQL External Routine Programming*.
To create a Teradata PT API diagnostic log file when using the Teradata PT API in an XSP, the XSP must have write access to the directory when the log file is generated. On Linux, this requires write access being given to the others group.

**Getting Started with the External Stored Procedure Example**

To run the example external stored procedure example

Follow these steps to build, install, and run this example:

1. Install the Teradata PT API package.
   
   Install the Teradata PT API package on the Teradata Database’s machine or on a client machine. If installed on a client machine, the machine needs to be the same platform as the Teradata Database machine.

2. Build the Example External Stored Procedure Library.
   
   Go to the Teradata PT API sample directory and build the example XSP library using the makefiles provided or the Visual Studio project file. On Linux, if the default Teradata PT API installation is used, the files for the sample XSP will be located in the following directory:
   
   /opt/teradata/client/tbuild/13.10/tptapi/sample/xsp
   
   Building the example XSP will require two files from the Teradata Database: the sqltypes_td.h header file and the udf library (udf.so on Linux, udf.lib on Windows). Refer to Database Administration for the location of these two files.

3. Move example External Stored Procedure Files onto the Teradata Database machine.
   
   **Note**: If the Teradata Database machine was used to build the example XSP library, this step can be skipped.
   
   Move the example XSP library into any directory on the Teradata Database machine. On the Linux platform, the example XSP library produced in step 2 is named xsp.so. On Windows platforms, both the xsp.lib and xsp.dll library files will be needed.
   
   In addition to the example XSP library, the following input files will be needed in order to run the example:
   
   - input.exp
   - input.lod
   - input.stm
   - input.upd
   - infile.littleEndian

4. Install all required Teradata PT API packages on the Teradata Database.
Chapter 8: Using Teradata PT API in an External Stored Procedure

Getting Started with the External Stored Procedure Example

See the beginning section of Appendix B: “Code Samples” for the list of packages which are mandatory to run the Teradata PT API samples. Install all of these packages on the Teradata Database machine.

5 Configure the Teradata Database to run the CREATE PROCEDURE SQL statement.

Depending on which platform the Teradata Database is running on, follow the above instructions for configuring the Teradata Database's environment so that the CREATE PROCEDURE SQL statement can be used to create the example Teradata PT API XSP.

For a Teradata Database running on Linux, move the following files into the /usr/tdbms/lib directory:

- libtelapi.so
- libpxicu.so

If the default Teradata PT API installation is used, these libraries can be found in the /opt/teradata/client/tbuild/13.10/lib64 directory.

6 Issue the CREATE PROCEDURE SQL statement.

Issue the following CREATE PROCEDURE SQL statement to the Teradata Database using a database user account that has the CREATE EXTERNAL PROCEDURE permission:

```
CREATE PROCEDURE xsptest
(
    IN operatorType VARCHAR(64),
    IN configfilename varchar(64),
    IN extraconfigfilename VARCHAR(64),
    IN datafilename VARCHAR(64),
    IN updatetime VARCHAR(64)
)
LANGUAGE CPP
MODIFIES SQL DATA
EXTERNAL NAME 'SP!CLI!SP!/xsptest/xsp.so!F!xsptest'
PARAMETER STYLE SQL;
```

Note that for this example CREATE PROCEDURE SQL statement “/xsptest/xsp.so” should be replaced with the name and current location of the example external stored procedure library.

7 Configure the Teradata Database for running Teradata PT API external stored procedures.

Depending on the Teradata Database platform, follow the above instructions for configuring the Teradata Database's environment for running Teradata PT API XSPs.

For a Teradata Database Linux installation, create a file with the following environment variable settings:

```
NLSPATH=/OPT/TERADATA/CLIENT/TBUILD/13.10/MSG64/%N
/* Default location of opermsgs.cat */
COPERR=/usr/lib64
COPLIB=/usr/lib64
LD_LIBRARY_PATH=/opt/teradata/client/tbuild/13.10/lib64
/* Default location of the driver libraries */
```

Add the current location of the example XSP library to the above LD_LIBRARY_PATH definition.

Next, create another file with the following information:
Chapter 8: Using Teradata PT API in an External Stored Procedure

Getting Started with the External Stored Procedure Example

CLIEnvFile:/<location and name of the first file created above>

Then use the Cufconfig utility to set the CLIEnvFile variable:
cufconfig -f/<location and name of the second file created above>

Restart the Teradata Database so that the new configuration takes effect.

Create tables for the external stored procedure examples

Use the following SQL commands to create the tables used in the example XSP:

```
CREATE SET TABLE tdexport, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT
(
    Associate_Id INTEGER,
    Associate_Name CHAR(25) CHARACTER SET LATIN NOT CASESPECIFIC,
    Salary FLOAT,
    DOJ date format 'YY/MM/DD',
    Designation VARCHAR(25) CHARACTER SET LATIN NOT CASESPECIFIC,
    Loan_Amount DECIMAL(5,2),
    Martial_Status CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC,
    No_Of_Dependents BYTEINT,
    update_date TIMESTAMP(6)
)
PRIMARY INDEX (Associate_Id);
```

```
CREATE SET TABLE tdexportA, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT
(
    Associate_Id INTEGER,
    Associate_Name CHAR(25) CHARACTER SET LATIN NOT CASESPECIFIC,
    Salary FLOAT,
    DOJ DATE FORMAT 'YY/MM/DD',
    Designation VARCHAR(25) CHARACTER SET LATIN NOT CASESPECIFIC,
    Loan_Amount DECIMAL(5,2),
    Martial_Status CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC,
    No_Of_Dependents BYTEINT,
    update_date TIMESTAMP(6)
)
PRIMARY INDEX (Associate_Id);
```

```
CREATE SET TABLE tdexportB, NO FALLBACK,
NO BEFORE JOURNAL,
NO AFTER JOURNAL,
CHECKSUM = DEFAULT
(
    Associate_Id INTEGER,
    Associate_Name CHAR(25) CHARACTER SET LATIN NOT CASESPECIFIC,
    Salary FLOAT,
    DOJ DATE FORMAT 'YY/MM/DD',
    Designation VARCHAR(25) CHARACTER SET LATIN NOT CASESPECIFIC,
    Loan_Amount DECIMAL(5,2),
    Martial_Status CHAR(1) CHARACTER SET LATIN NOT CASESPECIFIC,
    No_Of_Dependents BYTEINT,
    update_date TIMESTAMP(6)
)
PRIMARY INDEX (Associate_Id);
```

Create the tables on the same database that the example XSP is running on or on any other database. For the example scenarios which transfer data between multiple tables, the tables to export data from (source tables) can be on one database and the tables to load data into (target tables) can be on a different database.

Issue the SQL CALL statement.

Use one of the following SQL CALL statements to run the example XSP:

- Use the Load driver to load data from infile.littleEndian into the database:
  ```
  CALL xsptest ('TD_LOAD','input.lod',null,'infile.littleEndian',null);
  ```
• Use the Update driver to load data from `infile.littleEndian` into the database:
  ```sql
  CALL xsptest ('TD_UPDATE','input.upd',null,'infile.littleEndian',null);
  ```

• Use the Stream driver to load data from `infile.littleEndian` into the database:
  ```sql
  CALL xsptest ('TD_STREAM','input.stm',null,'infile.littleEndian',null);
  ```

• Use the Export driver to export data from the database into a file:
  ```sql
  CALL xsptest ('TD_EXPORT','input.exp',null,'outfile.littleEndian',null);
  ```

• Use the Export and Load drivers to transfer data between two tables:
  ```sql
  CALL xsptest ('TD_EXPORT2LOAD','input.exp','input.lod',null,null);
  ```

• Use the Export and Update drivers to transfer data from one table into two tables. The timestamp is used to determine which rows to transfer. In this example, only rows with timestamps newer than '2007-03-03 10:30:00' will transfer:
  ```sql
  CALL xsptest ('TD_EXPORT2UPDATE','input.exp2','input.lod',null,'2007-03-03 10:30:00');
  ```

• Use the Export and Stream drivers to transfer data between two tables. The timestamp is used to determine which rows to transfer. In this example, only rows with timestamps newer than '2007-03-03 10:30:00' will be transferred.
  ```sql
  CALL xsptest ('TD_EXPORT2STREAM','input.exp2','input.stm',null,'2007-03-03 10:30:00');
  ```

**Notes on Using the above SQL CALL Statements**

• When issuing the above SQL CALL statements, always specify the full path for any necessary input or output file. In the above statements `infile.littleEndian`, `outfile.littleEndian`, `input.lod` are examples of input and output files for which the full path should be specified.

• Read access is required for all input files and the directories in which they are located. On Linux, this requires giving read access to the `others` group.

• Write access is required for all output files and the directories where they will be located. On Linux, this requires giving write access to the `others` group.

• All examples using the Export driver require that there be pre-existing data in the table from which data will be exported.

• Refer to the `input.exp` input file for instructions on how to create the `input.exp2` input file.
CHAPTER 9

Converting TIME, TIMESTAMP, and INTERVAL Data Types

This chapter contains conversion information for TIME, TIMESTAMP, and INTERVAL data types for Teradata PT API.

ANSI/SQL DateTime Specifications

In Teradata PT API, the TIME, TIMESTAMP and INTERVAL data types are not part of the syntax as they are in SQL. No Teradata PT API data type exists for TIME or TIMESTAMP. Therefore, to load or export data, manually convert the desired data type to the ANSI/SQL DateTime data types by specifying the appropriate fixed CHAR column in the schema as specified in Table 32.

Table 32: ANSI/SQL DateTime Specifications

<table>
<thead>
<tr>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
</tr>
<tr>
<td>TIME (n)</td>
</tr>
<tr>
<td>Where n is the number of digits after the decimal point, 0 through 6. (Default = 6.)</td>
</tr>
<tr>
<td>Convert to: CHAR(8 + n + (1 if n &gt; 0, otherwise 0))</td>
</tr>
<tr>
<td>Format (n = 0): hh:mm:ss</td>
</tr>
<tr>
<td>Example: 11:37:58</td>
</tr>
<tr>
<td>Format (n = 4): hh:mm:ss.sss</td>
</tr>
<tr>
<td>Example: 11:37:58.1234</td>
</tr>
</tbody>
</table>

| TIMESTAMP |
| TIMESTAMP (n) |
| Where n is the number of digits after the decimal point, 0 through 6. (Default = 6.) |
| Convert to: CHAR(19 + n + (1 if n > 0, otherwise 0)) |
| Format (n = 0): yyyy-mm-dd hh:mm:ss |
| Example: 1998-09-04 11:37:58 |
| Format (n = 4): yyyy-mm-dd hh:mm:ss.sss |
| Example: 1998-09-04 11:37:58.1234 |

| TIME WITH TIME ZONE |
| TIME (n) WITH TIME ZONE |
Table 32: ANSI/SQL DateTime Specifications (continued)

<table>
<thead>
<tr>
<th>Specification</th>
<th>Where ( n ) is the number of digits after the decimal point, 0 through 6. (Default = 6.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert to:</td>
<td>CHAR(14 + n + (1 if ( n &gt; 0 ), otherwise 0))</td>
</tr>
<tr>
<td>Format ( (n = 0) ):</td>
<td>( hh:mm:ss[\pm]hh:mm )</td>
</tr>
<tr>
<td>Example:</td>
<td>11:37:58-08:00</td>
</tr>
<tr>
<td>Format ( (n = 4) ):</td>
<td>( hh:mm:ss.sss {\pm} ) ( hh:mm )</td>
</tr>
<tr>
<td>Example:</td>
<td>11:37:58.1234-08:00</td>
</tr>
</tbody>
</table>

**TIMESTAMP WITH TIME ZONE**

**TIMESTAMP \( (n) \) WITH TIME ZONE**

Where \( n \) is the number of digits after the decimal point, 0 through 6. (Default = 6.)

| Convert to: | CHAR(25 + n + (1 if \( n > 0 \), otherwise 0))                     |
| Format \( (n = 0) \): | \( yyyy-mm-dd \) \( hh:mm:ss[\pm]hh:mm \) |
| Example: | 1998-09-24 11:37:58+07:00                                      |
| Format \( (n = 4) \): | \( yyyy-mm-dd \) \( hh:mm:ss.sss \{\pm} \) \( hh:mm \)  |
| Example: | 1998-09-24 11:37:58.1234+07:00                                  |

**INTERVAL YEAR**

**INTERVAL YEAR \( (n) \)**

Where \( n \) is the number of digits, 1 through 4. (Default = 2.) Results include one leading blank space (for positive values) or a minus sign (for negative values).

| Convert to: | CHAR\((n+1)\)                                                     |
| Format \( (n = 2) \): | \( yy \)               |
| Example: | 98                                                                 |
| Format \( (n = 4) \): | \( yyyy \)              |
| Example: | 1998                                                                 |

**INTERVAL YEAR TO MONTH**

**INTERVAL YEAR \( (n) \) TO MONTH**

Where \( n \) is the number of digits, 1 through 4. (Default = 2.) Results include one leading blank space (for positive values) or a minus sign (for negative values).

| Convert to: | CHAR\((n + 3)\)                                      |
| Format \( (n = 2) \): | \( yy-mm \)            |
| Example: | 98-12                                                 |
| Format \( (n = 4) \): | \( yyyy-mm \)             |
| Example: | 1998-12                                               |

**INTERVAL MONTH**

**INTERVAL MONTH \( (n) \)**

Where \( n \) is the number of digits, 1 through 4. (Default = 2.) Results include one leading blank space (for positive values) or a minus sign (for negative values).
### ANSI/SQL DateTime Specifications

#### INTERVAL DAY

**INTERVAL DAY**

Where \( n \) is the number of digits, 1 through 4. (Default = 2.) Results include one leading blank space (for positive values) or a minus sign (for negative values).

<table>
<thead>
<tr>
<th>Convert to:</th>
<th>CHAR(( n ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format ( n = 2 ):</td>
<td>( dd )</td>
</tr>
<tr>
<td>Example:</td>
<td>31</td>
</tr>
<tr>
<td>Format ( n = 4 ):</td>
<td>( dddd )</td>
</tr>
<tr>
<td>Example:</td>
<td>0031</td>
</tr>
</tbody>
</table>

#### INTERVAL DAY TO HOUR

**INTERVAL DAY (\( n \)) TO HOUR**

Where \( n \) is the number of digits, 1 through 4. (Default = 2.) Results include one leading blank space (for positive values) or a minus sign (for negative values).

<table>
<thead>
<tr>
<th>Convert to:</th>
<th>CHAR(( n + 3 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format ( n = 2 ):</td>
<td>( dd \ hh )</td>
</tr>
<tr>
<td>Example:</td>
<td>31 12</td>
</tr>
<tr>
<td>Format ( n = 4 ):</td>
<td>( dddd \ hh )</td>
</tr>
<tr>
<td>Example:</td>
<td>0031 12</td>
</tr>
</tbody>
</table>

#### INTERVAL DAY TO MINUTE

**INTERVAL DAY (\( n \)) TO MINUTE**

Where \( n \) is the number of digits, 1 through 4. (Default = 2.) Results include one leading blank space (for positive values) or a minus sign (for negative values).

<table>
<thead>
<tr>
<th>Convert to:</th>
<th>CHAR(( n + 6 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format ( n = 2 ):</td>
<td>( dd \ hh:mm )</td>
</tr>
<tr>
<td>Example:</td>
<td>31 12:59</td>
</tr>
<tr>
<td>Format ( n = 4 ):</td>
<td>( dddd \ hh:mm )</td>
</tr>
<tr>
<td>Example:</td>
<td>0031 12:59</td>
</tr>
</tbody>
</table>

#### INTERVAL DAY TO SECOND

**INTERVAL DAY (\( n \)) TO SECOND**

**INTERVAL DAY TO SECOND (\( m \))**

**INTERVAL DAY (\( n \)) TO SECOND (\( m \))**

Where:

- \( n \) is the number of digits, 1 through 4. (Default = 2.)
- \( m \) is the number of digits after the decimal point, 0 through 6. (Default = 6.)

Results include one leading blank space (for positive values) or a minus sign (for negative values).
### ANSI/SQL DateTime Specifications (continued)

<table>
<thead>
<tr>
<th>Specification</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTERVAL HOUR</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INTERVAL HOUR</strong> (n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Where n is the number of digits, 1 through 4. (Default = 2.) Results include one leading blank space (for positive values) or a minus sign (for negative values).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convert to:</td>
<td>CHAR(n)</td>
<td></td>
</tr>
<tr>
<td>Format (n = 2):</td>
<td>hh</td>
<td></td>
</tr>
<tr>
<td>Example:</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Format (n = 4):</td>
<td>hhhh</td>
<td></td>
</tr>
<tr>
<td>Example:</td>
<td>0012</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specification</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTERVAL HOUR TO MINUTE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INTERVAL HOUR</strong> (n) TO MINUTE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Where n is the number of digits, 1 through 4. (Default = 2.) Results include one leading blank space (for positive values) or a minus sign (for negative values).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convert to:</td>
<td>CHAR(n + 3)</td>
<td></td>
</tr>
<tr>
<td>Format (n = 2):</td>
<td>hh:mm</td>
<td></td>
</tr>
<tr>
<td>Example:</td>
<td>12:59</td>
<td></td>
</tr>
<tr>
<td>Format (n = 4):</td>
<td>hhhh:mm</td>
<td></td>
</tr>
<tr>
<td>Example:</td>
<td>0012:59</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specification</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTERVAL HOUR TO SECOND</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INTERVAL HOUR</strong> (n) TO SECOND</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INTERVAL HOUR</strong> TO SECOND (m)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INTERVAL HOUR</strong> (n) TO SECOND (m)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Where:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• n is the number of digits, 1 through 4. (Default = 2.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• m is the number of digits after the decimal point, 0 through 6. (Default = 6.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Results include one leading blank space (for positive values) or a minus sign (for negative values).</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convert to:</td>
<td>CHAR(n + 6 + m + (1 if m &gt; 0, 0 otherwise))</td>
<td></td>
</tr>
<tr>
<td>Format (n = 2, m = 0):</td>
<td>hh:mm:ss</td>
<td></td>
</tr>
<tr>
<td>Example:</td>
<td>12:59:59</td>
<td></td>
</tr>
<tr>
<td>Format (n = 4, m = 4):</td>
<td>hhhh:mm:ss.ssss</td>
<td></td>
</tr>
<tr>
<td>Example:</td>
<td>0012:59:59:1234</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specification</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INTERVAL MINUTE</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>INTERVAL MINUTE</strong> (n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Where n is the number of digits, 1 through 4. (Default = 2.) Results include one leading blank space (for positive values) or a minus sign (for negative values).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 32: ANSI/SQL DateTime Specifications (continued)

<table>
<thead>
<tr>
<th>Specification</th>
<th>Convert to</th>
<th>Format (n = 2):</th>
<th>Example</th>
<th>Format (n = 4):</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTERVAL MINUTE TO SECOND</td>
<td>CHAR(n)</td>
<td>mm</td>
<td>59</td>
<td>mmm</td>
<td>0059</td>
</tr>
<tr>
<td>INTERVAL MINUTE (n) TO SECOND</td>
<td>CHAR(n) + m + (1 if m &gt; 0, 0 otherwise)</td>
<td>mm:ss</td>
<td>59:59</td>
<td>mmm:ss.sss</td>
<td>0059:59.1234</td>
</tr>
<tr>
<td>INTERVAL MINUTE (n) TO SECOND (m)</td>
<td>CHAR(n + m + (1 if m &gt; 0, 0 otherwise))</td>
<td>ss</td>
<td>59</td>
<td>sss.sss</td>
<td>0059.1234</td>
</tr>
<tr>
<td>INTERVAL SECOND</td>
<td>CHAR(n) + m + (1 if m &gt; 0, 0 otherwise)</td>
<td>ss</td>
<td>59</td>
<td>sss.sss</td>
<td>0059.1234</td>
</tr>
<tr>
<td>INTERVAL SECOND (n)</td>
<td>CHAR(n)</td>
<td>ss</td>
<td>59</td>
<td>sss.sss</td>
<td>0059.1234</td>
</tr>
<tr>
<td>INTERVAL SECOND (n, m)</td>
<td>CHAR(n + m + (1 if m &gt; 0, 0 otherwise))</td>
<td>ss</td>
<td>59</td>
<td>sss.sss</td>
<td>0059.1234</td>
</tr>
</tbody>
</table>
# Platform Compilers

Table 33: Teradata PT API Platform Compilers

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Platform Version</th>
<th>Compiler Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIX</td>
<td>Power PC AIX 5.3</td>
<td>VisualAge C++ Professional/C for AIX Compiler, V5 Using the compiler’s default STL</td>
</tr>
<tr>
<td></td>
<td>Power PC AIX 5.3</td>
<td>HP ANSI C++ B3910B X.03.37.01 for PA-RISC Using the compiler’s default STL</td>
</tr>
<tr>
<td>HP-UX</td>
<td>PA-RISC HP-UX 11v1 (11.11)</td>
<td>For 32-bit:</td>
</tr>
<tr>
<td></td>
<td>PA-RISC HP-UX 11v2 (11.23)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PA-RISC HP-UX 11v3 (11.31)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Itanium 64-bit HP-UX 11iv2 (11.23)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Itanium 64-bit HP-UX 11iv3 (11.31)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PA-RISC HP-UX 11v1 (11.11)</td>
<td>For 64-bit:</td>
</tr>
<tr>
<td></td>
<td>PA-RISC HP-UX 11v2 (11.23)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PA-RISC HP-UX 11v3 (11.31)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Itanium 64-bit HP-UX 11iv2 (11.23)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Itanium 64-bit HP-UX 11iv3 (11.31)</td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>x86 32-bit Red Hat® Linux Advanced Server 3.0</td>
<td>For 32-bit:</td>
</tr>
<tr>
<td></td>
<td>x86 32-bit Red Hat® Linux Advanced Server 4.0</td>
<td>g++ version 3.3.3 (V3.3.3)</td>
</tr>
<tr>
<td></td>
<td>x86 32-bit Red Hat® Linux Advanced Server 5.0</td>
<td>g++ 4.1 (V4.1.2) Using the compiler’s default STL</td>
</tr>
<tr>
<td></td>
<td>EM64T/Opteron 32-bit Red Hat® Linux Advanced Server 4.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EM64T/Opteron 32-bit Red Hat® Linux Advanced Server 5.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EM64T/Opteron 64-bit Red Hat® Linux Advanced Server 4.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EM64T/Opteron 64-bit Red Hat® Linux Advanced Server 5.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x86 32-bit SUSE® LINUX® Enterprise 9 SP1, SP2, and SP3</td>
<td>For 64-bit:</td>
</tr>
<tr>
<td></td>
<td>x86 32-bit SUSE® LINUX® Enterprise 10 SP1</td>
<td>g++ version 3.3.3 (V3.3.3)</td>
</tr>
<tr>
<td></td>
<td>EM64T/Opteron 32-bit SUSE LINUX® Enterprise 9 SP1, SP2, and SP3</td>
<td>g++ 4.1 (V4.1.2) Using the compiler’s default STL</td>
</tr>
<tr>
<td></td>
<td>EM64T/Opteron 64-bit SUSE LINUX® Enterprise 9 SP1, SP2, and SP3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EM64T/Opteron 32-bit SUSE LINUX® Enterprise 10 SP1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>EM64T/Opteron 64-bit SUSE LINUX® Enterprise 10 SP1</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** Build 32-bit user applications with the g++ version 3.3.3 compiler if the g++ 3.3.3 version of the Teradata PT API library was installed.
### Table 33: Teradata PT API Platform Compilers (continued)

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Platform Version</th>
<th>Compiler Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>z/Linux</td>
<td>RedHat Enterprise Linux on zSeries</td>
<td>• For 32-bit, g++ version 4.1 Using the compiler’s default STL</td>
</tr>
<tr>
<td></td>
<td>• 32-bit TTU on RHEL 5.0</td>
<td>• For 64-bit, g++ 4.1 Using the compiler’s default STL</td>
</tr>
<tr>
<td></td>
<td>• 64-bit TTU on RHEL 5.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Novell SUSE Enterprise Linux on zSeries</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 32-bit TTU on SUSE 10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 64-bit TTU on SUSE 10</td>
<td></td>
</tr>
<tr>
<td>Solaris Opteron™</td>
<td>• 32-bit Opteron running 64-bit Solaris 10</td>
<td>• Sun C++ 5.8 2005/10/13 Using the compiler’s default STL (libCstd)</td>
</tr>
<tr>
<td></td>
<td>• 64-bit Opteron running 64-bit Solaris 10</td>
<td><strong>Note:</strong> Applications linked with the STLport version of the STL library are incompatible with the Teradata PT API libraries. Link user applications with the default STL library instead.</td>
</tr>
<tr>
<td>Solaris SPARC™</td>
<td>SPARC Solaris 8,9,10</td>
<td>• Sun WorkShop 6 update 2 C++ 5.3 Using the compiler’s default STL (libCstd)</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> Applications linked with the STLport version of the STL library are incompatible with the Teradata PT API libraries. Link user applications with the default STL library instead.</td>
</tr>
<tr>
<td>Operating System</td>
<td>Platform Version</td>
<td>Compiler Version</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------</td>
<td>------------------</td>
</tr>
</tbody>
</table>
| Windows          | • x86 2000 Professional Server Advanced Server SP4  
                    • x86 32-bit XP Professional SP2  
                    • EM64T/Operton 32-bit XP Professional SP2  
                    • EM64T/Operton 64-bit XP Professional x64 Edition SP2  
                    • EM64T/Operton 64-bit WS 2003 Enterprise x64 Edition Standard x64 Edition Original and SP2  
                    • EM64T/Operton 64-bit WS 2008 Enterprise x64 Edition Standard x64 Edition  
                    • EM64T/Operton 32-bit Vista Enterprise edition  
                    • EM64T/Operton 64-bit Vista Enterprise edition  
                    • EM64T/Operton x64 Vista Enterprise edition | • MSVC_7.1 and MSVC_8.0 for 32-bit  
Using the compiler's default STL  
• MSVC_8.0 for 64-bit  
Using the compiler's default STL |
| z/OS             | • z/OS 1.7  
                    • z/OS 1.8  
                    • z/OS 1.9 | • V1.7 z/OS XL C++ |
Table 34 lists the Teradata Tools and Utilities 13.10 version of the following products that are required to run the code samples for Teradata PT API.

It is preferred that the products be installed in the order listed in the table, however, each product will state its prerequisites at install time.

For Teradata Tools and Utilities 13.10 products, the full version number for the packages is 13.10.00.xx. The last two numbers in the version string denote the e-fix version with the latest release having the greatest e-fix number. Before using an e-fix version of a product, review the corresponding e-fix ReadMe file for fix information and for documentation on any resulting usage changes.

See Appendix A: “Platform Compilers,” for the supported versions and releases of the platform compilers.

Refer to the following publications for the Teradata PT API default installation directory and for which environment variables need to be set prior to using Teradata PT API. The latest levels of these information products are available at http://www.info.teradata.com/.

- B035-2459-mmyx, the Teradata Tools and Utilities Installation Guide for UNIX and Linux
- B035-2407-mmyx, Teradata Tools and Utilities Installation Guide for Microsoft Windows
After you install Teradata PT API you should see the following directory structure:

- `\bin` (contains 32-bit .lib and .dll files)
- `\bin64` (contains 64-bit .lib and .dll files)
- `\bin\vc7` (contains 32-bit .lib and .dll files for vc7.1)
- `\tptapi\inc` (contains .h files)
- `\sample\blockloading` (Shows how to load rows in buffer mode with the load driver)
- `\sample\checkpoint` (Shows how checkpoint/restart works with the stream driver)
- `\sample\common` (Contains shared files used by all samples)
- `\sample\generic` (Shows how each driver works)
- `\sample\getbuffer` (Shows how to export the driver’s GetBuffer feature)
- `\sample\inc` (Contains shared .h files used by all samples)
- `\sample\multiple` (Shows how to use multiple drivers in the same application)
- `\sample\multi_node` (Shows how to use the multi-node protocol)
- `\sample\threads` (Shows how to write multi-threaded applications)
- `\sample\xsp` (Shows how to use Teradata PT API in an XSP)

**Note:** The 64-bit .lib and .dll files will only be in the 64-bit Windows package and the 32-bit .lib and .dll files will only be in the 32-bit Windows package.

Get familiar with the Teradata PT API methods in the sample programs (.cpp files).

In each sample solution file, you can see the following settings. Build the 32-bit samples in Visual Studio V7.1 or V8.0 and the 64-bit samples in Visual Studio V8.0.

- **C/C++ Additional Include Directories:**
  - `../inc; .inc; ..\common`
- **Linker Additional Library Directories**
  - `\..\..\..\bin; \TELAPI.lib` (32-bit samples)
  - `\..\..\..\bin64; \TELAPI.lib` (64-bit samples)

  **Note:** Edit the `makefile` and modify the values of the C++C and CC variables to the location of the compilers on your system.

Put the sample executable, data file, and all Teradata PT API dlls in the same directory or specify the Teradata PT API directory in the PATH environment variable. For example, the following directories should be appended to the PATH environment variable if they are not already there:

- `C:\Program Files\Teradata\Client\13.10\Teradata Parallel Transporter\bin` (32-bit version)
- `C:\Program Files\Teradata\Client\13.10\Teradata Parallel Transporter\msg` (32-bit version)
- `C:\Program Files\Teradata\Client\13.10\Teradata Parallel Transporter\bin64` (64-bit version)
- `C:\Program Files\Teradata\Client\13.10\Teradata Parallel Transporter\msg64` (64-bit version)
- `C:\Program Files\Teradata\Client\13.10\Teradata Parallel Transporter\bin\vc7` (32-bit version for vc7.1)

  **Note:** All 32-bit executables run with 32-bit dlls and all 64-bit executables run with 64-bit dlls.

Change the DBS machine name, DBS user name, and DBS user password in the BTEQ scripts `setupTables.bteq` and `cleanupTables.bteq` for each sample.

Run the BTEQ script `setupTables.bteq` to set up the tables for the current sample.

Execute the sample.

Run the BTEQ script `cleanupTables.bteq` to clean up the tables used by the current sample.
Appendix B: Code Samples
Solaris SPARC, Solaris Opteron, HP-UX and AIX

Note: To execute TPTAPI samples using VS2003 (vc7.1), copy libraries from the /bin/vc7 folder to /bin.

Solaris SPARC, Solaris Opteron, HP-UX and AIX

1. After you install Teradata PT API you should see the following directory structure:

```
\lib         (contains 32-bit .so/.sl files)
\lib64       (contains 64-bit .so/.sl files)
\tptapi
  \inc   (contains .h files)
  \sample
  \blockloading (Shows how to load rows in buffer mode with the load driver)
  \checkpoint (Shows how checkpoint/restart works with the stream driver)
  \common (Contains shared files used by all samples)
  \generic (Shows how each driver works)
  \getbuffer (Shows how to export the driver’s GetBuffer feature)
  \inc (Contains shared .h files used by all samples)
  \multiple (Shows how to use multiple drivers in the same application)
  \multi_node (Shows how to use the multi-node protocol)
  \threads (Shows how to write multi-threaded applications)
```

Note: The 32-bit and 64-bit .so/.sl files are in the Solaris, HP-UX PARISC, and AIX packages. Only the 64-bit .so files are in the HP-UX IA64 package.

2. Get familiar with the Teradata PT API methods in the sample programs (.cpp files).

3. In the sample makefile, you can see the following settings:

   • C/C++ Additional Include Directories:
     ```
     ./inc  (Teradata PT API common headers)
     ./.inc  (sample-only header)
     ./.common  (sample shared files)
     ```

   • Linker Additional Library Directories:
     ```
     -ltelapi
     ```
     Run the makefile to build the samples.

   Note: Edit the makefile and modify the values of the C++C and CC variables to the location of the compilers on your system. Use a GNU compatible version of make when building the samples on AIX.

Before running the makefile, verify that the directory containing the Teradata PT ICU Library (libpxicu.so/libpxicu.sl) is in the LD_LIBRARY_PATH environment variable (SHLIB_PATH for HP-UX AND LIBPATH for AIX).

4. If necessary, set environment variables. Put the sample executable, data file, and all Teradata PT API .so or .sl files in the same directory. Or specify the Teradata PT API directories in the PATH variables. Run all 32-bit executables with 32-bit .so or .sl files and run all 64-bit executables with 64-bit .so or .sl files.

   For example on Solaris SPARC and Solaris Opteron:

   a. ```
      export LD_LIBRARY_PATH = <=library path>:`$LD_LIBRARY_PATH
      ```
      ```
      export LD_LIBRARY_PATH=/opt/teradata/client/13.10/tbuild/lib:$LD_LIBRARY_PATH
      (32-bit)
      ```
      ```
      export LD_LIBRARY_PATH=/opt/teradata/client/13.10/tbuild/lib64:$LD_LIBRARY_PATH
      (64-bit)
      ```

   b. ```
      export NLSPATH=<directory path of the catalog>/%N:$NLSPATH
      ```
      ```
      export NLSPATH=/opt/teradata/client/13.10/tbuild/msg/%N:$NLSPATH (32-bit)
      ```
      ```
      export NLSPATH=/opt/teradata/client/13.10/tbuild/msg64/%N:$NLSPATH (64-bit)
      ```
**Note:** On HP-UX set the SHLIB_PATH and the NLSPATH environment variables and on AIX set the LIBPATH and the NLSPATH environment variables.

5 Set the CLlv2 environment variable `THREADONOFF` to 1 to run the Teradata PT API multi-threaded application.

6 Change the DBS machine name, DBS user name, and DBS user password in the BTEQ scripts `setupTables.bteq` and `cleanupTables.bteq` for each sample.

7 Run the BTEQ script `setupTables.bteq` to set up the tables for the current sample.

8 Execute the sample.

9 Run the BTEQ script `cleanupTables.bteq` to clean up the tables used by the current sample.

---

**Linux**

1 After you install Teradata PT API, you should see the following directory structure:

```
\lib (contains 32-bit .so files)
\lib64 (contains 64-bit .so files)
\lib\gcc_3.x (contains TPTAPI gcc_3.x 32-bit .so files)
\lib64\gcc_3.x (contains TPTAPI gcc_3.x 64-bit .so files)
\inc (contain .h files)
\sample
  \blockloading (Shows how to load rows in buffer mode with the load driver)
  \checkpoint (Shows how checkpoint/restart works with the stream driver)
  \generic (Shows how each driver works)
  \getbuffer (Shows how to export the driver’s GetBuffer feature)
  \inc (Contains shared .h files used by all samples)
  \multiple (Shows how to use multiple drivers in the same application)
  \multi_node (Shows how to use the multi-node protocol)
  \threads (Shows how to write multi-threaded applications)
  \xsp (Shows how to use Teradata PT API in an XSP)
```

2 Get familiar with the Teradata PT API methods in the sample programs (.cpp files).

3 In the sample `makefile`, you can see the following settings. Run the `makefile` to build the samples.

   - **C/C++ Additional Include Directories:**
     ```
     ./inc  (Teradata PT API common headers)
     ./inc  (sample-only header)
     ./common (sample shared files)
     ``

   - **Linker Additional Library Directories:**
     ```
     -ltelapi
     ``

     **Note:** Edit the `makefile` and modify the values of the C++C and CC variables to the location of the compilers on your system.

     Before running the makefile, verify that the directory containing the Teradata PT ICU Library (`libpxicu.so`) is in the LD_LIBRARY_PATH environment variable.

4 Set the following environment variables if necessary. Put the sample executable, data file, Teradata PT API .so file in the same directory. Or specify the Teradata PT API directory in the PATH variables. Run all 32-bit executables with 32-bit .so files and run all 64-bit executables with 64-bit .so files. For example:

   a  ```
   export LD_LIBRARY_PATH = < library path>$LD_LIBRARY_PATH
   ```

   ```
   export LD_LIBRARY_PATH=
   -/opt/teradata/client/13.10/tbuild/lib:$LD_LIBRARY_PATH (32-bit)
   ```
Appendix B: Code Samples

IBM z/OS

1. After installing Teradata PT API, these datasets will be present (where xx is the current release number):
   - STV.TIxxAPP.TWB.LOAD
   - STV.TIxxAPP.TWB..H
   - STV.TIxxAPP.TWB.SAMP

   (Contains .h files for the samples and data file.)
   (Contains the source code for the generic, multi-node and multi-threaded samples)

2. Get familiar with the Teradata PT API methods in the sample programs.

3. Build the samples (Provide a job card.):
   - Generic sample
     `STV.TIxxAPP.TWB.SAMP (MVSCMPBN)`
     This dataset contains the JCL to build the generic sample. Submit this job by opening this dataset and typing the command `sub` in the command line.
   - Multi-Node sample
     `STV.TIxxAPP.TWB.SAMP (MNMCMPBN)`
     `STV.TIxxAPP.TWB.SAMP (MMNCMPBN)`
     These datasets contain the JCL to build the master and slave instance respectively. To submit these jobs open each dataset and type the command `sub` in the command line. Submit these datasets in any order one after the other.
   - Threads sample
     `STV.TIxxAPP.TWB.SAMP (THDCMPBN)`
     This dataset contains the JCL to build the threads sample. Submit this job by opening this dataset and typing the command `sub` in the command line.

   **Note:** An output dataset must be created to hold the object modules in the samples. This dataset must have the record format as FB and Record Length as 80. It is named `UD18502.TEST.OBJ` and can be changed in the build jobs. The executable module will appear in `UD18502.TEST.LOAD` if left unchanged.

4. Change the DBS machine name (TDP), DBS user name, and DBS user password in the BTEQ scripts `setupTables.bteq` and `cleanupTables.bteq` for each sample.
Run the BTEQ script `setupTables.bteq` to set up the tables for the current sample.

Execute the sample (Provide a job card.).

- **Generic sample**
  
  `'STV.TIxxAPP.TWB.SAMP(MVSCMPBN)'`

  This dataset contains the JCL to execute the generic sample. Submit this job by opening this dataset and typing the command `sub` in the command line.

- **Multi-Node sample**

  `'STV.TIxxAPP.TWB.SAMP(MMNCPBNN)'`

  These datasets contain the JCL to execute the master and slave instances respectively. To submit these jobs open each dataset and type the command `sub` in the command line. Submit the master instance (`MNMSTRGO`) followed by the slave instance (`MNSLVEGO`).

- **Threads sample**

  `'STV.TIxxAPP.TWB.SAMP(THDCMPBN)'`

  This statement contains the JCL to execute the Threads sample. Submit this job by opening this dataset and typing the command `sub` in the command line.

  **Note:** The user can change the destination of the input and output files in the SCRIPT datasets. The names of the files are `GENINPUT`, `MNMINPUT`, `MNSINPUT` and `PARINPUT` in the `STV.TIxxAPP.TWB.SAMP` dataset.

Run the BTEQ script `cleanupTables.bteq` to drop data tables used by the current sample and any error tables that may have been created.
APPENDIX C

Compiling and Linking Options

Of note for this release:

- The Teradata PT API header files were changed. Applications built with previous versions of Teradata PT API must be recompiled with the updated header files.
- All Teradata PT API applications must explicitly link with the Teradata PT ICU library using the `-lpxicu` option.

Refer to the following solution files or `makefile`
s.

- **32-bit Application for Windows**
  See `generic.sln` in sample\generic.
  See `generic_vc7.sln` in sample\generic for vc7.1
  For multi_threaded applications, see `TelapiThreadTest.sln` in sample\threads.

- **64-bit Application for Windows**
  See `generic64.sln` in sample\generic.
  For multi_threaded applications, see `TelapiThreadTest64.sln` in sample\threads.

- **32-bit and 64-bit Applications for Solaris Opteron**
  See `makedriver.so1.o` in sample\generic
  For multi_threaded applications, see `makeparallel.so1.o` in sample\threads.

- **32-bit and 64-bit Applications for Solaris SPARC**
  See `makedriver.so1.s` in sample\generic
  For multi_threaded applications, see `makeparallel.so1.s` in sample\threads.

- **32-bit and 64-bit Applications for HP_UX**
  See `makedriver.hpux` in sample\generic
  For multi_threaded applications, see `makeparallel.hpux` in sample\threads.

- **32-bit and 64-bit Applications for AIX**
  See `makedriver.aix` in sample\generic.
  For multi_threaded applications, see `makeparallel.aix` in sample\threads.

- **32-bit and 64-bit Applications for LINUX**
  See `makedriver.linux` in sample\generic.
  For multi_threaded applications, see `makeparallel.linux` in sample\threads.
• 32-bit and 64-bit Applications for z/Linux
  See `makedriver.linux_390` in sample\generic.
  For multi-threaded applications, see `makeparallel.linux_390` in sample\threads.

• z/OS platforms
  See `STV.TIxxAPP.TWB.SAMP(MVSSMPBN)` for the JCL to invoke the generic sample compilation PROC libraries. Additional details can be found in `STV.TIxxAPP.TWB.SAMP(CBCC)` and `STV.TIxxAPP.TWB.SAMP(SBCXB)`. 
**Glossary**

**A**

**administrator** A special user responsible for allocating resources to a community of users.

**Access Module Processor (AMP)** A virtual processor that receives steps from a parsing engine (PE) and performs database functions to retrieve or update data. Each AMP is associated with one virtual disk, where the data is stored. An AMP manages only its own virtual disk and not the virtual disk of any other AMP.

**access rights** See privilege.

**AMP** See Access Module Processor (AMP).


**application** A complete, self-contained program that performs a specific function directly for the user. Contrast this to system software (two examples of system software are operating system kernels and libraries) which exists to support applications.

**Application Programming Interface (API)** An interface (calling conventions) by which an application accesses an operating system and other services. An API is defined at source code level and provides a level of abstraction between the application and the kernel (or other privileged utilities) to ensure the portability of the code.

An API can also provide an interface between a high-level language and lower-level utilities and services written without consideration for the calling conventions supported by compiled languages. In this case, the API may translate the parameter lists from one format to another and interpret call-by-value and call-by-reference arguments in one or both directions.

**B**

**Basic Teradata Query (BTEQ)** A CLI application program used to interact with the Teradata Relational Database Management System (RDBMS). BTEQ commands are used for controlling sessions, submitting Teradata SQL requests, formatting results, and handling output data. BTEQ may also be used to verify the installation of Teradata client utilities.

**C**

**Call-Level Interface Version 2 (CLITv2)** Specifically for network-attached clients, a collection of callable service routines that provide an interface to the Teradata Database. Specifically, CLI is the interface between the application program and the Micro Teradata Directory Program (for network-attached clients). CLI builds parcels that MTDP packages for sending to the Teradata Database using the Micro Operating System Interface (MOSI) and
provides the application with a pointer to each of the parcels returned from the Teradata Database.

column In the relational model of Teradata SQL, databases consist of one or more tables. In turn, each table consists of fields, organized into one or more columns by zero or more rows. All of the fields of a given column share the same attributes.

cost This is the outlay of database resources used by a given query.

data definition The statements and facilities that manipulate database structures and the Data Dictionary information kept about these structures. These statements include CREATE, DROP, ALTER, and MODIFY.

Data Definition Language (DDL) In Teradata SQL, the statements and facilities that manipulate database structures (such as CREATE, MODIFY, DROP, GRANT, REVOKE, and GIVE) and the dictionary information kept about those structures. In the typical, pre-relational data management system, data definition and data manipulation facilities are separated, and the data definition facilities are less flexible and more difficult to use than in a relational system.

Data Dictionary In the Teradata Database, the information automatically maintained about all tables, views, macros, databases, and users known to the Teradata Database system. It includes information about ownership, space allocation, accounting, and access right relationships between those objects. Data Dictionary information is updated automatically during the processing of Teradata SQL data definition statements. It is used by the parser to obtain information needed to process all Teradata SQL statements.

database A related set of tables that share a common space allocation and owner. A collection of objects that provide a logical grouping for information. The objects include, tables, views, macros, triggers, and stored procedures.

DBA Acronym for Database Administrator.

DBS Acronym for Database System or Database Software.

DBS Control information DBS Control information is a group of fields used by Teradata Database for debugging and diagnostic purposes, establishing known global system values, and conducting performance tuning.

DBW Acronym for Database Windows.

driver In Teradata PT API, the term that refers to combination of the operators from the Teradata PT product and application programming interface program code.

domain name A group of computers whose hostnames (the unique name by which a computer is known on a network) share a common suffix, that is the domain name.
**E**

***endianness***  The byte ordering convention of data that is represented with multiple bytes. Big-endian is an order in which the “big end” (most significant value in the sequence) is stored first (at the lowest storage address). Little-endian is an order in which the “little end” (least significant value in the sequence) is stored first. For example, in a big-endian computer, the number 256 is indicated as 0x01 0x00. In a little-endian computer, the number 256 is indicated as 0x00 0x01.

***export***  This refers to extracting or transferring system information from the tables and views of a given Teradata Database and saving it so it can be manipulated or pulled into another system.

**Extended Binary Coded Decimal Interchange Code (EBCDIC)**  A character encoding used on IBM mainframe systems.

**F**

***field***  The basic unit of information stored in the Teradata Database. A field is either null, or has a single numeric or string value. Also see column, row, and table.

**G**

***Globally Distributed Objects (GDO)***  A data structure that is shared by all of the virtual processors in the Teradata Database system configuration.

**I**

***ICU***  An acronym for a library of routines that handle Unicode UTF16/UTF8 session character sets.

***import***  The process of pulling system information into a program. To add system information from an external source to another system. The system receiving the data must support the internal format or structure of the data.

**J**

***join***  A SELECT operation that combines information from two or more tables to produce a result.

**L**

***log***  A record of events. A file that records events. Many programs produce log files. Often you will look at a log file to determine what is happening when problems occur. Log files have the extension “.log”.

**macro** A set of Teradata SQL statements stored by the Teradata Database and executed by a single EXECUTE statement. Each macro execution is implicitly treated as a transaction.

**multi-threading** An option in Teradata SET that enables you to speed up your export and import operations with multiple connections.

**name** A word supplied by the user that refers to an object, such as a column, database, macro, table, user, or view.

**null** The absence of a value for a field.

**object** In object-oriented programming, a unique instance of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages, or methods, defined by its class.

**object definition** The details of the structure and instances of the objects used by a given query. Object definitions are used to create the tables, views, and macros, triggers, join indexes, and stored procedures in a database.

**operator** The term used in the Teradata Parallel Transporter (Teradata PT) product to describe the modules of code responsible for handling particular tasks usually relating to one of the Teradata Database protocols. These operators reside in dynamically-linked libraries.

**Open Database Connectivity (ODBC)** Under ODBC, drivers are used to connect applications with databases. The ODBC driver processes ODBC calls from an application, but passes SQL requests to the Teradata Database for processing.

**parameter** A variable name in a macro for which an argument value is substituted when the macro is executed.

**privilege** In Teradata SQL, a user’s right to perform the TeradataSQL statements granted to him or her against a table, database, user, macro, or view. Also known as access right.

**production system** A Teradata Database used in a live environment. A system that is actively used for day to day business operations. This differs from a test or development system that is used to create new queries or test new features before using them on the production system.

**random AMP sample (RAS)** An arbitrary sample from an Access Module Processor (AMP). These are samples of the tables in a query or all of the tables in a given database. Also known as RAS.
**Relational Database Management System (RDBMS)** A database management system in which complex data structures are represented as simple two-dimensional tables consisting of columns and rows. For Teradata, RDBMS is referred to as “Teradata Database.”

**request** In host software, a message sent from an application program to the Teradata Database.

**result** The information returned to the user to satisfy a request made of the Teradata Database.

**row** The fields that represent one entry under each column in a table. The row is the smallest unit of information operated on by data manipulation statements.

**session** In client software, a logical connection between an application program on a host and the Teradata Database. It permits the application program to send one request to and receive one response from the Teradata Database at a time.

**single sign-on (SSO)** Allows users of the Teradata Database on Windows 2000 systems to access the Teradata Database based on their authorized network usernames and passwords. This feature simplifies the procedure requiring users to enter an additional username and password when logging on to Teradata Database via client applications.

**statement** A request for processing by the Teradata Database that consists of a keyword verb, optional phrases, and operands. It is processed as a single entity.

**statistics** These are the details of the processes used to collect, analyze, and transform the database objects used by a given query.

**Standard Template Library (STL)** Standard Template Library is a software library of C++ algorithms, container classes and iterators. It is available on most platforms. Teradata PT API uses features in the STL. The C++ Standard Library is defined by ISO/IEC 14882.

**Structured Query Language Call-Level Interface (SQL/CLI)** A programming interface designed to support SQL access to databases from shrink-wrapped application programs. SQL/CLI provides and international standard implementation-independent CLI to access SQL databases. Client-server tools can easily access database through dynamic link libraries. It supports and encourages a rich set of client-server tools.

**stored procedure** Combinations of procedural and non-procedural statements run using a single call statement.

**table** A two-dimensional structure made up of one or more columns with zero or more rows that consist of fields of related information. See also target table.

**test system** A Teradata Database where you want to import Optimizer-specific information to emulate a target system and create new queries or test new features.
**Target Level Emulation (TLE)**  Permits you to emulate a target environment (target system) by capturing system-level information from that environment. The captured information is stored in the relational tables SystemFE.Opt_Cost_Table and SystemFE.Opt_RAS_Table. The information in these tables can be used on a test system with the appropriate column and indexes to make the Optimizer generate query plans as if it were operating in the target system rather than the test system.

**target table**  For the Load, Update, and Stream drivers, the target table is the table on a Teradata Database which will receive the data and/or contains data to be modified. For the Export driver, the target table is the table from which the data will be exported.

**Teradata Extract and Load Application Programming Interface (TELAPI)**  This is the former name of Teradata PT API.

**Teradata Parallel Data Pump (TPump)**  A utility that provides an alternative to MultiLoad for the low volume batch maintenance of large databases under control of a Teradata Database. TPump enables acquisition of all data from the client with low processor utilization.

**test system**  A Teradata Database where you want to import Optimizer-specific information to emulate a target system and create new queries or test new features.

**trigger**  One or more Teradata SQL statements associated with a table and executed when specified conditions are met.

**user**  A database associated with a person who uses the Teradata Database. The database stores the person’s private information and accesses other Teradata Databases.

**view**  An alternate way of organizing and presenting information in the Teradata Database. A view, like a table, has rows and columns. However, the rows and columns of a view are not directly stored by the Teradata Database. They are derived from the rows and columns of tables (or other views) whenever the view is referenced.
Index

A
AddArrayAttribute 42
AddAttribute 42
AddColumn 51
AddDMLGroup 43
AddDMLOption 53
AddSchema 43
AddSerializeOn
  arguments 35
  syntax 53
AddStatement 53
AddUseList 53
  used with DMLGroup object 36
AIX
  code samples 177
  reporting events
    Export driver 132
    Load driver 67
    Update driver 84
AMPS
  nonparticipant during Update tasks 94
  offline during Update tasks 94
  Single-AMP Systems 95
ANSI/SQL DateTime data type
  specifications, table of 165
ApplyRows 43
array attributes
  defined 24
array support
  AddArraySupport DMLGroup class object 52
  default settings 103
B
BTEQ
  DROP TABLE statement 72, 91
  restarting the Update job 93
  scripts used with code samples
    Linux 179
    SPARC, Opteron, HP-UX and AIX 178
    Windows 176
    z/OS 180
C
CALL, SQL statement
  examples 162
usage notes 163
checkpoint and restart
  checkpoint arguments 33, 34
  Checkpoint, Connection class object 44
  Load driver event (rows checkpointed) 71
  setting in the Connection object 33
  Stream driver event (rows checkpointed) 116
  Update driver event (rows checkpointed) 90
Class Constructor
  Connection class 44
  DMLGroup class 53
  Schema class 51, 52, 54
CLIEnvFile, CLiv2 variable 157
CLISPBDAT 73, 93, 119, 139
CLiv2
  buffer contents 28
  CLI Error status message 59
  CLIEnvFile variable
    on Linux 157
  GetErrorInfo 45
  making direct calls 37
  required levels 4, 175
  TD_BUFFER_SIZE 79
  TD_BUFFER_SIZE defaults 63
  TD_Evt_CLIError 70, 136
    Stream driver event 115
    Update driver event 88
  TD_Evt_ConnectStatus 71
  TD_NOTIFY_LEVEL settings 66
  TD_OPER_CLI, TD_TRACE_LEVEL attribute 58
  TD_TRACE_LEVEL setting 69
Connection class objects
  AddArrayAttribute 42
  AddAttribute 42
  AddDMLGroup 43
  AddSchema 43
  ApplyRows 43
  Checkpoint 44
  EndAcquisition 44
  GetBuffer 45
  GetErrorInfo 44, 45
  GetEvent 46
  GetRow 46
  GetSchema 46
  GetTELINFO 47
  Initiate 47
  PutBuffer 47
PutEvent 48
PutRow 48
PutTELINFO 49
Restart 49
Terminate 49
UseDMLGroups 50

Connection object
creating 23
initiating the connection 27
macro support (Stream driver) 121
parameters 23
reusing 37
terminating 36
z/OS support 41, 51, 52

converting
TIMESTAMP and INTERVAL data types 165
CREATE PROCEDURE, SQL statement
code samples 159, 161
configuring DBS 161
linking libraries 158
On Linux 156
On Windows 157
CREATE SET TABLE, SQL statement 162
Cufconfig utility 156, 157, 162

DateTime specifications, table of 165
DBCHCLN, CLI function 37
deadlocks, Teradata Database 35
DMLGroup class objects
AddDMLOption 52, 53
AddSerializeOn 53
AddStatement 53
AddUseList 53
DMLGroup objects
Class Constructor 53

EndAcquisition 44
endian modes 144
error codes
21044, invalid restart argument values 34
no error return values 33
using GetErrorInfo 33
error tables
duplicate records
Load driver 72
reusing table names
Load driver 72
Stream driver 117
Update driver 91
Update driver 90
using existing tables

Load driver 64
Stream driver 103, 105
Update driver 80

external stored procedure See XSP

FastExport 21, 47, 51
FastLoad
column definitions for schema 51
initiating sessions 47
protocols effect on restart log table
Load driver 64
Update driver 81

GetBuffer
object defined 45
GetErrorInfo 45
GetEvent
Export driver events 136
Load driver events 70
object defined 46
Stream driver events 114
Update driver events 88
using for run-time statistics 34
using in PutBuffer loading 28
GetRow 46
GetTELINFO, Connection class object 47

header files 181
HP-UX
code sample 177
notify exit routine default name
Load driver 66
Stream driver 108
Update driver 83

IBM z/OS
code samples 179
ICU
specifying directories in Linux PATH variables 178
IGNORE_DUPLICATE_INSERT_ROWS, DML Group option 54
IGNORE_DUPLICATE_ROWS, DML Group option 54
IGNORE_DUPLICATE_UPDATE_ROWS, DML Group option 54
IGNORE_EXTRA_DELETE_ROWS, DML Group option 54
IGNORE_EXTRA_ROWS, DML Group option 54
IGNORE_EXTRA_UPDATE_ROWS, DML Group option 54
INDEX

P
performance considerations, factors of, 139
PERIOD data type
   specifying for the schema 51
product version numbers 4
PutBuffer 47
   arguments 29
   method of loading rows 28
PutEvent
   arguments 35
   modifier data 136
PutRow 48
PutTELINFO, Connection class object 49

Q
query banding optional attribute
   defined 37
   Export driver 133
   Load driver 68
   Stream driver 110
   Update driver 85

R
required attributes
   Stream driver 102
required privileges
   Load driver 73
   Stream driver 118
   Update driver 93
restart log table
   using existing restart log table
      Load driver 64
      Update driver 81
Restart, Connection class object 49
return codes 59, 144
run-time statistics 34

S
Schema class objects
   AddColumn 51
   Class Constructor 51, 52, 54
   SetType 52
Schema object
   AddColumn data types
      TD_BIGINT 56
      TD_BYTE 56
      TD_BYTEINT 56
      TD_CHAR 56
      TD_DATE 56
      TD_DATE_ANSI 56
      TD_DECIMAL 56
      TD_FLOAT 56
Index

- TD_GRAPHIC 56
- TD_INTEGER 56
- TD_LONGVARCHAR 56
- TD_LONGVARGRAPHIC 56
- TD_NONE 56
- TD_PERIOD_DATE 56
- TD_PERIOD_TIME 56
- TD_PERIOD_TIME_TZ 56
- TD_PERIOD_TS 56
- TD_PERIOD_TS_TZ 57
- TD_SMALLINT 57
- TD_VARBYTE 57
- TD_VARCHAR 57
- TD_VARGRAPHIC 57

AddSchema function 43
- defined 24
- designating as input or output 24
- using to specify macro parameters 121

serialization
- enabling with DMLGroup object 35
- using with Stream driver 35

session limits
- Export driver 139
- Load driver 73
- Stream driver 118
- Update driver 93

SetType 52

software releases
- supported 3

Solaris SPARC and Opteron
- code samples 177

reporting events
- Export driver 132
- Load driver 67
- Stream driver 109
- Update driver 84

status messages
- CLI Error 59
- DBS Error 59
- TD_Call_EndAcq 59
- TD_END_Method 59
- TD_Error 59
- TD_Success 59
- TD_SYNC_Barrier 60
- TD_SYNC_TELINFO 60
- TD_Unavailable 60
- TPTAPI Error 59

STL (Standard Template Library) 172, 173, 187

T

- TD_ACCOUNT_ID
  - Export driver 129
  - Load driver 62

Stream driver 103

- TD_amp_check
  - settings for nonparticipant AMPs 95
  - using with TD_DELETE_TASK 80

- TD_APPENDERRORTABLE
  - Stream driver 103

- TD_Attribute 56

- TD_BLOCK_SIZE
  - Load driver 61
  - Stream driver 102

- TD_BUFFER_HEADER_SIZE
  - Export driver 129

- TD_BUFFER_LENGTH_SIZE
  - Export driver 129

- TD BUFFER_MAX_SIZE
  - Export driver 129

- TD_BUFFER_MODE
  - Load driver 63
  - Stream driver 104
  - Update driver 79

- TD_BUFFER_TRAILER_SIZE
  - Export driver 130

- TD_CHARSET
  - Export driver 130
  - Load driver 63
  - Stream driver 104
  - Update driver 79

- TD_DATA_ENCRYPTION
  - Export driver 130
  - Stream driver 104
  - Update driver 79

- TD_DataType 56
  - for AddColumn function in the Schema object 56
  - using in the Schema object 24

- TD_DATE_FORM
  - Export driver 130
  - Load driver 64
  - Stream driver 104
  - Update driver 79

- TD_DELETE_TASK
  - Update driver 80

- TD_DROPERRORTABLE
  - Load driver 64
  - Stream driver 105
  - Update driver 80

- TD_DROPLOGTABLE
  - Load driver 64
  - Update driver 81

- TD_DROPWORKTABLE
  - Update driver 81

- TD_ERROR_LIMIT
  - Load driver 64
Stream driver 105
Update driver 81
TD_ERROR_TABLE
Stream driver 106
TD_ERROR_TABLE_1
Load driver 65
Update driver 82
TD_ERROR_TABLE_2
Load driver 65
Update driver 82
TD_EVT_ApplyCount
Load driver 71
Stream driver 115
Update driver 89
TD_EVT_BlockCount
Export driver 136
Load driver 70
Stream driver 115
Update driver 88
TD_EVT_CLIError
Export driver 136
Load driver 70
Stream driver 115
Update driver 88
TD_EVT_ConnectStatus
Export driver 136
Load driver 71
Stream driver 115
Update driver 88
TD_EVT_CPUTime
Export driver 136
Load driver 71
Stream driver 116
Update driver 88
TD_EVT_DBSError
Export driver 136
Load driver 70
Stream driver 115
Update driver 88
TD_EVT_ErrorTable1
Load driver 72
Stream driver 116
Update driver 89
TD_EVT_ErrorTable2
Load driver 72
Update driver 89
TD_EVT_ExitCode
Export driver 136
Load driver 72
Stream driver 116
Update driver 89
TD_EVT_ExportCount
Export driver 136
TD_EVT_RowsCheckpointed
Load driver 71
Stream driver 116
Update driver 90
TD_EVT_Runstats
Load driver 71
Stream driver 116
TD_EVT_Version
Export driver 136
Load driver 72
Stream driver 117
Update driver 90
TD_IGNORE_MAX_DECIMAL_DIGITS
Export driver 130
TD_INSTANCE_NUM
Export driver 127, 128
Load driver 62
Stream driver 102
TD_LOG_TABLE
Load driver 62
Stream driver 102
Update driver 78
TD_LOGON_MECH
Export driver 131
Load driver 65
Stream driver 106
Update driver 82
TD_LOGON_MECH_DATA
Export driver 131
Stream driver 106
Update driver 82
TD_LOGSQL
Export Driver 128
Load Driver 65
Stream Driver 107
Update Driver 82
TD_MACRODATABASE
Stream driver 107
TD_MAX_DECIMAL_DIGITS
Export driver 131
TD_MAX_INSTANCES
Export driver 128
Load driver 62
Stream driver 102
TD_MAX_SESSIONS
Export driver 131
Load driver 66
Stream driver 107
Update driver 83
Update driver values 93
TD_MIN_SESSIONS
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Export driver 131</td>
</tr>
<tr>
<td>Load driver 66</td>
</tr>
<tr>
<td>Stream driver 107</td>
</tr>
<tr>
<td>Update driver 83</td>
</tr>
<tr>
<td>Update driver values 93</td>
</tr>
<tr>
<td><strong>TD_MSG_ENCODING</strong></td>
</tr>
<tr>
<td>Export driver 131</td>
</tr>
<tr>
<td>Load driver 66</td>
</tr>
<tr>
<td><strong>TD_NOTIFY_EXIT</strong></td>
</tr>
<tr>
<td>Export driver 131</td>
</tr>
<tr>
<td>Load driver 66</td>
</tr>
<tr>
<td>Stream driver 108</td>
</tr>
<tr>
<td>Update driver 83</td>
</tr>
<tr>
<td><strong>TD_NOTIFY_LEVEL</strong></td>
</tr>
<tr>
<td>Export driver 132</td>
</tr>
<tr>
<td>Load driver 66</td>
</tr>
<tr>
<td>Stream driver 108</td>
</tr>
<tr>
<td>Update driver 84</td>
</tr>
<tr>
<td><strong>TD_NOTIFY_METHOD</strong></td>
</tr>
<tr>
<td>Export driver 132</td>
</tr>
<tr>
<td>Load driver 67</td>
</tr>
<tr>
<td>Stream driver 109</td>
</tr>
<tr>
<td>Update driver 84</td>
</tr>
<tr>
<td><strong>TD_NOTIFY_STRING</strong></td>
</tr>
<tr>
<td>Export driver 132</td>
</tr>
<tr>
<td>Load driver 67</td>
</tr>
<tr>
<td>Stream driver 109</td>
</tr>
<tr>
<td>Update driver 84</td>
</tr>
<tr>
<td><strong>TD_OperatorType</strong></td>
</tr>
<tr>
<td>58</td>
</tr>
<tr>
<td><strong>TD_OperatorType attributes</strong></td>
</tr>
<tr>
<td><strong>TD_EXPORT</strong> 58</td>
</tr>
<tr>
<td><strong>TD_LOAD</strong> 58</td>
</tr>
<tr>
<td><strong>TD_NO_OPERATOR</strong> 58</td>
</tr>
<tr>
<td><strong>TD_STREAM</strong> 58</td>
</tr>
<tr>
<td><strong>TD_UPDATE</strong> 58</td>
</tr>
<tr>
<td><strong>TD_PACK</strong></td>
</tr>
<tr>
<td>Stream driver 109</td>
</tr>
<tr>
<td><strong>TD_PACKMAXIMUM</strong></td>
</tr>
<tr>
<td>Stream driver 109</td>
</tr>
<tr>
<td><strong>TD_PAUSE_ACQ</strong></td>
</tr>
<tr>
<td>Load driver 67</td>
</tr>
<tr>
<td>Update driver 85</td>
</tr>
<tr>
<td><strong>TD_QUERY_BAND_SESS_INFO</strong></td>
</tr>
<tr>
<td>Export driver 133</td>
</tr>
<tr>
<td>Load driver 68</td>
</tr>
<tr>
<td>Stream driver 110</td>
</tr>
<tr>
<td>Update driver 85</td>
</tr>
<tr>
<td><strong>TD_REPLICATION_OVERRIDE</strong></td>
</tr>
<tr>
<td>Stream driver 110</td>
</tr>
<tr>
<td><strong>TD_RESTARTMODE</strong></td>
</tr>
<tr>
<td>Export driver 128</td>
</tr>
<tr>
<td>Load driver 62</td>
</tr>
<tr>
<td>Stream driver 102</td>
</tr>
<tr>
<td><strong>TD_ROBUST</strong></td>
</tr>
<tr>
<td>Stream driver 111</td>
</tr>
<tr>
<td><strong>TD_SELECT_STMT</strong></td>
</tr>
<tr>
<td>Export driver 128</td>
</tr>
<tr>
<td><strong>TD_SYSTEM_OPERATOR</strong></td>
</tr>
<tr>
<td>Export driver 128</td>
</tr>
<tr>
<td>Load driver 62</td>
</tr>
<tr>
<td>Stream driver 102</td>
</tr>
<tr>
<td><strong>TD_TARGET_TABLE</strong></td>
</tr>
<tr>
<td>Load driver 62</td>
</tr>
<tr>
<td><strong>TD_TDP_ID</strong></td>
</tr>
<tr>
<td>Export driver 134</td>
</tr>
<tr>
<td>Load driver 68</td>
</tr>
<tr>
<td>Stream driver 111</td>
</tr>
<tr>
<td><strong>TD_TENACITY_HOURS</strong></td>
</tr>
<tr>
<td>Export driver 134</td>
</tr>
<tr>
<td>Load driver 68</td>
</tr>
<tr>
<td>Stream driver 112</td>
</tr>
<tr>
<td>use with <strong>TD_DELETE_TASK</strong> 80</td>
</tr>
<tr>
<td><strong>TD_TENACITY_SLEEP</strong></td>
</tr>
<tr>
<td>Export driver 134</td>
</tr>
<tr>
<td>Load driver 68</td>
</tr>
<tr>
<td>Stream driver 112</td>
</tr>
<tr>
<td>use with <strong>TD_DELETE_TASK</strong> 80</td>
</tr>
<tr>
<td><strong>TD_TRACE_LEVEL</strong></td>
</tr>
<tr>
<td>Export driver 135</td>
</tr>
<tr>
<td>Load driver 69</td>
</tr>
<tr>
<td>Stream driver 113</td>
</tr>
<tr>
<td>Update driver 87</td>
</tr>
<tr>
<td><strong>TD_TRACE_LEVEL attributes</strong></td>
</tr>
<tr>
<td><strong>TD_OFF</strong> 58</td>
</tr>
<tr>
<td><strong>TD_OPER</strong> 58</td>
</tr>
<tr>
<td><strong>TD_OPER_ALL</strong> 58</td>
</tr>
<tr>
<td><strong>TD_OPER_CLI</strong> 58</td>
</tr>
<tr>
<td><strong>TD_OPER_NOTIFY</strong></td>
</tr>
<tr>
<td><strong>TD_OPER_OPCOMMON</strong></td>
</tr>
<tr>
<td><strong>TD_TRACE_OUTPUT</strong></td>
</tr>
<tr>
<td>Export driver 135</td>
</tr>
<tr>
<td>Load driver 69</td>
</tr>
<tr>
<td>Stream driver 114</td>
</tr>
<tr>
<td>Update driver 87</td>
</tr>
<tr>
<td><strong>TD_USER_NAME</strong></td>
</tr>
<tr>
<td>Export driver 128</td>
</tr>
<tr>
<td>Load driver 62</td>
</tr>
<tr>
<td>Stream driver 102</td>
</tr>
<tr>
<td><strong>TD_USER_PASSWORD</strong></td>
</tr>
<tr>
<td>Export driver 128</td>
</tr>
<tr>
<td>Load driver 62</td>
</tr>
<tr>
<td>Stream driver 102</td>
</tr>
<tr>
<td><strong>TD_WILDCARDINSERT</strong></td>
</tr>
<tr>
<td>Load driver 70</td>
</tr>
<tr>
<td><strong>TD_WORK_TABLE</strong></td>
</tr>
<tr>
<td>Update driver 87</td>
</tr>
<tr>
<td><strong>TD_WORKINGDATABASE</strong></td>
</tr>
<tr>
<td>Export driver 135</td>
</tr>
</tbody>
</table>
Stream driver 114
Update driver 88
Teradata ICU 4, 175
Teradata PT API
building the connection object 23
parameters 54
status messages 59
TeraGSS 4, 175
Terminate 49
Terminate method, Connection object 36
TIMESTAMP data type
converting 165
definition 165
TPump 51, 101

UNIX
code samples 177
notify exit routine default name
Load driver 66
Stream driver 108
Update driver 83
reporting events
Export driver 132
Load driver 67
Update driver 84
use lists, DMLGroup object 36
UseDMLGroups 50
UTF16
Class Constructor
Connection Class 44
DMLGroup Class 54
Schema Class constructor 52
encoding objects and messages 38
errors 39
schema considerations 38
setting TD_CHARSET 37
using TD_MSG_ENCODING attribute 39
UTF-16 character encoding, see UTF16
UTF8
as Unicode default 38
Class Constructor
Connection Class 44
DMLGroup Class 54
Schema Class 52
schema considerations 38
using TD_MSG_ENCODING 39

Windows
code samples 176
notify exit routine default name
Load driver 66
Stream driver 108
Update driver 83
prerequisites for understanding Teradata PT API 4
reporting events
Export driver 132
Load driver 67
Update driver 83, 84
work tables
Update driver
TD_WORK_TABLE attribute 87
usage 93
workspace errors 95
using existing tables
Update driver 81

XSP
configuring on
Linux 156
Windows 157
defined
software dependencies 156

z/OS
code samples 179