The product or products described in this book are licensed products of Teradata Corporation or its affiliates.

Teradata, BYNET, DBC/1012, DecisionCast, DecisionFlow, DecisionPoint, Eye logo design, InfoWise, Meta Warehouse, MyCommerce, SeeChain, SeeCommerce, SeeRisk, Teradata Decision Experts, Teradata Source Experts, WebAnalyst, and You've Never Seen Your Business Like This Before are trademarks or registered trademarks of Teradata Corporation or its affiliates.
Adaptec and SCSISelect are trademarks or registered trademarks of Adaptec, Inc.
AMD Opteron and Opteron are trademarks of Advanced Micro Devices, Inc.
BakBone and NetVault are trademarks or registered trademarks of BakBone Software, Inc.
EMC, PowerPath, SRDF, and Symmetrix are registered trademarks of EMC Corporation.
GoldenGate is a trademark of GoldenGate Software, Inc.
Hewlett-Packard and HP are registered trademarks of Hewlett-Packard Company.
Intel, Pentium, and XEON are registered trademarks of Intel Corporation.
IBM, CICS, RACF, Tivoli, and z/OS are registered trademarks of International Business Machines Corporation.
Linux is a registered trademark of Linus Torvalds.
LSI and Engenio are registered trademarks of LSI Corporation.
Microsoft, Active Directory, Windows, Windows NT, and Windows Server are registered trademarks of Microsoft Corporation in the United States and other countries.
Novell and SUSE are registered trademarks of Novell, Inc., in the United States and other countries.
QLogic and SANbox are trademarks or registered trademarks of QLogic Corporation.
SAS and SAS/C are trademarks or registered trademarks of SAS Institute Inc.
SPARC is a registered trademark of SPARC International, Inc.
Sun Microsystems, Solaris, Sun, and Sun Java are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries.
Symantec, NetBackup, and VERITAS are trademarks or registered trademarks of Symantec Corporation or its affiliates in the United States and other countries.
Unicode is a collective membership mark and a service mark of Unicode, Inc.
UNIX is a registered trademark of The Open Group in the United States and other countries.
Other product and company names mentioned herein may be the trademarks of their respective owners.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS-IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. IN NO EVENT WILL TERADATA CORPORATION BE LIABLE FOR ANY INDIRECT, DIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS OR LOST SAVINGS, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The information contained in this document may contain references or cross-references to features, functions, products, or services that are not announced or available in your country. Such references do not imply that Teradata Corporation intends to announce such features, functions, products, or services in your country. Please consult your local Teradata Corporation representative for those features, functions, products, or services available in your country.

Information contained in this document may contain technical inaccuracies or typographical errors. Information may be changed or updated without notice. Teradata Corporation may also make improvements or changes in the products or services described in this information at any time without notice.

To maintain the quality of our products and services, we would like your comments on the accuracy, clarity, organization, and value of this document. Please e-mail: teradata-books@lists.teradata.com

Any comments or materials (collectively referred to as "Feedback") sent to Teradata Corporation will be deemed non-confidential. Teradata Corporation will have no obligation of any kind with respect to Feedback and will be free to use, reproduce, disclose, exhibit, display, transform, create derivative works of, and distribute the Feedback and derivative works thereof without limitation on a royalty-free basis. Further, Teradata Corporation will be free to use any ideas, concepts, know-how, or techniques contained in such Feedback for any purpose whatsoever, including developing, manufacturing, or marketing products or services incorporating Feedback.

Copyright © 2005-2010 by Teradata Corporation. All Rights Reserved.
Preface

Purpose

This book provides information on how to use Teradata Parallel Transporter (Teradata PT), a Teradata® Tools and Utilities product.

Teradata PT provides high-performance data extraction, loading, and updating operations for the Teradata Database.

Teradata Tools and Utilities is a group of client products designed to work with the Teradata Database.

Audience

This book is intended for use by:

- System and application programmers
- System administrators
- Data administrators
- Relational database developers
- System operators
- Other database specialists using Teradata PT

Supported Releases

This book applies to the following releases:

- Teradata Database 13.10
- Teradata Tools and Utilities 13.10
- Teradata Parallel Transporter 13.10

Note: See “Verifying the Teradata PT Version” on page 37 to verify the Teradata Parallel Transporter version number.

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
- Relational database management systems
- SQL and Teradata SQL
- Basic concepts and facilities of the Teradata Database
- Connectivity software, such as ODBC or CLI
- Teradata utilities that load and retrieve data
- C programming (for NotifyExit Routines only)

## Changes to This Book

The following changes were made to this book in support of the current release. Changes since the last publication 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.

Some new functions and features for the current release of Teradata PT might not be documented in this manual. New Teradata PT features and functions are also documented in the other manuals listed in “Additional Information” on page 5.

<table>
<thead>
<tr>
<th>Date and Release</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>February 2010</td>
<td>New product release number scheme for this release.</td>
</tr>
<tr>
<td>13.10</td>
<td></td>
</tr>
<tr>
<td>April 2009</td>
<td>Clarified documentation of job variables.</td>
</tr>
<tr>
<td>13.00</td>
<td>Documented Teradata PT support for NoPI (No Primary Index) tables.</td>
</tr>
<tr>
<td>September 2008</td>
<td>Book rebranded</td>
</tr>
<tr>
<td>13.00</td>
<td></td>
</tr>
</tbody>
</table>
Additional Information

Additional information that supports this product and the Teradata Tools and Utilities is available at the following web sites.

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 mmyx 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| Book completely reorganized and revised. It now contains information on:  
• Teradata PT basics, including how to write a Teradata PT job script, Teradata Database effects on job scripts, and pre-job setup  
• Teradata PT job strategies  
• Teradata PT operators and access modules  
• Launching, managing, and troubleshooting a job  
• Advanced topics, including advanced Teradata Database considerations, advanced scripting strategies, extended character sets, and operational metadata  
• Job script examples  
• Teradata PT Wizard  
• JCL examples  
• Example logs |

### Additional Information

<table>
<thead>
<tr>
<th>Type of Information</th>
<th>Description</th>
<th>Access to Information</th>
</tr>
</thead>
</table>
| Release overview    | Use the Release Definition for the following information:  
• Overview of all of the products in the release  
• Information received too late to be included in the manuals  
• Operating systems and Teradata Database versions that are certified to work with each product  
• Version numbers of each product and the documentation for each product  
• Information about available training and the support center | 1 Go to [http://www.info.teradata.com/](http://www.info.teradata.com/).  
2 Under **Online Publications**, click **General Search**.  
3 Type 2029 in the **Publication Product ID** box.  
4 Click **Search**.  
5 Select the appropriate Release Definition from the search results. |
<p>| Late information    |            |                       |</p>
<table>
<thead>
<tr>
<th>Type of Information</th>
<th>Description</th>
<th>Access to Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additional product information</td>
<td>Use the Teradata Information Products website to view or download specific manuals that supply related or additional information to this manual.</td>
<td>1 Go to <a href="http://www.info.teradata.com/">http://www.info.teradata.com/</a>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2 Under the <strong>Online Publications</strong> subcategory, <strong>Browse by Category</strong>, click <strong>Data Warehousing</strong>.</td>
</tr>
<tr>
<td></td>
<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 <strong>Teradata Tools and Utilities</strong>, and then select an item under <strong>Releases</strong> or <strong>Products</strong>.</td>
<td>4 Select a link to any of the data warehousing publications categories listed.</td>
</tr>
<tr>
<td></td>
<td>• Select a link to any of the data warehousing publications categories listed.</td>
<td>Specific books related to Teradata PT are as follows:</td>
</tr>
</tbody>
</table>
|                                        | • **Teradata Tools and Utilities Access Module**  
  Programmer Guide  
  B035-2424-mmymA  
  Reference  
  B035-2425-mmymA  
  Operator  
  Programmer Guide  
  B035-2435-mmymA  
  Reference  
  B035-2436-mmymA  
  International Character Set Support  
  B035-1125-mmymA  
  Tools and Utilities Installation Guide for  
  Microsoft Windows  
  B035-2407-mmymA  
  UNIX and Linux  
  B035-2459-mmymA  
 IBM z/OS  
  B035-2458-mmymA | 5 For a list of Teradata Tools and Utilities documents, click **Teradata Tools and Utilities**, and then select an item under **Releases** or **Products**.                                                                                                                             | 6 Select a link to any of the data warehousing publications categories listed.                           |
|                                        | • Select a link to any of the data warehousing publications categories listed.                                                                                                                                                                                                                                                                     | Specific books related to Teradata PT are as follows:                                                        |
|                                        | • **Teradata Tools and Utilities Access Module**  
  Programmer Guide  
  B035-2424-mmymA  
  Reference  
  B035-2425-mmymA  
  Operator  
  Programmer Guide  
  B035-2435-mmymA  
  Reference  
  B035-2436-mmymA  
  International Character Set Support  
  B035-1125-mmymA  
  Tools and Utilities Installation Guide for  
  Microsoft Windows  
  B035-2407-mmymA  
  UNIX and Linux  
  B035-2459-mmymA  
  IBM z/OS  
  B035-2458-mmymA | 5 For a list of Teradata Tools and Utilities documents, click **Teradata Tools and Utilities**, and then select an item under **Releases** or **Products**.                                                                                                                             | 6 Select a link to any of the data warehousing publications categories listed.                           |
| CD-ROM images                          | Access a link to a downloadable CD-ROM image of all customer documentation for this release. Customers are authorized to create CD-ROMs for their use from this image.                                                                                                                                                              | 1 Go to [http://www.info.teradata.com/](http://www.info.teradata.com/).                                         |
| Ordering information for manuals       | Use the Teradata Information Products website to order printed versions of manuals.                                                                                                                                                                                                                                                                    | 2 Under the **Online Publications** subcategory, **Browse by Category**, click **Data Warehousing**.                                                     |
|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                   | 3 Click **CD-ROM List and Images**.                                                                        |
|                                        |                                                                                                                                                                                                                                                                                                                                                                                                                   | 4 Follow the ordering instructions.                                                                       |

Teradata Parallel Transporter User Guide
### 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
- Press releases, mentions, and media resources

### Access to Information

1. Go to Teradata.com.
2. Select a link.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preface</strong></td>
<td>3</td>
</tr>
<tr>
<td>Purpose</td>
<td>3</td>
</tr>
<tr>
<td>Audience</td>
<td>3</td>
</tr>
<tr>
<td>Supported Releases</td>
<td>3</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>3</td>
</tr>
<tr>
<td>Changes to This Book</td>
<td>4</td>
</tr>
<tr>
<td>Additional Information</td>
<td>4</td>
</tr>
</tbody>
</table>

## SECTION 1 Teradata PT Basics

### Chapter 1: Introduction to Teradata PT

- High-Level Description 25
- Basic Processing 28
- Teradata PT Parallel Environment 29
- Operator Types 31
- Access Modules 35
- Data Streams 36
- Verifying the Teradata PT Version 37
- Switching Versions 37
- Related Teradata PT Publications 37

### Chapter 2: Teradata PT Job Components

- Fast Track Job Scripting 39
- Understanding Job Script Concepts 41
- Creating a Job Script 44
Defining the Job Header and Job Name ............................................................44
Using Job Variables .........................................................................................46
Defining a Schema .........................................................................................48
Defining Operators ..........................................................................................50
Coding the Executable Section .......................................................................60
Defining Job Steps. ..........................................................................................63

SECTION 2 Pre-Job Setup

Chapter 3:
Job Setup Tasks ...............................................................................................67
Setting Up Configuration Files ........................................................................67
Setting Up the Job Variables Files ....................................................................68
Setting Up the Teradata Database ....................................................................69
Setting Up the Client System ...........................................................................71

Chapter 4:
Teradata Database Effects on Job Scripts ......................................................73
Teradata Database Logon Security .....................................................................73
Teradata Database Access Privileges ...............................................................78
Optimizing Job Performance with Sessions and Instances ...............................80
Limits on Teradata PT Task Concurrency .........................................................86

SECTION 3 Job Strategies

Chapter 5:
Moving External Data into Teradata Database ............................................91
Data Flow Description ......................................................................................91
Comparing Applicable Operators ......................................................................92
Using Access Modules to Read Data from an External Data Source ..............96
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
</table>

Using Teradata PT EasyLoader ............................................. 97
Common Jobs for Moving Data into a Teradata Database .................. 106

**Chapter 6:**
**Moving Data from Teradata Database to an External Target** . . 127

Data Flow Description ............................................................ 127
Comparing Applicable Operators ............................................... 128
Using Access Modules to Process Data Before Writing to External Targets ...... 130
Common Data Movement Jobs .................................................... 131

**Chapter 7:**
**Moving Data within the Teradata Database Environment** . . 139

Data Flow Description ............................................................ 139
Comparing Applicable Operators ............................................... 140
Common Jobs to Move Data within a Teradata Database .................. 140

**SECTION 4 Operator Usage Strategies**

**Chapter 8:**
**DataConnector Operator** ....................................................... 151

DataConnector Operator Capabilities ........................................ 151
DataConnector Operator as a Producer ....................................... 151
DataConnector Operator as a Consumer ...................................... 152
Key Specifications ..................................................................... 153
Operational Considerations ....................................................... 160

**Chapter 9:**
**DDL Operator** ................................................................. 161

DDL Operator Capabilities ....................................................... 161
Table of Contents

DDL Operator Definition ........................................162
Key Specifications .................................................163
Job Options .....................................................164
Operational Considerations ....................................166

Chapter 10:
Export Operator ..............................................167

Export Operator Capabilities ....................................167
FastExport Utility and the Export Operator .................168
Export Operator Definition ....................................169
Key Specifications ..............................................170
Job Options .....................................................172
Operational Considerations ....................................174

Chapter 11:
FastExport OUTMOD Adapter Operator ..................177

FastExport OUTMOD Adapter Operator Capabilities ..........177
FastExport OUTMOD Adapter Operator Definition ..........178
Key Specifications ..............................................178
Job Options .....................................................179
Operational Considerations ....................................179

Chapter 12:
FastLoad INMOD Adapter Operator ......................181

FastLoad INMOD Adapter Operator Capabilities ..........181
FastLoad INMOD Adapter Operator Definition ..........182
Key Specifications ..............................................182
Job Options .....................................................183
Operational Considerations ....................................183
Chapter 13: Load Operator

Load Operator Capabilities ........................................ 185
FastLoad Utility and the Load Operator ............................. 186
Load Operator Definition ........................................... 187
Key Specifications ...................................................... 188
Job Options ............................................................. 191
Operational Considerations ......................................... 192

Chapter 14: MultiLoad INMOD Adapter Operator

MultiLoad INMOD Adapter Operator Capabilities .............. 195
MultiLoad INMOD Adapter Operator Definition ................ 196
Key Specifications ...................................................... 196

Chapter 15: ODBC Operator

ODBC Operator Capabilities ........................................ 199
ODBC Operator Definition ........................................... 200
Key Specifications ...................................................... 201
Job Options ............................................................. 202
Operational Considerations ......................................... 205

Chapter 16: OS Command Operator

OS Command Operator Capabilities ................................. 207
OS Command Operator Definition .................................. 208
Key Specifications ...................................................... 208
<table>
<thead>
<tr>
<th>Chapter 17: SQL Inserter Operator</th>
<th>211</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Inserter Operator Capabilities</td>
<td>211</td>
</tr>
<tr>
<td>SQL Inserter Operator and the Load Operator</td>
<td>212</td>
</tr>
<tr>
<td>SQL Inserter Operator Definition</td>
<td>212</td>
</tr>
<tr>
<td>Key Specifications</td>
<td>213</td>
</tr>
<tr>
<td>Job Options</td>
<td>214</td>
</tr>
<tr>
<td>Operational Considerations</td>
<td>214</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 18: SQL Selector Operator</th>
<th>215</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Selector Operator Capabilities</td>
<td>215</td>
</tr>
<tr>
<td>SQL Selector Operator and the Export Operator</td>
<td>215</td>
</tr>
<tr>
<td>SQL Selector Operator Definition: Not for LOB Exporting</td>
<td>216</td>
</tr>
<tr>
<td>SQL Selector Operator Definition: For LOB Exporting</td>
<td>217</td>
</tr>
<tr>
<td>Key Specifications</td>
<td>217</td>
</tr>
<tr>
<td>Job Options</td>
<td>219</td>
</tr>
<tr>
<td>Operational Considerations</td>
<td>221</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 19: Stream Operator</th>
<th>223</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream Operator Capabilities</td>
<td>223</td>
</tr>
<tr>
<td>TPump Utility and the Stream Operator</td>
<td>224</td>
</tr>
<tr>
<td>Stream Operator Definition</td>
<td>226</td>
</tr>
<tr>
<td>Key Specifications</td>
<td>226</td>
</tr>
<tr>
<td>Job Options</td>
<td>229</td>
</tr>
<tr>
<td>Operational Considerations</td>
<td>234</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 20: Update Operator</th>
<th>239</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update Operator Capabilities</td>
<td>239</td>
</tr>
<tr>
<td>MultiLoad Utility and the Update Operator</td>
<td>241</td>
</tr>
</tbody>
</table>
 SECTION 5 Launching, Managing, and Troubleshooting a Job

 Chapter 21: LAUNCHING A JOB

 Setting tbuild Options

 Setting Checkpoint Options

 Launching a Teradata PT Job

 Chapter 22: MANAGING AN ACTIVE JOB

 Managing an Active Job

 Using twbstat to List Currently Active Jobs

 Using the twbcmd Command to Monitor and Manage Job Performance

 Using twbkill to Terminate a Job

 Chapter 23: POST-JOB CONSIDERATIONS

 Post-Job Checklist

 Exit Codes

 Accessing and Using Job Logs

 Accessing and Using Error Tables

 Effects of Error Limits

 Dropping Error Tables

 Restart Log Tables

 Strategies for Evaluating a Successful Job
# Chapter 24: Troubleshooting a Failed Job

- Detecting and Correcting the Cause of Failure ........................................ 293
- Common Job Failures and Remedies ......................................................... 294
- When the Job Fails to Begin Running ...................................................... 294
- When the Job Fails to Complete ............................................................. 301
- Operator-Specific Error Handling ......................................................... 303
- Load Operator Errors ................................................................. 304
- Stream Operator Errors ............................................................... 307
- Update Operator Errors ................................................................. 311
- SQL Selector Operator Errors ......................................................... 316
- Additional Debugging Strategies for Complex Job Failures ................. 317
- Restarting A Job ........................................................................... 318
- Removing Checkpoint Files .......................................................... 323

# Chapter 25: Notify Exit Routines

- Notify Exit Routines ......................................................................... 325
- Using Notify Exit Routines to Monitor Events .................................... 337
- Compiling and Linking Notify Exit Routines ....................................... 338
- Examples ......................................................................................... 343

## SECTION 6 Advanced Topics

# Chapter 26: Advanced Teradata Database Considerations

- Query Banding Considerations .......................................................... 365
- Large Objects ............................................................................... 366
- Data Conversions ........................................................................ 370
**Chapter 27:**  
**Advanced Scripting Strategies**  
Data Acquisition and Loading Options ........................................... 373  
Data Filtering and Conditioning Options ........................................ 378  
Reusing Definitions with the INCLUDE Directive ............................... 380  
Using the Multiple APPLY Feature .................................................. 381

**Chapter 28:**  
**Extended Character Sets** .......................................................... 385  
Using Extended Character Sets ..................................................... 385  
Using LONG VARCHAR with Unicode Character Sets ......................... 393

**Chapter 29:**  
**Operational Metadata** ............................................................. 395  
Metadata Types ............................................................................. 395  
Example Metadata Log Output ....................................................... 397  
Viewing Metadata ........................................................................ 398  
Exporting and Loading Metadata ................................................... 399  
Analyzing Job Metadata .................................................................. 399

**Appendix A:**  
**Job Script Examples** ................................................................. 401  
How to Use Job Script Examples ..................................................... 401  
Job Script Example Library ............................................................. 403

**Appendix B:**  
**Teradata PT Wizard** ................................................................. 449  
Launching TPT Wizard .................................................................... 449  
Overview ....................................................................................... 449  
Wizard Limitations ....................................................................... 450  
Main Window .................................................................................. 451
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a New Script</td>
<td>453</td>
</tr>
<tr>
<td>Stop, Restart, Delete, Edit Jobs</td>
<td>473</td>
</tr>
<tr>
<td>View Job Output</td>
<td>475</td>
</tr>
<tr>
<td>Menus and Toolbars</td>
<td>477</td>
</tr>
<tr>
<td><strong>Appendix C:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>z/OS Sample Files</strong></td>
<td></td>
</tr>
<tr>
<td>Job Script Examples</td>
<td>479</td>
</tr>
<tr>
<td>JCL Sample</td>
<td>480</td>
</tr>
<tr>
<td>Job Attribute File</td>
<td>480</td>
</tr>
<tr>
<td>Teradata PT Catalogued Procedure</td>
<td>481</td>
</tr>
<tr>
<td>Teradata PTLV Catalogued Procedure</td>
<td>481</td>
</tr>
<tr>
<td><strong>Appendix D:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Example Logs</strong></td>
<td></td>
</tr>
<tr>
<td>Example Logs for a Successful Job</td>
<td>483</td>
</tr>
<tr>
<td>Example Logs for an Unsuccessful Job</td>
<td>499</td>
</tr>
<tr>
<td><strong>Glossary</strong></td>
<td>509</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>513</td>
</tr>
</tbody>
</table>
List of Figures

Figure 1: Contrasting Traditional Utilities and Teradata PT ........................................ 29
Figure 2: Teradata PT Pipeline Parallelism................................................................. 30
Figure 3: Teradata PT Data Parallelism................................................................. 31
Figure 4: Job Flow Using a FastLoad INMOD Adapter Operator ............................. 34
Figure 5: Job Flow Using an INMOD Adapter Operator ......................................... 34
Figure 6: Job Flow Using an OUTMOD Adapter Operator ...................................... 34
Figure 7: Data Streams ......................................................................................... 37
Figure 8: Script Sections ...................................................................................... 41
Figure 9: Job Header and Job Name ....................................................................... 45
Figure 10: Example Schema Definition ................................................................. 49
Figure 11: Defining producer operators ................................................................. 51
Figure 12: Export Operator Definition ................................................................. 53
Figure 13: Defining consumer operators ............................................................... 53
Figure 14: Load Operator .................................................................................... 55
Figure 15: Example script for defining the DDL operator ...................................... 56
Figure 16: Multiple Insert Statements ................................................................. 60
Figure 17: SELECT Statement in an APPLY Statement ........................................ 61
Figure 18: Setup Tables Prior to Loading Data .................................................... 70
Figure 19: Copy Files from One Client Location to Another before Executing and Extract/Load Operation .......................................................... 72
Figure 20: Moving Data from a Non-Teradata Source into Teradata Database ......... 91
Figure 21: Job Example 1A -- Reading Data from a Flat File for High Speed Loading ... 107
Figure 22: Job Example 1B -- Reading Data from a Named Pipe for High Speed Loading. 108
Figure 23: Job Example 1C -- Reading Data from Multiple Flat Files for High Speed Loading . 108
Figure 24: Job Example 2A -- Reading Data from a Flat File ................................. 110
Figure 25: Job Example 2B -- Reading Data from a Named Pipe ......................... 110
Figure 26: Job Example 3 -- Loading BLOB and CLOB Data .................................. 112
Figure 27: Job Example 4 -- Pre-processing Data with an INMOD Routine before Loading114
Figure 28: Job Example 5A -- Read Transactional Data from JMS and Load Using the Stream Operator ................................................................. 116
Figure 29: Job Example 5B -- Read Transactional Data from MQ and Load Using the Stream Operator ................................................................. 117
List of Tables

Table 1: Comparison of Teradata PT Operators and Teradata Utilities .................. 26
Table 2: Operator Summary ................................................................. 34
Table 3: Producer Operators .............................................................. 51
Table 4: Consumer Operators ............................................................. 54
Table 5: Standalone Operators .......................................................... 55
Table 6: Comparing Update and Stream Operators ........................................ 95
Table 7: Valid FileName Syntax ............................................................. 157
Table 8: Export Operator Supported Features ............................................ 168
Table 9: SELECT Requests ................................................................. 171
Table 10: Load Operator Supported Features ............................................. 186
Table 11: Stream Operator Feature Support ............................................. 224
Table 12: Update Operator Feature Support ............................................. 241
Table 13: AMP Usage ................................................................. 250
Table 14: Checkpoint Types and Functions ............................................... 265
Table 15: Load Errors ................................................................. 304
Table 16: Format of ErrorTable1 .......................................................... 305
Table 17: Error Table Columns ............................................................ 308
Table 18: Task Example ................................................................. 310
Table 19: Acquisition Error Table Format for the Update Operator ................. 313
Table 20: Application Error Table Format for the Update Operator .................. 314
Table 21: Task Example ................................................................. 315
Table 22: Export Operator Notify Event Codes .......................................... 326
Table 23: Export Operator Events That Create Notifications .......................... 327
Table 24: Load Operator Notify Event Codes ........................................... 329
Table 25: Load Operator Events That Create Notifications ............................ 330
Table 26: Update Operator Notify Event Codes ........................................ 331
Table 27: Update Operator Events That Create Notifications ......................... 333
Table 28: Stream Operator Notify Event Codes ......................................... 334
Table 29: Stream Operator Events that Create Notifications ......................... 336
Table 30: UNIX Syntax for Notify Exit Routines ........................................ 339
Table 31: HP-UX Syntax for Notify Exit Routines ....................................... 340
Table 32: Linux Syntax for Notify Exit Routines ....................................... 340
List of Tables

Table 33: IBM AIX Syntax for Notify Exit Routines ................................................. 341
Table 34: z/OS Syntax for Notify Exit Routines ...................................................... 343
Table 35: Data Conditioning Syntax Using SELECT ................................................. 378
Table 36: Specifying Unicode character sets in a Teradata PT Job Script .............. 389
Table 37: Menu Items ............................................................................................ 477
Table 38: Toolbar .................................................................................................... 477
Table 39: Job Script Examples ............................................................................... 479
SECTION 1 Teradata PT Basics
Introduction to Teradata PT

The chapter provides an overview of the Teradata PT product.

Topics include:

- High-Level Description
- Basic Processing
- Teradata PT Parallel Environment
- Operator Types
- Access Modules
- Data Streams
- Verifying the Teradata PT Version
- Switching Versions
- Related Teradata PT Publications

High-Level Description

Teradata PT is an object-oriented client application that provides scalable, high-speed, parallel data:

- Extraction
- Loading
- Updating

These capabilities can be extended with customizations or with third-party products.

Teradata PT uses and expands on the functionality of the traditional Teradata extract and load utilities, that is, FastLoad, MultiLoad, FastExport, and TPump, also known as standalone utilities.

Teradata PT supports:

- **Process-specific operators**: Teradata PT jobs are run using operators. These are discrete object-oriented modules that perform specific extraction, loading, and updating processes.
- **Access modules**: These are software modules that give Teradata PT access to various data stores.
- **A parallel execution structure**: Teradata PT can simultaneously load data from multiple and dissimilar data sources into, and extract data from, Teradata Database. In addition,
Teradata PT can execute multiple instances of an operator to run multiple and concurrent loads and extracts and perform inline updating of data. Teradata PT maximizes throughput performance through scalability and parallelism.

- **The use of data streams**: Teradata PT distributes data into data streams shared with multiple instances of operators to scale up data parallelism. Data streaming eliminates the need for intermediate data storage: data is streamed through the process without being written to disk.

- **A single SQL-like scripting language**: Unlike the traditional standalone utilities that each use their own scripting language, Teradata PT uses a single script language to specify extraction, loading, and updating operations.

- **An application programming interface (API)**: Teradata PT can be invoked with scripts or with the Teradata PT set of open APIs. Using the Teradata PT open APIs allows third-party applications to execute Teradata PT operators directly. This makes Teradata PT extensible.

- **A GUI-based Teradata PT Wizard**: The Teradata PT Wizard helps you generate simple Teradata PT job scripts.

### Teradata PT and the Teradata Utilities

Teradata PT replaces Teradata Warehouse Builder. For example, instead of running FastLoad, Teradata PT uses the Load operator. Instead of running MultiLoad, Teradata PT uses the Update operator.

Table 1 compares Teradata PT operators with Teradata utilities.

**Table 1: Comparison of Teradata PT Operators and Teradata Utilities**

<table>
<thead>
<tr>
<th>Teradata PT Operator</th>
<th>Utility Equivalent</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataConnector operator</td>
<td>Data Connector (PIOM)</td>
<td>Reads data from and writes data to flat files</td>
</tr>
<tr>
<td>DataConnector operator with</td>
<td>same with Data Connector (PIOM)</td>
<td>Reads data from IBM WebSphere MQ</td>
</tr>
<tr>
<td>WebSphere MQ© Access Module</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataConnector operator with</td>
<td>same with Data Connector (PIOM)</td>
<td>Reads data from a named pipe</td>
</tr>
<tr>
<td>Named Pipes Access Module</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDL operator</td>
<td>BTEQ</td>
<td>Executes DDL, DCL, and self-contained DML SQL statements</td>
</tr>
<tr>
<td>Export operator</td>
<td>FastExport</td>
<td>Exports data from Teradata Database (high-volume export)</td>
</tr>
<tr>
<td>FastExport OUTMOD Adapter operator</td>
<td>FastExport OUTMOD Routine</td>
<td>Preprocesses exported data with a FastExport OUTMOD routine before writing the data to a file</td>
</tr>
<tr>
<td>FastLoad INMOD Adapter operator</td>
<td>FastLoad INMOD Routine</td>
<td>Reads and preprocesses data from a FastLoad INMOD data source</td>
</tr>
<tr>
<td>Load operator</td>
<td>FastLoad</td>
<td>Loads an empty table (high-volume load)</td>
</tr>
</tbody>
</table>
Platforms

Teradata PT 13.10 is certified to operate on most platform environments that Teradata Clients support.

For a detailed list of supported platform environments for Teradata PT, as well as other Teradata Tools and Utilities, see Teradata Tools and Utilities 13.10 Supported Platforms and Product Versions, B036-3119-xxxA. For information about how to access this and other related publications, see “Supported Releases” on page 3.

Note: The 13.10 TPT products are compiled on the AIX 5.3 using the the xlC version 9 and must run on the AIX machine with the same level or higher C++ runtime library version 9.0 and C runtime library version 5.3. For more information, see Teradata Tools and Utilities Installation Guide for UNIX and Linux B035-2459-xxxA

Compatibilities

Observe the following information about job script compatibility.

- Scripts written for the former Teradata Warehouse Builder work with Teradata PT without modification, but Teradata Warehouse Builder scripts cannot employ new Teradata PT features. Teradata recommends that all new scripts be written using the Teradata PT scripting language.
- Scripts written for Teradata standalone utilities are incompatible with Teradata PT. Teradata recommends that existing standalone utility scripts be reworked using Teradata PT scripting language. Contact Professional Services for help.
Other Vendors
ETL vendor products can be used with Teradata PT to generate scripts for load operations or to make API calls:

- **Extract, Transform, and Load (ETL)** vendors add value by performing:
  - Data extractions and transformations prior to loading Teradata Database. Teradata PT provides the ability to condition, condense, and filter data from multiple sources through the Teradata PT SELECT statement.
  - Data extractions and loading, but leaving all the complex SQL processing of data to occur inside the Teradata Database itself. Like ETL vendors, Teradata PT can condition, condense, and filter data from multiple sources into files.
- **The Teradata PT API** provides additional advantages for third-party ETL/ELT vendors. For more information, see *Teradata Parallel Transporter Application Programming Interface Programmer Guide*.

Basic Processing
Teradata PT can load data into, and export data from, any accessible database object in the Teradata Database or other data store using Teradata PT operators or access modules.

Multiple targets are possible in a single Teradata PT job. A data target or destination for a Teradata PT job can be any of the following:

- Databases (both relational and non-relational)
- Database servers
- Data storage devices
- File objects, texts, and comma separated values (CSV)

**Note:** Full tape support is not available for any function in Teradata PT for network-attached client systems. To import or export data using a tape, a custom access module must be written to interface with the tape device. See *Teradata Tools and Utilities Access Module Programmer Guide*, B035-2424-mmyx for information about how to write a custom access module.

When job scripts are submitted, Teradata PT can do the following:

- Analyze the statements in the job script.
- Initialize its internal components.
- Create, optimize, and execute a parallel plan for completing the job by:
  - Creating instances of the required operator objects.
  - Creating a network of data streams that interconnect the operator instances.
  - Coordinating the execution of the operators.
- Coordinate checkpoint and restart processing.
- Restart the job automatically when the Teradata Database signals restart.
- Terminate the processing environments.
Between the data source and destination, Teradata PT jobs can:

- Retrieve, store, and transport specific data objects using parallel data streams.
- Merge or split multiple parallel data streams.
- Duplicate data streams for loading multiple targets.
- Filter, condition, and cleanse data.

**Teradata PT Parallel Environment**

Although the traditional Teradata standalone utilities offer load and extract functions, these utilities are limited to a serial environment.

Figure 1 illustrates the parallel environment of Teradata PT.

Teradata PT uses data streams that act as a pipeline between operators. With data streams, data basically flows from one operator to another.

Teradata PT supports the following types of environments:

- **Pipeline Parallelism**
- **Data Parallelism**

**Pipeline Parallelism**

Teradata PT pipeline parallelism is achieved by connecting operator instances through data streams during a single job.
Figure 2 shows:

- An export operator on the left that extracts data from a data source and writes it to the data stream.
- A filter operator extracts data from the data stream, processes it, then writes it to another data stream.
- A load operator starts writing data to a target as soon as data is available from the data stream.

All three operators, each running its own process, can operate independently and concurrently.

As the figure shows, data sources and destinations for Teradata PT jobs can include:

- Databases (both relational and non-relational)
- Database servers
- Data storage devices, such as tapes or DVD readers
- File objects, such as images, pictures, voice, and text

Figure 2: Teradata PT Pipeline Parallelism

Data Parallelism

Figure 3 shows how larger quantities of data can be processed by partitioning a source data into a number of separate sets, with each partition handled by a separate instance of an operator.
Operator Types

Teradata PT provides four functional operator types:

- **Producer operators**: read data from a source and write to a data stream.
- **Consumer operators**: read from data streams and write to a data target.
- **Filter operators**: read data from data streams, perform data filtering functions such as selection, validation, cleansing, and condensing, and then write filtered data to data streams.
- **Standalone operators**: perform processing that does not involve receiving data from or sending data to the data stream.

**Producer Operators**

Producer operators read data from a source and write it to a data stream.

Teradata PT includes the following producer operators:

- Export operator
- SQL Selector operator
- DataConnector operator, when reading data:
  - Directly from a flat file
Chapter 1: Introduction to Teradata PT
Operator Types

- Through an access module that reads data from an external source
- FastLoad INMOD Adapter operator
- MultiLoad INMOD Adapter operator
- ODBC operator

Producer operators are summarized in Table 2 on page 34. For detailed descriptions and required syntax, see Teradata Parallel Transporter Reference.

Consumer Operators

Consumer operators “consume” data from a data stream and write it to a file or database.

Teradata PT provides the following consumer operators:

- Load operator
- Update operator
- Stream operator
- SQL Inserter operator
- DataConnector operator, when interfacing an access module that writes data to an external destination
- FastExport OUTMOD Adapter operator

Consumer operators are summarized in Table 2 on page 34. For details, see Teradata Parallel Transporter Reference.

Filter Operators

Filter operators can both consume data from an input data stream and produce data for an output data stream. Filter operators prevent the output of any data row that contains column values that fail to satisfy filter conditions.

Although Teradata PT does not include any specific filter operators, the following filter operations can be accomplished using Teradata PT:

- Teradata PT job scripts can invoke user-written filter operators that are coded in the C or C++ programming languages. For more information about creating customized operators, see Teradata Parallel Transporter Operator Programmer Guide.
- Teradata PT includes the MultiLoad INMOD Adapter filter-type operator.
- Teradata PT supports several filtering capabilities, specifically the WHERE clause and CASE DML expressions in APPLY statements. These can handle most filtering operations.

Functioning between producer and consumer operators, filter operators can perform the following functions:

- Validating data
- Cleansing data
- Condensing data
- Updating data
Filter operators are summarized in Table 2 on page 34. For details, see Teradata Parallel Transporter Reference.

Standalone Operators

Standalone operators perform specialty processes that do not involve sending data to, or receiving data from, a data stream. In other words, standalone operators solely use input data from job scripts as their source.

Standalone operators can perform the following functions:
- Execute DDL and other self-contained SQL statements
- Execute host operating system commands
- Execute a DELETE task on the Teradata Database

Teradata PT includes the following standalone-type operators:
- OS Command operator
- DDL operator
- The Update operator, when it is executing the Delete Task and if no data is required.

Standalone operators are summarized in Table 2 on page 34. For details, see Teradata Parallel Transporter Reference.

Custom Operators

In addition to the four functional operator types, Teradata PT provides the means to develop custom operators using the Teradata PT API.

Custom operators must:
- Be written in the “C” or “C++” programming languages. (C is the preferred language for coding customer operators.)
- Comply with the requirements of the Teradata PT operator interface.

For more information, see Teradata Parallel Transporter Operator Programmer Guide.

INMOD and OUTMOD Adapter Operators

INMOD Adapter Operators

Input modification (INMOD) adapter operators are user-written INMOD routines that can preprocess data before it is sent to the Load or Update operator and then to the Teradata Database.

An INMOD routine, which can be invoked by the INMOD adapter operator, cannot send data directly to the consumer operators. The INMOD routine and the INMOD adapter operator can together act as a produce operator to pass data to the Load or Update operators.

Figure 4 shows a sample job flow using the FastLoad INMOD Adapter Operator.
Figure 4: Job Flow Using a FastLoad INMOD Adapter Operator

Figure 5 shows a sample job flow using the INMOD Adapter Operator.

Figure 5: Job Flow Using an INMOD Adapter Operator

For detailed information, see “FastLoad INMOD Adapter Operator” or “MultiLoad INMOD Adapter Operator” in the Teradata Parallel Transporter Reference.

OUTMOD Adaptor Operators

Output modification (OUTMOD) adaptor operators are user-written routines that process extracted data prior to delivering the data to its final destination.

An OUTMOD routine cannot be directly invoked by the Teradata PT Export operator. Rather, OUTMOD routines are invoked by the Teradata PT FastExport OUTMOD adapter operator, which acts as a consumer operator to read data from the Export operator. Figure 6 shows a sample flow.

Figure 6: Job Flow Using an OUTMOD Adapter Operator

For more information, see “FastExport OUTMOD Adapter Operator” in the Teradata Parallel Transporter Reference.

Operator Summary

Table 2 summarizes the function, type, and purpose of the Teradata PT operators.

For detailed information about operators, see Teradata Parallel Transporter Reference.

Table 2: Operator Summary

<table>
<thead>
<tr>
<th>Teradata PT Operator Needed</th>
<th>TYPE Definition</th>
<th>Action</th>
<th>Standalone Equivalent</th>
</tr>
</thead>
</table>
| DataConnector operator as a consumer | TYPE DATACONNECTOR CONSUMER | • Writes to flat files  
• Interfaces with access modules | Data Connector |
Access Modules

Access modules are software modules that encapsulate the details of access to various data stores, for example, CD-R, CD-RW, tape (via DataConnector or FastLoad OUTMOD Adapter operators), subsystems (such as Teradata Database servers, IBM’s WebSphere MQ).
Access modules provide Teradata PT with transparent, uniform access to various data sources. Access modules isolate Teradata PT from the following:

- Device dependencies, for example, disk versus tape (only on mainframe) (embedded into the Teradata standalone utilities today)
- Data source/target location, for example, local versus remote
- Data store specifics, for example, sequential file versus indexed file versus relational table

Access modules can be used with the DataConnector operator to read from different types of external data storage devices.

The following access modules are supported. These access modules support only reading (importing) of data, not writing:

- **Named Pipes Access Module** for Teradata PT allows you to use Teradata PT to load data into the Teradata Database from a UNIX named pipe. A pipe is a type of data buffer that certain operating systems allow applications to use for the storage of data.
- **WebSphere MQ Access Module** for Teradata PT allows you to use Teradata PT to load data from a message queue using IBM’s WebSphere MQ (formerly known as MQ Series) message queuing middleware.
- **JMS Access Module** for Teradata PT allows you to use Teradata PT to load data from a JMS-enabled messaging system using JMS message queuing middleware.
- **Custom Access Modules.** You can also create custom access modules to use with the DataConnector operator for access to specific systems.

For more information about creating and using custom access modules, see *Teradata Tools and Utilities Access Module Programmer Guide*.

### Data Streams

In Teradata PT, data streams (or buffers in memory that temporarily hold data) enable the passing of data between operators without intermediate storage. Data streams allow Teradata PT to automate parallelism and scalability.

A Teradata PT job moves data from the sources to the targets through data streams. Data moves from producer operator(s) through the data streams to consumer operator(s):

- Producer operators take data from the source, moving the data into data streams.
- At that point, filter operators can access the data, perform updates, and return updated data to the data streams for further processing by consumer operator(s).
- In the absence of a filter operator, data passes from the producer operator(s), straight through the data streams, to the consumer operator(s).

In all cases, the data flows through the data streams as shown in Figure 7.
Verifying the Teradata PT Version

To verify the version of Teradata PT you are running, issue a `tbuild` command (on the command line) with no options specified, as follows:

```
tbuild
```

Switching Versions

Multiple versions of Teradata Warehouse Builder (Teradata WB) and Teradata PT can be installed.

To switch between them, or between multiple versions of Teradata PT, refer to the instructions in these publications:


Related Teradata PT Publications

User documentation for Teradata PT is distributed among the following documents:

<table>
<thead>
<tr>
<th>Publication</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Teradata Parallel Transporter User Guide</em> (This book) B035-2445-mmyA</td>
<td>An overview of Teradata PT capabilities. Example jobs showing how those capabilities can be used to accomplish common ETL tasks. Strategies for planning, scripting, optimizing, launching, and debugging Teradata PT jobs.</td>
</tr>
</tbody>
</table>
## Related Teradata PT Publications

<table>
<thead>
<tr>
<th>Publication</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Teradata Parallel Transporter Reference</em>&lt;br&gt;B035-2436-mmyA</td>
<td>Reference material including detailed descriptions and required syntax for all Teradata PT scripting elements.</td>
</tr>
<tr>
<td><em>Teradata Parallel Transporter Operator Programmer Guide</em>&lt;br&gt;B035-2435-mmyA</td>
<td>Provides information on developing custom operators, including all interface functions that allow communication between operators and the Teradata PT infrastructure.</td>
</tr>
</tbody>
</table>
CHAPTER 2

Teradata PT Job Components

This chapter provides an overview of the components available for use in a Teradata PT job, a brief description of their function, and how these components work together in a job script.

Topics include:
- Fast Track Job Scripting
- Understanding Job Script Concepts
- Creating a Job Script

Fast Track Job Scripting

Although this book contains all the information required to create job scripts, it is not necessary to create the entire script. Teradata PT provides a comprehensive collection of job script examples that require only simple modifications to accomplish most common ETL tasks.

<table>
<thead>
<tr>
<th>Scripting Aid</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example jobs and scripts</td>
<td>This User’s Guide contains the following scripting aids that you can use as the basis for your own job scripts. A comprehensive collection of commonly used Teradata PT jobs is shown in:</td>
</tr>
</tbody>
</table>
|                        | • Chapter 5: “Moving External Data into Teradata Database.”  
|                        | • Chapter 6: “Moving Data from Teradata Database to an External Target.”  
|                        | • Chapter 7: “Moving Data within the Teradata Database Environment.”  
|                        | Script examples for the common jobs shown in Chapters 5 through 7 are available in Appendix A: “Job Script Examples.” |
| Sample scripts file    | The Teradata PT software also includes a directory of sample scripts that provide an additional resource beyond those contained in Appendix A.  
|                        | For details on accessing the sample script library and editing the scripts for your unique needs, see “Accessing and Editing Sample Scripts” on page 40. |
| Teradata PT Wizard     | The Teradata PT Wizard, included with the Teradata PT software, offers a simple GUI interface that prompts the user through each stage of job script creation. Although its main function is as a teaching tool, you can also use it to create rudimentary single-step job scripts, which can be copied and pasted into more a complex script.  
|                        | For information, see Appendix B: “Teradata PT Wizard.” |
Accessing and Editing Sample Scripts

Teradata recommends that, where possible, you use Teradata PT sample scripts as the basis for all job scripts you create. This approach is easier, more efficient, and will save a lot of time. The following sections list script locations and methods for editing them, according to Teradata client operating system.

To edit sample scripts for UNIX clients:

1. Copy scripts from Appendix A: “Job Script Examples.” or from the Sample directory that is installed in the same directory as Teradata PT. The default directory is:
   `/opt/teradata/client/<version>/tbuild/sample/etl`
2. Use UNIX vi or another Unix editor to edit the sample script.
   
   **Note:** Remember that UNIX is case-sensitive.
3. Save the file.

To edit sample scripts for Windows clients:

1. Copy scripts from Appendix A: “Job Script Examples.” or from the Sample directory that is installed in the same directory as Teradata PT. The default directory is:
   `C:\Program Files\Teradata\Client\<version>\Teradata Parallel Transporter\sample\etl`
2. Use Windows Notepad or another Windows-compatible editor to edit a script.
3. Save the file.

To edit sample scripts for z/OS clients:

1. On a z/OS system, log on to the Time Sharing Option (TSO).
2. At the **Master Application** menu, type *P*, then press **Enter** to execute the Program Development Facility (PDF, also known as ISPF).
3. At the **ISPF Primary Option** menu, type **2** (Edit), then press **Enter** to execute the TSO Editor.
4. In the **Project**, **Group**, and **Type** boxes, type the name of your Teradata PT JCL library.
5. In the **Member** box, type the name of the member file containing the script you want to execute, then press **Enter**.
6. Edit the file.
7. Save the file.
Understanding Job Script Concepts

It is important to understand the following basic concepts about job script components and structure before attempting to create or edit a Teradata PT job script.

Script Sections

Every Teradata PT job script has the following sections:

- An optional job header, consisting of C-style comments that can be used to record such useful information as who created the script, when it was created, and what it does and how it works.
- The declarative section of the script uses DEFINE statements to define the Teradata PT objects needed for the job. Objects identify the schemas of data sources and targets and the operators that extract, filter and load data.
- The executable section of the script specifies the processing statements that initiate the actions that read/extract, filter, insert, update, and delete data, by APPLYing tasks to the specific objects that will execute them. APPLY statements specify the operations to be performed, the operators to be used, the source and destination of data, filtering options, and the optional degree of parallelism for each operator used. APPLY statements can employ SELECT statements, WHERE clauses, and CASE DML or CASE value expressions.

Figure 8: Script Sections

Statement Types

A Teradata PT script consists of the following types of statements:

Object Definition Statements

In the declarative section of the script, definition statements define all of the Teradata PT objects referenced in the script. For detailed information on required syntax for each types of DEFINE statement, see Teradata Parallel Transporter Reference.

Definition statements include:
Chapter 2: Teradata PT Job Components
Understanding Job Script Concepts

- **DEFINE JOB** (required) - Names a Teradata PT job, but is not necessarily the same as the file name of the script. Also optionally identifies the character set being used. Contains the definitions of all job objects, as well as one or more processing statements.

- **DEFINE SCHEMA** (required) - Defines the data structure for the data an operator will process. Each unique data structure addressed by the job script requires a separate DEFINE SCHEMA object.

- **DEFINE OPERATOR** (required) - Defines an operator and specifies the operator attributes to which values can be assigned.

**Processing Statements**

In the executable section of the script, APPLY statements specify all operations to be performed by the job and the objects that will perform them. For detailed information on APPLY, see Teradata Parallel Transporter Reference.

Processing statement specifications include the following:

- APPLY...TO to specify:
  - the operators that will be used to load or update the data.
  - the number of instances to be used for the operators
  - operator attribute values (optional)

- SELECT...FROM to specify:
  - the operators that will be used to acquire, and if necessary, filter the data
  - the number of instances to be used for the operator
  - the selected columns to be sent to the consumer operator
  - operator attribute values (optional)

- WHERE clauses, CASE DML or CASE value expressions, and SELECT derived column values to filter data between source and destination. See “Data Filtering and Conditioning Options” on page 378.

Set the degree of processing parallelism to be used for each operator. See “Optimizing Job Performance with Sessions and Instances” on page 80.

**Scripting Language**

Teradata PT uses an SQL-like scripting language to define extract, updating, and load functions in a job script. This easy-to-use language is based on SQL, making it familiar to most database users. All Teradata PT operators use the same language.

The language is declarative and tells Teradata PT exactly what operations to perform. A single job script can define multiple operators, the schema, data updates, and pertinent metadata to create complex extract and load jobs.

For an example, see “Script Example 1A: High Speed Bulk Loading from Flat Files into an Empty Teradata Database Table” on page 404.
Syntax Rules

A few simple syntax rules are important to note when creating Teradata PT job scripts:

- **Case Sensitivity**
  - Attribute names are case-insensitive.
  - Most attribute values are case-insensitive. However, attribute values, such as file names and directory names, may be case-sensitive depending on the platform.
  - Non-attribute object parameters, such as the syntax elements in a DEFINE JOB statement, are case-sensitive.

- **Defining Objects** - Every Teradata PT object must be defined before it can be referenced anywhere in the Teradata PT job script.

- **Keyword Restrictions** - Do not use Teradata PT reserved keywords, such as OPERATOR, SOURCE, DESCRIPTION in your job scripts as identifiers for column names, attributes, or other values. A complete list of these reserved keywords is provided in *Teradata Parallel Transporter Reference*.

- **Use of VARCHAR and INTEGER** - Use of the keywords VARCHAR and INTEGER to declare the attributes of an operator, as follows:
  - VARCHAR and INTEGER are required in a job script:
    - In a DEFINE SCHEMA statement, which may also require other keywords for datatype specification.
    - In a DEFINE OPERATOR statement when an attribute is declared but no attribute value is specified.
  - VARCHAR and INTEGER are *not* required in a job script:
    - In a DEFINE OPERATOR statement, if the attribute declaration includes a value.
  
  **Note:** VARCHAR and INTEGER keywords are unnecessary when assigning a value to an attribute in an APPLY statement because the data type of the attribute is specified when the operator is defined.

- **Quotation Marks** - Use the following rules when using quotes:
  - Character string literals must be enclosed in single quotes.
  - Values for VARCHAR attributes must be enclosed in single quotes, and embedded quotes must be escaped with two consecutive single quotes.
  - Values for INTEGER attributes require no quotes.

- **SQL Notation** - SQL statements that span multiple lines must have a space or tab character between the last character of a line and the first character in the next line. If not, the two lines are processed as if there is no line break, which inadvertently joins the two character strings, resulting in either an error or the processing of an unintended SQL statement.

  For example, the following code would produce an error if no space or tab was added between “FINAL” and “INSTANCE” because the Teradata Database would see the invalid keyword FINALINSTANCE:

  ```
  ('CREATE TYPE INSV_INTEGER AS INTEGER FINAL INSTANCE METHOD IntegerToFloat()
  RETURNS FLOAT
  LANGUAGE C
  ```
Creating a Job Script

Creating a job script requires that you define the job components in the declarative section of the job script, and then apply them in the executable section of the script to accomplish the desired extract, load, or update tasks. The object definition statements in the declarative section of the script can be in any order as long as they appear prior to being referenced by another object.

The following sections describe how to define the components of a Teradata PT job script.

- Defining the Job Header and Job Name
- Defining a Schema
- Defining Operators
- Coding the Executable Section
- Defining Job Steps

For required syntax and available options, see Teradata Parallel Transporter Reference.

Defining the Job Header and Job Name

A Teradata PT script starts with an optional header that contains general information about the job, and the required DEFINE JOB statement that names and describes the job, as shown in Figure 9.
Consider the following when creating the job header and assigning the job name.

- The *Script Name* shown in the job header is optional, and is there for quick reference. It can be the same as the *jobname* or it can be the filename for the script.

- The *jobname* shown in the DEFINE JOB statement is required. It is best to use a descriptive name, in the case of the example script, something like “Two Source Bulk Update.”

Note that the *jobname* shown in the DEFINE JOB statement is not necessarily the same as the “jobname” specified in the *tbuild* statement when launching the job, although it can be. The *tbuild* statement might specify something like “Two Source Bulk Updateddmmmyy,” to differentiate a specific run of the job.

For detailed information on *tbuild* and how to specify the job and job script, see “Setting *tbuild* Options” on page 261.
Using Job Variables

Most Teradata PT job script values can be coded as job variables, which can be used anywhere in the script except within quoted strings and in comments. Once variables have been defined, they can be reused in any job where the value of the variable is valid. A common use of variables is for the values of operator attributes.

If the attribute is:

- A character data type, the job variable value must be a quoted string.
- An integer data type, the job variable value must be an integer.
- An array attribute, the variable value must be an array of values of the attribute data type.

**Note:** Job variables cannot be used between quoted strings unless they are concatenated. Job variables can represent entire quoted strings. For example, to insert the literal term “@item” into a column, use the string: 'Insert this @item into a column'. However, to use @item as a job variable, use the string: 'Insert this'|| @item || ' into a column'

Using job variables for job script parameters requires completion of two setup activities:

- Reference the variables in the job script.
- Assign values to the variables in the one of the following places, shown in processing order, from highest to lowest priority.
  - on the command line (highest priority)
  - in a local job variables file (next highest)
  - in the global job variables file (UNIX and Windows platforms) (next)
  - in the job script itself (lowest)

Setting Up Job Variables

Job variables can be set up in the following locations

- **Global job variables file** - The lowest priority for supplying values for job variables is storing them inside the global job variables file. The global job variables file is read by every Teradata PT job. Place common, system-wide job variables in this file, then specify the path of the global job variables file in the Teradata PT configuration file by using the GlobalAttributeFile parameter.
  
  **Note:** A global job variables file is available on UNIX and Windows.

- **Local job variables file** - The second highest priority for defining values for job variables is storing them inside a local job variables file. You can specify a local job variables file, which contains the values for job variables, using the -v option on the command line as follows:
  
  `tbuild -f weekly_update.tbr -v local.jobvars`
  
  **Note:** On z/OS, specify a local job variables file through the DDNAME of ATTRFILE.

For information, see “Setting Up the Job Variables Files” on page 68.
Referencing Job Variables in a Job Script

To specify a job variable in a job script, reference the variable where the value would normally appear. The job variable reference is composed of the @ symbol, followed by a unique identifier for the variable. You can use the attribute name or any other identifier to construct the variable.

Example: Specifying Variables for Attributes

```
DEFINE JOB CREATE_SOURCE_EMP_TABLE

  DEFINE OPERATOR DDL_OPERATOR
  DESCRIPTION 'Teradata Parallel Transporter DDL Operator'
  TYPE DDL
  ATTRIBUTES
    {
      VARCHAR UserName = @MyUserName,
      VARCHAR UserPassword = @MyPassword
    }

  APPLY
    ('DROP TABLE SOURCE_EMP_TABLE;'),
    ('CREATE TABLE SOURCE_EMP_TABLE(EMP_ID INTEGER, EMP_NAME CHAR(10));'),
    ('INSERT INTO SOURCE_EMP_TABLE(1,''JOHN'');'),
    ('INSERT INTO SOURCE_EMP_TABLE(2,''PETER'');')
  TO OPERATOR (DDL_OPERATOR());
```

In this example, the DDL operator issues two DDL and two DML statements that create a table and then populates that table with two rows. The values of the UserName and UserPassword operator attributes are coded as job variables.

Example: Specifying Non-Attribute Job Variables

Job variables can be used for object names and other parameters. In the following example, the values for @ConsumerOperator and @ProducerOperator can be assigned in the global job variables file, in a local job variables file, in a `tbuild` command, or in a job script using the `SET` directive.

```
APPLY
  'INSERT INTO TABLE xyz (:col1, :col2);'
TO OPERATOR ( @ConsumerOperator [1] )
SELECT * FROM OPERATOR ( @ProducerOperator[2] );
```

Job variable values can be stored in the locations shown in the following list, with the lowest priority locations listed first. Note that if values for a particular variable are stored in more than one of the listed locations, the value highest priority sources is used.
Assigning Job Variables on the Command Line

You can specify variables on the command line, as follows:

```
tbuild -f weekly_update.tbr -u "UsrID = 'user', Pwd = 'pass' "
```

For further information on specifying job variables on the command line, see “Assigning Job Variables on the Command Line” on page 263.

Defining a Schema

Teradata PT requires that the job script describe the structure of the data to be processed, that is the columns in table rows or fields in file records. This description is called the schema. Schemas are created using the DEFINE SCHEMA statement.

The value following the keyword SCHEMA in a DEFINE OPERATOR statement identifies the schema that the operator will use to process job data. Schemas specified in operator definitions must have been previously defined in the job script. To determine how many schemas you must define, observe the following guidelines on how and why schemas are referenced in operator definitions (except standalone operators):

- The schema referenced in a producer operator definition describes the structure of the source data.
- The schema referenced in a consumer operator definition describes the structure of the data that will be loaded into the target. The consumer operator schema can be coded as SCHEMA * (a deferred schema), which means that it will accept the scheme of the output data from the producer.
- You can use the same schema for multiple operators.
- You cannot use multiple schemas within a single operator, except in filter operators, which use two schemas (input and output).
- The column names in a schema definition in a Teradata PT script do not have to match the actual column names of the target table, but their data types must match exactly. Note, that when a Teradata PT job is processing character data in the UTF-16 character set, all CHAR(m) and VARCHAR(n) schema columns will have byte count values m and n, respectively, that are twice the character count values in the corresponding column definitions of the DBS table. Because of this, m and n must be even numbers.

**Note:** When using the UTF16 character set in a job script, the value of n in VARCHAR(n) and CHAR(n) in the SCHEMA definition must be an even and positive number.
The following is an example of a schema definition:

**Figure 10: Example Schema Definition**

```sql
DEFINE SCHEMA ATTENDEES
DESCRIPTION 'Employees who attended the training session'
(
    ATTENDEE_NAME     CHAR(24),
    TRAINING_FEEBACK  VARCHAR(256)
);

DEFINE SCHEMA ABSENTEES
DESCRIPTION 'Employees who failed to attend the training session'
(
    ABSENTEE_NAME  CHAR(24),
    EXCUSE         VARCHAR(256)
);

DEFINE SCHEMA PRESENTERS
DESCRIPTION 'Employees who gave presentations at the training session'
(
    PRESENTER_NAME      CHAR(24),
    PRESENTATION_TOPIC  VARCHAR(128)
);
```

**Using Multiple Source Schemas**

A single script often requires two schemas, one each for the source and target. It is also possible to use multiple schemas for the source data if all rows are UNION-compatible. Two schemas are UNION-compatible if their corresponding columns have exactly the same data type attributes (type, length, precision and scale); that is, other than their column names, the schemas are identical. If the schemas are UNION-compatible Teradata PT combines data from the sources, each being extracted by a different producer operator using a different schema, into a single output data stream using its UNION ALL feature. For information, see “UNION ALL: Combining Data from Multiple Sources” on page 373.

**Example: Multiple Schemas in a Job Script**

```sql
DEFINE SCHEMA ATTENDEES
DESCRIPTION 'Employees who attended the training session'
(
    ATTENDEE_NAME     CHAR(24),
    TRAINING_FEEBACK  VARCHAR(256)
);

DEFINE SCHEMA ABSENTEES
DESCRIPTION 'Employees who failed to attend the training session'
(
    ABSENTEE_NAME  CHAR(24),
    EXCUSE         VARCHAR(256)
);

DEFINE SCHEMA PRESENTERS
DESCRIPTION 'Employees who gave presentations at the training session'
(
    PRESENTER_NAME      CHAR(24),
    PRESENTATION_TOPIC  VARCHAR(128)
);
```

**Explanation of Multiple Schema Example**

Consider the following when referring to the proceeding multiple schema example:

- Each schema must have a unique name within the job script.
- Schemas ATTENDEES and ABSENTEES are UNION-compatible. Schema PRESENTERS is not UNION-compatible, because VARCHAR(128) is not identical to VARCHAR(256).
Choosing operators for use in a job script is based on the type of data source, the characteristics of the target tables, and the specific operations to be performed.

Teradata PT scripts can contain one or more of the following operator types.

- **Producer operators** “produce” data streams after reading data from data sources.
- **Consumer operators** “consume” data from data streams and write it to target tables or files.
- **Filter operators** read data from input data streams, perform operations on the data or filter it, and write it to output data streams. Filter operators are optional.
- **Standalone operators** issue Teradata SQL statements or host operating system commands to set up or clean up jobs; they do not read from, or write to, the data stream.

**Note:** The following locations contain additional information about Teradata PT operators:

- For details about operator attributes and syntax, see *Teradata Parallel Transporter Reference*.
- For information about operator capabilities, see the chapters on using operators beginning with “Chapter 8 DataConnector Operator” on page 151.
- For examples of using operators to accomplish specific tasks, see “Chapter 5 Moving External Data into Teradata Database,” “Chapter 6 Moving Data from Teradata Database to an External Target,” and “Chapter 7 Moving Data within the Teradata Database Environment.”

### Operator Definition in a Teradata PT Job Script

Teradata PT operators must be defined in the declarative section of a job script, using a DEFINE OPERATOR statement.

Use the following procedure when defining an operator in a Teradata PT job script.

1. For DEFINE OPERATOR statement syntax, see *Teradata Parallel Transporter Reference*.
2. Specify the required syntax elements:
   - The **operator name** (a maximum of 255 characters with no spaces) is the name by which the job steps reference the operator.
   - The **operator TYPE**, for example LOAD or UPDATE.
   - The **schema name**, which can be either:
     - The name of a predefined schema object
     - A deferred schema specification, using SCHEMA *
     - An explicit schema definition that includes all of the column definitions
       - **Note:** Standalone operators do not extract/load data, so they do not specify a schema.
   - The declaration of all the attributes required by the operator with the associated values. All mandatory attributes must have values associated with them in the DEFINE OPERATOR statement.

For attribute definitions and syntax, see Teradata Parallel Transporter Reference.
• The declaration of any optional attributes that are meaningful to the operator with the optional assignment of values.

**Note:** An optional *operator description*, which can be defined in a Teradata PT job script after the *operator name* is defined, provides a short description of the operator to differentiate it from other operators of the same type. For instance, you may define several Load operators that are each configured to do a unique type of load job.

**Defining Producer Operators**

Producer operators “produce” a data stream after reading data from a Teradata Database or an external data store. Then they write the data into the data stream for further processing by consumer or filter operators.

Figure 11: Defining producer operators

A Teradata PT job script allows as many producer operators as data sources, as long as the output schema is the same; or you can use a single producer. The producer extracts data from the source and places it into the data stream, where other operators can use it.

Following is a list of Teradata PT producer operators:

**Table 3: Producer Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Produces Data from Teradata Database</strong></td>
<td></td>
</tr>
<tr>
<td>Export</td>
<td>The Export operator extracts data from Teradata tables and writes it to the data stream. The Export operator functions in a way similar to the standalone FastExport utility protocol.</td>
</tr>
<tr>
<td>SQL Selector</td>
<td>Selects data from Teradata tables using SQL sessions. The only producer operator that can handle LOB data.</td>
</tr>
<tr>
<td><strong>Produces Data from a Non-Teradata Data Source</strong></td>
<td></td>
</tr>
<tr>
<td>DataConnector (producer)</td>
<td>The DataConnector operator accesses files either directly or through an access module, and then writes it to the data stream.</td>
</tr>
</tbody>
</table>
Script Requirements

When you define a producer operator in a Teradata PT script, required specifications include:

- In the operator definition
  - A name for the operator (maximum of 255 characters, with no spaces).
  - The operator type.
  - The name of the input schema. A deferred schema, specified as SCHEMA *, is not supported for producer operators.
  - Declarations for all required attributes.
- In the APPLY statement
  - A SELECT . . . FROM clause that names the producer operator
Example: Producer Operator Definition

Following is a simplified example of an Export operator definition in a job script:

Figure 12: Export Operator Definition

```
DEFINE OPERATOR Export_operator_name
DESCRIPTION 'comments for the Export operator'
TYPE EXPORT
SCHEMA schema_name
ATTRIBUTES
  (VARCHAR PrivateLogName = 'filename2 log,
   INTEGER Blocksize = 64260,
   INTEGER TenacityHours = 1,
   INTEGER TenacitySleep = 1,
   INTEGER MaxSessions = 5,
   INTEGER MinSessions = 1,
   VARCHAR TdpId = 'TDP_ID'
   VARCHAR DateForm = 'ANSIDATE',
   VARCHAR UserName = 'Teradata_User',
   VARCHAR UserPassword = 'password',
   VARCHAR AccountID,
   VARCHAR SelectStmt = 'SELECT statement here;')
```

User-defined name for the Export operator for this job.
The schema, defined earlier in the script, is referenced here by the name you gave it.
Open parenthesis starts the attribute definitions, defined in any order, and separated by commas.
The maximum number of sessions for this operator, distributed among the instances.
The TDP ID, username, and password of the source table must be identified.
The SQL SELECT statement is passed to the Teradata Database to retrieve the rows and put them in the data streams.
Close parenthesis and semicolon to end the Export operator attributes.

Defining Consumer Operators

A consumer operator “consumes” data from the data stream in order to write it to the Teradata Database, or an external data target, such as a flat file.

Figure 13: Defining consumer operators

A script can have as many consumer operators as there are occurrences of the keyword APPLY. For more information, see “APPLY” in Teradata Parallel Transporter Reference.
Following is a list of Teradata PT consumer operators:

Table 4: Consumer Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operators that Write Data to a Teradata Database</strong></td>
<td></td>
</tr>
<tr>
<td>Load</td>
<td>The Load operator writes data into an empty Teradata table. It is based on</td>
</tr>
<tr>
<td></td>
<td>the standalone FastLoad utility protocol.</td>
</tr>
<tr>
<td>Update</td>
<td>The Update operator can perform INSERT, UPDATE, and DELETE operations on one</td>
</tr>
<tr>
<td></td>
<td>to five Teradata tables. It is based on the standalone MultiLoad utility</td>
</tr>
<tr>
<td></td>
<td>protocol.</td>
</tr>
<tr>
<td>Stream</td>
<td>The Stream operator continuously loads data into Teradata tables. It is</td>
</tr>
<tr>
<td></td>
<td>based on the standalone TPump utility protocol.</td>
</tr>
<tr>
<td>SQL Inserter</td>
<td>Inserts data into Teradata tables with SQL sessions.</td>
</tr>
<tr>
<td><strong>Operators that Write Data to an External Target</strong></td>
<td></td>
</tr>
<tr>
<td>DataConnector (consumer)</td>
<td>Writes data directly to an external flat file. The DataConnector operator</td>
</tr>
<tr>
<td></td>
<td>can also write data through an access module, which can provide an interface</td>
</tr>
<tr>
<td></td>
<td>with different types of external data storage devices.</td>
</tr>
<tr>
<td><strong>Operators that Process and Write Data to an External Target</strong></td>
<td></td>
</tr>
<tr>
<td>FastExport OUTMOD Adapter</td>
<td>Enables a standalone FastExport utility OUTMOD routine to be used to</td>
</tr>
<tr>
<td></td>
<td>post-process rows exported from Teradata tables, before writing them to</td>
</tr>
<tr>
<td></td>
<td>external flat files.</td>
</tr>
</tbody>
</table>

**Script Requirements**

When you define a consumer operator in a Teradata PT script, required specifications include:

- In the operator definition
  - A name for the operator (maximum of 255 characters, with no spaces).
  - The name of the output schema, if different than the input. Use SCHEMA * if the input and output schemas are the same.
  - Declarations for all required attributes.
- In the APPLY statement
  - An APPLY TO clause that names the consumer operator

Teradata PT limits the number of tables consumers can load simultaneously, as follows:

<table>
<thead>
<tr>
<th>Teradata PT Operator</th>
<th>Maximum Target Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>1</td>
</tr>
<tr>
<td>Update</td>
<td>5</td>
</tr>
<tr>
<td>Stream</td>
<td>127</td>
</tr>
</tbody>
</table>
Chapter 2: Teradata PT Job Components

Defining Operators

Example: Consumer Operator Definition

Figure 14: Load Operator

Table 5: Standalone Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDL</td>
<td>Executes SQL statements before or after the main extract and load job steps, for job setup or cleanup. For example, you can create tables and create indexes before starting a job, or drop work tables, as needed, after a job.</td>
</tr>
</tbody>
</table>
Defining Operators

Example:

Following is a simplified example of defining the DDL operator:

Figure 15: Example script for defining the DDL operator

```
DEFINE OPERATOR DDL_OPERATOR
DESCRIPTION 'TERADATA PT DDL OPERATOR'
TYPE DDL
ATTRIBUTES
{
    VARCHAR ARRAY TraceLevel,
    VARCHAR TdpId = 'my_database',
    VARCHAR UserName = 'my_user',
    VARCHAR UserPassword = 'my_password',
    VARCHAR AccountID,
    VARCHAR PrivateLogName = 'ddllog'
};
```

Specification of Operator Attributes

The specification of operator attributes in a DEFINE OPERATOR statement identifies the attributes that require specified values or that must use other than default attribute values.

Attribute specification requires two values; declaring attributes and assigning attribute values.

Declaring Attributes

The following rules describe how to declare attributes in a DEFINE OPERATOR statement:

- Attributes must be declared in the operator definition when:
  - they are required by the operator.
  - you want to assign a value that is different than the default attribute value.
  - you want the option of assigning an overriding value (for either the default or assigned value) in the APPLY statement.

- Declaring an attribute requires that you list the attribute name under ATTRIBUTES in the operator definition as follows:
  
  ATTRIBUTES
  {
    VARCHAR TraceLevel,
    VARCHAR TenacityHours=0,
    VARCHAR PrivateLogName='export.log',
    VARCHAR SelectStmt,
    ...}
Defining Operators

Note: The use of VARCHAR and INTEGER is optional when the attribute declaration includes a value assignment, such as for TenacityHours and PrivateLogName, above.

- All required attributes must be declared. Note that most attributes are not required.
- Optional attributes automatically assume their default attribute values. If the default value for an optional attribute is adequate for the purpose of a job, it need not be declared.

Note: Not all attributes have default values.

For information on operator attributes and default values, see the chapters on individual operators in the Teradata Parallel Transporter Reference.

Assigning Attribute Values

The following rules describe how to assign attribute values to attributes declared in a DEFINE OPERATOR statement:

- Assign values to attributes in the DEFINE OPERATOR statement if:
  - There is no default value, such as for the UserName and UserPassword attributes.
  - The job cannot use the default value and you do not want to assign a value in the APPLY statement that references the operator.

- Do not assign values for declared attributes if:
  - The operator uses the default attribute value; for example, the default On (enabled) for the Stream operator ArraySupport attribute.
  - The APPLY statement that references the operator assigns an attribute value.

Note: If an attribute value is assigned in the operator definition and is also assigned in the APPLY statement, the APPLY value overrides the value in the operator definition. The override value applies only to the occurrence of the operator where the override value is assigned. All other occurrences in the script are unaffected. For further information, see “Attribute Value Processing Order” on page 59.

- The value assigned to an attribute (anywhere in the script) is a job variable, using the form attributeName = @<jobVariableName>, then the variable will be replaced by the value from the highest priority job variable source.

  For further information on setting job variables and the processing order of job variable sources, see “Specifying Job Variables for Attribute Values” on page 59.

- The value assigned to an attribute must match the data type of the attribute.

Multivalued (Array Type) Attributes

Teradata PT allows specification of multiple values for a few operator attributes. Array attribute values can be specified as part of:

- A DEFINE OPERATOR statement
- A reference to an operator in an APPLY statement

Available array attributes are shown in the following table:
The following examples show how specification of an array value for an attribute would appear in a DEFINE OPERATOR statement or an APPLY statement:

```
VARCHAR ARRAY TraceLevel = [ 'CLI', 'OPER' ]
VARCHAR TraceLevel = [ 'CLI', 'OPER' ]
TraceLevel = [ 'CLI', 'OPER' ]
```

The syntax for using one or more array attributes in a DEFINE statement is shown in “Specification of Operator Attributes” on page 56.

Observe the following additional guidelines for use of array attributes.

- The Teradata PT compiler ensures that array attributes are assigned array-type (multiple) values and vice versa; multiple values are assigned only to array attributes.

- Array values can be assigned in a series as shown in the following examples:
  ```
  VARCHAR ARRAY TargetTable = [ 'table1', 'table2', ..., 'tableN']
  VARCHAR TargetTable = [ 'table1', 'table2', ..., 'tableN']
  TargetTable = [ 'table1', 'table2', ..., 'tableN']
  ```

  Using the ARRAY keyword in assigning an array value is optional.

- An array value containing a single member (for example, [ 'x' ] or [ 2 ]) is still considered a valid array value. In this case, the array dimension is 1. However, even this single value must be specified through array notation, that is to say, enclosed in [ ].

- To omit some of the values for the array attribute, for example, to assign a first and a third value, but not the second, you can do the following: specify a value of NULL. Following is an example of assigning certain attribute values while omitting others.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorList</td>
<td>DDL</td>
</tr>
<tr>
<td>• TargetTable</td>
<td>Load</td>
</tr>
<tr>
<td>• ErrorTable1</td>
<td></td>
</tr>
<tr>
<td>• ErrorTable2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TraceLevel</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• DDL</td>
<td></td>
</tr>
<tr>
<td>• Export</td>
<td></td>
</tr>
<tr>
<td>• Load</td>
<td></td>
</tr>
<tr>
<td>• ODBC</td>
<td></td>
</tr>
<tr>
<td>• OS Command</td>
<td></td>
</tr>
<tr>
<td>• SQL Inserter</td>
<td></td>
</tr>
<tr>
<td>• SQL Selector</td>
<td></td>
</tr>
<tr>
<td>• Stream</td>
<td></td>
</tr>
<tr>
<td>• Update</td>
<td></td>
</tr>
</tbody>
</table>

The following examples show how specification of an array value for an attribute would appear in a DEFINE OPERATOR statement or an APPLY statement:
• specify a value of NULL, as follows:
  VARCHAR FILE_NAMES = ['/first', NULL, '/third']
• specify the omitted value with commas, as follows:
  VARCHAR FILE_NAMES = ['/first', , '/third']
• Following example shows an array attribute value assigned as part of a SELECT statement:

  SELECT * FROM OPERATOR (reader ATTR (FILE_NAMES = ['/first',
  NULL,'/ third'], MODE = 'read')

  Note: Use of VARCHAR and INTEGER is optional when specifying an array attribute value. For detailed information on using VARCHAR and INTEGER, see “Syntax Rules” on page 43.

For details about how to use array attributes for a particular operator, see the chapter on that operator in Teradata Parallel Transporter Reference.

Specifying Job Variables for Attribute Values

When you declare an operator attribute in an operator definition, you have the option of delay its value assignment until you run the job script that contains the operator definition. To do this, specify the attribute value as a job variable. All three attributes in the following operator attribute list are assigned values at run time via job variables:

ATTRIBUTES
{
  VARCHAR UserName = @UsrID,
  VARCHAR UserPassword = @Pwd,
  VARCHAR Tdpid = @Tdpid
};

The job variable reference is composed of the @ symbol, followed by a unique identifier for the variable. You can use the attribute name or any other identifier.

When a job script is submitted for execution, the first thing that happens is that the character-string value of each job variable replaces all occurrences of that job variable in the script text, just as if the value had been part of the original script text at those places. Only then is the script compiled.

  Note: You can also reference a job variable for an attribute value in the APPLY statement.

Attribute Value Processing Order

Object attribute values can be assigned at several locations within the job script.

The following list shows the locations where attribute values can be assigned, in the order they are processed, from first to last. The last value processed is used in the job.

1  DEFINE OPERATOR
2  As part of an APPLY TO...SELECT FROM statement
Coding the Executable Section

After defining the Teradata PT script objects required for a job, you must code the executable (processing) statement to specify which objects the script will use to execute the job tasks and the order in which the tasks will be executed. The APPLY statement may also include data transformations by including filter operators or through the use of derived columns in its SELECT FROM.

A job script must always contain at least one APPLY statement, and if the job contains multiple steps, each step must have an APPLY statement.

For more information about syntax and the use, see “APPLY” in Chapter 3 of the Teradata Parallel Transporter Reference.

Coding the APPLY Statement

An APPLY statement typically contains two parts, which must appear in the order shown:

1. A DML statement (such as INSERT, UPDATE, or DELETE) that is applied TO the consumer operator that will write the data to the target, as shown in Figure 16. The statement may also include a conditional CASE or WHERE clause.

Figure 16: Multiple Insert Statements

```
APPLY

"INSERT INTO Target_table_1
  (column_1, column_2, column_3, etc);

,'INSERT INTO Target_table_2
  (column_1, column_2, column_3, etc);

,CASE WHEN (column_name = 'Value_A' OR column_name = 'Value_B' OR column_name = 'Value_C')
    THEN
      'INSERT INTO Target_table_3
        (column_1, column_2, column_3, etc);
    END

TO OPERATOR (update_operator_name[instances])
```
For most jobs, the APPLY statement also includes the read activity, which uses a SELECT FROM statement to reference the producer operator. If the APPLY statement uses a standalone operator, it does not need the SELECT FROM statement.

**Note:** In Figure 17, the SELECT statement also contains the UNION ALL statement to combine the rows from two SELECT operations against separate sources, each with its own operator.

Figure 17: SELECT Statement in an APPLY Statement

![SELECT Statement in an APPLY Statement](image)

**Derived Column Data Types**

Derived columns, which have values derived from the evaluation of expressions, require derived column names. A derived column name must be defined in the schema of a job script, and if multiple schemas are used, identically defined in all schemas.

The following warnings and errors can occur:

- Incompatibility between the schema-defined derived column and the resulting data type attributes of the expression, such as assigning a numeric value to a CHAR column. An error results when the script is compiled, and the job terminates.
- An incompatibility such as the value of a numeric expression being outside the range of the data type of its derived numeric column, which can be detected only during execution. An error results, and the job terminates.
- Truncated characters due to an incompatibility in character data type length. When the script is compiled, one warning is generated for every applicable derived column, but no run-time message is generated if truncation occurs.

**Using the DDL Operator in an APPLY Statement**

The DDL operator can be specified in the APPLY statement in either single or multi-statement format. To execute each statement as its own transaction, you should have one SQL statement per DML group (enclosed in parentheses).

If more than one statement is specified in a DML group, the operator combines them all into a single multi-statement request and sends it to the Teradata Database as one transaction. Teradata Database enforces the rule that a multi-statement DML group can have only one DDL statement and it must be the last statement in the transaction, which means the last statement in the group. The SQL statements are executed by groups in the order they are
specified in the APPLY statement. If any statement in the group fails, then all statements in that group are rolled back and no more groups are processed.

The following is a simplified example of a DDL operator in a single-statement format:

```sql
APPLY
'SQL statement1',
'SQL statement2',
........
'SQL statementN'
TO OPERATOR (operator_specifications)
```

The following is a simplified example of a DDL operator in a multi-statement format:

```sql
APPLY
('SQL statement1a', 'SQL statement1b', ........),
('SQL statement2a', 'SQL statement2b', ........),
........
('SQL statementNa', 'SQL statementNb', ........)
TO OPERATOR (operator_specification)
```

### Using the Update Operator to Delete Data

Use the Update operator with the DeleteTask attribute to delete data from the Teradata Database. The Update operator functions as either a standalone or a consumer operator, depending on whether or not data is required to complete the deletion.

Consider the following rules when using the DeleteTask feature:

- The Delete Task feature may not be used on a database view.
- Only one special session will be connected.
- Only one instance may be specified.
- Only one DML group may be specified.
- Only one DML DELETE statement in the DML group may be specified.
- Only one target table may be specified.
- The first of the error tables (the acquisition error table) is not used and is ignored.
- Only one data record may be provided if using a WHERE clause. For example, you can send more than one row to the data stream (from the producer operator), but only the first one is used.

For further information on use of the DELETE task with a standalone Update operator, see Chapter 20: “Update Operator.”
Defining Job Steps

Job steps are units of execution in a Teradata PT job. Using job steps is optional, but when used, they can execute multiple operations within a single Teradata PT job. Job steps are subject to the following rules:

- A job must have at least one step, but jobs with only one step do not need to use the STEP syntax.
- Each job step contains an APPLY statement that specifies the operation to be performed and the operators that will perform it.
- Most job steps involve the movement of data from one or more sources to one or more targets, using a minimum of one producer and one consumer operator.
- Some job steps may use a single standalone operator, such as:
  - DDL operator, for setup or cleanup operations in the Teradata Database.
  - The Update operator, for bulk delete of data from the Teradata Database.
  - OS Command operator, for operating system tasks such as file backup.

Using Job Steps

Job steps are executed in the order in which they appear within the DEFINE JOB statement. Each job step must complete before the next step can begin. For example, the first job step could execute a DDL operator to create a target table. The second step could execute a Load operator to load the target table. A final step could then execute a cleanup operation. For a complete example of a multi-step job, see Appendix A: "Job Script Examples."

The following is a sample of implementing multiple job steps:

```
DEFINE JOB multi-step
(  DEFINE SCHEMA...;
  DEFINE SCHEMA...;

  DEFINE OPERATOR...;
  DEFINE OPERATOR...;

  STEP first_step
  (    APPLY...; /* DDL step */
  );

  STEP second_step
  (    APPLY...; /* DML step */
  );

  STEP third_step
  (    APPLY...; /* DDL step */
  );
);```
Starting a Job from a Specified Job Step

You can start a job from step one or from an intermediate step. The `tbuild -s` command option allows you to specify the step from which the job should start, identifying it by either the step name, as specified in the job STEP syntax, or by the implicit step number, such as 1, 2, 3, and so on. Job execution begins at the specified job step, skipping the job steps that precede it in the script.

For information on using `tbuild -s` command, see Chapter 21: “Launching a Job.”

For a complete example of a multi-step job, see Appendix A: “Job Script Examples.”
SECTION 2 Pre-Job Setup
This chapter provides information on setup tasks, some of which can be done outside of the job and some that are most efficiently done as part of the job script.

Topics include:

- Setting Up Configuration Files
- Setting Up the Job Variables Files
- Setting Up the Teradata Database
- Setting Up the Client System

## Setting Up Configuration Files

Before beginning to run Teradata PT job scripts, Teradata recommends that you set up the following global configuration files. Set up is normally only required at Teradata PT installation.

<table>
<thead>
<tr>
<th>Configuration File Parameters</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Job Variables File</td>
<td>Allows you to specify global variables and values that can be used by many jobs.</td>
</tr>
<tr>
<td></td>
<td>Creating global variables helps eliminate the errors inherent in re-entering values in multiple job scripts.</td>
</tr>
<tr>
<td></td>
<td>It also allows you to keep such information as username and password out of scripts where they may be seen by unauthorized persons.</td>
</tr>
<tr>
<td>Checkpoint Directory</td>
<td>Setup for these directories is included in Teradata PT installation procedure.</td>
</tr>
<tr>
<td>Log Directory</td>
<td>For details, see Teradata Tools and Utilities Installation Guide.</td>
</tr>
</tbody>
</table>

The contents of these configuration files can then be shared by all jobs.

Teradata PT supports configuration files on the following operating systems for the specification of Teradata PT system-wide configuration options.
### Setting Up the Job Variables Files

You can create variables and assign variable values in two types of files:

- **Global Job Variables File** - Every Teradata PT job *automatically* reads the global job variables file. If there is a variable called out anywhere in the script, the job looks for the corresponding value in the global job variables file.

  **Note:** A global job variables file is available on UNIX and Windows.

- **Local Job Variables File** - You can also create a local job variables file to contain values for job variables. When the `tbuild` command specifies the `-v` option, the associated job will read the local job variables file and the variable value found there will be used in place of the value from the global job variables file. For example:

  ```
  tbuild -f weekly_update.tbr -v local.jobvars
  ```

  **Note:** On z/OS, specify a local job variables file through the DDNAME of `ATTRFILE`.

If possible, define known common system variables when you begin to employ Teradata PT. The use of variables enhances the efficiency and security of job scripting. You can add additional variable names and values when required.

Setting up variables requires two actions:

To specify configuration file parameters use this form: `<parameter> = <single-quoted string>;

For example, on UNIX:

```perl
GlobalAttributeFile = '/usr/tbuild/<version_number>/global.jobvariables.txt';
CheckpointDirectory = '/usr/tbuild/<version_number>/checkpoint';
LogDirectory = '/var/log/tbuild';
```

where:

- `GlobalAttributeFile` is the path to the global job variables file.
- `CheckpointDirectory` is where Teradata PT stores job checkpoint records.
- `LogDirectory` is the directory where Teradata PT stores job logs.

### Platform Configuration File

<table>
<thead>
<tr>
<th>Platform</th>
<th>Configuration File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td><code>&lt;installation directory&gt;\twbcfg.ini</code></td>
</tr>
<tr>
<td>UNIX</td>
<td><code>&lt;installation directory&gt;/twbcfg.ini</code></td>
</tr>
<tr>
<td></td>
<td>Configuration options can also be customized by placing the following configuration file in the home directory: <code>&lt;HOME&gt;/.twbcfg.ini</code></td>
</tr>
<tr>
<td>z/OS</td>
<td>Specified through the DDNAME of <code>ATTRFILE</code>.</td>
</tr>
</tbody>
</table>
Setting up script values to defer to local or global job variable values by specifying them using the form `<attribute name>=@<job variable name>`, as follows:

```
VARCHAR UserName=@Name
VARCHAR UserPassword=@Password
VARCHAR TdpId=@Tdp
```

• Entering job variable assignments in the global job variables file and the local job variables file separated by commas, or one assignment per line without commas, in the form `<job variable name>=‘value’`, as follows:

```
Name=‘userX’,
Password=‘secret’,
Tdp=‘Td32y’,
```

The location of the global job variables file and the local job variables file, as well as the name of the global and local job variables file, is user-definable in the context of the following:

• The user-defined path and filename of the global job variables file must put inside the twbcfg.ini file as an entry, as follows:

```
GlobalAttributeFile = ‘<userspath>/<usersGlobalJobVariablesName.’
```

• The user-defined path and filename of the local job variables files is specified after the `-v` option on the command line.

For further information on specifying job variables in the script, see “Setting Up Job Variables” on page 46.

For information on specifying job variables on the command line at job launch, see “Assigning Job Variables on the Command Line” on page 263.

## Setting Up the Teradata Database

Setting up the Teradata Database can be done in a preliminary job step, using the DDL operator. Most jobs require the use of the DDL operator to perform such setup each time the job is run. The database setup task is comprised of two parts:

1. Define the DDL operator
2. In the APPLY statement for the step that references the DDL operator, enter the SQL statement that will execute the setup in the Teradata Database, for instance a CREATE TABLE statement.

The DDL operator is shown in a preliminary step for most job examples shown in Chapters 5 through 7.

For information on the DDL operator, see “Chapter 9 DDL Operator” on page 161 and the related chapter in *Teradata Parallel Transporter Reference*. 

Teradata Parallel Transporter User Guide
**Objective:**

Drop error tables and then set up the target table for a job that loads data into Teradata Database.

**Note:** The DDL operator can also be used to drop staging tables from previous steps when employed in multi-step jobs.

**Data Flow Diagram**

Figure 20 shows a diagram of the elements for setting up the Teradata Database to receive data.

**Script Example**

For the script example that corresponds to this job, see “Script Example 1A: High Speed Bulk Loading from Flat Files into an Empty Teradata Database Table” on page 404. Note that many of the script examples have a similar setup step that uses DDL operator.

**Rationale**

This job uses the DDL operator because it can execute a wide variety of SQL statements in the Teradata Database to prepare for the main job tasks that occur in succeeding job steps.

**Usage Notes**

Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job.

For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 5 of this Users Guide.
**DDL Operator**
- Key Attributes
  - Tdpid
  - UserName
  - UserPassword
  - ErrorList
- Important Considerations
  - Enter the DDL to be executed in the APPLY statement for the step that references the DDL operator.
  - Use job variables for standard DDL statements that can be used for multiple jobs.

**Setting Up the Client System**

In addition to setting up the Teradata Database, job setup sometime requires that tasks such as file transfer be done on the client system, using operating system commands. A job step that performs such a task is composed of two elements:

1. Define the OS Command operator
2. In the APPLY statement for the step that references the OS Command operator, enter the operating system command that will execute the setup.

For information on the DDL operator, see “Chapter 16 OS Command Operator” on page 207, and the related chapter in *Teradata Parallel Transporter Reference*.

**Job Objective:**

Copy files from one client location to another before extracting them and sending them to Teradata Database for loading.

**Data Flow Diagram**

Figure 20 shows a diagram of the elements for setting up the Teradata Database to receive data.
Script Example

For the script examples that correspond to this job, see “Script Example 3: Loading BLOB and CLOB Data into Teradata Database” on page 414. Note that other script examples may execute a similar setup step that uses the OS Command operator.

Rationale

This job uses the OS Command operator because it is the only operator that can execute operating system commands as part of a Teradata PT job.

Usage Notes

Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job.

For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 5 of this Users Guide.

**OS Command Operator**

- Key Attributes
  - OsCmd
  - IgnoreError
- Important Considerations
  - The OS Command operator may not be equally supported on all platforms. Its functionality on z/OS platforms is limited.
  - Host operating system commands are by definition platform-specific. Some degree of platform-independence may be achieved for scripts that use the OS Command operator by using a job variable for the actual command text (the value of OS Command operator attribute OsCmd).
This chapter discusses Teradata Database requirements that affect the creation and use of Teradata PT job scripts.

Topics include:

- Teradata Database Logon Security
- Teradata Database Access Privileges
- Optimizing Job Performance with Sessions and Instances
- Limits on Teradata PT Task Concurrency

For additional information on Teradata Database-related scripting issues, see Chapter 26: “Advanced Teradata Database Considerations.”

Teradata Database Logon Security

Security for Teradata PT logons to the Teradata Database involves the following concepts:

- Specification of Security Attributes
- Teradata Database Authentication
- External Authentication
- Encryption
- z/OS Security

Specification of Security Attributes

The following security-related attributes may be required for logons to Teradata Database depending on the user authentication method employed.

- UserName
- UserPassword
- TdpId
- LogonMech
- LogonMechData

For information on how the attribute values vary with authentication method, see “Teradata Database Authentication” on page 75, and “External Authentication” on page 76.
Specifying Security Attribute Values
Values for the security attributes can be assigned in any the following statements, which are listed in the order they are processed, from lowest to highest priority.

• DEFINE OPERATOR
• in an APPLY statement, or SELECT clause of an APPLY statement

Note: Specifying an attribute value at a higher priority level (an APPLY statement) supersedes values for the attribute specified at a lower level (a DEFINE OPERATOR statement).

Security Strategy
Consider the following when deciding were to specify values for security attributes:

• Operators can be more generally applied if they are not required to carry values for the security-related attributes, although these values can be overridden in APPLY statements.
• When processing sensitive information with Teradata PT, specifying the UserName and UserPassword values as job variables avoids problems that may occur if such logon information is kept in plain view in job scripts.
• If a single user has the privileges necessary to run an entire job script, specify the UserName and UserPassword values as job variables rather than individually in the operators, other objects, or APPLY statements.
• If privilege requirements vary greatly among instances of the same object, specify the userName and userPassword values in the APPLY statement.

Teradata PT jobs log on to either the Teradata Database, an outside data source, or both. Logon requirements differ between Teradata Database and outside data sources.

When Accessing Non-Teradata Data Sources
The following operators access non-Teradata data sources. However, since they logon through an access module, they do not require logon information.

• DataConnector
• FastLoad INMOD Adapter
• FastExport OUTMOD Adapter
• MultiLoad INMOD Adapter

For these operators, logon information must be entered as part of the access module or INMOD/OUTMOD routine through which the operator accesses the outside data source.

Note: Although it also accesses outside data sources, the ODBC operator functions differently from other such operators, and allows the option of specifying the following in the job script:

• UserName
• UserPassword

For detailed information, see “ODBC Operator” in Teradata Parallel Transporter Reference.
**Chapter 4: Teradata Database Effects on Job Scripts**

**Teradata Database Logon Security**

When a user accessing a Teradata Database is authenticated by the Teradata Database, values for the security attributes should be specified as follows:

<table>
<thead>
<tr>
<th>Security Attribute</th>
<th>Description</th>
<th>Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserName</td>
<td>The Teradata Database username.</td>
<td>All users employing Teradata Database authentication must be defined in the Teradata Database. For information about creating users, see Database Administration. For information about assigning passwords, see Security Administration.</td>
</tr>
<tr>
<td>UserPassword</td>
<td>The Teradata Database password associated with the username</td>
<td>For information about creating users, see Database Administration. For information about assigning passwords, see Security Administration.</td>
</tr>
</tbody>
</table>
| TdpId              | Identifies the connection to the Teradata Database | Optional: If you don’t specify a TdpId, the system will use the default Tdpid, as defined in the Teradata Client clispb.dat. Specify either:  
  - For channel-attached clients, specify the identity of the Teradata Director Program through which Teradata PT connects to the database. For example: TDP6  
  - For network-attached clients, specify the name of the interface to the Teradata Database system, or logical host group. For example: cs4400S3 |
| LogonMech          | A security mechanism used to externally authenticate the user. | Not applicable for Teradata Database authentication. |
| LogonMechData      | Username, password, and other data required by an external authentication mechanisms to complete the logon. | Not applicable for Teradata Database authentication. |

**Note:** Make sure that any UserName specified in a Teradata PT job script has the privileges necessary to carry out all operations covered by the logon.
External Authentication

In some cases the username in a job script must be authenticated by an agent external to the Teradata Database, such as Kerberos or Active Directory. External authentication is only available for jobs launched from network-attached clients. It requires special setup.

**Note:** Do use external authentication to log on a Teradata PT job script until you understand the associated setup and logon requirements, as shown in *Security Administration.*

Specify security attributes for external authentication as follows:

<table>
<thead>
<tr>
<th>Security Attribute</th>
<th>Description</th>
<th>Strategy</th>
</tr>
</thead>
</table>
| UserName           | The name of the user being authenticated for access to Teradata Database | Required
|                    | Specify a Teradata Database username and password. |
| UserPassword       | The password associated with the UserName | Optional |
| TdpId              | Identifies the connection to the Teradata Database | Optional
|                    | If you don’t specify a TdpId, the system will use the default TdpId, as defined in the Teradata Client clispb.dat. |
|                    | To specify a TdpId, do the following: |
|                    | • For channel-attached clients; this value is the identity of the Teradata Director Program through which Teradata PT connects to the database. For example: TDP6 |
|                    | • For network-attached clients; this value is the name of the interface to the Teradata Database system, or logical host group. For example: cs4400S3 |
| LogonMech          | The security mechanism used to authenticate the user. TD2 is required for all Teradata Database authentication. | Optional, depending on security setup. TD 2 is the default mechanism and the system will automatically defer to it unless the default has been set to another mechanism or TD 2 has been disabled. |
| LogonMechData      | Data required only by external authentication mechanisms to complete the logon. | LogonMechData is not required for Teradata Database authentication, and is ignored. |
### Security Attribute | Description | Strategy
--- | --- | ---
User Name | The name used to log on to the network prior to launching the job script. | Optional:  
- For single sign-on: The username employed for the initial network logon must match a username defined in the Teradata Database. No additional username and password information is required.  
- For other external authentication methods (e.g. LDAP or Kerberos), specify the username and password values in one of the following ways:  
  - As values for the UserName and UserPassword attributes, except for logons that require use of LogonMechData (see below).  
  - As the value for the LogMechData attribute.  
  **Note:** Do not declare the UserName or UserPassword attributes if you plan to enter username and password data in LogonMechData.
User Password | The network password (not the Teradata Database password) associated with the UserName | Optional  
If you don’t specify a TdpId, the system will use the default Tdpid, as defined in the Teradata Client clispb.dat. Specify either:  
- For channel-attached clients, specify the identity of the Teradata Director Program through which Teradata PT connects to the database. For example: TDP6  
- For network-attached clients, specify the name of the interface to the Teradata Database system, or logical host group. For example: cs4400S3
TdpId | Identifies the connection to the Teradata Database | Required unless the external authentication mechanism is the default.  
Choose among the following, depending on authentication method:  
- Use LDAP for directory sign-on  
- Use KRB5 or NTLM for single sign-on and sign-on as logons
LogonMech | The security mechanism that authenticate the user. Similar to the .logmech statement in a Teradata Database logon string. | Optional  
LogonMechData | Data required by external authentication mechanisms to complete the logon. Similar to the .logdata statement in a Teradata Database logon string. | Optional  

#### Encryption
All Teradata PT operators that interface with the Teradata Database have the option to encrypt job data during transmission across the network. The data is then decrypted and checked for integrity when it is received by the Teradata Database. Encryption is only available for network-attached clients.

The following operators support data encryption:
- DDL
- Export
Set the `DataEncryption` attribute to ‘On’ to enable encryption. The default setting is ‘Off.’ Encryption can also be set in an APPLY statement and as a job variable.

**Note:** Encryption may result in a noticeable decrease in load/unload performance due to the time required to encrypt, decrypt, and verify the data, especially when the job involves the processing of very large quantities of data. Take care to encrypt data only when the security benefit is likely to outweigh the performance cost.

### z/OS Security

On the z/OS platform, all Teradata PT jobs run as batch applications through JCL, just like the standalone utilities. You should be able to run the Teradata PT infrastructure without making any special security provisions.

### Teradata Database Access Privileges

The user represented by the value of the `UserName` attribute in an operator definition must have the Teradata Database access privileges required for the actions that the operator will execute. Refer to the following list and make sure all users referenced in your job script have the access privileges necessary for job to run:

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

- **DDL operator:**
  - The DDL operator requires all privileges necessary to execute the SQL that it submits as part of a Teradata PT job, for example, CREATE TABLE privileges.
  - REPLCONTROL privilege to set the ReplicationOverride attribute.

- **Export operator:**
  - SELECT privileges on the Export target table.

- **SQL Inserter operator:**
  - REPLCONTROL privilege to set the ReplicationOverride attribute.

- **Stream operator:**
  - SELECT, INSERT, UPDATE, and DELETE privileges on all 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.
- REPLCONTROL privilege to set the ReplicationOverride attribute.

The Stream operator does not have any special protections on the database objects it creates. Therefore, administrators and users must establish the following privileges on the databases used by the Stream operator:
- CREATE TABLE privileges on the database where the restart log table is placed.
- CREATE TABLE privileges on the database where the error table is placed.
- CREATE/DROP MACRO privileges on the database where macros are placed.
- EXECUTE MACRO privileges on the database where the macros are placed.

Macros slightly complicate privileges. The remaining privileges necessary to run the Stream operator have two scenarios.
- When a Stream operator macro is placed in the same database as the table that it affects, the required privileges are INSERT/UPDATE/DELETE on the table affected by the DML executed.
- When a Stream operator macro is placed in a different database from the table it affects, the required privileges for the database where the macro is placed are INSERT/UPDATE/DELETE WITH GRANT OPTION in the table affected by the DML executed. You must also have EXECUTE MACRO rights on the database where the macro is placed.

To change a table, you must have the corresponding INSERT, UPDATE, or DELETE privileges for that table.

- Update operator:
  - SELECT and INSERT privileges on the Update target table
  - SELECT and INSERT privileges on the error tables, and DROP privileges on the database that contains the error tables.
  - SELECT, INSERT, and DELETE privileges on the restart log table, and DROP privileges on the database that contains the restart log table.
  - REPLCONTROL privilege to set the ReplicationOverride attribute.

For detailed information on how to GRANT such privileges to users, see Database Administration.

**Teradata PT Handling of Roles**

If database access privileges for the logon user of a Teradata PT script are defined by more than one Teradata Database role, the default user role (as set in the user profile) automatically applies when the user logs on. Each operator that communicates with the Teradata Database logs on separately, and Teradata PT scripts do not support use of the SET ROLE statement (except for the DDL operator). Since the default role cannot be reset for a Teradata PT session, make sure that Teradata PT user default role includes all the necessary privileges.
Optimizing Job Performance with Sessions and Instances

Job scripts can be constructed to maximize job performance by specifying multiple instances of an operator at the point where the operator is referenced in an APPLY statement. Operator instances then execute in parallel to complete the task.

Each operator used in a single-step job, or in a job step, will attempt to simultaneously log on to one Teradata Database session for each AMP configured on the Teradata Database system. This feature provides a high degree of parallelism to maximize operator performance.

The following operators can be configured to enhance job performance through the optimization of instances and sessions:

- DataConnector (instances only)
- Export
- Load
- SQL Inserter (instances only)
- Stream (instances only)
- Update

The following sections discuss how to optimize the specification of instances and sessions in a job script.

Determining the Optimum Number of Sessions

Each operator used in a single-step job, or in a job step, will attempt to simultaneously log on to one Teradata Database session for each AMP configured on the Teradata Database system. However, this may not be optimal for every job. For some jobs, the default parallelism may be excessive. In other cases, there may not be enough available AMPs to provide the sessions necessary to run the job efficiently. Teradata PT provides the following attributes to optimize session usage for the five operators that support session limits.

- MaxSessions determines the maximum number of sessions an operator can use.
- MinSessions, determines the minimum number of sessions that must be available in order for the job to run.

Setting Values for the MaxSessions Attribute

Consider the following factors when specifying a value for the MaxSessions attribute:

- If no value is set for MaxSessions, the operator attempts to connect to one session per available AMP.
- The DDL, ODBC, and SQL Selector operators are limited to a single concurrent session, that is, one session each per single-step job, or one each session per step in a multi-step job.
• If the value of the MaxSessions attribute for an operator is smaller than the number of operator instances, the job will abort.

• If the value of MaxSessions is set to a number greater than the number of available AMPs, the job runs successfully, but logs on only as many sessions as available AMPs.

• For some jobs, especially those running on systems with a large number of AMPs, the default session allocation (one per available Teradata Database system AMP) may not be advantageous, and you may need to adjust the MaxSessions attribute value to limit the number of sessions used. After the job has run, use the evaluation criteria shown in “Strategies for Balancing Sessions and Instances” on page 83 to help adjust and optimize the MaxSessions setting.

• The SQL Inserter operator supports only one session.

• The Stream operator uses an SQL protocol, so it is not seen as a “load job” by the Teradata Database. Therefore, Stream operator connects to as many sessions as requested, up to the number of sessions allowed by the Teradata Database.

The Effect of Operator Instances on the MaxSessions Value

The number of sessions specified by the value of the operator MaxSessions attribute are balanced across the number of operator instances. For example, if the Update operator is defined with two instances, and the MaxSessions attribute is set to 4, each instance of the defined Update operator will run two sessions, provided there are at least four AMPs on the system.

An Update operator uses a maximum of one session per available AMP on the Teradata Database system. This means that if your Teradata Database system has ten available AMPs, the MaxSessions value must be less than or equal to ten.

Examples of How the MaxSessions Value is Processed

If there are ten maximum sessions defined for an operator, the following combinations of instances and sessions are possible:

• One instance with ten sessions
• Two instances with five sessions each
• Three instances: two instances with three sessions and one instance with four sessions
• Four instances: two instances with three sessions and two instances with two sessions
• Five instances with two sessions each

Setting Values for the MinSessions Attribute

Use the MinSessions operator attribute to specify the minimum number of sessions needed to run a job. Larger systems are more likely to be enable to connect a sufficient number of sessions, whereas smaller, busier systems may often not have enough available AMPs to run the job efficiently. Proper setting of the MinSessions attribute prevents the job from running until there are enough AMPs for it to run at an acceptable rate.

Setting values for the MinSessions attribute should be done by running a job several times, observing the results, and adjusting the MinSessions value until it is optimal.
Specifying Instances

You can specify the number of instances for an operator in the APPLY TO or SELECT FROM statement in which it is referenced, using the form (operator_name [number of instances]), as shown in the following example:

```
APPLY <DML>...TO OPERATOR (UPDATE_OPERATOR [2]...)
```

In attempting to determine the right number of instances for your job, note that producer operators tend to use all of the instances specified in the script, while consumers often use fewer instances than the number specified. This difference results from the fact that consumers and producers use instances differently:

- Producers automatically balance the load across all instances, pumping data into the data stream as fast as they can.
- By default, consumers will use only as many instances as needed. If one instance can read and process the data in the data stream as quickly as the producers can write it, then the other instances are not used. If the first instance cannot keep up with the producer operators then the second instance is engaged, and so on.

The -C command line option overrides the default behavior by informing producer operators and their underlying data streams to ship data blocks to target consumer operators in a cyclical, round-robin manner, providing a more even distribution of data to consumer operators.

Consider the following when specifying operator instances:

- If the number of instances is not specified, the default is 1 instance per operator.
- Experiment. Start by specifying only one or two instances for any given operator.
- Teradata PT will start as many instances as specified, but it uses only as many as needed.
- Don’t create more instances than needed--instances consume system resources.
- Read the Teradata PT log file, which displays statistics showing how much data was processed by each instance. Reduce the number of instances if you see under-utilized instances of any operators. If all instances are used add more and see if the job runs better.
- If the number of instances exceeds the number of available sessions, the job aborts. Therefore, when specifying multiple instances make sure the MaxSessions attribute is set to a high enough value that there is at least one session per instance.
- After the job runs, use the evaluation criteria shown in “Strategies for Balancing Sessions and Instances” on page 83 to help adjust and optimize the number of operator instances.

Calculating Shared Memory Usage Based on Instances

Use the following formula to decide the shared memory usage based in instance:

Let $n$ and $m$ be the number of instances of the producer and consumer operators, respectively.

**Note:** Data from producers are multiplexed into consumers through data streams. In other words, the number of data streams to be used per job would be $n \times m$.

Let $q$ be the maximum queue depth (in terms of 64K buffers) of a data stream. (In Teradata PT TTU 7.0, two appears to be the most efficient number of buffers)
Examples

Note the following examples assume each of the data streams between producers and consumers is full during execution (for example, \( q = 2 \) buffers):

Shared memory used by 2 producers and 2 consumers:

\[ (((2 \times 2) \text{data streams}) \times 64K \times q) = 512K \]

Shared memory used by 4 producers and 4 consumers:

\[ (((4 \times 4) \text{data streams}) \times 64K \times q) = 2048K \]

Shared memory used by 24 producers and 1 consumer:

\[ (((24 \times 1) \text{data streams}) \times 64K \times q) = 3072K \]

Note: The maximum shared memory available for a job is 10M.

System Characteristics that Affect Sessions versus Instances

When specifying the number of sessions and instances to meet the goal of best overall job performance, consider these other factors:

- Number of nodes
  
  Larger Teradata Database systems provide more AMPs, and by default make available more sessions.

- Number and speed of CPUs
  
  - The greater the available processing power, the less need there may be for a large number of parallel sessions. A few connections (sessions) to a very powerful system can handle the same amount of throughput as a larger number of connections to a less powerful system.
  
  - An imbalance between the throughput capability of the source and target system may reduce the benefits of using parallel sessions. The operator that interfaces with the more powerful system may spend excessive time waiting for the operator that interfaces with the less powerful system to complete its tasks.

- Workload
  
  Always attempt to run a Teradata PT job at times when the workload leaves enough available AMPs for the optimal number of sessions.

- I/O bandwidth

- Structure of the source or target tables/records

- Volume of data being loaded

Strategies for Balancing Sessions and Instances

Without concrete performance data, no recommendations or guidelines exist for determining the optimum number of sessions or instances, but some strategies exist for finding good balances.
Balancing sessions and instances helps to achieve the best overall job performance without wasting resources.

- Logging on unnecessary sessions is a waste of resources.
- Starting more instances than needed is a waste of resources.

**Strategy 1**

Start with MaxSessions equal to the number of available AMPs or number of sessions you want to allocate for the job, using one instance of each operator (producers and consumer).

1. Run the job and record how long it took for the job to complete.
2. Then, increment the number of instances for the consumer operator. Do not change the number of sessions, because changing multiple variables makes it difficult to compare.
3. Rerun the job and examine the job output:
   - Did the job run faster?
   - Was the second consumer instance used?
   - How many rows were processed by the first vs. the second instance? (For example, were they about equal or was one doing 80% of the work while the other was only doing 20%?) If the work was balanced, another instance might improve performance. If the second instance did not do much work, a third one is likely to waste resources without much performance gain.
     
     If the work was unbalanced, another instance might be better. If the second instance did not do much work, a third one would not likely get engaged.
4. Repeat the process of increasing the number of instances for the consumer operator until you are using as many instances as it needs.

Now it is time to look at the producers. You can try increasing the number of each producer instance separately to see if it will feed data into the data stream at a higher rate.

1. Increase the number of instances for each producer operator separately. Again, do not change the number of sessions.
2. Rerun the job and compare the job output:
   - Did the job run faster? Remember this is the ultimate goal!
   - Was there a change in the number of consumer operator instances used? Because the work always is balanced across the producer instances, you should look at the impact on the consumer instances to see if the change impacted the job.
   - Was there a change in the balance of the consumer operator instances? You want to balance the number of rows being loaded across the number of instances, using as many instances as necessary.
     
     **Note:** Be careful not to trade off instance balance for overall job performance. Just because rows are read evenly across all instances, it does not necessarily mean that the balanced load makes the whole job run faster.
3. Depending on your results, you may want to increase the number of instances for the producer or consumer operators.
Now that you know an acceptable number of instances, you can modify the value of MaxSessions to see if there is an impact.

1. Decrease the value of MaxSessions. It is best to make MaxSessions a multiple of the number of instances for the operator so they are evenly balanced across the instances.

2. Rerun the job and compare the output:
   - Did the job run faster?
   - Was there a change in the number of consumer operator instances used?
   - Was there a change in the balance of data in the consumer operator instances?

3. Depending on the results, you may want to use the original MaxSessions, or continue experimenting. You may even want to revisit the number of instances you are using.

**Strategy 2**

Start with MaxSessions equal to the number of available AMPs or number of sessions allocated for the job, using four instances of each operator (producers and consumer).

1. Run the job and examine the output:
   - How long did it take for the job to complete?
   - How many consumer operator instances are being used?
   - How many rows are being processed by each consumer operator instance? We are looking for balance without wasted resources.

2. Make adjustments based on your results.
   - If the job is not using all the consumer operator instances:
     - Decrease the number of instances to eliminate the unused ones.
     - Decrease the number of producer instances by one. Avoid doing anything too drastic, or it will be difficult to determine the optimal number.
   - If the job is using all the consumer operator instances, and the workload is balanced:
     - Try increasing the number of consumer operator instances.
   - If the job is using all the consumer operator instances, but the workload not balanced:
     - Try increasing the number of producer operator instances.

3. Rerun the job and compare the output:
   - Did the job run faster? Remember, this is the ultimate goal!
   - Was there a change in the number of consumer operator instances used?
   - Was there a change in the balance of data in the consumer operator instances?

4. Repeat the process to optimize the number of producer and consumer instances.

Now that you know the best number of instances, you can modify the number of MaxSessions to see if there is an impact.

1. Decrease the number of MaxSessions. It is best to make MaxSessions a multiple of the number of instances for the operator so they are evenly balanced across the instances.

2. Rerun the job and compare the output:
   - Did the job run faster?
• Was there a change in the number of consumer operator instances used?
• Was there a change in the balance of data in the consumer operator instances?

3 Depending on the results, you may want to use the original MaxSessions, or continue experimenting. You may even want to re-visit the number of instances you are using.

**Limits on Teradata PT Task Concurrency**

The following factors limit task concurrency within and among Teradata PT jobs:

- Teradata Database limits
- Teradata Warehouse Manager limits

Note that the lowest, most restrictive limit imposed by these factors takes precedence and will be used to determine job concurrency.

**Teradata Database Task Concurrency Limits**

To ensure that the data remains accessible to users, Teradata Database enforces limits on the quantity of system resources that can be used by extract and load utilities. These limits affect use of the following operators:

- Export
- Load
- Update

Each time one of these operators is referenced in an APPLY statement counts as one task. Limits are controlled by two DBSControl settings:

- MaxLoadAWT
- MaxLoadTasks

The system counts the total number of “tasks” attempting to run concurrently and then the settings of the two fields are applied to that total, and task limits are enforced, where necessary.

To get a general idea of total concurrent Teradata PT tasks for your site, consider the following basic rules:

- Each operator specified in an APPLY statement that interfaces with the Teradata Database constitutes one task. For example:
  - A job that moves data between a Teradata Database and either a second, separate Teradata Database or an external source or target requires only one task, because only a single operator interfaces with each Teradata Database at one time.
  - A job that moves data within a single Teradata Database requires two tasks, because both the extract and load operators interface with the database at the same time.

Keep in mind that job steps within a single job execute sequentially, so the maximum number of concurrent tasks in a single job is two.
• If more than one job script is running concurrently, the total number of tasks applied to the MaxLoad calculations is based on the total number of operators that run concurrently in all jobs.

• Default MaxLoad settings allow concurrent execution of approximately 15 tasks, so you may need to reset the MaxLoad values.

Actual limits are subject to additional factors. For details on how to set the MaxLoadAWT and MaxLoadTasks fields in the DBSControl GDO, and how those settings will affect Teradata PT jobs, see the section on the DBSControl utility in Utilities, Volume 1.

**Teradata Warehouse Manager Task Concurrency Limits**

The Teradata Warehouse Manager utility provides the capability to limit the maximum number of load/unload tasks using the throttle feature. If such a throttle has been configured and is more restrictive than other load/unload task limiters, it will be used by the system to determine Teradata PT task limits.
SECTION 3 Job Strategies
This chapter describes several alternative methods for using Teradata PT to move data from a non-Teradata source into a Teradata Database. It includes the following topics:

- Data Flow Description
- Comparing Applicable Operators
- Using Access Modules to Read Data from an External Data Source
- Using Teradata PT EasyLoader
- Common Jobs for Moving Data into a Teradata Database

### Data Flow Description

Teradata PT offers several paths for moving data from a non-Teradata source into a Teradata Database, as shown in the following composite diagram.

Figure 20: Moving Data from a Non-Teradata Source into Teradata Database

Figure 20 shows a composite of the possible paths for moving data from an external source to a Teradata Database. Note that Job Example 5C (Figure 30) allows for writing data to an external data target in parallel with writing data to a Teradata Database.
Comparing Applicable Operators

Once you identify the requirements for moving data from an external data source to Teradata Database, you must select the components the script will use to execute the job. There are three types of components you need to consider:

- A producer operator that reads data directly from the external source and places it in the data stream.
- A producer operator that uses an INMOD routine or access module to access data from an external source and then pre-process the data before sending it into the data stream.
- A consumer operator that takes data from the data stream and writes it to the Teradata Database.

Producer Operators

The Teradata PT producer operators in this section read data from an external data source and place it in the data stream.

The Teradata PT job script invokes a producer operator using a SELECT statement within an APPLY statement. For further information on using SELECT to specify a producer operator, see “Coding the APPLY Statement” on page 82 and the section on APPLY in Teradata Parallel Transporter Reference.

The following table briefly describes and compares the function of each Teradata PT operator that can be used as a producer when moving data from an external source into Teradata Database:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operators that Read Data from External Sources</strong></td>
<td></td>
</tr>
</tbody>
</table>
| DataConnector Operator | Reads flat files from an external data source. Functions similarly to the standalone Teradata DataConnector utility. **Features:**  
  - Can read a specific file or can be used to scan a directory.  
  - Interfaces with all Teradata PT supported access modules.  
  **Limitations:**  
  - Cannot read data from ODBC-compliant data sources.  
  For usage information, see “Chapter 8 DataConnector Operator.” |
| ODBC Operator         | Reads data from most ODBC-compliant data sources. **Limitations:**  
  - Cannot interface with access modules.  
  For detailed usage information, see “Chapter 15 ODBC Operator.” |
Comparing Applicable Operators

Consumer Operators

The Teradata PT consumer operators in this section read data from the data stream and write it to a Teradata Database.

The Teradata PT job script invokes a consumer operator using an APPLY statement. For further information on using APPLY to specify a consumer operator, see “Coding the APPLY Statement” on page 82 and the section on APPLY in Teradata Parallel Transporter Reference.

The following table briefly describes and compares the function of each Teradata PT operator that can be used as a consumer when loading data into a Teradata Database:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastExport INMOD Adapter Operator</td>
<td>Uses FastExport INMOD routines to read data from external files and then process it before sending it to the data stream. For detailed usage information, see Chapter 11: “FastExport OUTMOD Adapter Operator.”</td>
</tr>
<tr>
<td>FastLoad INMOD Adapter Operator</td>
<td>Uses FastLoad INMOD routines to read data from external files and then process it before sending it to the data stream. For detailed usage information, see Chapter 12: “FastLoad INMOD Adapter Operator.”</td>
</tr>
<tr>
<td>MultiLoad INMOD Adapter Operator</td>
<td>Uses MultiLoad INMOD routines to read data from external files and then process it before sending it to the data stream. For detailed usage information, see “Chapter 14 MultiLoad INMOD Adapter Operator.”</td>
</tr>
</tbody>
</table>
| Load Operator                   | Inserts data at high speed into a single empty Teradata Database table. Function is similar to the standalone FastLoad utility. Features:
  • Best used for the initial data loads into Teradata Database tables. Limitations:
  • Does not support UPDATE, SELECT, or DELETE operations.
  • The target table must be empty, with no defined secondary indexes.
  • Multiple parallel instances of the Load operator can be used in a job, but they must all insert data into the same table. For detailed information, see “Chapter 13 Load Operator.” |
Chapter 5: Moving External Data into Teradata Database
Comparing Applicable Operators

Note: Consumer operators have a limit on the number of tables they can load simultaneously, as shown in the following:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Maximum Target Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>1</td>
</tr>
<tr>
<td>Update</td>
<td>5</td>
</tr>
<tr>
<td>Stream</td>
<td>127</td>
</tr>
<tr>
<td>SQL Inserter</td>
<td>1</td>
</tr>
</tbody>
</table>
Comparing Update and Stream Operators

Both the Update operator and the Stream operator can be used to update data in the Teradata Database, however:

- The Update operator *locks* the target tables that are being updated so that interactive data reads and writes *cannot* be performed concurrently.
- The Stream operator *does not lock* the target tables that are being updated so that interactive read and write activities *can* be performed concurrently.

This feature of the Stream operator enables it to perform update operations during periods of heavy table access by other users. Like the other Teradata PT operators, the Stream operator can use multiple sessions and multiple operator instances to process data from several data sources concurrently.

Unlike the Load and Update operators, Stream operator does not use its own protocol to access Teradata. Rather it uses Teradata SQL protocol.

Table 6: Comparing Update and Stream Operators

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Update Operator</th>
<th>Stream Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume</td>
<td>Performs high-volume updates against a large number of rows.</td>
<td>Works better for low-volume real-time updates.</td>
</tr>
<tr>
<td>Performance</td>
<td>Performance improves as the volume of updates increases.</td>
<td>Performance improved with multi-statement requests.</td>
</tr>
<tr>
<td>Lock Granularity</td>
<td>Bulk updates at block level. Must lock all tables, which prevents access until</td>
<td>Does not fully lock target tables during updates. Instead, uses standard SQL</td>
</tr>
<tr>
<td></td>
<td>complete. Rows are not available until the load job is complete.</td>
<td>locking protocols to lock individual rows as updates are applied, which permits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>concurrent read and write access to target tables by other users. Rows are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>immediately available for access once the transaction is complete.</td>
</tr>
<tr>
<td>Number of Tables</td>
<td>No more than 5.</td>
<td>Up to 127.</td>
</tr>
<tr>
<td>Timing</td>
<td>Batches transactions and applies them at a higher volume, but usually at a rate</td>
<td>Loads changes in near real-time.</td>
</tr>
<tr>
<td></td>
<td>that is much slower than real-time.</td>
<td></td>
</tr>
<tr>
<td>Concurrent</td>
<td>Requires an active task for each DEFINE OPERATOR statement in a script that</td>
<td>Does not require an active load task.</td>
</tr>
<tr>
<td>Operations</td>
<td>defines an Export, Load, or Update operator</td>
<td></td>
</tr>
<tr>
<td>Instances</td>
<td>Multiple parallel instances improve update performance.</td>
<td>Multiple parallel instances might or might not improve performance.</td>
</tr>
<tr>
<td>Sequencing</td>
<td>Data is processed in sequence all the time (but not in real-time).</td>
<td>Robust mode must be used if sequencing is needed.</td>
</tr>
</tbody>
</table>
Chapter 5: Moving External Data into Teradata Database

Using Access Modules to Read Data from an External Data Source

Access modules are dynamically attached software components of the Teradata standalone load and unload utilities. Some access modules are usable with Teradata PT and provide the input/output interface between operators and various types of external data storage devices. Any operator that uses access modules can interface with all available access modules.

Be careful to specify the Teradata Parallel Transporter version of any access module you use.

The following access modules can be used as part of a job to move data from an external data source to Teradata Database.

<table>
<thead>
<tr>
<th>Access Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Message Service (JMS)</td>
<td>Provides access to any JMS-enabled messaging system. Reads JMS transaction messages and sends the data to the DataConnector producer operator for transfer to a Teradata Database. Teradata JMS Access Module caches the message throughput data stream in a fallback data file that supports checkpoint and restart functions.</td>
</tr>
<tr>
<td>Named Pipes</td>
<td>Provides access to data from a named pipe and sends it to a DataConnector producer operator. The Teradata Named Pipes Access Module also caches the data input from a pipe in a fallback data file that supports checkpoint and restart functions. This access module is not available on z/OS platforms.</td>
</tr>
<tr>
<td>OLE DB</td>
<td>Provides read access to data from an OLE DB provider application, such as Connix or SQL Server, which can access flat files, spreadsheets, and databases in an external data store.</td>
</tr>
<tr>
<td>WebSphere MQ</td>
<td>Provides access to transactional data from IBM WebSphereMQ. The Teradata WebSphere MQ Access Module also caches the output message stream in a fallback data file that supports checkpoint and restart functions.</td>
</tr>
</tbody>
</table>

Table 6: Comparing Update and Stream Operators (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Update Operator</th>
<th>Stream Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>DML Statements</td>
<td>Uses actual DML statements.</td>
<td>Uses macros to modify tables rather than actual DML commands.</td>
</tr>
<tr>
<td>Work Tables</td>
<td>Requires one work table per target table.</td>
<td>Work tables not required.</td>
</tr>
</tbody>
</table>
Specifying an Access Module

Use the following attributes in the DataConnector operator definition to specify the optional use of an access module:

- **AccessModuleName**
  Each access module has a pre-assigned name depending on the operating system on which it is installed. For instance, the JMS access module running on HP-UX is named libjmsam.sl.

- **AccessModuleInitStr**
  Specifies the access module initialization string.

For detailed information about configuring and using access modules with Teradata PT, see *Teradata Tools and Utilities Access Module Reference*.

For information about creating custom access modules, see *Teradata Tools and Utilities Access Module Programmer Guide*.

z/OS Considerations

When using access modules residing within a z/OS program library, either a PDS or a PDSE, the access module name provided to the producer operator using the AccessModuleName attribute is the member name within the library. It may be a system library, private library, or even a temporary library.

If a system library contains the access module, no further JCL is required. However when a private or temporary library houses the access module, a JOBLIB DD statement or a STEPLIB DD statement is required within the jobstream to designate the library containing the access module. The following example shows a typical JOBLIB DD statement for a private library in a TPT jobstream:

```plaintext
//JOBLIB DD DISP=SHR,DSNAME=STV.TI70APP.TWB.LOAD
//        DD DISP=SHR,DSNAME=STV.TI70APP.APP.L
//        DD DISP=SHR,DSNAME=PROD.TERADATA.LOAD
```

where:

- STV.TI70APP.TWB.LOAD is the TPT installation library.
- STV.TI70APP.APP.L is the CLI installation library.
- PROD.TERADATA.LOAD is a private library that contains all Teradata related access modules.

The above is the recommended concatenation order for these libraries.

Using Teradata PT EasyLoader

Teradata PT EasyLoader is a command-line interface to Teradata PT for loading data from a delimited format flat file into a Teradata Database table without requiring you to write a Teradata PT script.
Chapter 5: Moving External Data into Teradata Database
Using Teradata PT EasyLoader

Required Tasks

The tasks in this section are required when using Teradata PT EasyLoader to load data from an external flat file. Execute Tasks 1, 2, and 4 completely and in the order presented. Tasks 3 and 5 are optional depending on job conditions and outcome.

Prerequisites

The following applies:

- The target table exists in the Teradata Database
- The flat file must be a text file that contains character data in delimited format. See “Sample Flat File” below.
- The layout of the data records in the flat file must match that of the target table.
- Unicode is not supported in the command line or in the job variables file.

The tdload Command

The Teradata PT EasyLoader command, `tdload`, has the following syntax:

`tdload jobOptions jobname`

where `jobOptions` are:

<table>
<thead>
<tr>
<th>Option</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-f filename</code></td>
<td>Required. The name of the flat file containing the data to be loaded. If you are not executing the tdload command in the directory where the flat file is stored, <code>filename</code> must be a fully qualified flat file name.</td>
</tr>
<tr>
<td><code>-t tablename</code></td>
<td>Required. The name of the target table. <strong>Note:</strong> If the target table resides in a database that is different from the default database of the specified Teradata user, you must also use the <code>--TargetWorkingDatabase</code> option.</td>
</tr>
<tr>
<td><code>-u username</code></td>
<td>Required. The logon id of the Teradata user with access privileges to the target table.</td>
</tr>
<tr>
<td><code>-p password</code></td>
<td>The password of the specified Teradata user. <strong>Note:</strong> If the option is not specified, tdload will prompt you for a password.</td>
</tr>
<tr>
<td><code>-h tdpId</code></td>
<td>The name by which the Teradata Data Warehouse Appliance 255x is known to the network. <strong>Recommendation:</strong> Specify this option to make sure you are connecting to the correct system.</td>
</tr>
</tbody>
</table>
### Option 
<table>
<thead>
<tr>
<th>Specifies</th>
</tr>
</thead>
</table>
| -d "delimiterCharacter" | The delimiter character used to separate the fields of the data records in a delimited format flat file.  
**Note:** The default delimiter character is a comma (","). This is different from the usage in a Teradata PT script where the default delimiter character is the pipe character ("|""). You must specify this option if the delimiter character in your flat file is not a comma. |
| -j jobVariableFile | The name of the job variables file. See “Define Job Variables File” below.  
If you are not executing the `tdload` command in the directory where the job variables file is stored, `jobVariableFile` must be a fully qualified filename. |
| -c characterSet | The character set encoding of the flat file. This option sets the client session character set.  
Specify this option if the flat file is not an ASCII file. |

where `jobname` is:

### Option 
<table>
<thead>
<tr>
<th>Specifies</th>
</tr>
</thead>
</table>
| `jobname` | A unique name that identifies the load job.  
**Recommendation:** Teradata strongly recommends that you specify names for your jobs when multiple load jobs are running simultaneously. |

You can also specify the above options using long option names with the ("--") syntax as follows:

- `--SourceFileName filename`
- `--SourceTable tablename`
- `--TargetUserName username`
- `--TargetUserPassword password`
- `--TargetTdpId tpid`
- `--SourceTextDelimiter "delimiterCharacter"`

In addition, the following options are also available:

### Option 
<table>
<thead>
<tr>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>--help</td>
</tr>
</tbody>
</table>
| --TargetWorkingDatabase `databaseName` | The database where the target table is located.  
**Note:** This option is required if the target table resides in a database that is different from the default database of the specified Teradata user. |
Chapter 5: Moving External Data into Teradata Database
Using Teradata PT EasyLoader

The following options are used to optimize job performance:

<table>
<thead>
<tr>
<th>Option</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>--TargetAccountld</td>
<td>The account associated with the specified Teradata user.</td>
</tr>
<tr>
<td>--TargetMaxSessions</td>
<td>The maximum sessions to be used in the load job. The default value is 32.</td>
</tr>
<tr>
<td>--TargetMinSessions</td>
<td>The minimum sessions to be used in the load job. The default value is 1.</td>
</tr>
<tr>
<td>--LoadInstances</td>
<td>The number of instances used to load data. The default value is 1.</td>
</tr>
<tr>
<td>--FileInstances</td>
<td>The number of instances used to read the data file. The default value is 1.</td>
</tr>
<tr>
<td>--TargetErrorLimit</td>
<td>The maximum errors allowed in data records. When the number of errors encountered exceeds this number, the load job terminates. The default value is 1.</td>
</tr>
</tbody>
</table>

Reference Information

<table>
<thead>
<tr>
<th>Information on...</th>
<th>Is available in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>delimited format files</td>
<td>“Key Specifications” on page 222</td>
</tr>
<tr>
<td>determining the optimum number of</td>
<td>“Optimizing Job Performance with Sessions and Instances” on page 98</td>
</tr>
<tr>
<td>sessions and instances</td>
<td></td>
</tr>
<tr>
<td>error limits</td>
<td>“Effects of Error Limits” on page 344</td>
</tr>
</tbody>
</table>

Task 1: Define a Job Variables File

Instead of entering the above options on the command line, you may specify any of them in a job variables file.

Use a text editor to create a local job variables file that contains a list of variables and their corresponding values.

**Note:** The following procedure assumes you have not yet created a job variables file. If you have, you can simply add job variables to it. A job variables file can contain job variables that multiple Teradata PT scripts use.

Execute the following procedure from a Teradata client configured with Teradata PT.
Chapter 5: Moving External Data into Teradata Database
Using Teradata PT EasyLoader

Procedure
1 Use a text editor to create a job variables file that contains a list of options and their corresponding values to be used with Teradata PT EasyLoader. Each job variable must be defined on a single line separated by commas, using the following format:

\[ \text{option} = \text{value} \]

where:

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>option</td>
<td>A single-letter or a multi-letter (long) option. Note: The option name is case-sensitive.</td>
</tr>
<tr>
<td>value</td>
<td>An integer or a character string. You must enclose character strings within single quotes.</td>
</tr>
</tbody>
</table>

2 Save the job variables file as a .txt file in the same directory as your data file. If you save the job variables file in a different directory, you will need to specify the full file name of the job variables file in the -j option when executing tdload.

Example
The following shows the contents of a sample job variables file for a Teradata PT EasyLoader job.

```
SourceFileName = 'employee_data.txt',
u = 'dbadmin',
TargetUserPassword = 'tdpasswd5',
h = 'tdat1',
TargetWorkingDatabase = 'Tables_Database',
TargetTable = 'Employee',
SourceTextDelimiter = '|',
TargetMaxSessions = 6
```

Task 2: Launch a Teradata PT EasyLoader Job

Procedure
Follow these steps to launch a Teradata PT EasyLoader job.

1 In a Command window, navigate to the directory where your data file is stored.
2 Enter the tdload command at the command prompt. For example:

   \[ \text{tdload} -f \text{filename} -u \text{username} -p \text{password} -h \text{tdpid} -t \text{tablename} \]

3 The screen will display the Job Id for the load job.

Example 1
The following tdload command loads data from the emp_data.txt file into the Tables_Database.Employee table. The name of the load job is EmpLoadJob.

```
tdload -f emp_data.txt -u dbadmin -p pw123 -h tdat1 -t employee -d "|" --
TargetWorkingDatabase Tables_Database EmpLoadJob
```
Example 2
The following example uses the empload_jobvars job variables file that this example assumes has specified all job options associated with the Teradata PT EasyLoader job in Example 1 above. Using the -j option eliminates the need to type the options when executing tdload.

```
toload -j empload_jobvars.txt
```

Sample Flat File
The following shows the contents of a flat file with data in delimited format. The delimiter character is the pipe ("|"). Teradata PT EasyLoader can only load a delimited format flat file.

The schema of this file matches the schema of the Teradata Employee table used in the other Teradata PT job examples.

```
10001|John Smith|93000.00|1954-10-21|Sr. Software Engineer|100|Y|5
10002|Mary Knotts|45000.00|1974-09-13|Secretary|100|Y|1
10005|Keith Muller|85000.00|1972-06-09|Sr. Software Engineer|100|Y|3
10021|David Crane|65000.00|1966-10-02|Technical Writer|101|Y|2
10022|Richard Dublin|60000.00|1965-03-19|Software Engineer|100|N|0
10023|Kelly O'Toole|65000.00|1955-04-08|Software Tester|102|N|2
10024|Brett Jackson|75000.00|1962-04-08|Software Engineer|100|Y|2
10025|Erik Wickman|79000.00|1965-03-08|Software Engineer|100|N|2
```

Task 3: Monitor and Manage a Teradata PT EasyLoader Job

You can monitor and manage a Teradata PT EasyLoader job just as you monitor and manage a Teradata PT job.

Teradata PT provides the capability to monitor and manage a job while it is running.

- The **twbcmd** command allows you to:
  - Pause and resume a job
  - View the status of a job
- The **twbkill** command allows you to stop a job

Procedure 1: Pause and Resume a Job

Once a Teradata PT job launches, you can pause and then resume the job using the twbcmd command.

1. From the command line in the working directory, enter the following command to pause the job:
   ```
twbcmd job_id JOB PAUSE
   where job_id is the job name followed by a dash (“-”) and the job sequence number generated by the system at launch.
   ```

2. When you are ready to continue, enter the following command to resume the job:
   ```
twbcmd job_id JOB RESUME
   The job resumes at the point at which it was paused.
   ```
Procedure 2: View the Job Status

Do the following to check the status of a running job.

1. From the command line in the working directory, enter the following command to check job status:

   \texttt{twbcmd job\_id JOB STATUS}

   where \texttt{job\_id} is the job name followed by a dash ("-") and the job sequence number generated by the system at launch.

Procedure 3: Terminate a Job

If you need to terminate a running job, for example if continuation of the job could either cause system failures or significantly impact overall system performance, you can use the \texttt{twbkill} command to force all executing job tasks to terminate immediately.

1. From the command line in the working directory, enter the following command to terminate a job:

   \texttt{twbkill job\_id}

   where \texttt{job\_id} is the job name followed by a dash ("-") and the job sequence number generated by the system at launch.

2. When the job terminates, check the logs as shown in “Examine the Teradata PT Job Logs” below to make sure you understand the problem that led to the job termination.

Reference Information

<table>
<thead>
<tr>
<th>Information on...</th>
<th>Is available in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>the twbcmd command</td>
<td>“Managing an Active Job” on page 327</td>
</tr>
<tr>
<td></td>
<td>Chapter 2: “Teradata PT Utility Commands” in \textit{Teradata Parallel Transporter Reference}.</td>
</tr>
</tbody>
</table>

Task 4: Evaluate a Completed Teradata PT EasyLoader Job

You can evaluate a completed Teradata PT EasyLoader job just as you evaluate a Teradata PT job.

Procedure 1: Examine Exit Codes

Each Teradata PT job returns an exit code upon job completion, which indicates job success or failure.
1 Examine the job exit code, which appears on the screen where you launched the job.

<table>
<thead>
<tr>
<th>Exit Code</th>
<th>Description.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Completed successfully.</td>
</tr>
<tr>
<td>4</td>
<td>Completed successfully, but issued one or more warnings.</td>
</tr>
<tr>
<td>8</td>
<td>Terminated due to a user error, such as a syntax error.</td>
</tr>
<tr>
<td>12</td>
<td>Terminated due to a fatal error.</td>
</tr>
<tr>
<td></td>
<td>A fatal error is any error other than a user error.</td>
</tr>
</tbody>
</table>

2 Determine whether or not further action is required.
   - If the exit code is 0, the job was successful and no further action is required.
   - If the exit code is 4, you can check the logs to examine the warning(s) and determine whether or not you should revise the area of the script that generated the warning to avoid a possible future failure.
   - If the exit code is 8 or 12, revise the script to correct the error.

3 For jobs that return an exit code that requires examination of the job logs, see “Examine the Teradata PT Job Log” immediately below.

**Procedure 2: Examine the Teradata PT Job Logs**

Examine the job logs to understand the details of how the job executed, what warnings were issued, and if the job failed, which errors caused the failure.

<table>
<thead>
<tr>
<th>Types of Log</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Console</td>
<td>The console log displays messages in the Command window where the tbuild command was issued.</td>
</tr>
<tr>
<td></td>
<td>This log contains high-level information about Teradata PT operators and infrastructure. It updates continuously while the job runs.</td>
</tr>
<tr>
<td>Public</td>
<td>The public log contains general information about the job. Use the tlogview command to access this log.</td>
</tr>
<tr>
<td>Private</td>
<td>The private log contains job performance metadata and a log of the activities and errors for each operator defined in the job. Use the tlogview command to access this log.</td>
</tr>
</tbody>
</table>

**Procedure 3: Examine the Teradata PT Error Tables, If Applicable**

Error tables provide information on Teradata Database errors encountered while writing data to the database, as well as detailed information about errors initially presented in the job logs.

If you have set error tables as attributes in your job script, examine the error tables. There are two types of error tables.
Task 5: Troubleshoot a Failed Teradata PT EasyLoader Job, If Necessary

You can troubleshoot a failed Teradata PT EasyLoader job just as you troubleshoot a Teradata PT job.

Reference Information

<table>
<thead>
<tr>
<th>Information on...</th>
<th>Is available in...</th>
</tr>
</thead>
<tbody>
<tr>
<td>detecting and correcting the cause of failure</td>
<td>“Chapter 24 Troubleshooting a Failed Job” on page 351</td>
</tr>
<tr>
<td>common job failures and remedies</td>
<td>“Chapter 24 Troubleshooting a Failed Job” on page 351</td>
</tr>
<tr>
<td>operator-specific error handling</td>
<td>“Chapter 24 Troubleshooting a Failed Job” on page 351</td>
</tr>
</tbody>
</table>
You can use any valid combination of producer and consumer operators, and where necessary access modules, to create a job script to move data into Teradata Database. The following list of topics includes examples of the most common scenarios.

**Note:** There are multiple variations for many of the jobs and a script example for each.

- Job Example 1: High Speed Bulk Loading into an Empty Table
- Job Example 2: Perform INSERT, UPDATE, and DELETE in Multiple Tables
- Job Example 3: Loading BLOB and CLOB Data
- Job Example 4: Pre-processing Data with an INMOD Routine Before Loading
- Job Example 5: Continuous Loading of Transactional Data from JMS or MQ
- Job Example 6: Loading Data from Other Relational Databases
- Job Example 7: Mini-Batch Loading
- Job Example 8: Batch Directory Scan
- Job Example 9: Active Directory Scan

**How to Use Job Examples**

The job examples shown in the following sections correspond to the job script examples provided in Appendix A: “Job Script Examples”. Each job example is comprised of a data flow diagram, a statement of the rationale for why the job uses the operators shown, a list of key operator attributes, and other important factors to consider when customizing the job for your particular use.

The purpose of these job examples is to provide a simple way to find a Teradata PT standard script example that you can customize to accomplish an ETL task instead of writing the entire script yourself. Use the job examples as follows:

1. Find the job example that most closely resembles the task you want to accomplish.
2. Examine the corresponding script example to gain an understanding of the job structure.
3 Assign values to the key attributes shown in the job example. If you are not already familiar with the purpose of each required attribute, read the attribute descriptions and syntax requirements shown in *Teradata Parallel Transporter Reference*. Refer to the chapters on operators, beginning with *Chapter 8: “DataConnector Operator”* for strategies on setting attribute values.

**Note:** Tdpid, UserName, and UserPassword are key attributes for all operators that connect to a Teradata Database, which must be configured according to individual site requirements. For detailed information, see “Teradata Database Logon Security” on page 91. Wherever possible use global job variables for these values.

4 Check the list of Important Considerations for other strategic hints and reminders.

5 Customize the example script for your needs.

6 Refer to the section beginning with *Chapter 21: “Launching a Job”* for information on launching and managing Teradata PT jobs.

**Job Example 1: High Speed Bulk Loading into an Empty Table**

**Job Objective:**
Read large amounts of data directly from an external flat file, or from an access module, and write it to an empty Teradata Database table. If the source data is an external flat file, this job is equivalent to using the Teradata FastLoad utility. If the data source is a named pipe, the job is equivalent to using the Teradata standalone DataConnector utility to access data from a named pipe, through an access module, and write it to a temporary flat file, and then running a separate FastLoad job to load the data from the temporary file.

**Note:** In cases where data is read from more than one source file, use UNION ALL to combine the data before loading into a Teradata Database table, as shown in Job Example 1C.

**Data Flow Diagrams**
*Figure 21 through Figure 23* show flow diagrams of the elements in each of the three variations of Job Example 1.

Figure 21: Job Example 1A -- Reading Data from a Flat File for High Speed Loading
Chapter 5: Moving External Data into Teradata Database
Common Jobs for Moving Data into a Teradata Database

Figure 22: Job Example 1B -- Reading Data from a Named Pipe for High Speed Loading

Figure 23: Job Example 1C -- Reading Data from Multiple Flat Files for High Speed Loading

Related Script Examples
For the script examples that correspond to the three variations of this job, see the following:

- “Script Example 1A: High Speed Bulk Loading from Flat Files into an Empty Teradata Database Table” on page 404
- “Script Example 1B: High Speed Bulk Loading from a Named Pipe into an Empty Teradata Database Table” on page 406
- “Script Example 1C: High Speed Bulk Loading from Two Flat Files into an Empty Teradata Database Table Using UNION ALL” on page 408

Rationale
This job uses:

- DDL operator because it can DROP/CREATE tables needed for the job prior to loading and DROP unneeded tables at the conclusion of the job.
- DataConnector operator because it is the only producer operator that reads data from external flat files and from the Named Pipes access module.
- Load operator because it is the consumer operator that offers the best performance for high speed writing of a large number of rows into an empty Teradata Database table.

**Usage Notes**

Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job. For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide.

**DDL Operator**

- Key attributes
  - TdpId
  - UserName
  - UserPassword
  - ErrorList

**DataConnector Operator**

- Key Attributes (reading data directly from flat file)
  - DirectoryPath
  - FileName
  - Format
  - TextDelimiter (required if the format is “delimited”)
  - IndicatorMode (default = No)
- Key Attributes (reading data through an access module)
  - AccessModuleName
  - AccessModuleInitStr
  - FileName
  - Format
  - OpenMode
  - TextDelimiter (required if the format is “delimited”)
  - IndicatorMode (default = No)
- Important Considerations
  - Specify TYPE DATACONNECTOR PRODUCER in the DataConnector operator definition to define that the DataConnector is reading data.

**Load Operator**

- Key Attributes
  - TdpId
  - UserName
  - UserPassword
  - LogTable
  - TargetTable
Chapter 5: Moving External Data into Teradata Database

Common Jobs for Moving Data into a Teradata Database

- ErrorTable1
- ErrorTable2
- Important Considerations
  - The Load operator can load only a single, empty target table.
  - The target is identified using the TargetTable attribute.

Job Example 2: Perform INSERT, UPDATE, and DELETE in Multiple Tables

Job Objective:
Read data directly from non-Teradata source files, or from an access module, and perform INSERT, DELETE, and UPDATE operations on multiple Teradata Database tables. The loading part of this job is equivalent to the most common use of the Teradata MultiLoad utility.

Data Flow Diagram
Figure 24 and Figure 25 show diagrams of the job elements for the two variations of Job Example 2.

Figure 24: Job Example 2A -- Reading Data from a Flat File

Figure 25: Job Example 2B -- Reading Data from a Named Pipe

Related Script Example
For the script examples that correspond to this job, see “Script Example 2A: Read Data Direct from Source Files and Perform INSERT, UPDATE, and DELETE on Multiple Teradata Database Tables” on page 474 and “Script Example 2B: Read Data from a Named Pipe and Perform INSERT, UPDATE, and DELETE on Multiple Teradata Database Tables” on page 477.
Rationale
This job uses:

- DDL operator because it can DROP/CREATE target tables and DROP work tables.
- DataConnector operator because it is the only producer operator that reads data from non-Teradata, non-ODBC data sources and from Named Pipes.
- Update operator as the consumer operator because it can perform INSERT, UPDATE, and DELETE operations into either new or pre-existing Teradata Database tables.

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job. For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide

DDL Operator
- Key attributes
  - TdpId
  - UserName
  - UserPassword
  - ErrorList

DataConnector Operator
- Key Attributes (reading direct from flat files)
  - DirectoryPath
  - FileName
  - Format
  - TextDelimiter (required if the format is “delimited”)
  - IndicatorMode (default = No)
- Key Attributes (reading from an access module)
  - AccessModuleName
  - AccessModuleInitStr
  - FileName
  - Format
  - TextDelimiter (required if the format is “delimited”)
  - IndicatorMode (default = No)
- Important Considerations
  - Specify TYPE DATACONNECTOR PRODUCER in the DataConnector operator definition to specify that the DataConnector is reading data.

Update Operator
- Key Attributes
  - TdpId
Chapter 5: Moving External Data into Teradata Database

Common Jobs for Moving Data into a Teradata Database

- UserName
- UserPassword
- LogTable
- TargetTable
- WorkTable
- ErrorTable1
- ErrorTable2

- Important Consideration:
  - The Update operator can load either an empty table or one with pre-existing data.
  - The APPLY statement that invokes the Update operator contains the SQL that defines the operation to be carried out (e.g. INSERT) and it identifies the Teradata Database table on which the operation will be performed.

**Job Example 3: Loading BLOB and CLOB Data**

**Job Objective:**
Extract inline BLOB/CLOB data from files and load it into one or more Teradata Database tables.

For detailed information on inline LOB processing, see “Large Objects” on page 426.

**Data Flow Diagram**
Figure 26 shows a diagram of the job elements for Job Example 3.

Figure 26: Job Example 3 -- Loading BLOB and CLOB Data

**Related Script Example**
For the script examples that correspond to this job, see “Script Example 3: Loading BLOB and CLOB Data into Teradata Database” on page 480.

**Rationale**
This job uses:
Common Jobs for Moving Data into a Teradata Database

• OS Command operator because it is the only operator that can copy a flat file from one directory to another on the client system.
• DDL operator because it can DROP work tables and CREATE target tables prior to loading
• DataConnector operator because it is the only producer operator that reads inline LOB data from external flat files.
• SQL Inserter operator as the consumer operator because it is the only operator that can load BLOB/CLOB data into Teradata Database tables.

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job.

For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide.

OS Command Operator
• Key Attributes
  • OsCmd
  • IgnoreError
• Important Considerations
  • The OS Command operator is not concerned with data structure, and therefore does not require specification of a schema.
  • The OS Command operator executes operating system commands on the client system that runs the job.

DDL Operator
• Key attributes
  • TdpId
  • UserName
  • UserPassword
  • ErrorList

DataConnector Operator
• Key Attributes
  • DirectoryPath
  • FileName
  • OpenMode
  • Format
  • TextDelimiter (required if the format is “delimited”)
  • IndicatorMode (default = No)
• Important Considerations
  • Specify TYPE DATACONNECTOR PRODUCER in the DataConnector operator definition to define that the DataConnector is reading data.
**SQL Inserter Operator**

- Key Attributes
  - TdpId
  - UserName
  - UserPassword

- Important Considerations
  - The SQL Inserter operator can load a target table that is either empty or that contains pre-existing data.
  - The APPLY statement that invokes the SQL Inserter operator contains the SQL that defines the operation to be carried out (INSERT) and it identifies the Teradata Database table into which the data will be inserted.

**Job Example 4: Pre-processing Data with an INMOD Routine Before Loading**

**Job Objective:**
Read data from external source files and pre-process it with an INMOD routine before loading it into Teradata Database tables. There are two variations of this job:

- Bulk loading of the data using the Load operator (FastLoad protocol).
- Using the data to perform INSERT, UPDATE, and DELETE operations using the Update operator (Multiload protocol).

**Data Flow Diagram**

Figure 27 shows a diagram of the job elements for Job Example 4.

**Related Script Example**
For script examples that support this job, see “Script Example 4A: Pre-process Data with an INMOD Routine Before Loading into an Empty Teradata Database Table (FastLoad Protocol)” on page 483 and “Script Example 4B: Pre-process Data with an INMOD Routine Before Loading into Teradata Database Tables (Multiload Protocol)” on page 485.
Rationale
The job uses:

- The producer for this job can be either:
  - FastLoad INMOD Adapter operator because it is the only operator that can read data from an INMOD routine that was written for the FastLoad protocol.
  - MultiLoad INMOD Adapter operator because it is the only operator that can read data from an INMOD routine that was written for the MultiLoad protocol
- The consumer operator for this job can be either:
  - The Load operator because it offers the best performance for high speed inserting of a large number of rows into empty Teradata Database tables (FastLoad protocol).
  - The Update operator because it can perform INSERT, UPDATE, and DELETE operations on up to 5 new or pre-existing Teradata Database tables (MultiLoad protocol).

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job. For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide.

FastLoad or MultiLoad INMOD Adapter Operator
- Key Attributes
  - InmodName
  - FileName
  - Format
  - OpenMode

Load Operator
- Key Attributes
  - TdpId
  - UserName
  - UserPassword
  - LogTable
  - TargetTable
  - ErrorTable1
  - ErrorTable2
- Important Considerations
  - The Load operator can load only a single, empty target table.
  - The target is identified using the TargetTable attribute.

Update Operator
- Key Attributes
Chapter 5: Moving External Data into Teradata Database

Common Jobs for Moving Data into a Teradata Database

- TdpId
- UserName
- UserPassword
- LogTable
- TargetTable
- WorkTable
- ErrorTable1
- ErrorTable2

Important Considerations
- The Update operator can perform DML operations on up to 5 tables, any of which can be empty or contain data.
- The APPLY statement that invokes the Update operator contains the SQL that defines the operation to be carried out and identifies the Teradata Database table in which the data will be updated.

Job Example 5: Continuous Loading of Transactional Data from JMS or MQ

Job Objective:
Read transactional data from MQ or JMS and perform continuous INSERT, UPDATE, and DELETE operations on one or more Teradata Database tables and optionally load an external flat file with the same data, using the Teradata PT duplicate data stream feature. In this job the Stream operation functions like the Teradata standalone utility TPump.

Data Flow Diagram
Figure 28 through Figure 30 show diagrams of the job elements for variations of Job Example 5.

Figure 28: Job Example 5A -- Read Transactional Data from JMS and Load Using the Stream Operator
Related Script Example

For script examples that correspond to this job, see:

- “Script Example 5A: Continuous Loading of Transactional Data from MQ” on page 487.
- “Script Example 5B: Continuous Loading of Transactional Data from JMS” on page 489.
- “Script example 5C: Intermediate File Logging Using Multiple APPLY Clauses with Continuous Loading of Transactional Data” on page 491.

Rationale

This job uses:

- DataConnector operator as both the producer operator and one of the two consumer operators because:
  - It is the only producer operator that can read data from an access module.
  - It is the only consumer operator that can write data to an external file.
- Stream operator because it can perform INSERT, UPDATE, and DELETE operations on up to 127 new or pre-existing Teradata Database tables, while queries are performed on the tables.
- Multiple APPLY statements to apply data from the producer operator to two different consumer operators, loading data into both a Teradata Database and an external flat file.
Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job. For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide.

DataConnector Operator
- Key Attributes (when reading data from an access module)
  - AccessModuleName
  - AccessModuleInitStr
  - FileName
  - Format
  - TextDelimiter (required if the format is “delimited”)
  - IndicatorMode (default = No)
- Key Attributes (when writing data to an external flat file)
  - DirectoryPath
  - FileName
  - Format
  - TextDelimiter (required if the format is “delimited”)
  - IndicatorMode (default = No)
- Important Considerations
  - The variations of Job Example 5 use DataConnector as both a producer and consumer operator. Job Example 5C requires two DataConnector operator definitions, because it uses both variations. Be sure to specify TYPE DATACONNECTOR PRODUCER or TYPE DATACONNECTOR CONSUMER in the DataConnector operator definition to define whether the DataConnector is reading or writing data.

Stream Operator
- Key Attributes
  - TdpId
  - UserName
  - UserPassword
  - LogTable
  - ErrorTable
- Important Considerations
  - The Stream operator can update up to 127 tables.
  - The APPLY statement that invokes the Stream operator contains the SQL that defines the operations to be carried out (e.g. INSERT) and it identifies the Teradata Database table(s) that will be updated.
Job Example 6: Loading Data from Other Relational Databases

Job Objective:
Read data from an ODBC-compliant relational databases such as Oracle, SQL Server, DB2, etc, and write it to Teradata Database tables.

Data Flow Diagram
Figure 31 shows a diagram of the job elements for Job Example 6.

Related Script Example
For script examples that correspond to this job, see “Script Example 6: Loading Data from Other Relational Databases into an Empty Teradata Database Table” on page 493.

Rationale
The job uses:
• ODBC operator because it is the only operator that can read data from ODBC-compliant external databases.
• Load operator because it offers the best performance for high speed writing of a large number of rows into empty Teradata Database tables.

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job. For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide.

ODBC Operator
• Key Attributes
  • DSNName
  • UserName
  • UserPassword
  • SelectStmt
• Important Considerations:
  • Before using the ODBC operator, you must designate the data source name (DSN) in ODBC Data Sources to point to the proper data source.
• Consider the differences in the rules for handling decimal data between Teradata and non-Teradata Databases.
• Consider the differences in the rules for handling DATE fields between Teradata and non-Teradata databases.
• Even though ODBC operator can communicate directly with the Teradata Database, it is more efficient to use the Load operator to write data to Teradata Database.
• The ODBC operator does not support multiple instances.

**Load Operator**

• Key Attributes:
  • TdpId
  • UserName
  • UserPassword
  • LogTable
  • TargetTable
  • ErrorTable1
  • ErrorTable2

• Important Considerations:
  • The Load operator can load only a single table. That table is identified using the TargetTable attribute.
  • Ensure that the target table is empty before loading data.

**Job Example 7: Mini-Batch Loading**

**Job Objective:**
Read data directly from one or more external flat files and write it to a Teradata Database table.

**Note:** This job represents a special case of high speed loading, where the destination table is already populated, or has join indexes or other restrictions that prevent it from being accessed by the Load operator. Because of this, the job includes an intermediate step that loads the data into a staging table and then uses the DDL operator with INSERT...SELECT to move the data into the final destination table.

**Data Flow Diagrams**
Figure 32 shows a flow diagram of the elements of Job Example 7.
Related Script Example

The script example correspond to this job: “Script Example 7: Mini-Batch Loading into Teradata Database Tables” on page 496.

Rationale

This job uses:

- DDL operator because it can DROP/CREATE staging tables and target tables prior to loading, DROP unneeded tables at the conclusion of the job, and load the production table from the staging table using INSERT…SELECT.
- DataConnector operator because it is the only producer operator that reads data from external flat files and from the Named Pipes access module.
- Load operator because it is the consumer operator that offers the best performance for high speed writing of a large number of rows into an empty Teradata Database table.

Usage Notes

Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job. For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide.

**DDL Operator**

- Key attributes
  - TdpId
  - UserName
  - UserPassword
  - ErrorList
**DataConnector Operator**

- Key Attributes (reading data directly from flat file)
  - DirectoryPath
  - FileName
  - Format
  - TextDelimiter (required if the format is “delimited”)
  - IndicatorMode (default = No)
- Important Considerations
  - Specify TYPE DATACONNECTOR PRODUCER in the DataConnector operator definition to define that the DataConnector is reading data.

**Load Operator**

- Key Attributes
  - TdpId
  - UserName
  - UserPassword
  - LogTable
  - TargetTable
  - ErrorTable1
  - ErrorTable2
- Important Considerations
  - The Load operator can load only a single, empty target table.
  - The target is identified using the TargetTable attribute.

**Job Example 8: Batch Directory Scan**

**Job Objective:**
Scan a directory for one or more flat files and then do high speed loading of the data into a Teradata Database table.

**Note:** If the Teradata Database table is populated, use the Update operator for the load operation.

For strategies on using this method, see “Batch Directory Scan” on page 435 and Chapter 8: “DataConnector Operator.”

**Data Flow Diagrams**
Figure 33 shows a flow diagram of the elements in Job Example 8.
Chapter 5: Moving External Data into Teradata Database
Common Jobs for Moving Data into a Teradata Database

Figure 33: Job Example 8 -- Batch Directory Scan

Related Script Example
The script that corresponds to this job: “Script Example 8: Batch Directory Scan” on page 500.

Rationale
This job uses:

- DDL operator because it can DROP/CREATE staging tables and target tables prior to loading and DROP unneeded tables at the conclusion of the job.
- DataConnector operator because it is the only producer operator that reads data from external flat files.
- Load operator because it is the consumer operator that offers the best performance for high speed writing of a large number of rows into an empty Teradata Database table.

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job. For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide.

**DDL Operator**
- Key attributes
  - TdpId
  - UserName
  - UserPassword
  - ErrorList

**DataConnector Operator**
- Key Attributes (reading data directly from flat file)
  - DirectoryPath
  - FileName
  - Format
Chapter 5: Moving External Data into Teradata Database
Common Jobs for Moving Data into a Teradata Database

- TextDelimiter (required if the format is “delimited”)
- IndicatorMode (default = No)

Important Considerations
- Specify TYPE DATACONNECTOR PRODUCER in the DataConnector operator definition to define that the DataConnector is reading data.
- Specify “wildcard” file names to select the files to be loaded.

**Load Operator**
- Key Attributes
  - TdpId
  - UserName
  - UserPassword
  - LogTable
  - TargetTable
  - ErrorTable1
  - ErrorTable2

Important Considerations
- The Load operator can load only a single, empty target table.
- The target is identified using the TargetTable attribute.

**Job Example 9: Active Directory Scan**

**Job Objective:**
Periodically scan for transactional data files that continuously appear in two directories. Read the data from each new file and use it to perform updates on Teradata Database table(s).

For strategies on how to set up this job, see “Active Directory Scan: Continuous Loading of Transactional Data” on page 436 and Chapter 8: “DataConnector Operator.”

**Data Flow Diagrams**
Figure 34 shows a flow diagram of the elements in Job Example 9.
Related Script Example
The script that corresponds to this job: “Script Example 9: Active Directory Scan” on page 502.

Rationale
This job uses:

- DataConnector operator because it is the only producer operator that can scan a directory periodically for new files and extract data from only the files that are new since the previous scan.
- Stream operator because it is the only operator that can perform continuous updates of Teradata Database tables.

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job. For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide.

DDL Operator
- Key attributes
  - TdpId
  - UserName
  - UserPassword
  - ErrorList
**DataConnector Operator**

- **Key Attributes**
  - ArchiveDirectoryPath
  - DirectoryPath
  - EnableScan (must = yes)
  - FileName
  - Format
  - IndicatorMode (default = No)
  - TextDelimiter (required if the format is “delimited”)
  - VigilStartTime
  - VigilStopTime
  - VigilWaitTime
  - VigilSortFile
  - VigilElapsedTime

- **Important Considerations**
  - Specify TYPE DATACONNECTOR PRODUCER in the DataConnector operator definition to define that the DataConnector is reading data.
  - Choose values for the various Vigil attributes based on the arrival rates of new files in the directories to be scanned and on the amount of data in the files. Some experimentation will likely required to arrive at optimal settings. For information, see Chapter 8: “DataConnector Operator.”
  - Specify “wildcard” file names to select the files to be loaded.

**Stream Operator**

- **Key Attributes**
  - TdpId
  - UserName
  - UserPassword
  - LogTable
  - ErrorTable

- **Important Considerations**
  - The target is identified in the APPLY statement.
This chapter describes several alternative methods for using Teradata PT to move data from a Teradata Database into a non-Teradata target. It includes the following topics:

- Data Flow Description
- Comparing Applicable Operators
- Using Access Modules to Process Data Before Writing to External Targets
- Common Data Movement Jobs

### Data Flow Description

Teradata PT offers several paths for moving data from a Teradata Database into a non-Teradata target, as shown in the following composite diagram.

**Figure 35: Moving Data from a Teradata Database into a Non-Teradata Target**

Note that many of the blocks in Figure 35 allows you to choose among several operators and access modules. Read the following sections to understand how to make the best choices for specific data movement jobs.
Chapter 6: Moving Data from Teradata Database to an External Target
Comparing Applicable Operators

Comparing Applicable Operators

Once you identify the requirements for moving data from Teradata Database to an external data source, you must select the components that the script will use to execute the job. There are three types of components you need to consider:

- A producer operator that reads data from a Teradata Database and places it in the data stream.
  
- A consumer operator that takes data from the data stream and writes it to the data target.
  
- A consumer operator that uses an OUTMOD routine or access module to post-process the data before loading the data target.

Producer Operators

The Teradata PT producer operators in this section read data from a Teradata Database and write it to the data stream.

The Teradata PT job script invokes a producer operator, which employs the user-specified SQL SELECT statement to access Teradata Database tables. For further information on using APPLY/SELECT to specify a producer operator, see “Coding the APPLY Statement” on page 60 and the section on APPLY in Teradata Parallel Transporter Reference.

The following table briefly describes and compares the function of each Teradata PT operator that can be used as a producer when extracting data from a Teradata Database:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Export Operator</td>
<td>Extracts large volumes of data from a Teradata Database at high speed. Function is similar to the standalone Teradata FastExport utility.</td>
</tr>
</tbody>
</table>

**Features:**
- Allows use of multiple parallel instances.
- 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 sent to the client in sorted order.

**Limitations:**
- Cannot be used to retrieve data in TEXT mode and write it to target files in the TEXT or VARTEXT (delimited) format. Use SQL Selector for this where possible.
- A sorted answer set requires that only a single instance of the Export operator can be used. Specifying ORDER BY in the SELECT statement and multiple Export operator instances results in an error.

For detailed usage information, see Chapter 10: “Export Operator.”
Consumer Operators

The Teradata PT consumer operators in this section read data from the data stream and write it to an external target.

The Teradata PT job script invokes a consumer operator using an APPLY statement. For further information on using SELECT to specify a producer operator, see “Coding the APPLY Statement” on page 60 and the section on APPLY in Teradata Parallel Transporter Reference.

The following table briefly describes and compares the function of each Teradata PT operator that can be used as a consumer when moving data from Teradata Database to an external data target:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL Selector Operator</td>
<td>Submits a single SQL SELECT statement to the Teradata Database to retrieve data from a table.</td>
</tr>
<tr>
<td>Features:</td>
<td>• Use to retrieve data in TEXT mode and write it to target files in the TEXT or VARTEXT (delimited) format.</td>
</tr>
<tr>
<td></td>
<td>• Can retrieve LOB data from the Teradata Database.</td>
</tr>
<tr>
<td>Limitations:</td>
<td>• Much slower than Export operator.</td>
</tr>
<tr>
<td>For detailed usage information,</td>
<td>see Chapter 20: “Update Operator.”</td>
</tr>
<tr>
<td></td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Operators that Write Data to a non-Teradata Target</td>
<td></td>
</tr>
<tr>
<td>DataConnector Operator</td>
<td>Writes data to flat files and functions similarly to the DataConnector standalone utility.</td>
</tr>
<tr>
<td>Features:</td>
<td>• Can write directly to an external file or through an access module.</td>
</tr>
<tr>
<td>For usage information,</td>
<td>see Chapter 8: “DataConnector Operator.”</td>
</tr>
<tr>
<td></td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Operators that Pre-process Data before Writing to a non-Teradata Target</td>
<td></td>
</tr>
<tr>
<td>FastExport OUTMOD Adapter Operator</td>
<td>Uses a FastExport OUTMOD routine to pre-process data before writing it to the data target.</td>
</tr>
<tr>
<td>For detailed usage information,</td>
<td>see Chapter 11: “FastExport OUTMOD Adapter Operator.”</td>
</tr>
<tr>
<td></td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>

For detailed information, see the specific operator you want to use in Section 4: Operators.
Using Access Modules to Process Data Before Writing to External Targets

Access modules are dynamically attached software components of the Teradata standalone load and unload utilities. Some access modules are usable with Teradata PT job scripts, and provide the input/output interface between operators and various types of external data storage devices. Any operator that uses access modules can interface with all available access modules.

The following access modules can be used as part of a job to move data from Teradata Database to an external data target.

<table>
<thead>
<tr>
<th>Access Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLE DB</td>
<td>Provides write access to a flat file or a table in an OLE DB-compliant DBMS, such as SQL Server, Oracle or Connix.</td>
</tr>
</tbody>
</table>

Specifying an Access Module

Use the AccessModuleName attribute in the DataConnector (consumer) operator to specify the optional use of an access module to interface with the target database. The DataConnector operator definition must also specify a value for the AccessModuleInitStr attribute, to define the access module initialization string.

For detailed information on requirements for using access modules with Teradata PT, see Teradata Tools and Utilities Access Module Reference.

For information on using access modules with z/OS, see “Using Access Modules to Read Data from an External Data Source” on page 96.
Common Data Movement Jobs

You can use any valid combination of producer and consumer operators, and where necessary access modules, to create a job script for your data movement needs. However, the following list includes examples of some of the most common job scenarios. Evaluate the examples and if possible use one of the associated sample job scripts before creating your own.

- Job Example 10: Extracting Rows and Sending Them in Delimited Format
- Job Example 11: Extracting Rows and Sending Them in Indicator-mode Format
- Job Example 12: Export Data and Process It with an OUTMOD Routine
- Job Example 13: Export Data and Process It with an Access Module
- Job Example 14: Extract BLOB/CLOB Data and Write It to an External File

How to Use Job Examples

The job examples shown in the following sections correspond to the job script examples provided in Appendix A: “Job Script Examples”. Each job example is comprised of a data flow diagram, a statement of the rationale for why the job uses the operators shown, a list of key operator attributes, and other important factors to consider when customizing the job for your particular use.

The purpose of these job examples is to provide a simple way to find a Teradata PT standard script example that you can customize to accomplish an ETL task instead of writing the entire script yourself. Use the job examples as follows:

1. Find the job example that most closely resembles the task you want to accomplish.
2. Examine the corresponding job script example to gain an understanding of the job structure.
3. Check the list of key attributes shown in the job example. These are the attributes for which you must specify values based on your unique job. If you are not already familiar with the meaning of these required attributes, read the attribute descriptions and syntax requirements shown in Teradata Parallel Transporter Reference. Refer to Chapters 8 through 20 in this User’s Guide for operator-specific strategies on setting attribute values and other hints for customizing job scripts.

   **Note:** Tdpid, UserName, and UserPassword are key attributes for all operators that connect to Teradata Database. They must be configured according to individual site security requirements. For detailed information, see “Teradata Database Logon Security” on page 73.

4. Check the list of Important Considerations for other strategic hints and reminders.
5. Customize the example script for your needs.
6. Refer to the section beginning with Chapter 21: “Launching a Job” for information on launching and managing the job.
Job Example 10: Extracting Rows and Sending Them in Delimited Format

Job Objective:
Extract rows from Teradata Database tables and write them to an external target file as delimited data.

Data Flow Diagram
Figure 36 shows a diagram of the job elements for Job Example 10.

Figure 36: Job Example 10 -- Extracting Rows and Sending Them in Delimited Format

Script Example
For the script examples that correspond to this job, see “Script Example 10: Extracting Rows and Writing Them in Delimited Format” on page 436.

Rationale
This job uses:

- SQL Selector because it the only operator that can read data from a Teradata Database in field mode (character format).
- DataConnector operator because it is the only operator that can write character data to an external flat file in delimited format.

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job.

For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide.

SQL Selector Operator
- Key Attributes:
  - TdpId
  - UserName
  - UserPassword
  - SelectStmt
  - ReportModeOn
• Important Considerations:
  • The SQL Selector operator allows only one session and one instance to read data from a Teradata Database.
  • ReportModeOn must be set to Yes, because sending delimited data requires that the data be in character format.

**DataConnector Operator**

• Key Attributes
  • DirectoryPath
  • FileList
  • FileName
  • OpenMode
  • Format
  • TextDelimiter

**Note:** FileList is an important feature for the consumer Data Connector operator to write to multiple files in parallel.

• Important Considerations
  • Specify TYPE DATACONNECTOR CONSUMER in the DataConnector operator definition to define that the DataConnector is writing data.

**Job Example 11: Extracting Rows and Sending Them in Indicator-mode Format**

**Job Objective:**

Extract rows from Teradata Database tables using Export operator and write them to an external target as indicator-mode data.

**Data Flow Diagram**

Figure 37 shows a diagram of the job elements for Job Example 11.

Figure 37: Job Example 11 -- Extracting Rows and Sending Them in Binary or Indicator-mode Format

**Script Example**

For the script examples that correspond to this job, see “Script Example 11: Export Rows and Write Them as Binary or Indicator Mode Data” on page 437.
Chapter 6: Moving Data from Teradata Database to an External Target
Common Data Movement Jobs

**Rationale**
This job uses the operators shown for the following reasons:

- Use Export operator because it can extract large amounts of data from a Teradata Database table at high speeds.
- DataConnector operator because it can write data to an external flat file.

**Usage Notes**
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job.

For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 5 of this Users Guide.

**Export Operator**
- Key Attributes
  - TdpId
  - UserName
  - UserPassword
  - SelectStmt

**DataConnector Operator**
- Key Attributes
  - DirectoryPath
  - FileName
  - OpenMode
  - Format
  - IndicatorMode
- Important Considerations
  - Specify TYPE DATACONNECTOR CONSUMER in the DataConnector operator definition to define that the DataConnector is writing data.
  - The IndicatorMode attribute must be set to Yes.
Job Example 12: Export Data and Process It with an OUTMOD Routine

Job Objective:
Export data from a Teradata Database table and send it to an OUTMOD for post-processing before loading into an external target. This job is applicable to OUTMODs written for the FastExport utility.

Data Flow Diagram
Figure 38 shows a diagram of the job elements for Job Example 12.

Figure 38: Job Example 12 -- Export Data and Process It with an OUTMOD Routine

Script Example
For the script examples that correspond to this job, see “Script Example 12: Export Data and Process It with an OUTMOD Routine” on page 438.

Rationale
The job uses:

• Export operator because it is the fastest way to extract large amounts of data from a Teradata Database.

  Note: The SQL operator extracts data more slowly than the Export operator. Use the SQL Selector operator only if the Teradata Database is short on load tasks, because SQL Selector operator instances are not counted as load tasks.

• FastExport OUTMOD Adapter because it is the only operator that can interface with an OUTMOD routine written for the FastExport utility.

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job. For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 5 of this Users Guide.

Export Operator
• Key Attributes
  • TdpId
  • UserName
  • UserPassword
Chapter 6: Moving Data from Teradata Database to an External Target
Common Data Movement Jobs

- SelectStmt

FastExport OUTMOD Adapter Operator
- Key Attributes
  - OutmodName
  - OpenMode
  - Format
  - FileName
  - DirectoryPath
- Important Considerations
  - The FastExport OUTMOD Adapter operator allows use of only a single instance.

Job Example 13: Export Data and Process It with an Access Module

Job Objective:
Export rows from a Teradata Database table and send them to an Access Module for processing before loading the data into an external target.

Data Flow Diagram
Figure 39 shows a diagram of the job elements for Job Example 13.

Figure 39: Job Example 13 -- Export Data and Process It with an Access Module

Script Example
For the script examples that correspond to this job, see “Script Example 13: Export Data and Process It with an Access Module” on page 439.

Rationale
The job uses:
- Export operator because it is the fastest at extracting large amounts of data from a Teradata Database.
  Note: The SQL operator extracts data more slowly than the Export operator. Use the SQL Selector operator only if the Teradata Database is short on load tasks, because SQL Selector operator instances are not counted as load tasks.
- DataConnector operator because it is the only consumer operator that can interface with all Teradata PT-supported access modules.
Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job. For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 5 of this Users Guide.

Export Operator
- Key Attributes
  - TdpId
  - UserName
  - UserPassword
  - SelectStmt

DataConnector Operator
- Key Attributes
  - Access ModuleInitStr
  - AccessModuleName
  - FileName
  - Format
  - IndicatorMode
- Important Considerations
  - Specify TYPE DATACONNECTOR CONSUMER in the DataConnector operator definition to define that the DataConnector is writing data.

Job Example 14: Extract BLOB/CLOB Data and Write It to an External File

Job Objective:
Extract rows that include BLOB/CLOB data from a Teradata Database table and write them to an external flat file.

Data Flow Diagram
Figure 40 shows a diagram of the elements for Job Example 14.

Figure 40: Job Example 14 -- Extract BLOB/CLOB Data and Write It to an External File

Script Example
For the script examples that correspond to this job, see “Script Example 14: Extract BLOB/CLOB Data and Write It to an External Target” on page 440.
**Rationale**

This job uses the operators shown for the following reasons:

- Use SQL Selector operator because it is the only operator that can read BLOB and CLOB data from a Teradata Database and write it to separate external data files. One data file stores data for one LOB column.
- Use DataConnector operator because it is the only operator that can write LOB data to an external file.

**Usage Notes**

Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job.

For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 5 of this Users Guide.

**SQL Selector Operator**

- **Key Attributes**
  - TdpId
  - UserName
  - UserPassword
  - SelectStmt
  - LobDirectoryPath
  - LobFileBaseName
  - LobFileExtension

- **Important Considerations**
  - The SQL Selector operator allows only one session and one instance to read data from a Teradata Database.

**DataConnector Operator**

- **Key Attributes**
  - DirectoryPath
  - FileName
  - OpenMode
  - Format
  - TextDelimiter
  - IndicatorMode

- **Important Considerations**
  - Specify TYPE DATACONNECTOR CONSUMER in the DataConnector operator definition to define that the DataConnector is writing data.

For information on LOBs, see “Large Objects” on page 366.
CHAPTER 7

Moving Data within the Teradata Database Environment

This chapter describes the alternatives for using Teradata PT to move data within the Teradata Database environment, either from one place to another within the same system, or between Teradata Database systems.

The chapter includes the following topics:

- Data Flow Description
- Comparing Applicable Operators
- Common Jobs to Move Data within a Teradata Database

Data Flow Description

Teradata PT offers several methods for moving data within the Teradata Database environment, as shown in the following composite diagram.

Figure 41: Moving Data within the Teradata Database Environment

Note that some of the blocks in Figure 41 allow you to choose from among multiple operators. Read the following sections to understand how to make the best choice for a specific data movement job.
Comparing Applicable Operators

Once you identify the requirements for moving data within the Teradata Database environment, you must select the components that the script will use to execute the job. There are two types of components you need to consider:

- A producer operator that reads data from a Teradata Database and places it in the data stream.
  and
- A consumer operator that takes data from the data stream and writes it to the same, or another, Teradata Database.
  or,
- A standalone operator that acts as both producer and consumer.

Note: All applicable operators have already been discussed in preceding chapters. See the following references for operator information.

- For introductory information on operators that read data from Teradata Database, see Chapter 6: “Moving Data from Teradata Database to an External Target.”
- For information on operators that write data to Teradata Database, see “Chapter 5 Moving External Data into Teradata Database.”
- For more detailed information on using operators, see the chapter on the applicable operator in Section 4.

Common Jobs to Move Data within a Teradata Database

You can use any valid combination of producer and consumer operators to create a job script that moves data within the Teradata Database environment. However, the following section provides examples of some of the most common job scenarios. Evaluate the examples and, if possible, use one of the associated sample job scripts before creating your own.

- Job Example 15: Export Data from a Table and Load It into an Empty Table
- Job Example 16: Export Data and then Use It to Perform Conditional Updates Against Production Tables
- Job Example 17: Bulk Delete of Data from a Teradata Database
- Job Example 18: Export BLOB/CLOB Data from One Teradata Database Table to Another

How to Use Job Examples

The job examples shown in the following sections correspond to the job script examples provided in Appendix A: “Job Script Examples”. Each job example is comprised of a data flow diagram, a statement of the rationale for why the job uses the operators shown, a list of key
operator attributes, and other important factors to consider when customizing the job for your particular use.

The purpose of these job examples is to provide a simple way to find a Teradata PT standard script that you can customize to accomplish an ETL task instead of writing the entire script yourself. Use the job examples as follows:

1. Find the job example that most closely resembles the task you want to accomplish.
2. Examine the related script example to gain an understanding of the job structure.
3. Check the list of key attributes shown in the job example. These are the attributes for which you must specify values based on your unique job. If you are not already familiar with the meaning of these required attributes, read the attribute descriptions and syntax requirements shown in Teradata Parallel Transporter Reference. Refer to Chapters 8 through 20 in this User’s Guide for operator-specific strategies on setting attribute values and other hints for customizing job scripts.
4. Check the list of Important Considerations for other strategic hints and reminders.
5. Customize the script example for your needs.
6. Refer to the section beginning with Chapter 21: “Launching a Job” for information on launching and managing the job.

**Job Example 15: Export Data from a Table and Load It into an Empty Table**

**Job Objective:**
Export data from Teradata Database staging tables before loading it into an empty production table in either the same or a different Teradata Database.

**Data Flow Diagram**
Figure 42 shows a diagram of the job elements for Job Example 15.

Figure 42: Job Example 15 -- Exporting Data and Loading It into Production Tables
Related Script Example
For the script example that corresponds to this job, see “Script Example 15: Extract Data from a Teradata Database Staging Table and Load It into a Production Table” on page 441.

Rationale
This job uses:

• DDL operator because it can DROP work tables and CREATE new target tables prior to loading.
• Export operator because it can extract large amounts of data from a Teradata Database table at high speed.

  Note: The SQL operator extracts data more slowly than the Export operator. However, the SQL Selector operator can be used if the Teradata Database is short on load tasks, because SQL Selector operator instances are not counted as load tasks.
• Use Load operator because it can load large amounts of data into an empty Teradata Database table at high speed.

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job.

For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 5 of this Users Guide.

DDL Operator
• Key attributes
  • TdpId
  • UserName
  • UserPassword
  • ErrorList

Export Operator
• Key Attributes
  • TdpId
  • UserName
  • UserPassword
  • SelectStmt

Load Operator
• Key Attributes
  • TdpId
  • UserName
  • UserPassword
  • LogTable
Job Example 16: Export Data and then Use It to Perform Conditional Updates Against Production Tables

Job Objective:
Export data from Teradata Database tables and perform conditional updates using CASE logic against existing production tables in the same or in another Teradata Database.

Data Flow Diagram
Figure 43 shows a diagram of the job elements for Job Example 16.

Related Script Example
For the script examples that correspond to this job, see “Script Example 16: Export Data and then Use It to Perform Conditional Updates Against Production Tables” on page 443.

Rationale
This job uses the operators shown for the following reasons:

- Use Export operator because it can extract large amounts of data from a Teradata Database table at high speeds.
- Use Update operator as the consumer operator because it can perform INSERT, UPDATE, and DELETE operations in Teradata Database tables.

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job.
For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 4 of this Users Guide.

**DDL Operator**
- Key attributes
  - TdpId
  - UserName
  - UserPassword
  - ErrorList

**Export Operator**
- Key Attributes
  - TdpId
  - UserName
  - UserPassword
  - SelectStmt

**Update Operator**
- Key Attributes
  - TdpId
  - UserName
  - UserPassword
  - LogTable
  - TargetTable
  - WorkTable
  - ErrorTable1
  - ErrorTable2
- Important Considerations
  - The Update operator can perform Insert, Update, and Delete operations on up to 5 tables.
Job Example 17: Bulk Delete of Data from a Teradata Database

Job Objective:
Delete data from a Teradata Database table.

Data Flow Diagram
Figure 44 shows a diagram of the job elements for Job Example 17.

Figure 44: Job Example 17: Delete Data from a Teradata Database Table

Related Script Example
For the script examples that correspond to this job, see “Script Example 17: Bulk Delete of Data from a Teradata Database” on page 447.

Rationale
The job uses the Update operator because it is the only operator that can do stand alone bulk delete of data in a Teradata Database table.

Usage Notes
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job.

For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 5 of this User Guide.

Update Operator
• Key Attributes
  • TdpId
  • UserName
  • UserPassword
  • DeleteTask
  • LogTable
  • ErrorTable2
• Important Considerations
  • Delete Task uses a single session and a single instance of the Update operator.
  • Only a single DELETE statement is used against a single target table.
Chapter 7: Moving Data within the Teradata Database Environment

Common Jobs to Move Data within a Teradata Database

- Remember to specify TYPE UPDATE STANDALONE when using the Update Operator as a standalone operator.
  
  TYPE UPDATE identifies an Update operator that is used as a consumer operator.
  TYPE UPDATE STANDALONE identifies an Update operator that is used as a standalone operator.

Job Example 18: Export BLOB/CLOB Data from One Teradata Database Table to Another

**Job Objective:**
Move rows of data containing BLOB/CLOB data between two Teradata Database tables.

**Data Flow Diagram**
Figure 45 shows a diagram of the job elements for Job Example 18.

**Related Script Example**
For the script examples that correspond to this job, see “Script Example 18: Exporting BLOB/CLOB Data from One Teradata Database Table and Loading It into Another” on page 447.

**Rationale**
This job uses the operators shown for the following reasons:

- Use the SQL Selector and SQL Inserter operators because they are the only operators that can export and load rows containing BLOB/CLOB data.

**Usage Notes**
Make sure to evaluate, and where necessary configure, the following key attributes for the operators defined for this job.

For detailed information on key attributes and other important considerations, see the chapter on the applicable operator in Section 5 of this Users Guide.

**SQL Selector Operator**
- Key Attributes
  - TdpId
Chapter 7: Moving Data within the Teradata Database Environment
Common Jobs to Move Data within a Teradata Database

- UserName
- UserPassword
- SelectStmt
- ReportModeOn
- LobDirectoryPath
- LobFileBaseName
- LobFileExtension

- Important Considerations
  - The SQL Selector operator allows only one session and one instance to read data from a Teradata Database.

**SQL Inserter Operator**

- Key Attributes
  - TdpId
  - UserName
  - UserPassword

- Important Considerations
  - The SQL Inserter operator can load a target table that is either empty or contains data.
  - The APPLY statement that invokes the SQL Inserter operator contains the SQL that defines the operation to be carried out (INSERT) and it identifies the Teradata Database table into which the data will be inserted.

For information on LOBs, see “Large Objects” on page 366.
SECTION 4 Operator Usage Strategies
Section 4: Operator Usage Strategies
CHAPTER 8

DataConnector Operator

This chapter provides usage information for the most commonly specified DataConnector operator features.

Topics include:

- DataConnector Operator Capabilities
- DataConnector Operator as a Producer
- DataConnector Operator as a Consumer
- Key Specifications
- Operational Considerations

DataConnector Operator Capabilities

The DataConnector operator can either:

- Read data from flat files or access modules. As a reader, it is considered a producer operator, that is, one that produces a data stream.
- Write data to flat files or access modules. As a writer, it is considered a consumer operator, that is, one that consumes a data stream.

The DataConnector operator can also be used to scan directories.

Parallel processing can be accomplished by specifying multiple instances of the operator.

DataConnector Operator as a Producer

The DataConnector operator can read flat files in a number of ways:

- Read a single flat file by specifying a directory path and file name.
- Read a directory of flat files by specifying a directory path and a wildcard character (*) with the file name. The wildcard character allows a directory that contains hundreds of files to be treated as a single data source. Use multiple instances of the DataConnector operator to process multiple files in parallel, as long as the files are UNION-compatible.

**Note:** On Windows platforms, using the wildcard character (*) in the 'filename' operator attribute may inadvertently include more files than you desire. For example, if you specify ".dat", a directory scan of the folder will find files as if you had specified "*.dat"; for example,
files with the extension .data, .date, and .dat071503 will also be found. Therefore, you may need to first remove extraneous files from your folder.

- Read data using the supported access modules.

**Definition**

Following is an example of the definition of a DataConnector operator as a producer. Only key attributes are shown (in bold).

```sql
DEFINE DATACONNECTOR OPERATOR
DESCRIPTION 'DataConnector'
TYPE DATACONNECTOR PRODUCER
SCHEMA PRODUCT_SOURCE_SCHEMA
ATTRIBUTES
  (VARCHAR AccessModuleName,
   VARCHAR AccessModuleInitStr,
   VARCHAR FileName,
   VARCHAR Format,
   VARCHAR OpenMode,
   VARCHAR TextDelimite,r
   VARCHAR ArchiveDirectoryPath,
   VARCHAR DirectoryPath,
   VARCHAR PrivateLogName,
   VARCHAR VigilElapsedTime,
   VARCHAR VigilWaitTime,
   VARCHAR VigilStartTime,
   VARCHAR VigilStopTime,
   VARCHAR VigilSortField
  );
```

For required syntax and descriptions of all DataConnector operator attributes, their capabilities and valid values, see *Teradata Parallel Transporter Reference*.

For descriptions of typical uses of the DataConnector operator, see Chapter 5: “Moving External Data into Teradata Database.”

**DataConnector Operator as a Consumer**

**Definition**

Following is an example of the definition of a DataConnector operator as a consumer. Only key attributes are shown.

```sql
DEFINE DATACONNECTOR OPERATOR
DESCRIPTION 'DataConnector'
TYPE DATACONNECTOR CONSUMER
SCHEMA CONSUMER_SOURCE_SCHEMA
ATTRIBUTES
  (VARCHAR AccessModuleName,
   VARCHAR AccessModuleInitStr,
   VARCHAR FileName,
   VARCHAR Format,
   VARCHAR OpenMode,
   VARCHAR TextDelimite,
   VARCHAR ArchiveDirectoryPath,
   VARCHAR DirectoryPath,
   VARCHAR PrivateLogName,
   VARCHAR VigilElapsedTime,
   VARCHAR VigilWaitTime,
   VARCHAR VigilStartTime,
   VARCHAR VigilStopTime,
   VARCHAR VigilSortField
  );
```
Key Specifications

DataConnector Operator as a Producer in the Teradata PT Job Script

To define the DataConnector operator as a producer operator in a Teradata PT job script, you must provide the following required specifications:

- TYPE DATACONNECTOR PRODUCER
- Key operator attributes

Key Attributes: DataConnector Operator

As a Producer (Without an Access Module)

The following are the key attributes for DataConnector operator as a producer (not through an access module):

- FileList
- FileName
- Format
- MultipleReaders
- OpenMode
- TextDelimiter
- PrivateLogName

As a Producer (With an Access Module)

The following are the key attributes for the DataConnector operator as a producer (through an access module):

- AccessModuleName
- AccessModuleInitStr
- FileName
- Format
- OpenMode
- TextDelimiter
- PrivateLogName

For required syntax and descriptions of all DataConnector operator attributes, their capabilities and valid values, see Teradata Parallel Transporter Reference.

For descriptions of typical uses of the DataConnector operator, see Chapter 6: “Moving Data from Teradata Database to an External Target.”.
Chapter 8: DataConnector Operator

Key Specifications

**As a Producer (Using Batch Directory Scan)**

The following are the key attributes for the DataConnector operator as a producer (using Batch Directory Scan):

- ArchiveDirectoryPath
- DirectoryPath
- EnableScan
- FileName
- Format
- OpenMode
- TextDelimiter
- PrivateLogName

**As a Producer (Using Active Directory Scan)**

The following are the key attributes for the DataConnector operator as a producer (using Active Directory Scan):

- ArchiveDirectoryPath
- DirectoryPath
- EnableScan
- FileName
- Format
- OpenMode
- TextDelimiter
- VigilElapsedTime
- VigilStartTime
- VigilSortFile
- VigilWaitTime
- PrivateLogName

For required syntax and descriptions of all DataConnector (as a producer) operator attributes, their capabilities and valid values, see *Teradata Parallel Transporter Reference*.

**DataConnector Operator as a Consumer in a Teradata PT Job Script**

When you define the DataConnector operator as a consumer operator in a Teradata PT job script, you must provide the following required specifications:

- TYPE DATACONNECTOR CONSUMER
- Key operator attributes
**Key Attributes: Data Connector Operator**

**As a Consumer (Without an Access Module)**

The following are the key attributes for DataConnector operator as a consumer (not through an access module):

- File List
- FileName
- Format
- OpenMode
- TextDelimiter
- PrivateLogName

**As a Consumer (With an Access Module)**

The following are the key attributes for the DataConnector operator as a consumer (through an access module):

- AccessModuleName
- AccessModuleInitStr
- FileName
- Format
- OpenMode
- TextDelimiter
- PrivateLogName

For required syntax and descriptions of all DataConnector (as a consumer) operator attributes, their capabilities and valid values, see *Teradata Parallel Transporter Reference*.

**FileName**

The use of the FileName attribute varies depending on operator type and operating system:

- **DataConnector Producer Operator**

  When using the DataConnector operator as a producer, the wildcard character (*) is allowed in a FileName attribute if you want to process all matching files or members within a named UNIX directory or the z/OS partitioned data set (PDS or PDSE).

  The following conditions also apply depending on your operating system:

  - On UNIX systems, the FileName attribute limit is 255 bytes. FileName can either be a complete pathname of the file, or the name of a file within a directory. But if the directory is not defined in the optional DirectoryPath attribute, filename is expected to be found in the default directory. See Table 7 for examples.

  - On z/OS systems, FileName can specify the fully-qualified data set name of the script file, including the member name if the data set is a PDS or PDSE, the member name of a PDS or PDSE script library, or 'DD:<ddname>'. If only a member name is specified for the FileName, then the (PDS or PDSE) data set name containing the member must be specified by the DirectoryPath attribute. See Table 7 for examples.
• **DataConnector Consumer Operator**
  
  When using the DataConnector operator as a consumer, the FileName attribute becomes the complete file specification, and the FileName cannot contain the wildcard character (*).

  On UNIX systems, unless you specify a pathname, the FileName is expected to be found in the default directory. See Table 7 for examples.

• **Combining FileName and FileList attributes**
  
  The FileList attribute extends the capabilities of the FileName attribute. Adding FileList = ‘Y’ indicates that the file identified by FileName contains a list of files to be processed as input or used as containers for output. The file names found within the FileName file are expected to be full path specifications. If no directory name is included, the files are expected to be located within the current directory. Supplying full paths for output files enables you to write files to multiple directories or disks. You cannot use the DirectoryPath attribute in conjunction with this feature.

  When the combination of FileName and FileList attributes are used to control output, the supplied file list must have the same number of files as there are defined consumer instances; a mismatch results in a terminal error. At execution, rows are distributed to the listed files in a round-robin fashion if the `build -C` option is used. Without the option, rows may not be evenly distributed across the listed files.

  You cannot combine this feature with the archiving feature. Any attempt to use the archive feature (for example, by defining the ArchiveDirectoryPath attribute) results in a terminal error.

  If the pathname that you specify with the FileName attribute (as filename) contains any embedded pathname syntax (“/” on UNIX or “\” on Windows), the pathname is accepted as the entire pathname. However, if the DirectoryPath attribute is present, the FileName attribute is ignored, and a warning message is issued.

  Table 7 contains valid syntax examples for the FileName attribute.
Chapter 8: DataConnector Operator

Key Specifications

Note: On Windows platforms, using the wildcard character (*) in filename can inadvertently include undesired files. For example, specifying *.dat is the same as specifying *.dat*, which can include files with extensions such as .data, .date, and .dat071503. Therefore, it is recommend that extraneous files be removed from your folder.

Format

Several file formats can be processed by the DataConnector operator. Specify file format with the Format attribute.

- **Format = 'Binary'** - Each record contains a two-byte integer data length (n) followed by n bytes of data.
- **Format = 'Text'** - Each record is entirely character data, an arbitrary number of bytes followed by one of the following end-of-record markers:
  - A single-byte line feed (X'0A') on UNIX platforms
  - A double-byte carriage-return/line-feed pair (X'0D0A') on Windows platforms

Table 7: Valid FileName Syntax

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Valid Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>z/OS</td>
<td>FileName = '//' 'name.name(member)'''</td>
<td>z/OS PDS DSN Name.Name(Member)</td>
</tr>
<tr>
<td></td>
<td>FileName = '//' 'name.name''</td>
<td>z/OS DSN (sequential) Name.Name</td>
</tr>
</tbody>
</table>
|                  | FileName = 'DD:ddname'                | z/OS DSN described in JCL DD statement name “ddname.” If no DD statement is specified, the following occurs:  
  - For input, the fopen library function tries to open an HFS file named DD:ddname in the home directory. If the file is not found, the fopen library function returns an error that is displayed in the SYSOUT.  
  - For output, the fopen library function tries to create an HFS file named DD:ddname in the home directory. If the file already exists, the previous contents are overwritten. |
|                  | FileName = 'member'                   | z/OS PDS member expected to reside in the dsn that is defined in the DirectoryPath attribute. |
| UNIX             | FileName = '/tmp/user/ filename'      | UNIX pathname. |
|                  | FileName = 'filename'                 | If the DirectoryPath attribute is undefined, filename is located in the default directory. |
| Windows          | FileName = '\\tmp\user-filename'    | Windows path name. |
|                  | FileName = 'filename'                 | Windows file name expected to be found in the directory defined in the DirectoryPath attribute.  
  - If the DirectoryPath is not defined, filename is located in the default directory. |
Chapter 8: DataConnector Operator

Key Specifications

- **Format = 'Delimited'** - Each record is in variable-length text record format, but they contain fields (columns) separated by one or more delimiter characters, as defined with the 'TextDelimiter' attribute, which has the following limitations:
  - It can only be a sequence of characters.
  - It cannot be any character that appears in the data.
  - It cannot be a control character other than a tab.

With this file format, all of the data types in the DEFINE SCHEMA must be VARCHARs. And, if not provided, the TextDelimiter attribute defaults to the pipe character ( | ).

**Note:** There is no default escape character when using delimited data. Use the DataConnector operator’s EscapeTextDelimiter optional attribute to define the escape character.

- **Format = 'Formatted'** - Each record is in a format traditionally known as FastLoad or Teradata format, which is a two-byte integer (n) followed by n bytes of data, followed by an end-of-record marker (X’0A’ or X’0D0A).

- **Format = 'Unformatted'** - The data does not conform to any predefined format. Instead, the data is entirely described by the columns in the schema definition of the DataConnector operator.

**OpenMode**

Attribute that specifies the read/write access mode. Read means read-only access. Write means write-only access. If a mode value is not specified for OpenMode, it defaults to Read for a producer instance and Write for a consumer instance.

**TextDelimiter**

To help avoid accidental or ambiguous delimiters, the DataConnector operator allows multibyte VARTEXT delimiters, meaning that delimiters embedded in delimited data can be any length without concern for multiple characters.

To embed a delimiter, precede the delimiter with a backslash ( \ ) or escape. For example, if the default delimiter is the pipe ( | ), the following record:

```
abcd | ef\ | gh | ijk\ | | lmno | pqrst |
```

is broken into the following five text formatted columns:

```
abcd ef|gh ijk| lmno pqrst
```

In another example, the following data uses comma delimiters, and the second record is invalid because a comma appears with the data in the third column:

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Note</th>
<th>Valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>John,</td>
<td>Doe,</td>
<td>Gold Customer</td>
<td>valid</td>
</tr>
<tr>
<td>Sam,</td>
<td>Smith,</td>
<td>Silver Customer, since 1999</td>
<td>invalid</td>
</tr>
</tbody>
</table>
To resolve the invalid record due to the ambiguity of the comma in the third column, use one of the following solutions:

- Use multiple single-byte characters (a single multi-byte character also works) as in the following example:

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Note</th>
<th>Valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>John,,</td>
<td>Doe, ,</td>
<td>Gold Customer</td>
<td>valid</td>
</tr>
<tr>
<td>Sam,,</td>
<td>Smith, ,</td>
<td>Silver Customer, since 1999</td>
<td>valid</td>
</tr>
</tbody>
</table>

- Use escape delimiter characters preceded with a backslash to escape any commas as in the following example:

<table>
<thead>
<tr>
<th>First Name</th>
<th>Last Name</th>
<th>Note</th>
<th>Valid?</th>
</tr>
</thead>
<tbody>
<tr>
<td>John,</td>
<td>Doe,</td>
<td>Gold Customer</td>
<td>valid</td>
</tr>
<tr>
<td>Sam,</td>
<td>Smith,</td>
<td>Silver Customer, since 1999</td>
<td>valid</td>
</tr>
</tbody>
</table>

**Note:** Use the TextDelimiter attribute to specify the delimiter. Use the EscapeTextDelimiter attribute to change the default escape delimiter to something other than the back slash character (\).

**PrivateLogName**

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the diagnostic trace messages produced by the operator.

For details, see *Teradata Parallel Transporter Reference*.

**AccessModuleName**

Optional attribute that specifies the name of the access module file.

For details, see *Teradata Parallel Transporter Reference*

**AccessModuleInitStr**

Optional attribute that specifies the initialization string for the specified access mode.

**EnableScan**

Optional attribute that bypasses the directory scan logic when using access modules.
Chapter 8: DataConnector Operator
Operational Considerations

**DirectoryPath**

Attribute that supports the FileName attribute wildcard feature. Use this attribute to specify an existing base directory path for the location of the file indicated by the FileName attribute.

This attribute defaults to the directory in which the job is executing (the job working directory specified in the DEFINE JOB statement). For details, see *Teradata Parallel Transporter Reference*.

**MultipleReaders**

Use the MultipleReaders attribute to process a single, very large file with multiple instances of the Data Connector operator. For details, see *Teradata Parallel Transporter Reference*.

**SkipRows and SkipRowsEveryFile**

Optional attributes that work together to indicate rows to be skipped during processing. The SkipRows attribute expects an integer that indicates the number of rows to be skipped; valid values for the SkipRowsEveryFile attribute are Y[es] and N[o].

For example, if SkipRows = 5 and SkipRowsEveryFile = 'Y', the system skips the first five rows in every file and begins processing each file at the sixth row. You might use this method to bypass header rows that appear at the beginning of each file. Alternatively, if SkipRows = 1000 and SkipRowsEveryFile = 'N', the system skips the first 1000 rows of the job and begins processing on row 1001. If, for example, the job contains five files of 300 rows each, the system skips the first three files (900 rows) and begins processing the fourth file on line 101, skipping the required 1000 rows. You might use this method to begin reprocessing a failed job to eliminate rows that have already been processed.

**Vigil Attributes**

For information on all vigil attributes, see *Teradata Parallel Transporter Reference*.

**Operational Considerations**

**File Size Restrictions**

Windows, Sun Solaris SPARC, AIX, z/OS, Linux, and HP-UX operating systems place no 2-gigabyte file size restriction, provided system parameters are appropriately set.

**z/OS Considerations**

- The vigil window feature is not available for z/OS PDS files.
- Load distribution of input files across instances are based on file count rather than file size.
- The input of text formatted data, including delimited data, is supported only for files that are generated by the DataConnector operator. Currently, files that originate in a native z/OS editor or FTP are not properly formatted.
This chapter provides usage information for the most commonly specified DDL operator features.

Topics include:

- DDL Operator Capabilities
- DDL Operator Definition
- Key Specifications
- Job Options
- Operational Considerations

### DDL Operator Capabilities

The main task of DDL operator is to send SQL statements to Teradata Database to perform setup operations prior to executing load or export tasks. For example, The DDL operator is frequently used to create or drop error tables or indexes required by Teradata PT operators that will perform the main job tasks. The DDL operator is also used to perform mini-batch loading tasks. The DDL operator is neither a producer nor consumer operator. It does not retrieve data records from the Teradata PT data stream nor put records into the data stream.

**Note**: Because conditions in the database may change between uses of a particular job script, you should always evaluate the need to add or change the DDL operator in the script before running a particular TPT job.

Figure 46 shows the DDL operator interface.

Figure 46: DDL Operator Interface
The DDL Operator Function in a Teradata PT Job

When you use the DDL operator in a Teradata PT job, Teradata PT directs an instance of the DDL operator to:

- Log on to the Teradata Database using your user name, password, Teradata Database, and account ID information specified in the job script.
- The operator terminates when scripts contain invalid or unsupported statements.
- Submit a request (this can be one statement or multiple statements) to the Teradata Database and processes returned parcels from these statements.
- Issue, if an error occurs, an error message and terminate, unless the error is specified in the ErrorList attribute, in which case the DDL operator ignores the error and continues processing the job.
- Log off the Teradata Database.
- Put out an end status indicating how many requests succeeded, failed, or were ignored.
- Resume, in the case of a restart, at the group that was being executed, but failed to complete, in the previous execution.

DDL Operator Definition

Following is an example definition of the DDL operator.

Only key attributes are shown (in bold).

```
DEFINE OPERATOR DDL_OPERATOR
DESCRIPTION 'Teradata PT DDL OPERATOR Sample'
TYPE DDL
ATTRIBUTES
{
  VARCHAR TdpId,
  VARCHAR UserName,
  VARCHAR UserPassword,
  VARCHAR ErrorList,
  VARCHAR PrivateLogName
};
```

For required syntax and descriptions of all DDL operator attributes, their capabilities and valid values, see Teradata Parallel Transporter Reference.

For descriptions of typical uses of the DDL operator, see Chapter 5: “Moving External Data into Teradata Database” and Chapter 7: “Moving Data within the Teradata Database Environment”.

162 Teradata Parallel Transporter User Guide
Key Specifications

Defining a DDL Operator in a Teradata PT Script

When you define the DDL operator in a Teradata PT job script, you must provide the following required specifications:

- TYPE DDL
- Key operator attributes:
  - TdpId
  - UserName
  - UserPassword
  - ErrorList
  - PrivateLogName

Note: The first three key attributes listed above are “security attributes.”

In addition, other optional operator attributes can be specified depending on how the operator will be used.

For more information about available DDL operator attributes and the capabilities they provide, see DDL Operator in Teradata Parallel Transporter Reference.

Security Attributes

TdpId
Optional attribute that specifies the name of the Teradata Database machine) non-mainframe platforms or TDP (mainframe platforms) for the DDL operator job.

If you do not specify the value for the TdpId attribute, the operator uses the default TdpId established for the user by the system administrator.

For details, see “Teradata Database Logon Security” on page 73.

UserName
Attribute that specifies the Teradata Database user name.

For details, see “Teradata Database Logon Security” on page 73.

UserPassword
Attribute that specifies the password associated with the user name.

For details, see “Teradata Database Logon Security” on page 73.

ErrorList
The ErrorList attribute identifies specific error conditions to ignore.

For details, see Teradata Parallel Transporter Reference.
Dropped Tables

If you want the DDL operator to overlook or ignore errors about attempting to drop tables that do not exist, specify the error in the ErrorList attribute.

PrivateLogName

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the diagnostic trace messages produced by the operator.

For details, see Teradata Parallel Transporter Reference.

Job Options

Supported SQL Statements

The DDL operator supports every SQL statement except statements that:

- Return data to the operator, such as the SELECT, HELP, and SHOW statements.
  The CREATE REPLICATION GROUP statement is supported. The user needs the group id and/or the token after a replication group is created.
  The INSERT...SELECT statement is supported. It returns the total number of rows being inserted in the table.
- Require the operator to send data back to the Teradata Database.

Because standalone operators do not send data to or retrieve data from data streams, the DDL operator does not support the USING clause with the INSERT, UPDATE, and DELETE DML SQL statements.

Note: The Stream operator and the Update operator automatically employ the USING clause where needed.

All data values used by the SQL statement used by the DDL operator must be hard-coded into the submitted SQL statements.

Specifying DDL Statements in the Teradata PT APPLY Statement

The SQL statements that the DDL operator supports are specified in the Teradata PT APPLY statement.

The following examples show how to specify DDL statements in the APPLY statement:

- One SQL statement per group

  APPLY
  'ddl stmt1',
  'ddl stmt2',
  ........
  'ddl stmt3'
  TO OPERATOR (operator specification)

- Multiple SQL statements in a group, but only one group
Chapter 9: DDL Operator
Job Options

Chapter 9: DDL Operator
Job Options

APPLY
 ('ddl stmt1','ddl stmt2', ... ,'ddl stmtn')
TO OPERATOR (operator specification)

• Multiple SQL statements per group, and multiple groups
  APPLY
  ('ddl stmt11','ddl stmt12', ... ,'ddl stmt1n'),
  ...
  ('ddl stmtxn1','ddl stmtxn2', ... ,'ddl stmtxnm')
  TO OPERATOR (operator specification)

If more than one statement is specified in one group, then the DDL operator combines all
statements into one single multistatement request and sends it to the Teradata Database as one
implicit transaction. This means that any statement failure or any error rolls back the entire
transaction.

The SQL statements are executed by groups in the order they are specified in the APPLY
statement.

Grouping SQL Statements in the Teradata PT APPLY Statement

You can group several SQL statements together to perform a desired logical database task and
still take advantages of the automatic rollback feature on Teradata Database in case of any
statement failures or any errors occurring during the transaction.

However, you should have one SQL statement per group if you desire to execute each
statement in its own transaction.

When multiple SQL statements are specified in one DML group in the APPLY statement,
Teradata Database enforces the rule that if the group contains a DDL statement, it must be the
last statement in the implicit transaction, which means the last statement in a group.

Therefore, given that the information in parentheses (below) represents a group, the validity
of the statements can be determined as follows:

• Group 1: (DDL) is valid.
• Group 2: (DDL, DDL) is invalid because only one DDL statement is allowed.
• Group 3: (DML, DML, DDL) is valid.
• Group 4: (DML, DML, DML) is valid, even though the group contains no DDL statement.
• Group 5: (DML, DDL, DML) is invalid because the DDL statement is not the last
  statement in the group.

If a script contains unsupported or invalid statements, the job terminates so the script can be
fixed before continuing.
Operational Considerations

Checkpointing and Restarting

Because SQL statements are sent to Teradata Database by group in the order in which they are specified in the Teradata PT APPLY statement, the DDL operator can take a checkpoint after each group is executed. A group can contain one or more SQL statements. A checkpoint, with respect to the DDL operator, marks the last group of DDL/DML SQL statements to execute successfully.

The DDL operator restarts at the beginning of the group of SQL statements whose execution is interrupted by an abnormal termination. If the interrupted group has only one SQL statement, the DDL operator restarts at that statement.

If the last request was successful prior to a restart, the operator can resume at the next request in line. If the last request failed prior to a restart, then the operator resumes at the failed request.

For information on checkpointing and restarting, see Chapter 24: “Troubleshooting a Failed Job.”
CHAPTER 10
Export Operator

This chapter provides usage information for the most commonly specified Export operator features.

Topics include:

- Export Operator Capabilities
- FastExport Utility and the Export Operator
- Export Operator Definition
- Key Specifications
- Job Options
- Operational Considerations

Export Operator Capabilities

The Export operator, a producer operator, uses Teradata FastExport protocol to extract large amounts of data at high speeds from Teradata Database using block transfers over multiple sessions.

The Export operator reads data from one or more tables in Teradata Database. Figure 47 shows the Export operator interface.

Figure 47: Export Operator Interface

The Export Operator Function in a Teradata PT Job

When you use the Export operator in a Teradata PT job, the Export operator does the following:
1. Logs on to the Teradata Database, using your user name, password, Teradata Database, and account ID information specified in the job script.

2. Sends the SELECT request in the SelectStmt attribute information and retrieve the exported data from the Teradata Database.

3. Logs off from the Teradata Database.

4. If the Export operator job is successful, the job completes and returns information about the job, such as:
   - Total number of records exported from the Teradata Database.
   - Records exported by each instance.

5. If the job is unsuccessful, the job terminates and provides information about the job so that you can correct any problem and restart the job.

FastExport Utility and the Export Operator

Table 8 lists the operating features and capabilities of the Teradata FastExport utility and indicates whether such features and capabilities are supported by the Export operator.

Table 8: Export Operator Supported Features

<table>
<thead>
<tr>
<th>FastExport Utility Feature</th>
<th>Teradata PT Operator Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Modules</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>ANSI Date</td>
<td>Supported</td>
</tr>
<tr>
<td>Blocksize Specification</td>
<td>Supported</td>
</tr>
<tr>
<td>Character sets</td>
<td>Supported, using the USING CHARACTER SET clause before the DEFINE JOB statement</td>
</tr>
<tr>
<td>Checkpoint/restart</td>
<td>Supported</td>
</tr>
<tr>
<td>Configuration file</td>
<td>Supported, using the tbuild command line job attribute option (-v)</td>
</tr>
<tr>
<td>DECIMALDIGITS option</td>
<td>Supported</td>
</tr>
<tr>
<td>DISPLAY command</td>
<td>Not supported</td>
</tr>
<tr>
<td>Environment variables</td>
<td>Not supported</td>
</tr>
<tr>
<td>IF-ELSE-ENDIF commands</td>
<td>Not supported</td>
</tr>
<tr>
<td>Indicator mode</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>INMOD routines</td>
<td>Not supported</td>
</tr>
<tr>
<td>IMPORT command</td>
<td>Not supported</td>
</tr>
<tr>
<td>Maximum/minimum sessions</td>
<td>Supported</td>
</tr>
</tbody>
</table>
Table 8: Export Operator Supported Features (continued)

<table>
<thead>
<tr>
<th>FastExport Utility Feature</th>
<th>Teradata PT Operator Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Teradata SQL SELECT statements</td>
<td>Supported, with same layout, not supported with different layouts</td>
</tr>
<tr>
<td>Record formats</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>Nonindicator mode</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>Notify</td>
<td>Supported</td>
</tr>
<tr>
<td>OUTFILE option</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>OUTLIMIT specification</td>
<td>Supported</td>
</tr>
<tr>
<td>OUTMOD routines</td>
<td>Supported, using the FastExport OUTMOD Adapter operator</td>
</tr>
<tr>
<td>ROUTE MESSAGES command</td>
<td>Not supported</td>
</tr>
<tr>
<td>RUN FILE command</td>
<td>Supported by Teradata PT script language</td>
</tr>
<tr>
<td>SET command</td>
<td>Not supported</td>
</tr>
<tr>
<td>Show version information</td>
<td>Supported</td>
</tr>
<tr>
<td>No Spooling</td>
<td>Supported</td>
</tr>
<tr>
<td>Teradata SQL SELECT Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL Statements (other such as CREATE TABLE, DROP TABLE, etc.)</td>
<td>Supported, using the DDL operator</td>
</tr>
<tr>
<td>SQL DATABASE Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>SYSTEM Command</td>
<td>Supported, using the OS Command operator</td>
</tr>
<tr>
<td>Tenacity</td>
<td>Supported</td>
</tr>
<tr>
<td>User-defined variables</td>
<td>Limited support via script syntax</td>
</tr>
</tbody>
</table>

Export Operator Definition

Following is an example of a Export operator definition.

Only key attributes are shown (in bold).

DEFINE OPERATOR EXPORT
DESCRIPTION 'EXPORT OPERATOR'
TYPE EXPORT
SCHEMA PRODUCT_SOURCE_SCHEMA
ATTRIBUTES
{
  VARCHAR TdpId,
  VARCHAR UserName,
  VARCHAR UserPassword,
Chapter 10: Export Operator

Key Specifications

Export Operator in a Teradata PT Job Script

When you define an Export operator in a job script, you must provide the following required specifications:

- TYPE EXPORT
- Key operator attributes:
  - TdpId
  - UserName
  - UserPassword
  - SelectStmt
  - PrivateLogName

**Note:** The first three key attributes listed above are “security attributes.”

In addition, other optional operator attributes can be specified depending on how the operator will be used.

For more information about available Export operator attributes and the capabilities they provide, see Export Operator in *Teradata Parallel Transporter Reference*.

Security Attributes

**TdpId**

Optional attribute that specifies the name of the Teradata Database machine (non-mainframe platforms) or TDP (mainframe platforms) for the Export operator job.

If you do not specify the value for the TdpId attribute, the operator uses the default TdpID established for the user by the system administrator.

For details, see “Teradata Database Logon Security” on page 73.

**UserName**

Attribute that specifies the user name on the Teradata Database.

For details, see “Teradata Database Logon Security” on page 73.
**UserPassword**

Attribute that specifies the password associated with the user name.

For details, “Teradata Database Logon Security” on page 73.

**SelectStmt**

SelectStmt (SELECT statement) is the required attribute that the Export operator uses to perform data selection from Teradata Database tables. Multiple parallel instances of the Export operator and multiple sessions within each instance can improve the performance of an export.

A Select request within an Export script can have multiple SELECT statements. A SELECT statement can be optionally preceded by a LOCKING modifier.

However, Export SELECT requests cannot:

- Specify a USING modifier.
- Access non-data tables, such as SELECT DATE or SELECT USER.
- Be satisfied by one or two AMPs, such as SELECT statement which accesses rows based on the primary index or unique secondary index of a table.
- Contain character large object (CLOB) or binary large object (BLOB) data types.

The following table describes types of Select requests.

<table>
<thead>
<tr>
<th>Type of Select Request</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contains multiple SELECT statements</td>
<td>The Teradata Database might execute the requests in parallel, but response are still returned in the order of the requests, for the first statement first, then for the second, and so on. If the structure of response rows differs, an error results and the job terminates.</td>
</tr>
</tbody>
</table>
| Uses a LOCKING modifier | The specified lock remains in effect during the execution of all statements within the request that contains the modifier. The Teradata Database does the following:  
  - Implements all resource locks for the entire request before executing any of the statements in the request.  
  - Maintains the locks until all of the response data for the request is moved to spool tables.  
Following is a valid SELECT request using the LOCKING modifier:  
LOCKING TABLE MYTABLE FOR ACCESS SELECT COL1, COL2 FROM MYTABLE;  
Note that the LOCKING modifier can precede the SELECT statement. |
| Uses an ORDER BY clause | Specify one Export instance.  
Following is a valid ORDER BY clause:  
SELECT COL1, COL2 FROM MYTABLE ORDER BY COL1; |
Multiple SELECT Statements
A single Export operator can read data from multiple tables as long as their schemas are the same.

For example, if Table1 and Table2 have the same schema, use the following SELECT statement in the attribute section of the Export operator definition:

```
VARCHAR SelectStmt='SELECT * FROM Table1;SELECT * FROM Table2'
```

PrivateLogName
Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the messages produced by the operator.

By default, no diagnostic trace messages are produced. Diagnostic trace messages are produced only when the user sets a valid value for the TraceLevel attribute.

For details, see Teradata Parallel Transporter Reference.

Job Options

Limits on Export Jobs

The Export operator requires one designated Teradata Database loader task.

The number of concurrent loader tasks is, however, configurable in the Teradata Database environment using the same MaxLoadTasks field control used by FastLoad, FastExport, and MultiLoad. For example, one FastLoad job equates to one Teradata PT Load job in the count for MaxLoadTasks.

For more information, see “DBS Control Utilities” in Utilities.

Limiting Output

Use the OUTLIMIT attribute to limit the number of rows that an export job returns. OUTLIMIT expects a number greater than zero as input. Specifying an invalid number terminates the export operator and returns an error message. If OUTLIMIT is not specified, the export operator returns all records. OUTLIMIT controls the number of rows exported for each instance.

You can also use one of the following techniques to limit record output:

- The WHERE clause can limit the number of rows that are exported by specifying conditions that must be met. For more information on the WHERE clause, see Teradata Parallel Transporter Reference.
- The SAMPLE function is an SQL function that can limit the number of random rows returned. For more information, see SQL Data Manipulation Language.
Checkpointing and Restarting

The Export operator takes a checkpoint only when all data is sent to the data stream. If a restart occurs, the operator either must send all of the data or none of the data depending on whether the checkpoint has taken place.

- If all the data is sent, then the operator displays the following message and does not resend any of the data:
  
  \textit{Restart indicates that this export job completed.}

- If all the data is not sent, the operator terminates. Restart the job from the beginning.

- If none of the data is sent, the operator sends the data.

The Export operator does \textit{not} support a user-defined restart log table.

\textbf{Caution:} If a checkpoint interval is specified on the \texttt{tbuild} command line, the checkpoints incurred between the start of data loading and the end of data loading are ignored by the Export operator.

For information on checkpointing and restarting, see Chapter 24: "Troubleshooting a Failed Job."

Using the Large Decimal Teradata Database Feature

The Export operator optional attribute \textit{MaxDecimalDigits} allows Teradata PT to use the Large Decimal functionality of the Teradata Database. But what if \textit{MaxDecimalDigits} is set and the Teradata Database or the CLIv2 does not support this feature? Using the \textit{IgnoreMaxDecimalDigits} attribute allows the Teradata PT job to continue. This attribute is available to the Export and SQL Selector operators.

Figure 48 shows the logic flow used to determine if the Teradata PT job will continue. The main decision points are:

- Is \textit{MaxDecimalDigits} a valid value? Then \(18 \leq \textit{MaxDecimalDigits} \leq 38\). If the value does not fall in this range, a warning message is issued and the job will continue.

- Next, if the \textit{IgnoreMaxDecimalDigits} attribute is set, continue to determine if the Teradata Database and CLI versions support the Large Decimal feature.

  The job will terminate with an Exit code of eight and an error message if \textit{IgnoreMaxDecimalDigits} has not been set and either the Teradata Database or CLI levels do not support the Large Decimal feature.

  If the Teradata Database and CLI versions do not support the Large Decimal feature and the \textit{IgnoreMaxDecimalDigits} attribute is set, continue the job.

  The Export and SQL Selector operators will continue the job when the Large Decimal feature in not supported because the maximum returned decimal digits is 18 which is in the valid range.

- If the versions are supported and the \textit{MaxDecimalDigits} attribute is \textit{not} a valid value, display a message and continue the job with a Exit code of four.

- If either the Teradata Database or CLI version do not support the Large Decimal feature, and \textit{MaxDecimalDigits} is a valid value but the \textit{IgnoreMaxDecimalDigits} attribute is not set, display a message and continue the job with an Exit code of 4.
For complete information on the `MaxDecimalDigits` and `IgnoreMaxDecimalDigits` attributes, see Teradata Parallel Transporter Reference.

Figure 48: How the Teradata Database Large Decimal Feature Affects Job Continuation

Operational Considerations

Performance

The exporting ability of the Export operator is designed to outperform the SQL Selector operator (similar to BTEQ Export) in transferring large amounts of data from Teradata tables.
This is because, among other factors, the Export operator is multisessioned and able to run in multiple parallel instances. It performs block exports in contrast to the SQL Selector operator, which performs single-session and one-row-at-a-time exports.

Export functions in Teradata PT do not make Teradata Database perform faster. Rather, Teradata PT allows for the greater use of multiple parsing engines and AMPs.

**Sessions and Instances**

When running the Export operator, the actual session limit is determined by the first limiting factor that is encountered.

Multiple parallel instances of the Export operator can be used to improve the performance of the export over a single instance.

For detailed information about sessions and instances, see “Optimizing Job Performance with Sessions and Instances” on page 80.

**Multiple Databases**

When reading data from multiple Teradata Databases with different login information, specify values for the UserName and Password attributes in the SELECT portions of the APPLY statements that specify each database.
FastExport OUTMOD Adapter Operator

This chapter provides usage information for the most commonly specified FastExport OUTMOD Adapter operator features.

Topics include:

- FastExport OUTMOD Adapter Operator Capabilities
- FastExport OUTMOD Adapter Operator Definition
- Key Specifications

**FastExport OUTMOD Adapter Operator Capabilities**

The FastExport OUTMOD Adapter operator is a consumer operator that allows you to use Teradata FastExport OUTMOD routines within the Teradata PT.

The term OUTMOD is an acronym for “output modification.” OUTMOD routines are user exit routines the FastExport OUTMOD can call to provide enhanced processing functions on output data records from Teradata Database before they are sent to the client system.

For information on creating and using FastExport OUTMOD Routines, see *Teradata FastExport Reference*.

**Note:** The FastExport OUTMOD Adapter supports only FastExport OUTMOD routines, but not the OUTMOD routines for any other Teradata standalone load or unload utility.

**Figure 49** shows the FastExport OUTMOD Adapter operator interface.

**Figure 49: FastExport OUTMOD Adapter Operator Interface**

[Diagram showing the FastExport OUTMOD Adapter Operator Interface]
FastExport OUTMOD Adapter Operator Definition

Following is an example of a FastExport OUTMOD Adapter operator definition.

Only key attributes are shown (in bold).

DEFINE OPERATOR FASTEXPORT OUTMOD ADAPTER
  DESCRIPTION 'FASTEXPORT OUTMODE ADAPTER OPERATOR'
  TYPE EXPORT
  SCHEMA PRODUCT_SOURCE_SCHEMA
  ATTRIBUTES
    (VARCHAR OutmodName,
    VARCHAR Format,
    VARCHAR FileName,
    VARCHAR PrivateLogName
  )

For required syntax and descriptions of all FastExport OUTMOD Adapter operator attributes, their capabilities and valid values, see Teradata Parallel Transporter Reference.

For descriptions of typical uses of the FastExport OUTMOD operator, see Chapter 6: “Moving Data from Teradata Database to an External Target”.

Key Specifications

OutmodName

Attribute that specifies the name of the FastExport OUTMOD routine to be loaded.

When you specify an OUTMOD name in a Teradata PT job script, it is the named routine rather than a data source that provides the output records.

For details, see Teradata Parallel Transporter Reference.

Format

The internal format of each output record must conform to the specified schema.

The FastExport OUTMOD Adapter operator requires each field of the output record to be of a specified length. When no length is specified, the operator assumes the conventional default length.

The FastExport OUTMOD Adapter operator supports both fixed- and variable-length output records.

For records with a two-column schema such as:

{ COL1 INTEGER,
  COL2 VARCHAR(10))

each record is comprised of:

- A four-byte integer, because 4 is the INTEGER length
- A two-byte length indicator for the VARCHAR field
- The CHAR data of the length indicated in the two-byte length indicator field

The following examples show the byte order when running Teradata PT on an Intel processor.

- If, for example, the integer has a decimal value of 100 and the VARCHAR is ASCII '12345', then the output record appears in hexadecimal notation as:
  \[\text{x'64 00 00 00 05 00 31 32 33 34 35'}\]
- If the data in the output record is in indicator mode, there is a single preceding indicator byte:
  \[\text{x'00 64 00 00 00 05 00 31 32 33 34 35'}\]

**FileName**

Optional attribute that specifies the name of the file. On UNIX, limited to 255 bytes.

For details, see *Teradata Parallel Transporter Reference*.

**PrivateLogName**

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the diagnostic trace messages produced by the operator.

For details, see *Teradata Parallel Transporter Reference*.

**Job Options**

The FastExport OUTMOD Adapter operator provides the same functionality and is used in the same circumstances as the DataConnector operator.

**Operational Considerations**

The FastExport OUTMOD Adapter provides the same functionality and is used in the same circumstances as the DataConnector operator.
This chapter provides usage information for the most commonly specified FastLoad INMOD Adapter operator features.

Topics include:
- FastLoad INMOD Adapter Operator Capabilities
- FastLoad INMOD Adapter Operator Definition
- Key Specifications

**FastLoad INMOD Adapter Operator Capabilities**

The FastLoad INMOD Adapter operator is a producer operator that allows you to use Teradata FastLoad utility INMOD routines within Teradata PT. The term INMOD is an acronym for “input modification.” INMOD routines are user exit routines the FastLoad INMOD adapter can call to provide enhanced processing functions on input data records before they are sent to the Teradata Database.

You can use the FastLoad INMOD Routine with the FastLoad INMOD Adapter to read and preprocess input data values from files on the client system. These then provide the USING data for a subsequent SELECT statement. For information on creating and using FastLoad INMOD Routines, see *Teradata FastLoad Reference*.

**Note:** The FastLoad INMOD Adapter operator does not support the INMOD routines for any other Teradata standalone load or unload utility.

*Figure 50* shows the FastLoad INMOD Adapter operator interface.

*Figure 50: FastLoad INMOD Adapter Operator Interface*
FastLoad INMOD Adapter Operator Definition

Following is an example of a FastLoad INMOD operator definition.

Only key attributes are shown (in bold).

```sql
DEFINE OPERATOR FASTLOAD INMOD ADAPTER OPERATOR
DESCRIPTION 'FASTLOAD INMOD ADAPTER OPERATOR'
TYPE FASTLOAD INMOD
SCHEMA PRODUCT_SOURCE_SCHEMA
ATTRIBUTES
  ( VARCHAR InmodName,
    VARCHAR FileName,
    VARCHAR Format,
    VARCHAR PrivateLogName
  );
```

For required syntax and descriptions of all FastLoad INMOD Adapter operator attributes, their capabilities and valid values, see Teradata Parallel Transporter Reference.

For descriptions of typical uses of the FastLoad INMOD operator, see Chapter 5: “Moving External Data into Teradata Database”.

Key Specifications

**InmodName**

Attribute that specifies the name of the FastLoad INMOD routine.

For details, see Teradata Parallel Transporter Reference.

**PrivateLogName**

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the diagnostic trace messages produced by the operator.

For details, see Teradata Parallel Transporter Reference.

**Format**

The internal format of each input record with the FastLoad INMOD Adapter operator must conform to the schema.

The FastLoad INMOD Adapter operator requires each field of an input record to a certain length. When no length is specified, the operator assumes the conventional default length.

The FastLoad INMOD Adapter operator supports both fixed- and variable-length input records. For records with a two-column schema such as:

```sql
  ( COL1 INTEGER,
```

...
each record must be comprised of:

- A four-byte integer, because 4 is the INTEGER default,
- A two-byte length indicator for the VARCHAR field, and
- The CHAR data of the length indicated in the two-byte length indicator field.

The following examples show the byte order when Teradata PT is running on an Intel processor.

- For example, if the integer has a decimal value of 100 and the VARCHAR is ASCII '12345', then the input record appears in hexadecimal notation, as:

  X'64 00 00 00 05 00 31 32 33 34 35'

- If the data in the input record is in indicator mode, there is a single preceding indicator byte:

  X'00 64 00 00 00 05 00 31 32 33 34 35'

**Job Options**

The FastLoad INMOD Adapter operator provides the same functionality and is used in the same circumstances as the DataConnector operator.

**Operational Considerations**

The FastLoad INMOD Adapter operator provides the same functionality and is used in the same circumstances as the DataConnector operator.
This chapter provides usage information for the most commonly specified Load operator features.

Topics include:
- Load Operator Capabilities
- FastLoad Utility and the Load Operator
- Load Operator Definition
- Key Specifications
- Job Options
- Operational Considerations

**Load Operator Capabilities**

The Load operator, a consumer operator, uses Teradata FastLoad protocol to load a large volume of data at high speed into an empty table in Teradata Database.

The Load operator is typically used for initial loading of Teradata tables as it inserts data it consumes from data streams into individual rows of a target table.

**Load Operator Interface**

Figure 51 shows the Load operator interface.

Figure 51: Load Operator Interface
The Load Operator Function in a Teradata PT Job

When you use the Load operator in a Teradata PT job, Teradata PT directs each parallel instance of the Load operator to do the following:

1. Log on to the Teradata Database, using your user name, password, Teradata Database, and account ID information specified in the job script.
2. Load the input data into the Load target table on the Teradata Database.
3. Log off the Teradata Database.
4. If the Load operator job is successful, terminate the job and provide information about the job in the log or to the user, such as:
   - Total number of records read and sent to the Teradata Database
   - Number of errors posted to the error tables
   - Number of inserts applied
   - Number of duplicate rows
   - A status code indicating the success of the job
5. If the job is unsuccessful, terminate the job and provide information about the job so that you can correct any problem and restart the job.

Load Operation Phases

Load operations have two phases:

- **Acquisition Phase** - Data from the input stream is transmitted to the AMPs. The AMPs 'acquire' the data. The acquisition phase is not complete until all data sources are exhausted and all data rows are on the appropriate AMPs. The data is not yet sorted or blocked, and therefore, is not yet accessible.
- **Application Phase** - Data is sorted, blocked, and put into its final format. All activity in this phase is AMP-local. Data is accessible after the completion of this phase.

FastLoad Utility and the Load Operator

Table 10 lists the operating features and capabilities of the Teradata FastLoad utility, and indicates whether they are supported by the Load operator.

<table>
<thead>
<tr>
<th>FastLoad Utility Feature</th>
<th>Teradata PT Operator Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access Modules</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>ANSI Date</td>
<td>Supported</td>
</tr>
<tr>
<td>Checkpoint/Restart</td>
<td>Supported</td>
</tr>
</tbody>
</table>
Following is an example of a Load operator definition.

Only key attributes are shown (in bold).

```sql
DEFINE OPERATOR LOADOP
DESCRIPTION 'Teradata PT LOAD OPERATOR'
TYPE LOAD
SCHEMA INPUT_SCHEMA
ATTRIBUTES
  (Character Sets Supported, using the USING CHARACTER SET clause before the DEFINE JOB statement
  Configuration File Supported, using the `tbuild` command line job attribute option (-v)
  CREATE TABLE Statement Supported, using the DDL operator
  DATABASE Statement Supported
  DELETE Statement Supported, using the DDL operator
  DROP TABLE Statement Supported, using the DDL operator
  Error Limit Supported
  Indicator Mode Supported, using the DataConnector operator
  INMOD Routines Supported, using the FastLoad INMOD Adapter operator
  Maximum/Minimum Sessions Supported
  Nonindicator Mode Supported, using the DataConnector operator
  Notify Supported
  NULLIF Clauses Supported, but not the XB, XC, and XG data types
  Operating System Command Supported, using the OS Command operator
  Record Formats Supported, using the DataConnector operator
  RECORD n THRU m Supported, in a limited form by the DataConnector operator, allowing you to read in the first "m" rows of a file, effectively allowing "RECORD 1 THRU m"
  Show Version Information Supported
  Tenacity Supported
  Wildcard INSERT Supported

Table 10: Load Operator Supported Features (continued)

<table>
<thead>
<tr>
<th>FastLoad Utility Feature</th>
<th>Teradata PT Operator Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character Sets</td>
<td>Supported, using the USING CHARACTER SET clause before the DEFINE JOB statement</td>
</tr>
<tr>
<td>Configuration File</td>
<td>Supported, using the <code>tbuild</code> command line job attribute option (-v)</td>
</tr>
<tr>
<td>CREATE TABLE Statement</td>
<td>Supported, using the DDL operator</td>
</tr>
<tr>
<td>DATABASE Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>DELETE Statement</td>
<td>Supported, using the DDL operator</td>
</tr>
<tr>
<td>DROP TABLE Statement</td>
<td>Supported, using the DDL operator</td>
</tr>
<tr>
<td>Error Limit</td>
<td>Supported</td>
</tr>
<tr>
<td>Indicator Mode</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>INMOD Routines</td>
<td>Supported, using the FastLoad INMOD Adapter operator</td>
</tr>
<tr>
<td>Maximum/Minimum Sessions</td>
<td>Supported</td>
</tr>
<tr>
<td>Nonindicator Mode</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>Notify</td>
<td>Supported</td>
</tr>
<tr>
<td>NULLIF Clauses</td>
<td>Supported, but not the XB, XC, and XG data types</td>
</tr>
<tr>
<td>Operating System Command</td>
<td>Supported, using the OS Command operator</td>
</tr>
<tr>
<td>Record Formats</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>RECORD n THRU m</td>
<td>Supported, in a limited form by the DataConnector operator, allowing you to read in the first &quot;m&quot; rows of a file, effectively allowing &quot;RECORD 1 THRU m&quot;</td>
</tr>
<tr>
<td>Show Version Information</td>
<td>Supported</td>
</tr>
<tr>
<td>Tenacity</td>
<td>Supported</td>
</tr>
<tr>
<td>Wildcard INSERT</td>
<td>Supported</td>
</tr>
</tbody>
</table>
Key Specifications

Defining a Load Operator in a Teradata PT Script

The Load operator definition must provide the following specifications:

- TYPE LOAD
- Key operator attributes:
  - TdpId
  - UserName
  - UserPassword
  - TargetTable
  - LogTable
  - ErrorTable1
  - ErrorTable2
  - PrivateLogName

Note: The first three key attributes listed above are “security attributes.”

In addition, other optional operator attributes can be declared depending on how the operator will be used.

For more information about all Load operator attributes, their capabilities, and valid values, see “Load Operator” in Teradata Parallel Transporter Reference.

Security Attributes

**TdpId**

Optional attribute that specifies the name of the Teradata Database machine (non-mainframe platforms) or TDP (mainframe platforms) for the Load operator job.
If you do not specify the value for the TdpId attribute, the operator uses the default TdpId established for the user by the system administrator.

For details, see “Teradata Database Logon Security” on page 73.

**UserName**

Attribute that specifies the Teradata Database user name.

For details, see “Teradata Database Logon Security” on page 73.

**UserPassword**

Attribute that specifies the password associated with the user name.

For details, see “Teradata Database Logon Security” on page 73.

**TargetTable**

A target table must already exist on a Teradata Database and be empty, with no defined secondary or join indexes, before a Load job is run.

For target tables defined as NoPI, the restriction that the target table must be empty does not apply.

**LogTable**

A restart log table, which contains restart information written during job runs, is required for any execution of the Load operator. Specify a restart log table in scripts with the LogTable attribute.

Restarts (as discussed in “Staged Loading” on page 192) are common during staged loading operations. When additional data is available after a job is paused, the job is restarted by submitting a second script that specifies the additional data. The Load operator recognizes the job as a restart, reads the restart log to determine the status of the job, then loads the additional file.

Restarts can also occur following any unexpected error on a database. For example, if a table runs out of available space during a load operation, the job terminates, the table is paused, and a checkpoint is recorded in the restart log. Pausing the job in this way allows you to manually increase the available space, if needed, then restart the job because the load operation can restart a job from the last checkpoint in the restart log.

**ErrorTable1**

Load operations create an error table that captures errors during job runs. Jobs can use the default names of the error tables, or names can be user-specified as an attribute of the Load operator.

Error Table1 contains most of the errors connected with data and the data environment. The following types of errors are captured:

- **Constraint violations** - Records that violate a range or value constraint defined for specific columns of a table.
• **Unavailable AMP** - Records to be written to a non-fallback table about an offline AMP.
• **Data conversion errors** - Records that fail to convert to a specified data type.

### ErrorTable2

Load operations create a duplication error table that captures errors during job runs. Jobs can use the default names of the error tables, or names can be user-specified as an attribute of the Load operator.

Error Table2 contains all of the rows that have violations of the unique primary index. This error table is not used when the target table has a non-unique primary index.

### Auto-Generation of Error Tables

Some tables must be created by the user before the job begins and some are created during the execution of a job. A target table must exist on the Teradata Database when the Load operator job is executed.

If you have not specified error tables (specifying them is optional), the Load operator automatically creates names of the error tables as follows:

- The first error table is `ttname_ET`
- The second error table is `ttname_UV`

where `ttname` is the name of the corresponding target table.

**Warning:** The value of the TargetTable attribute is truncated to 27 characters before the suffixes "_ET" and "_UV" are appended. This means that if the value of the TargetTable attribute is a fully qualified table name and that fully qualified name exceeds the maximum supported length of a database object (30 characters), the generated names for the error tables may not be what you intend. In such a situation, Teradata recommends that you provide the names of the error tables and not rely on the Load operator to generate the names for these tables automatically.

For example, if the following is specified when no error table specifications exist,

```sql
VARCHAR TargetTable = 'ttname'
```

the Load operator creates the following error tables:

```sql
targetable_ET
targetable_UV
```

### PrivateLogName

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the messages produced by the operator.

By default, no diagnostic trace messages are produced. Diagnostic trace messages are produced only when the user sets a valid value for the TraceLevel attribute.

For details, see *Teradata Parallel Transporter Reference*. 
Job Options

Duplicate Rows

Duplicate rows, which are exact duplicates of existing table rows, are never inserted, even if the target table is defined as a multiset table, which usually permits duplicate rows. Duplicate row violations are thus not captured in either Error Table 1 or Error Table 2. Instead, they are counted and reported in the status log at the end of a job.

If the taget table is defined as a NoPI table, duplicate rows are inserted. NoPI tables are inherently multiset since no duplicate row checking is possible; with NoPI tables, duplicate rows can be on different AMPs. Therefore, no duplicate row elimination is performed.

If a table has a unique primary index, a circumstance where there is duplicate row takes precedence over a duplicate primary index error, meaning that the offending row is counted and reported, but it is not captured in Error Table 2.

ErrorLimit

While loading large amounts of data, it is possible that a single data error might be repeated for each input record. Because an error can often be corrected long before errors are generated for all of the records in a job run, consider using the ErrorLimit attribute to specify a limit to the number of errors that can be tolerated before a job is terminated.

This limit, which is specified with the ErrorLimit attribute, represents the total number of errors written to the first error table per instance of the Load operator, not to all instances combined. Therefore, if an error limit is set to 1,000, a single load instance must detect that 1,000 rows are inserted into the error table before the job is terminated.

The error limit can also be reached at checkpoint time; see the examples below.

Error Limit Examples

To illustrate how the Load operator determines if the number of errors has reached the Error Limit, consider these examples if there are two instances running and the Error Limit has been set to 1000.

- If either instance by itself reaches 1000, it will terminate the job by returning a fatal error.
- If instance #1 processes 500 error rows and instance #2 processes 500 error rows but does not reach a checkpoint. The job will continue processing.
- If instance #1 processes 500 error rows and instance #2 processes 500 error rows but does reach a checkpoint. The total number of error rows for all instances combined is determined at checkpoint time and at the end of the Acquisition Phase. If the total of all instances exceeds the error limit at that time, the job will terminate with an error.
Staged Loading

Staged loading is the ability to pause an active load operation until additional data is available. If a single table needs to be filled with the contents of three files, for example, usually all three files are streamed to look like a single source to Teradata PT. But if one of the files will not exist until the next day, it is possible to load in stages. In other words, Teradata PT can load the first two files, pause the Load operation, and then load the third file when it is available.

Staged loading is set by the attribute `PauseAcq = 'Y'`, which prevents the Load operator from proceeding to the application phase. Each stage of the load operation is accomplished with a separate job script: one for the acquisition phase, and one for the application phase.

For example, to accomplish the scenario with three files (one of which is unavailable until the next day), run Job1 on Day 1 using the two existing files as input to the load, with the `PauseAcq = 'Y'` setting.

When this stage of the job is finished, the target table is paused and becomes inaccessible to users. Attempts to access the target table (or the error tables) return the following error message:

```
Operation Not Allowed <tablename> is being loaded
```

On Day 2, restart the paused job by running Job2 using the third, newly available file as input. For this stage, set `PauseAcq = 'N'`. When this stage of the job finishes, the table is fully loaded and ready for access.

**Caution:** A paused table, though inaccessible, can be dropped.

Operational Considerations

Performance

Multiple parallel instances can be used to improve the performance of the load. Multiple instances of the Load operator can be run in a job, but all of them load data into the same table.

During a Load job, the Load operator inserts the data from each record it receives from the data streams into one row of the target table on Teradata Database. Because Teradata PT can filter out rows based on criteria in the script, some input rows may not be loaded.

The Load operator does not support Update, Select, and Delete operations on the target table. The Load operator only support Insert operations.

Space Requirements

Always estimate the final size of the Load target table to ensure that the destination on a Teradata Database has enough space to accommodate a Load operator job.

If the system that owns a Load target table, log table, or error tables runs out of space, Teradata Database returns an error message, then the Load operator job terminates. Additional space must be allocated to the database before the job can be restarted.
Sessions and Instances

A minimum and a maximum number of sessions (the session limits) can be specified for the Load operator. Both can also be specified for the Export and Update operators.

Consider the following usage notes which apply to all three types of operators.

- The maximum sessions connected can never exceed the number of available AMPs in the system, even if a larger number is specified.
- The default is one session per available AMP.
- For the MinSessions attribute, the minimum specification is one.
- The MaxSessions attribute can be set to a number smaller than the number of AMPs on the database server if fewer sessions are suitable for the job.
- Network protocol software might also impose limits on network-attached systems.
- Platform limits for maximum sessions per application differ:
  - On channel-attached z/OS client system, use the TDP SET MAXSESSIONS command to specify a platform limit.
  - On network-attached UNIX, Linux, and Windows client systems, this value is defined in the CLI file, clispb.dat, under the max_num_sess variable.
  - On channel-attached z/OS client systems, this value is defined in the HSHSPB parameter under the IBCSMAX setting.

The max_num_sess value in the clispb.dat file (or HSHSPB) specifies the total number of sessions allowed to be connected by a single application at one time. For the Export, Load, and Update operators, two types of sessions are counted toward the max_num_sess value: SQL sessions, and specialized data loading sessions that have a restriction of one per AMP.

The max_num_sess pertains to all sessions connected, both SQL and data loading.

For details, see “Optimizing Job Performance with Sessions and Instances” on page 80.

Limits on Load Jobs

The Load operator requires one designated Teradata Database loader task.

The number of concurrent load tasks is, however, configurable in the Teradata Database environment using the same MaxLoadTasks field control used by FastLoad, FastExport, and MultiLoad. For example, one FastLoad job equates to one Teradata PT Load job in the count for MaxLoadTasks. For more information, see “DBS Control Utilities” in Utilities - Volume 1.

Caution: Simultaneously running many Teradata PT jobs might impact other running Teradata Database processes and applications.

Checkpointing and Restarting

The Load operator takes checkpoints at the beginning and end of the acquisition phase. More granular checkpoints during the acquisition phase can be specified using the command line option -z when running Teradata PT using the tbuild command. The -z option specifies checkpoint intervals in terms of seconds.
The following command string is an example of the -z option:

```
tbuild -f <file name> -z 30
```

In this command, the -f option indicates the script that is input to `tbuild`, and the -z option indicates that a checkpoint will be taken every 30 seconds.

The `DEFINE JOB` statement can also be used to specify a checkpoint value. For more information, see “DEFINE JOB” in *Teradata Parallel Transporter Reference*.

Checkpointing during the application phase is managed internally by the Teradata Database, and therefore is *not* user-controlled.

For information on checkpointing and restarting, see Chapter 24: “Troubleshooting a Failed Job.”

---

**Load Operator as Standalone Operator**

In addition to its other capabilities, the Load operator can function as a standalone operator and supports an `APPLY` statement with no `SELECT` statement and no `INSERT` statement. To use LOAD as a standalone operator, use one of the following definitions:

```
TYPE LOAD STANDALONE
TYPE LOAD
```

**Note:** The STANDALONE keyword is optional.

In the following example, the STANDALONE keyword is omitted:

```
DEFINE JOB LOAD_USER_DATA
(
    DEFINE OPERATOR LOAD_OPERATOR
    TYPE LOAD
    (    
        .
        .
    );
    .
    .

    APPLY
    TO OPERATOR (LOAD_OPERATOR[1]);
);
```

**Note:** Use LOAD as a standalone operator to apply data on the target table without sending more data.
This chapter provides usage information for the most commonly specified MultiLoad INMOD Adapter operator features.

Topics include:
- MultiLoad INMOD Adapter Operator Capabilities
- MultiLoad INMOD Adapter Operator Definition
- Key Specifications

**MultiLoad INMOD Adapter Operator Capabilities**

The MultiLoad INMOD Adapter operator allows you to use Teradata MultiLoad utility INMOD routines with Teradata PT. This operator can be used as a producer or a filer operator. The term INMOD is an acronym for “input modification.” INMOD routines are user exit routines the MultiLoad INMOD adapter can call to provide enhanced processing functions on input records before they are sent to the Teradata Database. You can use the MultiLoad INMOD Routine with the MultiLoad INMOD Adapter to read and preprocess input data values from files on the client system. These then provide the USING data for a subsequent SELECT statement. For information on creating and using FastLoad INMOD Routines, see *Teradata MultiLoad Reference*.

Figure 52 and Figure 53 show the MultiLoad INMOD Adapter operator as a producer operator or as a filter operator.

Figure 52: MultiLoad INMOD Adapter Operator as Producer Operator
MultiLoad INMOD Adapter Operator Definition

Following is an example of a MultiLoad INMOD Adapter operator definition.

Only key attributes are shown (in bold).

```
DEFINE OPERATOR MULTILOAD INMOD ADAPTER
DESCRIPTION 'MULTILOAD INMOD ADAPTER'
TYPE MULTILOAD INMOD
SCHEMA PRODUCT_SOURCE_SCHEMA
ATTRIBUTES

( VARCHAR InmodName,
  VARCHAR FileName,
  VARCHAR Format,
  VARCHAR PrivateLogName
);
```

For required syntax and descriptions of all MultiLoad INMOD Adapter operator attributes, their capabilities and valid values, see Teradata Parallel Transporter Reference.

For descriptions of typical uses of the MultiLoad INMOD Adapter operator, see Chapter 5: “Moving External Data into Teradata Database”.

**Key Specifications**

**InmodName**

Required attribute that specifies the name of the MultiLoad INMOD routine.

When you specify the name in a Teradata PT job script, it is the named routine rather than a data source that provides the input data records.

For details, see Teradata Parallel Transporter Reference.
**Format**

The internal format of each input record intended for use with the MultiLoad INMOD Adapter operator must conform to the specified schema.

The MultiLoad INMOD Adapter operator requires each field to be of a specified length. When no length is specified, the operator assumes the conventional default length.

The MultiLoad INMOD Adapter operator supports both fixed- and variable-length input records.

For records with a two-column schema such as:

```java
(COL1 INTEGER,
 COL2 VARCHAR(10));
```

each record is comprised of:

- A four-byte integer, because 4 is the INTEGER length,
- A two-byte length indicator for the VARCHAR field, and
- The CHAR data of the length indicated in the two-byte length indicator field.

The following examples show the byte order when running Teradata PT on an Intel processor:

- If, for example, the integer has a decimal value of 100 and the VARCHAR is ASCII '12345', then the input record appears in hexadecimal notation as:
  `X'64 00 00 00 05 00 31 32 33 34 35'`
- If the data in the input record is in indicator mode, a single precedes the indicator byte:
  `X'00 64 00 00 00 05 00 31 32 33 34 35'`

**PrivateLogName**

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the diagnostic trace messages produced by the operator.

For details, see *Teradata Parallel Transporter Reference*. 
This chapter provides usage information for the most commonly specified ODBC operator features.

Topics include:

- ODBC Operator Capabilities
- ODBC Operator Definition
- Key Specifications
- Job Options
- Operational Considerations

**ODBC Operator Capabilities**

The ODBC operator is a producer operator that enables universal open data access to retrieve data from many ODBC compliant data sources. This interface is certified for:

- Oracle
- SQL Server
- DB2

The ODBC operator is, however, expected to work with all ODBC-compliant sources.

The ODBC operator can read data close to the source (from the same machine on which Teradata PT is running, as opposed to across a network), and then feed the data (via a data stream) directly to the Teradata Database without using any intermediate storage.

Multiple ODBC operators can read from an Oracle source table using different key ranges (to provide data parallelism), use the UNION ALL statement to combine the source data, and then use multiple instances of the Load operator to write to the same Teradata Database table.

**Note:** Use of the ODBC operator requires that a copy of the DataDirect ODBC driver, version 5.3, be installed on the system from which the Teradata PT job script is run.

The ODBC operator also provides parallel access to different data sources. For example, in a single job script, you can use multiple ODBC operators to read from multiple data sources (for example an Oracle server, a Microsoft SQL server, Microsoft Access, Teradata), use the UNION ALL statement to combine the imported data, and then use multiple instances of the Update operator to insert/update multiple tables in the Teradata Database.

**Figure 54** shows the ODBC operator interface.
The ODBC Operator Function in a Teradata PT Job

When you use the ODBC operator in a Teradata PT job, Teradata PT directs one instance of the operator to do the following:

1. Log on to a SQL session with an ODBC-compliant data source.
2. Send the SELECT request specified in the operator attribute 'SelectStmt' to the data source.
3. Retrieve the data returned from the data source.
4. Send data to the Teradata PT data streams.
5. Log off the SQL session.
6. If the ODBC operator job is successful, terminate the job and report the total number of records exported from Teradata Database.
7. If the job is unsuccessful, terminate the job and provide information about the job so that you can correct any problem and restart the job.

The ODBC operator does the following:

- Accesses many ODBC-compliant data sources, for example, Oracle, SQL Server, DB2, etc.
- Runs on all Teradata PT supported platforms
- Reads data close to the sources (from the same machine where Teradata PT is running, as opposed to across a network)
- Feeds data (via a data stream) directly to the Teradata Database without the need of an intermediate staging platform.

ODBC Operator Definition

Following is a simplified example of defining an ODBC operator.

Only key attributes are shown (in bold).

```
DEFINE OPERATOR ODBC_OPERATOR
  DESCRIPTION 'ODBC_OPERATOR'
  TYPE ODBC
  SCHEMA PRODUCT_SOURCE_SCHEMA
  ATTRIBUTES
```
Key Specifications

The ODBC Operator in a Teradata PT Job Script

To define an ODBC operator in a job script, provide the following required specifications:

- TYPE ODBC
- Key operator attributes
  - UserName
  - UserPassword
  - DSNName
  - SelectStmt
  - PrivateLogName

**Note:** The first two key attributes listed above are “security attributes.”

In addition, optional operator attributes can be declared depending on how the operator will be used.

For more information about all ODBC operator attributes, their capabilities, and valid values, see *Teradata Parallel Transporter Reference*.

Security Attributes

**UserName**

Optional attribute that specifies the user name of the account or database in the data source.

If not specified, then the ODBC driver looks in the initialization file for the user name information.

For details, see “Teradata Database Logon Security” on page 73.

**UserPassword**

Optional attribute that specifies the password associated with the UserName of the account or database in the data source.
If not specified, then the ODBC driver will look in the ODBC initialization file for the user password information.

For details, “Teradata Database Logon Security” on page 73.

**DSNName**

Optional attribute that specifies the name of the data source.

Only the system data source name can be used. User data source name does not work.

For details, see *Teradata Parallel Transporter Reference*.

**SelectStmt**

When using the SelectStmt attribute to define the ODBC operator, multistatement SELECTs are allowed as long as the ODBC driver supports them.

The ODBC operator does not check the attribute for the presence of a multistatement SELECT.

If there is a multistatement SELECT, and the ODBC driver does not support it, the driver returns an error (although the error might not state that a multistatement SELECT was attempted) and the job terminates.

Also, the schema returned from the SELECT must match the schema as defined in the Teradata PT job script.

For detail, see *Teradata Parallel Transporter Reference*.

**Caution:** Oracle 9.2.0.1.0 incorrectly processes 1 MB SELECT statements, and might cause a loss of data. For more information, consult your Teradata field representative.

**PrivateLogName**

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the diagnostic trace messages produced by the operator.

For details, see *Teradata Parallel Transporter Reference*.

**Job Options**

**Multiple Parallel Instances**

The ODBC operator does not support multiple parallel instances.

**Rules for Handling Decimal Data**

The following rules apply to the ODBC Operator running on z/OS, UNIX, and Windows platforms, when taking DECIMAL data from non-Teradata Databases (such as Oracle) and moving it to Teradata Database.
If the precision values between the source and target tables are not equal, the data will be checked at runtime. If the data does not fit, the job will be terminated.

If the scale factors do not match, one of the following will happen:

- If the source scale factor is less than the target, then Teradata PT will add 0's (zeros) to the end of the source scale factor until they do match, and then the job will continue.
- If the source scale factor is greater than the target, the job will terminate before attempting to process the data, due to possible loss of data.

For source column NUMBER \((m, n)\) and target column DECIMAL \((M, N)\), the job will do the following:

- Terminate with error if \(n > N\)
- Run if \(n < N\) or \(n = N\)
- Run if \(m <= M\), assuming \(n < N\) or \(n = N\)
- Attempt to run if \(m > M\), assuming \(n < N\) or \(n = N\)

**TruncateData**

Attribute that specifies whether data is to be truncated, when it arrives from the data source, to fit into the Teradata-type of column.

For details, see *Teradata Parallel Transporter Reference*.

**Using ODBC Operator with Oracle DATE Fields**

The following limitations apply when the ODBC Operator runs on z/OS, UNIX, and Windows platforms.

- Whether the TruncateData flag is ON or OFF, the ODBC operator will truncate the Oracle Date to Teradata Date if the target column is mapped to INTDATE or ANSIDATE. If you don’t want to truncate the Oracle Date then you must map the target column to TIMESTAMP or CHAR(19).
- Normal "DATE" fields in Oracle cannot copy correctly over to Teradata’s DATE field (in ANSI mode).
- On some non-Teradata Databases, the DATE column type does not equate to Teradata’s ANSI Date field.

For example, on Oracle, the column type DATE is equivalent to the column type DATETIME, which has a format of YYYY-MM-DD HH:MM:SS (equivalent to a CHAR(19)). Teradata’s ANSI DATE has a format of YYYY-MM-DD (equivalent to CHAR(10)).

**Configuring ODBC Initialization Files**

Before using the ODBC operator, you must configure the ODBC initialization file on your platform to designate the DSN (data source name) in 'ODBC Data Sources' to point to the proper data source.
Following is a sample `ODBC.ini` file for a Solaris client platform. It shows data source information for Teradata, ORACLE, and SQLServer data sources.

```
[ODBC]
InstallDir=/usr/odbc
Trace=0
TraceFile=/usr/joe/odbcusr/trace.log
TraceAutoStop=0

[ODBC Data Sources]
testdsn=tdata.so
default=tdata.so
Teradata=tdata.so
Oracle=DataDirect 4.10 Oracle
SQLServer Wire Protocol=DataDirect 4.10 SQL Server Wire Protocol

[testdsn]

[default]

[Teradata]

[Oracle]
Driver=/opt/odbc/lib/ivor818.so
Description=DataDirect 4.10 Oracle
LogonID=oracle
Password=oracle
ServerName=DBCName=200.001.001.01
CatalogOptions=0
ProcedureRetResults=0
EnableDescribeParam=0
EnableStaticCursorsForLongData=0
ApplicationUsingThreads=1

[SQLServer Wire Protocol]
Driver=/opt/odbc/lib/ivmsss18.so
Description=DataDirect 4.10 SQL Server Wire Protocol
Database=db
LogonID=uid
Password=pwd
Address=sqlserverhost,1433
QuotedId=No
 AnsiNPW=No
```

**Prerequisites**

Because using the ODBC operator in a Teradata PT job script requires:

- That a copy of the ODBC.INI (initialization) file has been installed on your Client platform, and
- That the ODBC.INI (initialization) file has been configured on your Client platform to specify the:
  - DSNName
  - HostName
  - ServiceName.
This enables your Teradata PT script (through the DSNName in the ODBC.INI file and the same DSNName attribute for the ODBC operator) to know that name of the system on which the Oracle database resides (identified in the ODBC.INI file by the Oracle HostName and ServiceName).

For the procedure for configuring the ODBC.INI file on your Client platform with the DSNName and the Oracle HostName and ServiceName, see chapter 2.

**Using the ODBC Operator on HP-UX**

Observe the following limitations when using the ODBC operator on the HP-UX platform:

- The user *must* set the LD_PRELOAD environment variable to the path and name of the pthread library. For example:
  
  ```
  LD_PRELOAD=/usr/lib/libpthread.1
  ```

- ODBC operator cannot run simultaneously with the Named Pipes Access Module due to a conflict in the setting of an environment variable associated with the ODBC Driver Manager.

**ODBC Operator and Teradata Database**

Even though this operator can communicate with the Teradata Database, it is more efficient to use either the Export operator or the SQL Selector operator to extract data from the Teradata Database because these operators use the Teradata Call-Level Interface (CLIv2).

The Export operator is the fastest way of extracting data from the Teradata Database.

**Operational Considerations**

**Checkpointing and Restarting**

The ODBC operator does not support checkpoint and restart operations because it is unknown how the databases it can connect to handle restarts.

For information on checkpointing and restarting, see Chapter 24: “Troubleshooting a Failed Job.”
CHAPTER 16

OS Command Operator

This chapter provides usage information for the most commonly specified OS Command
operator features.

Topics include:

• OS Command Operator Capabilities
• OS Command Operator Definition
• Key Specifications

OS Command Operator Capabilities

OS Command is a standalone operator that executes operating system commands on the
client system that runs the job.

Any output from executed commands is written to the job log.

For Teradata PT jobs running on z/OS, the OS Command operator has limited functionality.
It cannot execute z/OS system commands, although it can execute USS commands.

Figure 55 shows the OS Command operator interface.

Figure 55: OS Command Operator Interface

The OS Command Operator Function in a Teradata PT Job

When you use the OS Command operator in a Teradata PT job, Teradata PT directs one
instance of the operator to:

1. Spawn an OS command.

2. Read the output from the OS command.

3. Send the output to the Teradata PT Logger.
OS Command Operator Definition

Following is an example of an OS Command operator definition. Only key attributes are shown (in bold).

```
DEFINE OPERATOR OS_COMMAND
DESCRIPTION 'OS COMMAND OPERATOR'
TYPE OS COMMAND
ATTRIBUTES
  ( VARCHAR OsCmd,
    VARCHAR IgnoreError,
    VARCHAR PrivateLogName
  );
```

For required syntax and descriptions of all OS Command operator attributes, their capabilities and valid values, see Teradata Parallel Transporter Reference.

For descriptions of typical uses of the OS Command operator, see Chapter 5: “Moving External Data into Teradata Database”.

Key Specifications

**OSCmd**

Name of the operating system command.

To specify more than one command, use the following syntax:

```
VARCHAR OSCMD = ['cmd1', 'cmd2', 'cmd3'...]
```

Each command is executed in a separate call to the operating system.

For details, see Teradata Parallel Transporter Reference.

**IgnoreError**

Optional attribute that specifies whether to ignore an error that is returned from the operating system or to terminate.

Valid values are:

- 'Yes' (or 'Y') = Continue the Teradata PT job when an error returns from the execution of the operating system execution (default).
- 'No' (or 'N') = Terminate the Teradata PT job when an error returns from the execution of the operating system.

For details, see Teradata Parallel Transporter Reference.
PrivateLogName

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the diagnostic trace messages produced by the operator.

For details, see Teradata Parallel Transporter Reference.
This chapter provides usage information for the most commonly specified SQL Inserter operator features.

Topics include:

- SQL Inserter Operator Capabilities
- SQL Inserter Operator and the Load Operator
- SQL Inserter Operator Definition
- Key Specifications
- Job Options
- Operational Considerations

**SQL Inserter Operator Capabilities**

The SQL Inserter operator is a consumer operator inserts the data into a Teradata Database table. An SQL Inserter operation is similar in function to a BTEQ import operation.

The SQL Inserter operator supports multiple instances and each instance logs on as a single SQL session. To take advantage of the parallel processing of Teradata PT, users can specify multiple instances, which means multiple session, in order to insert data records simultaneously into a target table on Teradata Database. A target table can be an empty table or a table with existing data in it.

Figure 56 shows the SQL Inserter operator interface.

Figure 56: SQL Inserter Operator Interface
The SQL Inserter Operator Function in a Teradata Job Script

When you use the SQL Inserter operator in a Teradata PT job, it does the following:

1. Log on to the Teradata Database, using your user name, password, Teradata Database, and account ID information specified in the job script.
2. Load the data from the Teradata PT data stream into a target table specified in the SQL INSERT statement defined in the Teradata PT APPLY statement. Only one INSERT statement is allowed to be specified in the APPLY statement. If more than one INSERT statement is found in the APPLY statement, the SQL Inserter operator will issue an error message to both console and private logs and terminate the job.
3. Log off from the session with the Teradata Database.
4. If the SQL Inserter operator job is successful, terminate the job and report the total number of records successfully sent to the Teradata Database.
5. If the job is unsuccessful, terminate the job and provide information about the job so that you can correct any problem and restart the job.

SQL Inserter Operator and the Load Operator

To load large amounts of data, it is generally better to use the Load operator, but for smaller load jobs, the SQL Inserter might perform better than most load jobs because it does not need to set up multiple sessions to run.

Also, the SQL Inserter operator does not require an active load job. It simply uses standard SQL protocol on a single session. If it is difficult to acquire DBS resources for running concurrent tasks, the SQL Inserter operator has advantages over the Load operator.

SQL Inserter Operator Definition

Following is an example of a SQL Inserter operator definition.

Only key attributes are shown (in bold).

```sql
DEFINE OPERATOR SQL_INSERTER
DESCRIPTION 'Teradata PT SQL Inserter Operator'
TYPE INSERTER
SCHEMA *
ATTRIBUTES
(
    VARCHAR Tdpid,
    VARCHAR UserName,
    VARCHAR UserPassword,
    VARCHAR PrivateLogName
);
```

For required syntax and descriptions of all SQL Inserter operator attributes, their capabilities and valid values, see Teradata Parallel Transporter Reference.
For descriptions of typical uses of the SQL Inserter operator, see Chapter 5: “Moving External Data into Teradata Database” and Chapter 7: “Moving Data within the Teradata Database Environment”.

Key Specifications

Defining an SQL Inserter Operator in a Teradata PT Script

The SQL Inserter operator definition must provide the following specifications:

- TYPE SQL INSERTER
- Key operator attributes:
  - TdpId
  - UserName
  - UserPassword
  - PrivateLogName

Note: The first three key attributes listed above are “security attributes.”

In addition, other optional operator attributes can be declared depending on how the operator will be used.

For more information about all Load operator attributes, their capabilities, and valid values, see “SQL Inserter Operator” in Teradata Parallel Transporter Reference.

Security Attributes

TdpId
Optional attribute that specifies the name of the Teradata Database machine (non-mainframe platforms) or TDP (mainframe platforms) for the Load operator job.

If you do not specify the value for the TdpId attribute, the operator uses the default TdpId established for the user by the system administrator.

For details, see “Teradata Database Logon Security” on page 73.

UserName
Attribute that specifies the Teradata Database user name.

For details, see “Teradata Database Logon Security” on page 73.

UserPassword
Attribute that specifies the password associated with the user name.

For details, see “Teradata Database Logon Security” on page 73.
Chapter 17: SQL Inserter Operator

Job Options

PrivateLogName

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the diagnostic trace messages produced by the operator.

For details, see *Teradata Parallel Transporter Reference*.

Job Options

Support for LOBs

For details, see “Large Objects” on page 366.

Data Integrity

To protect data integrity, the SQL Inserter operator treats the entire loading job as a single explicit transaction.

If any error is encountered during the insert operation, the SQL Inserter operator backs out all rows of data inserted up to that point. For example, the SQL Inserter operator immediately terminates a load job if a duplicate data row is inserted into a target table.

Data Loading

The SQL Inserter operator is not used for massive data loading because it is slower than the other load operators.

Operational Considerations

Checkpointing and Restarting

The SQL Inserter operator takes two basic checkpoints, a start-of-data checkpoint and an end-of-data checkpoint, during a load operation so that the SQL Inserter operator can restart automatically when the load operation is interrupted by a Teradata Database restart, should one occur.

When Teradata PT loads LOB data into a target table on Teradata Database from external data files, the SQL Inserter operator can take user-defined interval checkpoints during the load job.

For information on checkpointing and restarting, see Chapter 24: “Troubleshooting a Failed Job.”
This chapter provides usage information for the most commonly specified SQL Selector operator features.

Topics include:
- SQL Selector Operator Capabilities
- SQL Selector Operator and the Export Operator
- SQL Selector Operator Definition: Not for LOB Exporting
- SQL Selector Operator Definition: For LOB Exporting
- Key Specifications
- Job Options
- Operational Considerations

**SQL Selector Operator Capabilities**

The SQL Selector operator is a producer operator that submits a single Teradata SQL SELECT statement to a Teradata Database to retrieve data from a table.

Figure 57: SQL Selector Operator Interface

**SQL Selector Operator and the Export Operator**

The main differences between the SQL Selector operator and the Export operator are seen in their performance and feature sets.

- The Export operator allows multiple sessions and multiple instances to extract data from the Teradata Database. When exporting a large number of rows, it is usually better to use the Export operator.
Chapter 18: SQL Selector Operator

The SQL Selector operator only allows one session and one instance to extract rows from the Teradata Database. When exporting a small number of rows, the SQL Selector operator usually performs better.

The SQL Selector operator support of a single session and a single instance is similar to a BTEQ Export operation.

Advantages of the SQL Selector Operator

The SQL Selector operator has features not found in the Export operator such as field mode processing:

- The SQL Selector operator has a Report Mode, known as Field Mode in the BTEQ environment. All data retrieved in this mode is converted to character strings.
- The SQL Selector operator is the only operator that can retrieve data from Teradata Database in Field Mode and send data to the DataConnector operator in order to have it written to an external target in the VARTEXT (or TEXT) format. The Export operator cannot be used to extract data from a Teradata Database table and write it to an external target in VARTEXT (or TEXT) format.
- The SQL Selector operator can extract LOB data from Teradata Database.
- The SQL Selector operator does not require an active load job. Instead, standard SQL protocol is used on the single session. If it is difficult to acquire DBS resources for running concurrent tasks, the SQL Selector operator is a logical choice rather than the Export operator.

SQL Selector Operator Definition: Not for LOB Exporting

Following is an example of an SQL Selector operator definition.

Only key attributes are shown (in bold).

```
DEFINE OPERATOR SQL_SELECTOR
DESCRIPTION 'SQL OPERATOR'
TYPE SELECTOR
SCHEMA PRODUCT_SOURCE_SCHEMA
ATTRIBUTES
(   VARCHAR TdpId,
    VARCHAR UserName,
    VARCHAR UserPassword,
    VARCHAR SelectStmt,
    VARCHAR ReportModeOn,
    VARCHAR PrivateLogName
) ;
```

For required syntax and descriptions of all SQL Selector operator attributes, their capabilities and valid values, see Teradata Parallel Transporter Reference.
SQL Selector Operator Definition: For LOB Exporting

Following is an example of an SQL Selector operator definition when used for exporting Large Objects (LOBs).

Only key attributes are shown (in bold).

```sql
DEFINE OPERATOR SQL_SELECTOR
DESCRIPTION 'SQL OPERATOR'
TYPE SELECTOR
SCHEMA PRODUCT_SOURCE_SCHEMA
ATTRIBUTES
(
   VARCHAR TdpId,
   VARCHAR UserName,
   VARCHAR UserPassword,
   VARCHAR SelectStmt,
   VARCHAR PrivateLogName,
   VARCHAR LobDirectoryPath,
   VARCHAR LobFileBaseName,
   VARCHAR LobFileExtension
);
```

For required syntax and descriptions of all SQL Selector operator attributes, their capabilities and valid values, see *Teradata Parallel Transporter Reference*.

For descriptions of typical uses of the SQL Selector operator, see Chapter 6: “Moving Data from Teradata Database to an External Target” and Chapter 7: “Moving Data within the Teradata Database Environment”.

Key Specifications

**Defining an SQL Selector Operator in a Teradata PT Script**

The SQL Selector operator definition must provide the following specifications:

- TYPE SQL Selector
- Key operator attributes:
  - TdpId
  - UserName
  - UserPassword
  - SelectStmt
  - PrivateLogName
  - For LOB Exporting: LobDirectoryPath
  - For LOB loading: LobFileBaseName
  - For LOB loading: LobFileExtension
  - Not for LOB loading: ReportModeOn
**Note:** The first three key attributes listed above are “security attributes.”

In addition, other optional operator attributes can be declared depending on how the operator will be used.

For more information about all Load operator attributes, their capabilities, and valid values, see “SQL Inserter Operator” in *Teradata Parallel Transporter Reference.*

### Security Attributes

**TdpId**

Optional attribute that specifies the name of the Teradata Database machine (non-mainframe platforms) or TDP (mainframe platforms) for the Load operator job.

If you do not specify the value for the TdpId attribute, the operator uses the default TdpId established for the user by the system administrator.

For details, see “Teradata Database Logon Security” on page 73.

**UserName**

Attribute that specifies the Teradata Database user name.

For details, see “Teradata Database Logon Security” on page 73.

**UserPassword**

Attribute that specifies the password associated with the user name.

For details, see “Teradata Database Logon Security” on page 73.

### SelectStmt

A SELECT request is a Teradata SQL SELECT statement. Certain restrictions apply when using a Teradata SQL SELECT statement in the SQL Selector operator:

- Do not specify a WITH clause with a SELECT statement. Use of a WITH or WITH BY clause produces an error message.
- Do not specify multiple Teradata SQL SELECT statements. If you specify multiple select statements in the SelectStmt attribute, the operator produces an error message stating that multiple SELECT statements are not allowed, and the job terminates.
- Do not specify a USING modifier with a SELECT statement. Use of a USING modifier produces an error message.

**Note:** The SQL Selector operator does not support the use of multiple Teradata SQL SELECT statements because this offers no value. The SQL Selector operator logs on only one session with the Teradata Database. Multiple SELECT statements are therefore sent as only one request to the Teradata Database. The Teradata Database executes these multiple statements sequentially. Thus, the SQL Selector operator can not take advantage of the parallel processing environment that Teradata PT offers. Instead, break multiple Teradata SQL SELECT statement requests into single SELECT statements carried by multiple SQL Selector operators, and then unite these data sources together with the UNION clause.
For details, see *Teradata Parallel Transporter Reference*.

**PrivateLogName**

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the diagnostic trace messages produced by the operator.

For details, see *Teradata Parallel Transporter Reference*.

**LOB Loading Attributes**

The following attributes:

- LobDirectoryPath
- LobFileBaseName
- LobFileExtension

are only for extracting LOB data in deferred mode. That is, when LOB columns are defined as follows in a schema:

BLOB AS DEFERRED BY NAME  
CLOB AS DEFERRED BY NAME

**Job Options**

**Using the Large Decimal Teradata Database Feature**

The Export operator optional attribute *MaxDecimalDigits* allows Teradata PT to use the Large Decimal functionality of the Teradata Database. But what if *MaxDecimalDigits* is set and the Teradata Database or the CLIv2 does not support this feature? Using the *IgnoreMaxDecimalDigits* attribute allows the Teradata PT job to continue. This attribute is available to the Export and SQL Selector operators.

*Figure 58* shows the logic flow used to determine if the Teradata PT job will continue. The main decision points are:

- Is *MaxDecimalDigits* a valid value? Then 18 <= *MaxDecimalDigits* <= 38. If the value does not fall in this range, a warning message is issued and the job will continue.
- Next, if the *IgnoreMaxDecimalDigits* attribute is set, continue to determine if the Teradata Database and CLI versions support the Large Decimal feature.
  
The job will terminate with an Exit code of eight and an error message if  
*IgnoreMaxDecimalDigits* has not been set and either the Teradata Database or CLI levels do not support the Large Decimal feature.

If the Teradata Database and CLI versions do not support the Large Decimal feature and the *IgnoreMaxDecimalDigits* attribute is set, continue the job.
The Export and SQL Selector operators will continue the job when the Large Decimal feature is not supported because the maximum returned decimal digits is 18 which is in the valid range.

- If the versions are supported and the `MaxDecimalDigits` attribute is *not* a valid value, display a message and continue the job with an Exit code of four.
- If either the Teradata Database or CLI version do not support the Large Decimal feature, and `MaxDecimalDigits` is a valid value but the `IgnoreMaxDecimalDigits` attribute is not set, display a message and continue the job with an Exit code of four.

For complete information on the `MaxDecimalDigits` and `IgnoreMaxDecimalDigits` attributes, see the Teradata Parallel Transporter Reference.

Figure 58: How the Teradata Database Large Decimal Feature Affects Job Continuation
Operational Considerations

Checkpointing and Restarting

The SQL Selector operator takes a checkpoint only when all of the data is sent to the Teradata PT data stream.

Then, on restart, the operator either sends none of the data, all of the data, or terminates with an error message, depending on the status of the data:

- If all of the data is sent, then the operator displays the following message and does not resend any of the data:
  
  \textit{Restart indicates that this export job completed.}

- If all of the data is not sent, then the operator terminates. Restart the job from the beginning.

- If none of the data is sent, then the operator sends the data.

For details, see Chapter 24: "Troubleshooting a Failed Job."
This chapter provides usage information for the most commonly specified Stream operator features.

Topics include:

- Stream Operator Capabilities
- TPump Utility and the Stream Operator
- Stream Operator Definition
- Key Specifications
- Job Options
- Operational Considerations

**Stream Operator Capabilities**

The Stream operator, a consumer operator, emulates the Teradata TPump utility to perform high-speed parallel Inserts, Updates, Deletes, and Upserts in a near-real-time to one or more empty or preexisting Teradata Database tables without locking target tables.

Row level locking, as used in the Stream operator’s SQL transactions, allows load operations in the background during normal system use.

The Stream operator instances can update up to 127 tables on the Teradata Database. Multiple parallel instances can be used to improve the performance of the update.

Figure 59 shows the Stream operator interface.

Figure 59: Stream Operator Interface
The Stream Operator Function in a Teradata PT Script

When you use the Stream operator in a Teradata PT job, the Stream operator:

1. Logs on to the Teradata Database, using your user name, password, Teradata Database, and account ID information specified in the job script.
2. Loads the data from the Teradata PT data streams into the Teradata Database.
3. Logs off from the Teradata Database.
4. If the Stream operator job is successful, terminate the job and provide information about the job in the log or to the user, such as:
   - Total number of records sent to the Teradata Database
   - Number of errors posted to the error table
   - Number of inserts, updates, and deletes applied
5. If the job is unsuccessful, terminate the job and provide information about the job so that you can correct any problem and restart the job.

For information about using the Stream operator, see Chapter 5: “Moving External Data into Teradata Database.”

TPump Utility and the Stream Operator

Table 11 lists the operating features and capabilities of the Teradata TPump utility and indicates whether they are supported by the Stream operator or another Teradata PT operator.

<table>
<thead>
<tr>
<th>TPump Utility Feature</th>
<th>Stream Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute field positioning (handled by the FIELD command)</td>
<td>Not supported</td>
</tr>
<tr>
<td>Access Modules</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>ANSI Date</td>
<td>Supported</td>
</tr>
<tr>
<td>ARRAY support for DML statements</td>
<td>Supported</td>
</tr>
<tr>
<td>Checkpoint/Restart</td>
<td>Supported</td>
</tr>
<tr>
<td>Character Sets</td>
<td>Supported, using the USING CHARACTER SET clause before the DEFINE JOB statement</td>
</tr>
<tr>
<td>Configuration File</td>
<td>Supported, using the tbuild command line job attribute option (-v)</td>
</tr>
<tr>
<td>CREATE TABLE Statement</td>
<td>Supported, using the DDL operator</td>
</tr>
<tr>
<td>DATABASE Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>DELETE Statement</td>
<td>Supported, using the DDL operator</td>
</tr>
</tbody>
</table>
Table 11: Stream Operator Feature Support (continued)

<table>
<thead>
<tr>
<th>TPump Utility Feature</th>
<th>Stream Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>DROP TABLE Statement</td>
<td>Supported, using the DDL operator</td>
</tr>
<tr>
<td>Environment variable</td>
<td>Not supported</td>
</tr>
<tr>
<td>Error Limit</td>
<td>Supported</td>
</tr>
<tr>
<td>IF-ELSE-ENDIF constructs</td>
<td>Not supported</td>
</tr>
<tr>
<td>Indicator Mode</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>INMOD Routines</td>
<td>Supported, via the MultiLoad INMOD Adapter operator</td>
</tr>
<tr>
<td>Maximum/Minimum Sessions</td>
<td>Supported</td>
</tr>
<tr>
<td>MARK/IGNORE EXTRA UPDATE/DELETE DML Option</td>
<td>Supported, using the MARK/IGNORE EXTRA UPDATE/DELETE DML option in the APPLY Statement</td>
</tr>
<tr>
<td>Nonindicator Mode</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>Notify</td>
<td>Supported</td>
</tr>
<tr>
<td>NULLIF Clauses</td>
<td>Supported, but not the XB, XC, and XG data types</td>
</tr>
<tr>
<td>Operating System Command</td>
<td>Supported, using the OS Command operator</td>
</tr>
<tr>
<td>Periodicity runtime parameter</td>
<td>Supported</td>
</tr>
<tr>
<td>Predefined system variables (i.e., SYSDATE, SYSDAY)</td>
<td>Not supported</td>
</tr>
<tr>
<td>QUEUETABLE</td>
<td>Supported</td>
</tr>
<tr>
<td>-m runtime parameter to keep macros</td>
<td>Supported</td>
</tr>
<tr>
<td>RATE keyword for limiting the rate at which statements are sent to the Teradata Database</td>
<td>Supported</td>
</tr>
<tr>
<td>Record Formats</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>RECORD n THRU m</td>
<td>Supported in a limited form by the DataConnector operator, allowing you to read in the first &quot;m&quot; rows of a file, effectively allowing &quot;RECORD 1 THRU m&quot;</td>
</tr>
<tr>
<td>Replication Services override</td>
<td>Supported</td>
</tr>
<tr>
<td>Routing of messages to alternate file</td>
<td>Supported via Logger Services</td>
</tr>
<tr>
<td>RUN FILE</td>
<td>Supported via Teradata PT script language</td>
</tr>
<tr>
<td>Schemas, more than one</td>
<td>Not supported</td>
</tr>
<tr>
<td>Show Version Information</td>
<td>Supported</td>
</tr>
<tr>
<td>Tenacity</td>
<td>Supported</td>
</tr>
<tr>
<td>TPump Monitor Interface</td>
<td>Not supported</td>
</tr>
</tbody>
</table>
Stream Operator Definition

Following is a Stream operator definition.

Only key attributes are shown (in bold).

```
DEFINE OPERATOR STREAM_LOAD_OPERATOR
DESCRIPTION 'STREAM OPERATOR FOR Teradata PT'
TYPE STREAM
SCHEMA *
ATTRIBUTES
  (VARCHAR TdpId,
   VARCHAR UserName,
   VARCHAR UserPassword,
   VARCHAR LogTable,
   VARCHAR ErrorTable,
   VARCHAR AppendErrorTable,
   VARCHAR DropErrorTable,
   VARCHAR PrivateLogName)
```

For required syntax and descriptions of all Stream operator attributes, their capabilities and valid values, see *Teradata Parallel Transporter Reference*.

For descriptions of typical uses of the Stream operator, see Chapter 5: “Moving External Data into Teradata Database” and Chapter 7: “Moving Data within the Teradata Database Environment”.

### Key Specifications

**Defining the Stream Operator in a Teradata PT Script**

When you define a Stream operator in a Teradata PT job script, you must provide the following required specifications:

- **TYPE STREAM**
- Key operator attributes:
  - TdpId
  - UserName
  - UserPassword
  - LogTable
Chapter 19: Stream Operator
Key Specifications

- ErrorTable
- AppendErrorTable
- DropErrorTable
- PrivateLogName
- DML statement

*Note:* The first three key attributes listed above are “security attributes.”

In addition, other optional operator attributes can be declared depending on how the operator will be used.

For more information about all Stream operator attributes, their capabilities, and valid values, see “Stream Operator” in *Teradata Parallel Transporter Reference*.

**Security Attributes**

**TdpId**

Attribute that specifies the name of the Teradata Database machine (non-mainframe platforms) or TDP (mainframe platforms) for the Stream operator job.

If you do not specify the value for the TdpId attribute, the operator uses the default TdpID established for the user by the system administrator.

For details, see “Teradata Database Logon Security” on page 73.

**UserName**

Attribute that specifies the Teradata Database user name.

For details, see “Teradata Database Logon Security” on page 73.

**UserPassword**

Attribute that specifies the password associated with the user name.

For details, see “Teradata Database Logon Security” on page 73.

**LogTable**

A restart log table, which contains restart information written during job runs, is required for any execution of the Stream operator. Specify a restart log table in scripts with the LogTable attribute.

Restarts can occur following any unexpected error on a database. For example, if a table runs out of available space during a load operation, the job terminates, the table is paused, and a checkpoint is recorded in the restart log. Pausing the job in this way allows you to manually increase the available space, if needed, then restart the job because the load operation can restart a job from the last checkpoint in the restart log.
**ErrorTable**

Attribute that specifies the name of the error table. This table contains information concerning data conversion errors, constraint violations, and other error conditions.

This attribute must be a new table name, unless the AppendErrorTable attribute is set to YES. Do not use a name that duplicates the name of an existing table unless you are restarting a Stream operator job. If the name is not supplied, it is created by the Stream operator. User-supplied names for error tables must not exceed 30 bytes.

**Keeping the Error Table**

By default, the Stream operator drops the error table after data loading, unless the error table is non-empty. The error table can be kept after data loading by setting the DropErrorTable attribute to NO, as follows:

```
VARCHAR DropErrorTable = NO
```

Keeping the error table allows the error table to be re-used for future jobs and saves the time required to drop the table.

**Re-using the Error Table**

By default, the Stream Operator will create the error table. If the error table already exists, Stream Operator will terminate the job with an error message. The exception is on a job restart. On a job restart, Stream Operator will continue.

- When the value of the AppendErrorTable attributes is YES, the Stream operator reuses the error table and displays the number of rows already existing in the error table.
- If the error table does not exist, the Stream operator creates the error table before data loading and continue.
- If the structure of the existing error table is not compatible with the error table the Stream operator expects, it terminates the job with an error message before data loading.

**Note:** A new column, RowInsertTime, was added to the Stream Operator error table for version 13.10. Therefore, you can no longer reuse error tables from prior versions, as valid Stream Operator error tables now have ten columns rather than nine.

**Error Limit**

Attribute that specifies the approximate number of records that can be stored in the error table before the Stream operator job is terminated.

This number is approximate because the Stream operator sends multiple rows of data at a time to Teradata Database. By the time Teradata PT processes the message indicating that the error limit has been exceeded, it may have loaded more records into the error table than the actual number specified in the error limit.

The ErrorLimit specification must be greater than 0. Specifying an invalid value will cause the Stream operator job to terminate. By default, the ErrorLimit value is unlimited.

**Note:** The ErrorLimit specification applies to each instance of the Stream operator.
**Job Options**

**SERIALIZE**

The Serialize option only applies to the Stream operator. Use the Serialize option when correct sequencing of transactions is required. For example, when a job contains a transaction that inserts a row to open a new account, and another transaction updates the balance for the account, then the sequencing of the transactions is critical.
Using the Serialize option in APPLY statements, the Stream operator ensures that operations for a given row occur in the order they are received from the input stream.

To use this option, associate a sequencing key (usually the primary index) with the target table. Each input data row is hashed based on the key to determine the session assigned to process each input row. This allows all rows with the same key to be processed in sequence by the same session, which is especially important if rows are distributed among many sessions.

When using the Serialize option, only one instance of the Stream operator is allowed. Specifying more than one instance causes the Stream operator to terminate with an error.

**SERIALIZE OFF**

When the Serialize option is set to OFF, transactions are processed in the order they are encountered, then they are placed in the first available buffer. Buffers are sent to parsing engine (PE) sessions and PEs process the data independently of other PEs. In other words, transactions might occur in any order.

If the Serialize option is not specified, the default is OFF unless the job contains an upsert operation, which causes Serialize to switch the default to ON.

**SERIALIZE ON**

If the Serialize option is set to ON, operations on a row occur serially in the order submitted.

The sequencing key of SERIALIZE ON is specified as one or more column names occurring in the input data SCHEMA definition. These SCHEMA columns are collectively referred to as the key. Usually the key is the primary index of the table being updated, but it can be a different column or set of columns. For example:

```
APPLY
  ('UPDATE emp SET dept_name = :dept_name
   WHERE empno = :empno;')
  SERIALIZE ON (empno)
TO TARGET_TABLE[1]
```

This APPLY statement guarantees that all data rows with the same key (empno) are applied to the database in the same order received they are received from the producer operator. In this case, the column empno is the primary index of the Emp table.

Note that SERIALIZE ON is local to a specific DML statement. In the following example, a group DML is specified, but only the first statement uses the Serialize option:

```
APPLY
  ('UPDATE emp SET dept_num = :dept_num
   WHERE empno = :empno; ')
  SERIALIZE ON (empno)
  ('UPDATE dept SET dept_name = :dept_name
   WHERE deptno = :deptno; ')
TO TARGET_TABLE[1]
```

Following are some of the advantages to using the Serialize option, and might improve performance:

- SERIALIZE ON can eliminate the lock delays or potential deadlocks caused by primary index collisions coming from multiple sessions.
SERIALIZE ON can also reduce deadlocks when rows with non-unique primary index values are processed.

Robust and Non-Robust Mode

For more robust restartability, use robust mode, which causes every DML operation to be checkpointed and ensures on restart that no operation is applied more than once.

The robust mode requires more writes to a restart log, which might impact performance more, however, using robust mode ensures that a restart avoids reprocessing rows that a normal interval checkpoint might necessitate.

Robust is the default mode for all Stream operator jobs. The Robust attribute turns the mode on or off. If uncertain whether to use robust restart logic, it is always safe to set the Robust parameter to 'Yes'.

- **Robust Mode** - Setting the attribute to “yes” tells the Stream operator to use robust restart logic.
  
  VARCHAR Robust = 'Yes' (or 'Y')

  Robust mode causes a row to be written to the log table each time a buffer successfully completes its updates. Mini-checkpoints are written for each successfully processed row. These mini-checkpoints are deleted from the log when a checkpoint is taken, and are used at restart to identify the rows that have been successfully processed, which permits them to be bypassed at restart. In robust mode, each row is processed only once. The larger the Pack factor, the less overhead is involved in this activity.

  Choosing the Robust mode is particularly useful to avoid problems with data integrity and unacceptable performance. Robust mode is recommended in the following situations to avoid having an adverse affect on restarts:

  - **INSERTs into multi-set tables** - Robust mode prevents the insertion of duplicate rows, which could insert the same row a second time.
  
  - **UPDATEs based on calculations** - Robust mode prevents the duplicate application of calculations.
  
  - **Large Pack factors** - Robust mode does not involve the application and rejection of duplicate rows after restarts, which is a time-consuming process of logging errors to the error table.
  
  - **Time-stamped data** - Robust mode prevents the possibility of stamping identical rows with different time stamps, resulting in duplicate rows.

  If rows are reapplied in non-robust mode, each reapplied row is marked with a time stamp that is different from the original row even though all of the other data is identical. To Teradata, these reapplied rows are different rows with the same primary index value, so they are inserted even though they are duplicates.

- **Non-Robust Mode** - Setting the attribute to “no” tells the Stream operator to use simple restart logic rather than robust logic.
  
  VARCHAR Robust = 'No' (or 'N')

  In a non-robust mode, restarts begin where the last checkpoint occurs in a job. Because some additional processing will most likely take place after the checkpoint is written, the
requests that occur after the checkpoint are resubmitted by the Stream operator as part of the restart process. For Deletes, Inserts and Upserts, this does not usually cause a problem or harm the database; however, re-running statements generates more rows in the error table because the operator will be attempting to insert rows that already exist and to delete rows that do not exist.

Re-attempting updates can also be a problem if update calculation, for example, is based on existing data in the row, such as adding 10% to an amount. Doing the update calculation a second time add an additional 10% to the amount, thus compromising data integrity. In this type of update, it is best to use robust mode to ensure that no DML operation is applied more than once.

The non-robust (or simple restart) method does not involve the extra overhead that comes with the additional inserts to the restart log table that are needed for robust logic, so overall processing is notably faster.

**Recovery Logic and Overhead**

In Robust mode, the Stream operator writes one row in the restart log table for each request issued. This collection of rows in the restart log table can be referred to as the request log. Because a request is guaranteed by the Teradata Database to either completely finish or completely roll back, the request log will always accurately reflect the completion status of a load operation. Thus, the request log overhead for restart logic decreases as the number of statements packed per request increases. During the checkpoint process, the Stream operator flushes all pending changes from internal storage to the database and also deletes the request log rows. The larger the checkpoint interval, the larger the request log is going to grow. In the event of a Robust mode restart, the Stream operator will use the request log to avoid the erroneous reapplication of database changes.

The Stream operator, in simple (non-robust) mode, provides basic checkpoints. If a restart occurs, then some requests will likely be reprocessed. This is adequate protection under some circumstances. Simple logic is adequate in certain DML statements that can be repeated without changing the results of the operation.

Examples of statements that are *not* simple include the following:

- Inserts into tables that allow duplicate rows (MULTISET tables).
- Self-referencing DML statements like: "UPDATE FOO SET A=A+1...", or "UPDATE FOO SET A = 3 WHERE A=4"

**Data Quality Affects Performance**

It is more important to have error-free data when using the Stream operator than with other Teradata PT operators. If data contains errors, a large Pack factor can slow performance because of the way Stream handles errors.

For example, if several hundred statements are packed, when an error occurs, the entire request is rolled back. The Stream operator then removes the error-producing statement(s) and then reissues the entire request. Such a process can be costly from a performance standpoint.
**Statement Packing**

To provide optimal performance, the Stream operator packs individual DML statements into a larger multistatement request based on the rate specified by the Pack attribute. This type of processing requires less overhead than multiple individual requests.

The Stream operator submits these multistatement requests using macros which it creates to hold the requests. The macros are then executed instead of running each individual DML statement.

The macros are automatically removed after the job is complete. The use of macros in place of lengthy requests helps to minimize both network and parsing overhead.

**Specifying the Pack Rate**

The Pack attribute specifies the number of statements in a multistatement request. Specifying a Pack rate improves network/channel efficiency by reducing the number of sends and receives between Teradata PT and the Teradata Database. A maximum of 2400 statements can be specified.

Trial and error might be required to determine the best Pack rate for a Stream job. As the Pack rate is increased, the throughput improvement is usually great at first, then falls off. In other words, going from a Pack rate of 1 to 2 could provide huge performance gains, and going from 2 to 4 could be just as beneficial, but moving from 8 to 16 might cause a performance drop.

If the PackMaximum attribute is set to 'Yes', the Stream operator determines the maximum pack for the job, and then reports it.

Two factors to consider are:

- The maximum Pack factor based on Stream operator restrictions
- The optimal Pack factor for a particular job

These two factors might not be equal. The maximum rate lets you know the upper limit, but performance might improve at a smaller rate. For this reason, it is recommended that PACKMAXIMUM not be used for production jobs until you determine the optimal Pack factor.

**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 operator’s logged output. Specify that determined value in the Pack attribute on subsequent runs. Set the PackMaximum to 'No'.

Alternatively, the Stream Driver TD_Evt_PackFactor event returns the current pack factor when queried. This value is available after a connection is initiated and before it is terminated.
Array Support

The Array Support feature allows DML requests containing only a single statement to be executed once for each of multiple rows of input data, each row specified being one of the members of the array. The DML must be contained within an APPLY statement that includes Stream operator and does not set ArraySupport to Off.

Use of array support improves performance by:

- Increasing the amount of data that can be sent per request from a total of approximately 64KB to a total of approximately 1MB, with a limit of 64KB per input data row.
- Improvements in internal Teradata Database processing as a result of using such requests.

Latency Interval

Latency is the interval value, expressed in seconds, between the flushing of stale buffers. Latency interval is an option that is exclusively used by the Stream operator.

In normal operations (without latency), the Stream operator reads data from the data stream until its buffer is full, then it writes all buffered rows to the Teradata Database. The data is written to the Teradata Database only when the buffer is full or when a checkpoint is taken. However, a latency interval (for example, set to 5400 seconds) causes the following:

- The Stream operator reads data from the data stream, and empties its buffer, writing the contents to the Teradata Database every 90 minutes (5400 seconds) regardless of whether it is full.
- If the buffer fills up within the time period (in this case, 90 minutes), it writes to the Teradata Database as it would during normal operation.

To set the latency interval, use the following syntax:

```
tbuild -l <LatencyInterval> -f <file name>
```

The value used for the latency interval must be a non-zero unsigned integer. The guiding factor is how stale you are willing to allow data to be.

For example, to run a continual load script with a latency interval of two hours, enter:

```
tbuild -l 7200 -f continualload
```

Operational Considerations

NoPI Tables

Operations other than Insert on target tables defined as NoPI are very non-performant, unless there is an appropriate secondary index path to the data rows. For details, see volume 4 of SQL.

Space Requirements

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 operator terminates the Stream operator job. When this happens, you must allocate more space to the database before you can restart the job.

**Sessions and Instances**

Because the Stream operator uses Teradata SQL sessions in its communication with Teradata Database, it does not use load tasks. The Stream operator provides continuous updates using row level locking, allowing constant load operations in the background during normal system use.

Both a minimum and a maximum number of sessions can be used by the Stream operator. The minimum specification is one. The default is one session for each operator instance.

The number of sessions is evenly distributed among the number of operator instances. If 20 sessions are requested and four instances of the operator are invoked, then each instance will receive five of the sessions.

For information on Sessions and Limits, see “Optimizing Job Performance with Sessions and Instances” on page 80.

**Other Session Limits**

The values that you specify with the Stream operator MinSessions and MaxSessions attributes are not the only factors that limit the number of sessions that the Stream operator establishes with the Teradata Database. The other limiting factors are:

- The platform limit on the maximum number of sessions per application.
  - On network-attached UNIX, Linux, and Windows client systems, this value is defined in the COP Interface software file, `clispb.dat`, under the `max_num_sess` variable.
  - On channel-attached z/OS client systems, this value is defined in the `HSHSPB` parameter under the IBCSMAX setting.
  - On channel-attached z/OS client system, use the TDP SET MAXSESSIONS command to specify a platform limit.
- The limit of the network protocol software on network-attached systems.

When the Stream operator executes, the actual session limit is determined by the first limiting factor that is encountered.

**Checkpointing and Restarting**

Checkpoint options control how often a row is written to the checkpoint file for the purposes of restarting a job. Unless otherwise specified, a checkpoint is taken at the start of and at the end of the input data. Since this process does not provide granular restartability in the case of longer running jobs, checkpoint intervals can be user-specified in terms of minutes or seconds.

- Seconds - Use the command line option `-z`. For example, the following command indicates that a checkpoint will be taken every 30 seconds:
tbuild -f <script file name> -z 30

- Minutes or seconds - Specify as part of a DEFINE JOB statement. For example:

```sql
DEFINE JOB test_job
SET CHECKPOINT INTERVAL 30 SECONDS
or
SET CHECKPOINT INTERVAL 10 MINUTES
```

For information on checkpointing and restarting, see Chapter 24: “Troubleshooting a Failed Job.”

### Dropped Tables

Some tables are created by the Teradata Database during the execution of a job, and others must be user-created before the job begins. The log table and error table are automatically created by Teradata PT when you run the job script. The error table is dropped during the Cleanup phase if no errors were detected during the job, unless the DropErrorTable attribute is set to ‘No’. 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, manually drop these tables by running a BTEQ script.

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

### DML Option Categories

The DML Options can be classified into four logical categories, which can be specified in any order:

- ARRAY SUPPORT
- SERIALIZE
- USE
- DML ERROR OPTIONS, which include:
  - MARK
  - IGNORE
  - INSERT FOR

The following restrictions apply:

- Multiple ARRAY SUPPORT Options cannot be specified.
- Multiple SERIALIZE Options cannot be specified.
- Multiple USE Options cannot be specified.
- Multiple DML Error Options (MARK,IGNORE, INSERT FOR) can be specified in one block. However these cannot be interleaved with the other three DML Option categories.
Temporal Tables

Beginning with version 13.10, Teradata Database supports temporal tables. When using the Stream operator to load data into temporal tables, the temporal qualifiers, such as CURRENT VALIDTIME and SEQUENCED VALIDTIME, must precede the DML statements, as in the following example.

```
APPLY
('SEQUENCED VALIDTIME INSERT INTO <target table> ... ;')
TO OPERATOR (STREAM_OPERATOR)
SELECT * FROM OPERATOR (<producer operator>);
```

When using an atomic upsert, the temporal qualifier must precede the UPDATE statement, not the INSERT statement, as in the following example.

```
APPLY
('SEQUENCED VALIDTIME UPDATE <target table> SET ... WHERE ... ;',
 'INSERT INTO <target table> ... ;')
INSERT FOR MISSING UPDATE ROWS
TO OPERATOR (STREAM_OPERATOR)
SELECT * FROM OPERATOR (<producer operator>);
```
This chapter provides information for using the most common Update operator features. Topics include:

- **Update Operator Capabilities**
- **MultiLoad Utility and the Update Operator**
- **Update Operator Definition**
- **Key Specifications**
- **Job Options**
- **Operational Considerations**
- **Delete Task Option**

### Update Operator Capabilities

The Update operator, a consumer operator, emulates the Teradata MultiLoad utility to perform highly scalable and parallel Inserts, Updates, Deletes, and Upserts into new or preexisting Teradata Database tables in a single pass. The Update operator can also be used to insert new rows into the database, but it *cannot* perform Select operations.

### Update Operator as Standalone Operator

The Update operator can also function as a standalone operator when it is executing the Delete task and if no data is required. The Delete Task permits the high-speed deletion of table rows based on a non-index value. For information on the Delete task option, see “Delete Task Option” on page 253. Figure 60 illustrates how the Update operator works.

Figure 60: Update Operator Interface
The Update Operator Function in a Teradata PT Job Script

When you use the Update operator in a Teradata PT job, each parallel instance of the Update operator:

1. Log on to the Teradata Database, using your user name, password, Teradata Database, and account ID information specified in the job script.
2. Loads the data from the Teradata PT data streams into the Teradata Database.
3. Logs off from the Teradata Database.
4. If the Update operator job is successful, terminates the job and provides information about the job in the log or to the user, such as:
   - Total number of records sent to the Teradata Database
   - Number of errors posted to the error tables
   - Number of inserts, updates, and deletes applied
   - A status code indicating the success of the job
5. If the job is unsuccessful, terminate the job and provide information about the job so that you can correct any problem and restart the job.

Functional Description of the Update Operator

During an Update job, the Update operator loads Teradata Database tables. Each instance retrieves rows from the Teradata PT data streams and sends them to the Teradata Database. Teradata Database Multiple parallel instances can be used to improve the performance of the update.

The Update operator reads a data block only once. The operator reads rows from a data stream, writes that data to a buffer, then sends that buffer of data across sessions to the available AMPs.

This process is dependent on all changes to tables being keyed on the primary index. Thus, all transactions must be primary index operations when using the Update operator. (An exception to this rule is the use of the Delete Task. For more information, see “Delete Task Option” on page 253.

Update Phases

Update operations have two phases:

- **Acquisition Phase** - Data from the input stream is transmitted to the AMPs, and access locks are placed on the target tables, limiting table access to read-only. The acquisition phase is complete when all data rows are on the appropriate AMPs where their changes will be applied.

  Records are sorted by the hash value of the primary index value. This sorting order becomes the sequence in which they are applied to a target table. Sorted records are placed in temporary work tables that require permanent space for the duration of the job.
One work table per target table is created by the Teradata Database. Because the acquisition phase involves writing only to work tables, target tables are left available for user access.

- **Application Phase** - Sorted input records are applied to data blocks of target tables using the appropriate DML commands (insert, update, delete). Each target block is read once into memory, and all changes are applied at that time. The access lock on target tables is upgraded to a write lock, so tables are not available for user access until the phase is complete. At the end of this phase, the work tables are dropped during a subsequent clean-up phase.

## MultiLoad Utility and the Update Operator

The following table lists the operating features and capabilities of the Teradata MultiLoad utility and indicates whether they are supported by the Update operator or another Teradata PT operator.

**Table 12: Update Operator Feature Support**

<table>
<thead>
<tr>
<th>MultiLoad Utility Feature</th>
<th>Update Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute field positioning (handled by the FIELD command)</td>
<td>Not supported</td>
</tr>
<tr>
<td>Access Modules</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>ANSI Date</td>
<td>Supported</td>
</tr>
<tr>
<td>Checkpoint.Restart</td>
<td>Supported</td>
</tr>
<tr>
<td>Character Sets</td>
<td>Supported, using the USING CHARACTER SET clause before the DEFINE JOB statement</td>
</tr>
<tr>
<td>Configuration File</td>
<td>Supported, using the <code>tbuild</code> command line job attribute option (-v)</td>
</tr>
<tr>
<td>CREATE TABLE Statement</td>
<td>Supported, using the DDL operator</td>
</tr>
<tr>
<td>DATABASE Statement</td>
<td>Supported</td>
</tr>
<tr>
<td>DELETE Statement</td>
<td>Supported, using the DeleteTask attribute and the APPLY statement</td>
</tr>
<tr>
<td>DROP TABLE Statement</td>
<td>Supported, using the DDL operator</td>
</tr>
<tr>
<td>Environment variable</td>
<td>Not supported</td>
</tr>
<tr>
<td>Error Limit</td>
<td>Supported</td>
</tr>
<tr>
<td>IF-ELSE-ENDIF constructs</td>
<td>Not supported</td>
</tr>
<tr>
<td>Indicator Mode</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>INMOD Routines</td>
<td>Supported, via the MultiLoad INMOD Adapter operator</td>
</tr>
</tbody>
</table>
### Update Operator Definition

Following is an example of an Update operator definition.

Only key attributes are shown (in bold).

```sql
DEFINE OPERATOR UPDATE_OPERATOR
DESCRIPTION 'Teradata PT UPDATE OPERATOR'
TYPE UPDATE
SCHEMA *
ATTRIBUTES
{
    VARCHAR TdpId,
    VARCHAR UserName,
}
```

<table>
<thead>
<tr>
<th>MultiLoad Utility Feature</th>
<th>Update Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum/Minimum Sessions</td>
<td>Supported</td>
</tr>
<tr>
<td>Nonindicator Mode</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>Notify</td>
<td>Supported</td>
</tr>
<tr>
<td>NULLIF Clauses</td>
<td>Supported, but not the XB, XC, and XG data types</td>
</tr>
<tr>
<td>Operating System Command</td>
<td>Supported, using the OS Command operator</td>
</tr>
<tr>
<td>Pause Acquisition</td>
<td>Supported</td>
</tr>
<tr>
<td>Predefined system variables (i.e., SYSDATE, SYSDAY)</td>
<td>Not supported</td>
</tr>
<tr>
<td>Record Formats</td>
<td>Supported, using the DataConnector operator</td>
</tr>
<tr>
<td>RECORD (n) THRU (m)</td>
<td>Supported in a limited form by the DataConnector operator, allowing you to read in the first &quot;m&quot; rows of a file, effectively allowing &quot;RECORD 1 THRU m&quot;</td>
</tr>
<tr>
<td>Replication Services override</td>
<td>Supported</td>
</tr>
<tr>
<td>Routing of messages to alternate file</td>
<td>Supported via Logger Services</td>
</tr>
<tr>
<td>RUN FILE</td>
<td>Supported via Teradata PT script language</td>
</tr>
<tr>
<td>Schemas, more than one</td>
<td>Not supported</td>
</tr>
<tr>
<td>Show Version information</td>
<td>Supported</td>
</tr>
<tr>
<td>SQL statements (such as CREATE TABLE, DROP TABLE, etc.)</td>
<td>Supported, using the DDL operator</td>
</tr>
<tr>
<td>Tenacity</td>
<td>Supported</td>
</tr>
<tr>
<td>User-defined variables</td>
<td>Limited support via script language</td>
</tr>
<tr>
<td>Wildcard INSERT</td>
<td>Not supported (cannot use &quot;INSERT INTO tablename.*;&quot;)</td>
</tr>
</tbody>
</table>
The Update Operator in a Teradata PT Job Script

When you define an Update operator in a Teradata PT job script, you must provide the following required specifications:

- TYPE UPDATE
- Key operator attributes:
  - TdpId
  - UserName
  - UserPassword
  - LogTable
  - WorkTable
  - TargetTable
  - ErrorTable1
  - ErrorTable2
  - PrivateLogName
- For Update operator as a standalone operator: DeleteTask

Note: The first three key attributes listed above are “security attributes.”

In addition, other optional operator attributes can be declared depending on how the operator will be used.

For more information about all Update operator attributes, their capabilities, and valid values, see “Update Operator” in Teradata Parallel Transporter Reference.
Chapter 20: Update Operator

Key Specifications

Security Attributes

**TdpId**
Optional attribute that specifies the name of the Teradata Database machine (non-mainframe platforms) or TDP (mainframe platforms) for the Update operator job.

If you do not specify the value for the TdpId attribute, the operator uses the default TdpId established for the user by the system administrator.

For details, see “Teradata Database Logon Security” on page 73.

**UserName**
Attribute that specifies the Teradata Database user name.

For details, see “Teradata Database Logon Security” on page 73.

**UserPassword**
Attribute that specifies the password associated with the user name.

For details, see “Teradata Database Logon Security” on page 73.

**LogTable**
A restart log table, which contains restart information written during job runs, is required for any execution of the Update operator. Specify a restart log table in scripts with the LogTable attribute.

Restarts (as discussed in “Staged Loading” on page 192) are common during staged loading operations. When additional data is available after a job is paused, the job is restarted by submitting a second script that specifies the additional data. The Update operator recognizes the job as a restart, reads the restart log to determine the status of the job, then loads the additional file.

Restarts can also occur following any unexpected error on a database. For example, if a table runs out of available space during a load operation, the job terminates, the table is paused, and a checkpoint is recorded in the restart log. Pausing the job in this way allows you to manually increase the available space, if needed, then restart the job because the load operation can restart a job from the last checkpoint in the restart log.

**TargetTable**
Required attribute that specifies the name of the Update target table to receive data from the client system. The table must already exist.

For details, see *Teradata Parallel Transporter Reference*.

The Update operator does not support target tables defined as NoPI.
ErrorTable

Update operations create two error tables for each target table. These error tables are similar to those used for the Load operator, but the Update error tables are typically named with the following suffixes to distinguish them.

Consider the following:

- Names for error tables can be defaulted or they can be explicitly named using the VARCHAR ErrorTable attribute.
- If a job generates no errors, the error tables will be empty. They are automatically dropped at the end of the job.
- If errors are generated, the tables are retained at the end of the job so error conditions can be analyzed.
- To rerun a job from the beginning, either delete the error tables, or rename them, otherwise an error message results, stating that error tables already exist.
- Conversely, if you restart a job (not from the beginning), an error tables must already exist. In other words, do not delete error tables to restart an update job.

Errors are separated into two tables, as follows:

- **Error Table (ET)** contains most of the errors relating to data and the data environment. The following types of errors are captured:
  - Constraint violations records that violate a range constraint defined for the table.
  - Unavailable AMP records that are written to a non-fallback table on an offline AMP.
  - Data conversion errors records that fail to convert to a specified data type.

  By default, this error table is assigned a name using the convention:
  
  `Target_Tablename_ET`

- **Uniqueness Violations (UV)** contains all of the rows that have violations of a unique primary index.

  By default, this error table is assigned a name using the following convention:
  
  `Target_Tablename_UV`

**Error Limits**

The Update operator provides the same capability as the Load operator for setting a limit on the number of errors captured before a job is terminated. When updating large amounts of data, it is not uncommon to encounter a data error that occurs repeatedly on each input record. Because an error can often be corrected long before errors are generated for all of the records in a job run, consider using the ErrorLimit attribute to specify a limit to the number of errors that can be tolerated before a job is terminated.

Note that the ET table contains errors in rows detected during the acquisition phase (the loading of data). These are commonly data conversion errors. The second table is the UV table and contains rows that are detected to be in error during the application phase of the job. These errors are commonly “uniqueness violation” errors (hence the name UV).
The ErrorLimit specification applies to each instance of the Update operator, not to all instances combined. For example, if the limit is set to 1,000, a single instance must detect that 1,000 rows were inserted into error tables to terminate the job.

This limit is specified with the `ErrorLimit` attribute. Errors are counted only during the Acquisition Phase, so the number of error rows being placed in the ET table are counted towards the number set in the `ErrorLimit` attribute. This applies to each instance of the Update operator, not to all instances combined. Therefore, if an error limit is set to 1,000, a single load instance must detect that 1,000 rows are inserted into the error tables before the job is terminated.

The error limit can also be reached at checkpoint time; see the examples below.

### Error Limit Examples

To illustrate how Teradata PT determines if the number of errors has reached the Error Limit, consider these examples if there are two instances running and the Error Limit has been set to 1000.

- If either instance by itself reaches 1000, it will terminate the job by returning a fatal error.
- If instance #1 processes 500 error rows and instance #2 processes 500 error rows but does not reach a checkpoint. The job will continue processing.
- If instance #1 processes 500 error rows and instance #2 processes 500 error rows but does reach a checkpoint. The total number of error rows for all instances combined is determined at checkpoint time and at the end of the Acquisition Phase. If the total of all instances exceeds the error limit at that time, the job will terminate with an error.

### Error Capture

When running insert, update, delete, or upsert requests, errors can occur due to missing or duplicate rows. When errors occur, the request is rolled back and the error is normally reported. Use the APPLY statement to specify how to handle this type of error:

- **MARK** means the error is to be captured and recorded.
- **IGNORE** means the error is not to be recorded.

Specify whether errors are marked or ignored with the following in mind:

- **DUPLICATE INSERT ROWS** means an attempt to insert a duplicate row.
- **DUPLICATE UPDATE ROWS** means an update will result in a duplicate row.
- **MISSING DELETE ROWS** means an attempt to delete a row that is missing.
- **MISSING UPDATE ROWS** means an attempt to update a row that is missing.

### WorkTable

Update operations create one work table for each target table, using the following naming convention:

```
Target_TableName_WT
```

As with the error tables, these default names can be overridden. Use the VARCHAR `WorkTable` attribute.
Auto-Generation of Error and Work Tables

Some tables are created during the execution of a job, and others must be created by the user before the job begins. The log table is created automatically when you run the Update job script. Target tables must exist on the Teradata Database when the Update operator job is executed.

If you have not specified error and work tables (specifying them is optional), the Update operator automatically creates the names of the error and work tables.

If you specify only one target table, use VARCHAR syntax, as follows:

```
VARCHAR TargetTable = 'ttname'
```

If you specify more than one target table, you can also use VARCHAR syntax, as follows:

```
VARCHAR TargetTable = ['table1', 'table2', ..., 'tableN']
```

You can, if you wish, use ARRAY syntax, as follows:

```
VARCHAR ARRAY TargetTable = ['table1', 'table2', ..., 'tableN']
```

**Note:** Using the ARRAY keyword in assigning an array value is optional.

In either case, the Update operator automatically creates the names of the error tables and the work table, associated with a target table, as follows:

- The first error table is `ttname_ET`
- The second error table is `ttname_UV`
- The work table is `ttname_WT`

where `ttname` is the name of the corresponding target table.

**Warning:** The value of the TargetTable attribute is truncated to 27 characters before the suffixes "_ET", "_UV", or "_WT" are appended. This means that if the value of the TargetTable attribute is a fully qualified table name and that fully qualified name exceeds the maximum supported length of a database object (30 characters), the generated names for the error and work tables may not be what you intend. In such a situation, Teradata recommends that you provide the names of the error and work tables and not rely on the Update operator automatically generating the names for these tables.

You cannot specify more error or work tables than there are target tables already defined, but you may specify fewer.

If the following is specified when no other error or work table specifications exist,

```
VARCHAR TargetTable = ['targtable1', 'targtable2', 'thirdtable']
```

the Update operator creates the following error tables and work tables:

```
targtable1_ET
ntargtable1_UV
targtable1_WT
targtable2_ET
targtable2_UV
targtable2_WT
```
Notice that each set of two error tables and one work table belong to a particular target table; the naming convention preserves the uniqueness of the tables associated with a target table.

If you specify the following, the Update operator creates the necessary missing table names:

```sql
VARCHAR TargetTable = ['ttname1','ttname2','ttname3']
VARCHAR ErrorTable1 = ['error_1']
VARCHAR ErrorTable2 = ['error_2']
VARCHAR WorkTable = ['work_1','work_2']
```

If you specify more error table names or work table names than there are target table names, the Update operator issues an error message and terminates the job.

### PrivateLogName

Optional attribute that specifies the name of a log that is maintained by the Teradata PT Logger inside the public log. The private log contains all of the messages produced by the operator.

By default, no diagnostic trace messages are produced. Diagnostic trace messages are produced only when the user sets a valid value for the TraceLevel attribute.

For details, see *Teradata Parallel Transporter Reference*.

### DeleteTask

See “Delete Task Option” on page 253.

### Job Options

**Checkpointing and Restarting**

The Update operator takes a checkpoint at the beginning and the end of the acquisition phase. More granular checkpoints during the acquisition phase can be specified using the `-z` option of the `tbuild` command, which specifies checkpoint intervals in terms of seconds. For example:

```bash
tbuild -f <file name> -z 30
```

In this example, the `-f` option specifies the name of the script that is input to `tbuild`, and the `-z` option indicates that a checkpoint is taken every 30 seconds.

For more information about checkpoints and restarts, see “Checkpoint Restartability” in Chapter 24: “Troubleshooting a Failed Job.”

The Update operator cannot be rolled back. Once changes are applied to target tables in the application phase, a job can only move forward.

Since a target table cannot be returned to its original state, it is advisable to archive tables prior to running Update operations against them.
To restart a job from the beginning (and bypass the restart log table) do the following:

1. Drop the restart log table.
2. Drop the error and work tables.
3. Drop the checkpoint files.

To discontinue an Update operation and drop the target tables, do the following:

1. Drop the restart log table.
2. Drop the error and work tables.
3. Drop the target tables.

**Upserts**

Upsert operations update rows if they already exist in a table, and insert new rows they do not already exist. Specified an upsert operation in an APPLY statement using the expression:

```
INSERT FOR MISSING UPDATE ROWS;
```

An upsert operation fails only if both the update and the insert fail.

Also, the following specification is usually included for upsert operations in anticipation of missing rows:

```
IGNORE MISSING UPDATE ROWS
```

**AMP Usage**

**Down AMPs**

The impact of down AMPs on the Update operator depends on the following:

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

Table 13 describes the impact of down AMPs on Update operator tasks on fallback and nonfallback tables.
Nonparticipant AMPs

An AMP can become nonparticipant for an Update operator task one of the following ways:

- When any AMP is down at the end of the acquisition phase/beginning of the application phase, the associated AMP becomes a nonparticipant if the AmpCheck 'None' option is specified. Because the Update operator does not run after the acquisition phase if an AMP is down and the target table is nonfallback, an AMP can become a nonparticipant only if the target table is defined as having fallback protection. The AmpCheck 'Apply' and 'All' options prevent the occurrence of nonparticipant AMPs in this situation.
- When I/O errors occur in certain Update operator tables during the application phase, the associated AMP becomes a nonparticipant if the I/O recovery operation stops the Update operator task.
- When a head/disk assembly (HDA) fails during the application phase, the associated AMP becomes a nonparticipant but it returns after the disk is replaced and the Disk Copy and Table Rebuild utilities are run.

In effect, the Update operator treats a nonparticipant AMP as if it were a down AMP. Thus, the Update operator does not run if a cluster has any combination of more than one AMP that is:

- Down
- Offline
- Nonparticipant
And, if more than one AMP in a cluster becomes a nonparticipant during the application phase, the Update operator tasks cannot continue. The target tables are considered unusable, and must be recovered from archives.

**One-AMP Systems**

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

**VARCHAR ARRAY Tables**

Target, error, and work table names can be specified in terms of VARCHAR ARRAY types if specifying more than one table, using the following syntax:

```
VARCHAR TargetTable = ['table1', 'table2', ..., 'tableN']
```

**Note:** Using the ARRAY keyword in assigning an array value is optional.

You cannot specify more error or work tables than there are target tables defined, but you may specify fewer. If fewer error/work table names are defined than target tables, the Update operator creates a name for the error/work table:

- The first error table is `ttnameN_ET`
- The second error table is `ttnameN_UV`
- The work table is `ttnameN_WT`, where `ttnameN` is the name of the corresponding target table

**Note:** Target table names are truncated to 27 characters before the suffixes "_ET", "_UV", or "_WT" are appended.

For example, if the following is specified when no other error/work table specifications exist,

```
VARCHAR TargetTable = ['targtable1', 'targtable2', 'thirdtable']
```

the Update operator creates the following error tables and work tables:

- `targtable1_ET`
- `targtable1_UV`
- `targtable1_WT`
- `targtable2_ET`
- `targtable2_UV`
- `targtable2_WT`
- `thirdtable_ET`
- `thirdtable_UV`
- `thirdtable_WT`

Note that each set of two error tables and one work table belong to a particular target table; the naming convention preserves the uniqueness of the associated target table.

If you specify the following, the Update operator creates the necessary missing table names:

```
VARCHAR TargetTable = ['ttname1','ttname2','ttname3']
VARCHAR ErrorTable1 = ['error_1']
VARCHAR ErrorTable2 = ['error_2']
VARCHAR WorkTable = ['work_1','work_2']
```
If you specify more error table names or work table names than there are target table names, the Update operator issues an error message and terminates the job.

**Dropped Tables**

Some tables are created during the execution of a job, and others must be created by the user before the job begins. With the Update operator, target tables must exist on the Teradata Database when the Update operator 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 operator during the cleanup phase if no errors are detected during the acquisition phase or the application phase. The work table is dropped by the Update operator during the cleanup phase if no errors are detected during the acquisition phase or the application phase. The log table is dropped by the Update operator after the job completes successfully.

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

**Caution:** Care must be taken if the target tables are manually dropped using a BTEQ script. If something goes wrong with an Update operator job, and you drop the target table manually and then try to rerun the job, you may lose the original data. This is because all rows are actually placed into worktables and they remain there until the Update operator job reaches the application phase, at which time the rows are copied to the real Target tables.

**Operational Considerations**

**Space Requirements**

Always estimate the final size of the Update target table, and ensure that 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 operator pauses the Update operator job. When this happens, you must allocate more space to the database before you can restart the job.

**Sessions and Instances**

The values that you specify with the Update operator MinSessions and MaxSessions attributes are not the only factors that limit the number of sessions that the Update operator establishes with the Teradata Database. The other limiting factors are:

- The Teradata Database limit of one session per available AMP
- The platform limit on the maximum number of sessions per application.
- On network-attached UNIX, Linux, and Windows client systems, this value is defined in the COP Interface software file, clispb.dat, under the `max_num_sess` variable.
• On channel-attached z/OS client systems, this value is defined in the **HSHSPB** parameter under the IBCSMax setting.

• On channel-attached z/OS client system, use the TDP SET MAXSESSIONS command to specify a platform limit.

• The limit of the network protocol software on network-attached systems

When the Update operator executes, the actual session limit is determined by the first limiting factor that is encountered.

For information on sessions and instances, see “Optimizing Job Performance with Sessions and Instances” on page 80.

**Limits on Update Jobs**

The Update operator requires one active load job. In addition, an active task is needed and counted towards the maximum number of active load tasks permitted only when an operator is running, not for each DEFINE OPERATOR statement in a script that defines an Export, Load, or Update operator. A script may have many operators defined and only use one of them. The number of active load tasks is configurable in the Teradata environment.

In an APPLY-SELECT statement, there is one active task for each operator that is specified. An active task is counted for the database to which the operator has connected.

For example, an Export operator is connected to DBS “A,” sending data to the Load operator connected to DBS “B.” There is an active load job counted for DBS “A” and one on DBS “B.” Similarly, there could be one Export operator defined in a script, but two instances are active because there are two SELECT statements (connected with a UNION ALL) in the APPLY-SELECT statement.

So the number of active tasks does not necessarily relate to how many DEFINE OPERATOR statements that are in a script. The number of active tasks can be determined by how many operators are specified in the APPLY-SELECT statement.

In most Teradata systems, a maximum of 15 active load tasks is permitted; however, the number of concurrent load tasks can be dynamically allocated and may exceed 15, depending on the version of the Teradata Database being used. Consult the Teradata Parallel Transporter Reference for more information.

**Delete Task Option**

The Delete Task option is unique to the Update operator. It deletes rows more quickly than using a single DELETE SQL statement.

When the DeleteTask attribute is set to ‘Y’, rows are deleted from a single table based on values other than a unique primary index (UPI) equality condition. A Delete Task option uses a single session and a single instance.

The Delete Task option is a good choice when a large percentage of rows in a table need to be deleted, such as deleting all rows with a transaction date prior to a specified date.
The Delete Task option operates very similarly to the standard DELETE statement in the Update operator, with the following differences:

- Deleting based on non-index columns is normal for the Delete Task option.
- Deleting based on a primary index, although possible, has certain limitations:
  - An equality test of a UPI value is not permitted.
  - An inequality test of a UPI value is permitted.
  - An equality test of a NUPI value is permitted.
- A single DELETE statement is used in the APPLY statement.
- The Delete Task option does not include an acquisition phase because there are no varying input records to apply.
- The application phase reads each target block and deletes qualifying rows.
- Altered blocks are written back to disk.

When the Delete Task option is specified, the Update operator functions as a standalone operator, that is, not as the usual consumer operator that reads from a source data stream. The exception is when the Delete Task is invoked by an APPLY statement that includes a WHERE clause, and the source data stream contains only a single row. In this case, the Update operator with the Delete Task option still functions as a consumer operator.

The following rules apply to a Delete Task operation regardless of whether it functions as a standalone or consumer operator:

- Only one session is connected.
- Only one instance is specified.
- Only one DML group is specified.
- Only one DML statement in the DML group is specified.
- Only a single DELETE statement is used.
- Only one target table is specified.
- The first error table is not used and is ignored.
- Only one data record is provided if using a WHERE clause.

**Using Delete Task**

Using the Delete Task option of the Update operator can run as either a consumer operator or as a standalone operator, depending on the construction of the APPLY statement:

- As a standalone operator that does *not* attempt to retrieve a row from the data stream:

  ```sql
  APPLY
  <SQL DELETE statement>
  ```

- As a consumer operator that attempts to read a row from the data stream:

  ```sql
  APPLY
  <SQL DELETE statement>
  SELECT FROM ...
  ```

Specify a single DELETE statement in the APPLY statement:

```sql
APPLY
```
In this case, the Update operator runs as a standalone operator. (The Update operator is a consumer operator and there is no producer operator, so there is no SELECT statement).

You can also specify a DELETE statement in which information in the DELETE requires some data. In this case, the APPLY needs a SELECT statement:

```sql
APPLY
(
    'DELETE FROM TABLE xyz WHERE Field1 = :Field1;'
) SELECT * FROM OPERATOR (FILE_READER [1] . . .
```

In this case, the Update operator is running as a consumer operator and it requires exactly one row of data. That row of data is passed to the Teradata Database, which extracts the data from the column as specified in the WHERE clause.

Another example, similar to the first one, is where the Update operator runs as a standalone operator but the DELETE statement has a WHERE clause:

```sql
APPLY
(
    'DELETE FROM TABLE xyz WHERE Field1 = ''abc'';'
) ;
```

In this case, there is a WHERE clause, but the information in the WHERE clause does not require data from a producer operator.

**Note:** When the Update operator runs as a standalone operator, no schema is necessary. That is, you do not need to define a schema using the DEFINE SCHEMA statement. This is because no data is needed from a producer operator for the job.

**Why Choose the Delete Task Option?**

A simple SQL DELETE statement can usually accomplish the same result as the Delete Task option, but the Delete Task option is usually preferred because it requires fewer system resources, and therefore generally performs better than an SQL DELETE.

- The Delete Task option does not use the Transient Journal so it uses less disk space, requires less I/O, and runs faster.
- The Delete Task option aborts without rollback as opposed to the SQL DELETE, which uses the Transient Journal to roll back all changes. The Delete Task option only moves forward.

**Example 1: Delete Task Option**

Following is an example of a job that uses the Delete Task option to delete rows from a table named Customer, where customer_number is less than the hard-coded job script value of 100000:

```sql
DEFINE JOB DELETE_TASK
```
DESCRIPTION 'Hard-coded DELETE FROM CUSTOMER TABLE'
{
  DEFINE OPERATOR UPDATE_OPERATOR
  DESCRIPTION 'Teradata PT UPDATE OPERATOR'
  TYPE UPDATE
  SCHEMA *
  ATTRIBUTES
  {
    VARCHAR TargetTable = 'Customer',
    VARCHAR TdpId = @Tdpid,
    VARCHAR UserName = @UserId,
    VARCHAR UserPassword = @Pwd,
    VARCHAR AccountId,
    VARCHAR LogTable = 'DeleteTask_log',
    VARCHAR DeleteTask = 'Y'
  };

  APPLY
  {
    'DELETE FROM CUSTOMER WHERE CUSTOMER_NUMBER LT 100000;' 
  }
}

TO OPERATOR( UPDATE_OPERATOR [1] );
);

Notice the following about this script:

- Setting the attribute DeleteTask to 'Y' makes this execution of the Update operator a Delete Task.
- The example uses the hard-coded value of 100000 in the deletion criterion.
  For increased flexibility, this value could be specified as a job variable, such as:
  'DELETE FROM CUSTOMER WHERE CUSTOMER_NUMBER LT ' || @Custno || ';
  The value for the variable in this expression can come from a job variable file, or it can come from the command line:
  tbuild -f <job file name> -u "Custno = '100000'"
  
- The script still requires the SCHEMA * clause even though the Update operator is functioning as a standalone operator.
- The APPLY statement must specify a single SQL DELETE statement.
- The LogTable attribute is always required for the Update operator.

For additional information about using variables, see Chapter 27: “Advanced Scripting Strategies.”

**Example 2: Delete Task Option**

The following example accomplishes the same purpose as the previous Delete Task example, but with a slightly different technique. Rather than hard-coding values in the deletion criterion, the value in this example is supplied from an external file by the DataConnector operator and a data stream. The SQL host variable (:CustNo) represents the value in the DELETE statement.
In this case, the Update operator, used as a Delete Task, is a consumer operator because it receives input from a data stream. Differences between this approach and the first example are shown in bold text.

```
DEFINE JOB DELETE_TASK_PARAM
DESCRIPTION 'External File DELETE FROM CUSTOMER TABLE'
(
  DEFINE SCHEMA CUST_NUM_SCHEMA
  DESCRIPTION 'CUSTOMER NUMBER SCHEMA'
  (
    Cust_Num INTEGER
  );

  DEFINE OPERATOR UPDATE_OPERATOR
  DESCRIPTION 'Teradata PT UPDATE OPERATOR'
  TYPE UPDATE
  SCHEMA CUST_NUM_SCHEMA
  ATTRIBUTES
  ( VARCHAR TargetTable = 'Customer',
    VARCHAR TdpId = @Tdpid,
    VARCHAR UserName = @Userid,
    VARCHAR UserPassword = @Pwd,
    VARCHAR AccountId,
    VARCHAR LogTable = 'DeleteTask_log',
    VARCHAR DeleteTask = 'Y'
  );

  DEFINE OPERATOR DATA_PRODUCER
  DESCRIPTION 'DATA CONNECTOR OPERATOR'
  TYPE DATACONNECTOR PRODUCER
  SCHEMA CUST_NUM_SCHEMA
  ATTRIBUTES
  ( VARCHAR OpenMode = 'Read',
    VARCHAR Format = 'Formatted',
    VARCHAR IndicatorMode,
    VARCHAR FileName = 'Single_Row_File'
  );
  APPLY
  ( 'DELETE FROM CUSTOMER WHERE CUSTOMER_NUMBER LT :CustNo')
  TO OPERATOR (UPDATE_OPERATOR [1])
  SELECT * FROM OPERATOR (DATA_PRODUCER[1]);
)
```

**Explanation**

Notice the following in this script:

- When using this approach, the DataConnector operator write a single row to the data stream so its input file contains a single record.
- A schema must be defined in order for the input value to be read.
• An SQL host variable (:CustNo) is used in the DELETE statement.
• The colon (:) symbol prefixed to CustNo specifies to the Teradata Database that the value comes from a source external to the SQL DELETE statement.
SECTION 5 Launching, Managing, and Troubleshooting a Job
CHAPTER 21
Launching a Job

This chapter explains how to set up Teradata PT job management options in the \texttt{tbuild} command before using that command to launch a job.

Topics include:

- Setting \texttt{tbuild} Options
- Setting Checkpoint Options
- Launching a Teradata PT Job

## Setting \texttt{tbuild} Options

The \texttt{tbuild} command, which is used to launch a Teradata PT job, enables you to specify job management options \textit{before} you launch a job. Take time to become familiar with the available \texttt{tbuild} options, determine which may be useful for your job, and whether or not you need to reset default values before you launch the job.

The following sections describe commonly used \texttt{tbuild} options and how to employ them. For details about all \texttt{tbuild} options, see \textit{Teradata Parallel Transporter Reference}.

### Specifying a File Name

The \texttt{-f <file name>} option is required in the \texttt{tbuild} command. The filename references the name of the file containing the job script you want to launch, as follows:

- If you run \texttt{tbuild} from the same directory that contains the job script, only the file name of the job script is required.
- If you run \texttt{tbuild} from a directory other than the directory that contains the job script file, the \texttt{-f <file name>} option must contain the path to the file.

### Specifying a Job Name

Specification of a job name differs depending on operating system.

**On UNIX or Windows**

Although it is not required, Teradata recommends that all jobs specify a job name on the \texttt{tbuild} command line. Lack of a specified job name complicates later access to other job-related features, such as checkpoint files.
Teradata PT allows any name specification within the 30 character limit. A common practice is to use the name of the job as specified in the DEFINE JOB statement of the script, followed by some form of sequence number (possibly a date stamp) that uniquely identifies the particular run of the job.

If you do not name the job in the `tbuild` command on UNIX and Windows, Teradata PT uses the logon userid followed by the hyphen and a Teradata PT-generated job sequence number.

**On z/OS**

Job scripts running on z/OS are executed via JCL, and require a JOB statement, which in turn requires specification of a jobname. Many users employ the TSO userid with a unique character appended.

**Special Considerations for Running Unnamed Jobs**

If you do not uniquely name your jobs on UNIX or Windows, or you do not supply a unique high-level qualifier for your z/OS jobs, Teradata recommends that you do the following:

- Run Teradata PT jobs only one at a time.
- Restart any interrupted job before running any other jobs. If you need to complete the interrupted job first, or if it is unable to complete successfully upon restart, then you must manually delete the checkpoint files from the checkpoint directory or from the z/OS catalog.

**Effect of Unspecified Jobname on Checkpoint Files**

Naming a Teradata PT job using the `jobname` option in the `tbuild` command is strongly recommended so that each job can have unique checkpoint file names.

If a jobname is not specified in the `tbuild` command, Teradata PT uses a default jobname to name the checkpoint files. The checkpoint files for all jobs executed under that userid will have the *same* name. The result for any job that follows a failed job will be for it to try to restart using the failed job checkpoint files, which are automatically retained by Teradata PT for all failed jobs. Whenever this happens, this newer job will not be successful.

For information on checkpoints, see “Setting Checkpoint Options” on page 264.

**Jobname Syntax**

The syntax to specify a job name is:

```
tbuild -f <filename> <jobname>
```

If the job name is omitted, the job will be given a default name:

```
<user name>-<Teradata PT job sequence number>
```

The default name consists of the symbolic username followed by a hyphen (“-”) and a sequence number that the Teradata PT increments for each job submitted.

**Example 1**

A valid `tbuild` command is shown below, if you are logged on to the system “labmachine” as user1, and you enter the `tbuild` command without a job name:
tbuild -f fivetableload

The resulting job ID would be user1-<sequence number>. Your user ID is the job name, and the “-<sequence number>” (say, “-38”) would be a sequential number that is incremented each time a Teradata PT job is run.

When a job is run using tbuild, a statement displays on the console to show the job ID and the system name. For example:

job id is user1-38 running on labmachine

Example 2

On the same system, if you specify a job name of week7 as shown below:

tbuild -f fivetableload week7

The specified job name overrides the default, and the job ID is week7-<sequence number>.

The specified job name is used, and the sequential number is incremented to -39 assuming that this is the next job run on this system.

Assigning Job Variables on the Command Line

Job variables are often used in place of attribute values and other specifications in a job script. Teradata PT provides two tbuild command options for assigning values to job variables on the command line. Values assigned to job variables through a command line option are in force only for the job being submitted.

-u Command Option Job Variable Assignments

The tbuild -u option allows you to assign values to one or more job variables on the command line. The set of assignments are enclosed in double quotes ("). If more than one assignment is made, they are separated by commas. Assigned values are enclosed in single quotes (’).

For example, if the job script contains the following attribute list:

ATTRIBUTES
(
    VARCHAR UserName = @UsrID,
    VARCHAR UserPassword = @Pwd
);

you can assign values to the variables “UsrId” and “Pwd” as follows:

tbuild -f daily_job.tpt -u "UsrID = 'John Doe', Pwd = 'ABC123' "

Values assigned to job variables on the command line take precedence over values assigned to the same variables by all other supported methods of job variable assignment.

-v Command Option Job Variables File

The tbuild -v option allows you to execute job variable assignments that are stored in a local job variables file that is identified on the command line. These assignments have the same format as those of the -u option, except that no comma is needed if you specify one assignment per line (or record on z/OS). To effect the same assignments as in the -u option using the -v option, the local job variables file my_attrs.txt would contain:
UsrId = 'John Doe'
Pwd = 'ABC123'

The associated `tbuild` command would be:
```
tbuild -f daily_job.tpt -v my_attrs.txt
```

A value assigned to a job variable from a local job variables file takes precedence over a value assigned to the same variable from any other source, except through the `-u` option. For setup details, see “Setting Up the Job Variables Files” on page 68.

### Specifying that the Job Can Continue Even If a Fatal Error Is Encountered

Teradata PT provides the `tbuild -n` option to allow a job to continue even if it encounters a fatal error. `tbuild -n` is valuable mainly for multi-step jobs that group related extract and loading steps together. If a step fails, the job can pick up from where it left off based on the checkpoint taken within that failed step. While grouping multiple extract and load steps into a job minimizes the use of system resources and reduces the number of jobs to be executed, you must take the following into consideration before using the `-n` option:

- Job steps must be independent of each other. The result of one step should have no effect on the other steps.
- Checkpoints of a previous failed step would be erased before the next step starts; which means the job is not restartable.
- Each of the failed steps needs to be evaluated and redoing a failed step may require a separate job or procedure to be executed. This is because the other steps within the same job might have been successfully executed or cannot be restarted.

### Setting Checkpoint Options

A checkpoint is a job restart point created by a Teradata PT job. Should the job be interrupted for any reason, it can be restarted from the checkpoint instead of from the beginning of the job. Checkpoints help guarantee that the work performed by an interrupted job up to the checkpoint will not have to be redone.

When a Teradata PT job takes a checkpoint, the producer operator in the currently-executing job step stops putting rows in the output data stream and the consumer operator processes all the rows in the input data stream, committing DBS updates or flushing output file buffers. All executing operators write records to the job checkpoint files with the information that will allow them to resume their processing, with no loss or duplication of data, at the point the checkpoint was completed.

Teradata PT offers the following options related to checkpointing:

- Specify an alternate location for checkpoint directory.
- Specify a user-defined checkpoint interval.
- Specify a limit to the number of times a job will automatically restart.
Types of Checkpoints

The following table describes the various types of checkpoints taken in Teradata PT jobs.

<table>
<thead>
<tr>
<th>Checkpoint Type</th>
<th>Function</th>
</tr>
</thead>
</table>
| Basic (default)       | If the **tbuild** command does not specify a checkpoint interval, the job will automatically take just two checkpoints during each job step that has consumer and producer operators:  
  - a Start-of-Data checkpoint  
  - an End-of-Data checkpoint  
  These two checkpoints allow Teradata PT jobs to restart automatically, without requiring user intervention, if the interruption was caused by a Teradata server restart or deadlock situation. The default checkpoints can also be used for a manual restart. In either case, the job will restart after the last checkpoint written to the files:  
  - If the End-of-Data checkpoint was taken, the work accomplished between these two checkpoints will not have to be repeated by the restarted job.  
  - If the job failed before the End-of-Data checkpoint was taken whatever work was accomplished after the Start-of-Data checkpoint was taken will have to be repeated by the restarted job.  
  This default checkpoint protection against redoing work is quite minimal, so Teradata recommends the use of interval checkpointing.                                                                                                     |
| Interval Checkpointing | When you specify a checkpoint interval for a Teradata PT job using the **tbuild** `-z` command, the job will take a checkpoint for each specified interval (in seconds), for each job step with producer and consumer operators.  
  If a job with interval checkpointing fails to run to completion and is later restarted, the only work that will have to be performed over again is the work done after the last checkpoint was taken. This option offers increased fault tolerance for long running jobs containing a substantial amount of data to be loaded/exported.                                                                                           |
| Direct Command        | A Teradata PT job can also be directed to take a checkpoint at any time through the **twbcmd** command, either explicitly with the JOB CHECKPOINT command option, or implicitly with the JOB PAUSE command option, which suspends job execution after the checkpoint is taken.  
  For details, see the section on Teradata PT utilities in *Teradata Parallel Transporter Reference*.                                                                                                                                                             |
| Operator Initiated    | The DataConnector operator automatically initiates a checkpoint after processing all the input files found during an interval-driven scan of a directory.                                                                                                                                   |

Specifying the Checkpoint Interval

Use one of the following methods to specify the checkpoint interval.

- On the **tbuild** command line using the `-z` option

  `tbuild -f <script file name> -z <checkpoint interval>`
The -z option sets the checkpoint interval to the number of seconds specified. Experiment with setting the checkpoint interval when doing trial runs of a job to determine the optimum interval. You can also use this method to override a checkpoint interval specified using the SET CHECKPOINT INTERVAL option in the DEFINE JOB statement for the job script, during a particular execution of the job, including restarts.

- In the job script using the SET CHECKPOINT INTERVAL statement, as shown in the following examples:
  
  SET CHECKPOINT INTERVAL 160 SEC
  
  or

  SET CHECKPOINT INTERVAL 12 MINUTES

  The checkpoint interval can be specified in a job script between the last DEFINE statement and the APPLY statement(s). This method is appropriate for established jobs for which the desired checkpoint interval has been determined and will seldom or never need to be changed from one run to another of the job script. For information, see the section on DEFINE JOB in Teradata Parallel Transporter Reference.

  The checkpoint interval must be specified either in SECONDS (or abbreviation SEC) or MINUTES (or abbreviation MIN).

  **Note:** If the checkpoint interval is specified both in the job script and with the `tbuild -z` command option, the -z option takes precedence.

  **Note:** If the checkpoint interval is set to zero, calls for checkpoint function to any access modules (attached via the DataConnector Operator) will be bypassed. Should an access modules checkpoint operations be particularly resource intensive, this feature allows for those checkpoint operations to be bypassed in cases where the user feels that checkpoint recovery is not critical.

### Effects of Interval Checkpointing on Job Performance

Checkpoints increase Teradata PT job overhead. In terms of resources, each executing operator must do the additional work of writing its internal operating state to the checkpoint file, so that it could be restarted from the information in the checkpoint file. In terms of running time, each executing operator must first finish all in-progress work, take its checkpoint, and then wait (when necessary) until all the other operators have finished taking their checkpoints.

Frequent checkpoints can guarantee that only a limited amount of work would have to be repeated if the job were interrupted and then later restarted, because it shortens the time between an error event and the checkpoint. However, specifying a very short checkpoint interval can significantly increase job running time. Choosing a checkpoint interval is a trade off between the cost in increased job run time and the potential reduction in repeated work if the job must be restarted.

Here is an example of a Teradata PT job that loads 20,000,000 rows with 4 instances each of the producer and consumer operators:

- Specifying a checkpoint interval of 10 seconds increased the job’s running time by 7.3% and its host CPU time by 3.3%.
• Specifying a checkpoint interval of 5 seconds increased the job’s running time by 20% and its host CPU time by 6.6%.

Even though interval checkpointing may have a substantial performance cost, its usefulness during a possible restart make interval checkpointing a Teradata “best practice” recommendation.

**How Checkpoints Affect Job Restarts**

Jobs sometimes fail to achieve a successful completion. Checkpoints enable failed jobs to be restarted from the last checkpoint before the failure. Before you restart a job try to understand what caused the failure. You may need to take remedial actions before restarting the job to prevent the failure from occurring again. For further information, see “Restarting A Job” on page 318.

When you are ready to restart the job, re-issue the same `tbuild` command used to submit the job the first time. The job will restart at the job step that failed, at the point in that step where the last checkpoint was taken prior to the failure.

**Launching a Teradata PT Job**

The following `tbuild` command executes the job script in file `/prod/hdqts.load` and uses the local job variables file named “attributeFile”:

```
tbuild -f /prod/hdqts.load -v attributeFile
```

**Command-Line Handling of String Delimiters in Script Parsing**

A few simple syntax rules govern the `tbuild` command, and the other Teradata PT commands, at the command prompt on the various supported platforms.

These examples suggest that single quotes should be avoided as string delimiters for command-line arguments on all platforms.

**On All Platforms**

- Double-quote characters ("") are interpreted as string delimiters, and are stripped from the character strings they enclose.
- A double-quote character as a data character in a string must be escaped with the backslash character (\). For example:

```
tlogview -j DTAC_FLD1@offshore36-746 -w "TASKNAME="SELECT_20001""
```

**On Windows Systems**

On UNIX systems, single quotes (') are stripped away so that quotes are not part of the string. But on Windows systems, the quote becomes part of the string. This means that if you run something like this at the Windows command prompt:

```
C:\>tlogview -l 'C:\Program Files\Teradata\client\<version>\Teradata Parallel Transporter\logs\testjob.out'
```

---

*Teradata Parallel Transporter User Guide*
the filename is read as 'C:\Program......testjob.out', including the quotes, which is not the right file name.

**On UNIX Systems**

- Single-quote characters are also recognized as string delimiters and are stripped from the character strings they enclose.
- A single-quote character can be a data character only when it occurs in a string delimited by double-quote characters. For example, in the command:
  
  ```
  tbuild -f test_job.twb -u "verb="Couldn't""
  ```

  the argument of the -u option passed to the tbuild program is the string verb= “Couldn’t”.

**On z/OS Batch Systems**

- Command line parameters are entered with the PARM field of the EXEC JCL statement for z/OS batch jobs.
- Single-quote characters are recognized as the PARM field delimiters and are stripped from the character strings they enclose.
Managing an Active Job

You can manage an active Teradata PT job using the following Teradata PT command line utilities:

- `twbstat`
- `twbcmd`
- `twbkill`

Using `twbstat` to List Currently Active Jobs

The `twbstat` command displays the names of currently active Teradata PT jobs.

What `twbstat` Does

**Note:** This command is not available on z/OS systems.

The following example uses the `twbstat` command to return a list of the Teradata PT jobs currently running on the system.

The following `twbstat` command:

```
twbstat
```

Creates the following output:

```
Using job directory/home/cl151001/jobs
Jobs running: 3
c1151001-112
lol42000-133
dcc13370-147
```
Using the twbcmd Command to Monitor and Manage Job Performance

The twbcmd command monitors and manages Teradata PT job performance.

**Note:** The twbcmd command is packaged as a z/OS load module in a single library, a required PDS/E, as part of the SMP/E installation procedure.

**What the twbcmd Command Does**

There are two kinds of twbcmd commands:

- Job-level commands
- Operator-level commands

**twbcmd Job-Level Commands**

The table below lists and briefly describes each twbcmd job-level command option. For the syntax and detailed descriptions for each command option, see *Teradata Parallel Transporter Reference*.

<table>
<thead>
<tr>
<th>Command Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JOB CHECKPOINT</td>
<td>Takes an immediate checkpoint, then continues the job.</td>
</tr>
<tr>
<td>JOB PAUSE</td>
<td>Takes an immediate checkpoint, then suspends processing.</td>
</tr>
<tr>
<td>JOB RESUME</td>
<td>Resumes a paused job.</td>
</tr>
<tr>
<td>JOB TERMINATE</td>
<td>Takes an immediate checkpoint, then terminates the job. The job retains the</td>
</tr>
<tr>
<td></td>
<td>checkpoint files, and is therefore restartable.</td>
</tr>
<tr>
<td>JOB STATUS</td>
<td>Writes a status record for each active operator instance to the TWB_STATUS</td>
</tr>
<tr>
<td></td>
<td>log, and displays row processing statistics while continuing the job.</td>
</tr>
</tbody>
</table>
twbcmd Job-Level Command Examples

The following examples show how to use twbcmd job-level commands to accomplish these job management objectives:

- Take a checkpoint
- Take a checkpoint and then terminate a job
- Pause and resume a job
- View and log the status of a job

**Force a job to take an immediate checkpoint**

An active job can be directed to take a checkpoint using the external command interface. Upon receiving the checkpoint request, each operator instance immediately takes a checkpoint rather than waiting for the checkpoint interval to expire. After the checkpoint completes, the job continues to process data.

Use one of the following commands to force a job to take a checkpoint, where *job ID* is the name of the target Teradata PT job (determined by using the `twbstat` command).

- On z/OS, send an external command to Teradata PT jobs using the console MODIFY command:
  
  ```
  F <job ID>,APPL=job checkpoint
  ```

- On all other platforms, use the following command:

  ```
  twbcmd <job ID> job checkpoint
  ```

**Force a job to take an immediate checkpoint and then terminate**

When the `twbkill` command is used to terminate a job, it does not automatically take a checkpoint, which means that restarting the terminated job reprocesses everything done after the last scheduled checkpoint. This can cause errors, such as the reinsertion of rows.

To avoid the problems caused by such reprocessing, use the following `twbcmd` option instead, which creates a checkpoint and then terminates the job.

Do one of the following, where *job ID* is the name of the target Teradata PT job (determined by using the `twbstat` command):

- On z/OS: External commands are sent to Teradata PT jobs using the console MODIFY command:
  
  ```
  F <job ID>,APPL=job job terminate
  ```

- On all other platforms:

  ```
  twbcmd <job ID> job terminate
  ```

**Pause and then resume a job**

Sometimes resources are tied up, tables are locked, or jobs get out of sync. External commands allow you to avoid terminating jobs under these conditions. Use the following procedure to temporarily suspend the flow of data to control job timing and system resources.
1. Do one of the following to pause a job, where job ID is the name of the target Teradata PT job (determined by using the twbstat command):
   - z/OS:
     F <job ID>,APPL=job pause
   - All other platforms:
     twbcmd <job ID> job pause

2. To resume the job, do one of the following:
   - z/OS:
     F <job ID>,APPL=job resume
   - All other platforms:
     twbcmd <job ID> job resume

Determine the status of all active jobs

Issue one of the following commands, where job ID is the name of the target Teradata PT job, to determine the status of all active jobs:

1. Issue one of the following commands, where job ID is the name of the target Teradata PT job (determined by using the twbstat command):
   - z/OS:
     F <job ID>,APPL=job status
   - All other platforms:
     twbcmd <job ID> job status

2. The following will happen:
   - All active operators write a status record to the TWB_STATUS log.
   - The console displays the current count for rows sent and received.
**twbcmd Operator-Level Command**

The table below lists and briefly describes `twbcmd` syntax elements for the operator-level command. For the complete syntax for the operator-level command, see *Teradata Parallel Transporter Reference*.

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
</table>
| rate=statementRate | Option that specifies the maximum number of DML statements per minute the Stream operator can submit to the Teradata Database. Use the `twbcmd` Rate option to slow down a Teradata PT job for other higher priority jobs and to speed it up again after the priority job has completed. **Note:** When a job step contains multiple occurrences of the Stream operator with differing Rate values, Teradata PT will automatically use the lowest rate value for all instances. The specified Rate value must be either:  
  - a whole number greater than zero  
  - unlimited  
  **Note:** The default statement rate, if not set using either the Stream operator Rate attribute or by `twbcmd`, is unlimited. Specifying ‘unlimited’ for the `twbcmd` Rate value means you are changing the value back to the default after having set the value in the Stream operator. When the `twbcmd` Rate option is used, the Stream operator changes the statement rate to the new value and displays a message showing the new value. If the specified rate is greater than the packing factor, the Stream operator will send the number of rows equal to the packing factor. |
| periodicity=periodicity | Option that specifies that the DML statements sent by the Stream operator to the Teradata Database be as evenly distributed as possible over each one minute interval. The *periodicity* value sets the number of sub-intervals per minute. For instance, if the *rate* is 1600 and the *periodicity* is 10, then the maximum number of statements submitted is 160 (1600/10) every 6 (60/10) seconds. Valid values are between 1 and 600. The default value is 4, that is, four 15 second intervals per minute. If the statement rate is unlimited, then the *periodicity* value will be ignored. **Note:** The periodicity can also be specified in a DEFINE OPERATOR statement, by using the Stream operator Periodicity attribute. When both values are present, the `twbcmd periodicity` value will supersede the Stream operator Periodicity attribute value. For information on the DEFINE OPERATOR command, see “DEFINE OPERATOR” in *Teradata Parallel Transporter User Guide*. |
twbcmd Operator-Level Command Example

The following example shows a typical case in which the allowable rate is changed using an operator-level twbcmd. For required syntax, see Teradata Parallel Transporter Reference.

A Teradata PT job named Sales_24_by_7 has a job step that employs the Stream operator. Stream operator executes continuously. The DEFINE OPERATOR statement for Stream operator in the job script includes the following attribute declarations:

```
INTEGER Rate = 50,
VARCHAR OperatorCommandID = 'Sales_Inflow'
```

Suppose the volume of incoming sales transactions is increasing and a backlog of unprocessed transactions is beginning to develop. To double the maximum rate (per minute) at which the Stream operator is allowed to send DML statements to the Teradata Database, issue the following command:

```
twbcmd Sales_24_by_7 Sales_Inflow rate=100
```

Or, use the following equivalent (MODIFY) command on z/OS, assuming the z/OS job name is SLS24X7 and the value of attribute OperatorCommandID is INFLOW MODIFY, as follows:

```
F SLS24X7,APPL=INFLOW RATE=100
```

For this command to be successful, the DEFINE OPERATOR statement for the Stream operator must declare the OperatorCommandID attribute and must assigned a value, either in the declaration itself (as above), or where it is referenced in the long-running job step. This enables the Teradata PT command processor to identify the operator process to which it will direct the requested change in the maximum DML statement rate.

**Note:** The statement rate can also be specified in a DEFINE OPERATOR statement, by using the Stream operator Rate attribute. When both values are present, the twbcmd rate value will supersede the Stream operator attribute Rate value.

Using twbkill to Terminate a Job

The twbkill command causes a Teradata PT job to immediately force all of its executing tasks to terminate without taking any checkpoints. Because of this, it should only be used in emergencies.

**twbkill Example**

Use the following twbkill command,

```
twbkill wilson-235
```

to terminate all tasks in the designated Teradata PT job.

An error message results if the termination is not successful.

The above command creates the following output:

```
# twbkill
Using job directory /home/wilson/jobs
wilson-235 killed
```
This chapter describes post-job considerations.

Topics include:

- Post-Job Checklist
- Exit Codes
- Accessing and Using Job Logs
- Accessing and Using Error Tables
- Effects of Error Limits
- Dropping Error Tables
- Restart Log Tables
- Strategies for Evaluating a Successful Job

Post-Job Checklist

The following procedure describes the tasks you should complete at the end of each job. The sections that follow describe each of these topics in greater detail.

1. Examine the exit code at the end of the job to determine whether or not the job was successful. Exit codes are also issued for each job step.
   For information, see “Exit Codes” on page 276.

2. Examine the job logs and error tables to understand the details of how the job executed, what warnings were issued, and if the job failed, which errors caused the failure.
   For information accessing job logs, see “Accessing and Using Job Logs” on page 277.
   - If the job failed to complete, refer to the troubleshooting procedure in Chapter 24: “Troubleshooting a Failed Job,” for instructions on using the job logs and error tables for failure analysis and determination of corrective action.
     
     Note: Some actions suggested below for successful jobs may also apply to the successful portions of failed jobs.

   - Even if the job completed successfully, action may still be required based on error and warning information in the job logs and error tables.

3. Determine whether or not further action is required.
   For information, see “Strategies for Evaluating a Successful Job” on page 288.

4. Take corrective actions to optimize the job and reduce the susceptibility to future failure.
Exit Codes

Each Teradata PT job returns exit codes indicating the success or failure of the job or job step. You can determine the exit code in the following ways:

- Monitor the console display to see how each job step runs. An exit code is returned at the successful completion of each step.
- Check the console display when the job completes to see the exit code for the entire job. The exit code at the end of the job is the highest level of error that occurred during execution of the job, and may not represent the level of errors or warnings that occurred within individual job steps.

The following table describes Teradata PT exit codes:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit Code = 0</td>
<td>The Teradata PT job or job step completed successfully with at most, minor warnings.</td>
</tr>
<tr>
<td>Exit Code = 4</td>
<td>The Teradata PT job or job step completed successfully, but issued one or more warnings.</td>
</tr>
<tr>
<td>Exit Code = 8</td>
<td>A user error, such as a syntax error in the script, terminated the job.</td>
</tr>
<tr>
<td>Exit Code = 12</td>
<td>A fatal error terminated the job. A fatal error is any error other than a user error, for example:</td>
</tr>
<tr>
<td></td>
<td>• Incompatible data types encountered during reading of data sources.</td>
</tr>
<tr>
<td></td>
<td>• Data errors exceeding the value specified in the ErrorLimit attribute.</td>
</tr>
<tr>
<td></td>
<td>• Insufficient system resources, such as shared memory or semaphores, to execute the job.</td>
</tr>
</tbody>
</table>

Observe the following when evaluating exit codes:

- Even though a job runs to completion, it may experience errors or warnings that require further action. Be sure to check the job logs and error tables of completed jobs to identify any errors or warnings that may have occurred, so that you can determine whether or not any action is required.

For information on the types of warnings that may occur and actions that may be required on successful jobs, see “Strategies for Evaluating a Successful Job” on page 288.

- If the tbuild -n option is used, it allows the Teradata PT job to continue even if there is a failure (an exit code of 8 or 12) in one of the steps.

For details on how to use -n to specify that the job can continue when an exit code of 8 or 12 is returned, see “Setting tbuild Options” on page 261.
Chapter 23: Post-Job Considerations

Accessing and Using Job Logs

Each time a Teradata PT job runs it generates log information that provides a running account of job activities and milestones, performance metadata, and any warnings or errors the job encountered.

Teradata PT automatically provides three types of job logs:

- The console log appears on the command line at the point the `tbuild` command was issued to launch the job. This log contains high-level information about Teradata PT operators and infrastructure, and it updates continuously while the job runs.
- The public log contains general information about the job, and is accessed using the `tlogview` command.
- The private log contains job performance metadata and a log of the activities and errors for each operator defined in the job. The private log can be accessed using the `tlogview` command.

Console Log

The console log continuously monitors job progress. It shows only an overview of the most important events related to the execution of the job, such as the completion of job steps or the occurrence of a job error.

In addition, the console will report SQL errors returned by the Teradata Database in response to DDL/DML statements submitted by the job script.

For an example of a console log, see “Console Log for a Successful Job” on page 483.

Public Log

The public log for a Teradata PT job is automatically generated and filed by the job name. The information is updated as the job runs and is presented in the sequence it is encountered.

Public Log Contents

The public log contains the following information about a job:

- Teradata PT Version Number
- An overview of the activities of each operator including stages of operator task execution, errors encountered, warnings, and a summary of data sent and received.
- Number Of Sessions
- Blocksize Used
- Number Of Blocks Created
- Task Status Codes
- Checkpoints Taken
- Restarts Attempted
- Job Elapsed Time
• Job CPU Time

Multiple operators can run within a single job. They all write asynchronously to the same public log. Information in the public log is not sorted, but is written to the log as it is received.

**Example Public Log**

For an example of a public log, see “Public Log for a Successful Job” on page 484.

**Accessing a Public Log by Job Name**

To access the public log for a particular Teradata PT job, enter the following `tlogview` command:

```bash
tlogview -j <job name>-<job sequence number>
```

where:

- `<job name>` is the name of a previously launched job, as specified in the `jobname` parameter of the `tbuild` command.
- `<job sequence number>` is a number generated by the Teradata PT that enumerates the Teradata PT jobs submitted under the current userid since its installation, and which appears in a console message as soon as the job starts executing.

**Accessing a Public Log with No Associated Job Name**

If no job name is specified in the `tbuild` command, Teradata PT automatically names the public log file using the logged-on user ID, a hyphen, and the `<job sequence number>`, resulting in public log file names of `<userid>-1.out`, `<userid>-2.out`, and so on.

To locate a public log with the default assigned name, execute the following command (UNIX shown):

```bash
tlogview -l $TWB_ROOT/logs/<userid>-<job sequence number>.out
```

where:

- `<userid>` is the username employed to log on the job.
- `<job sequence number>` is a number generated by the Teradata PT that enumerates the Teradata PT jobs submitted under the current userid since its installation, and which appears in a console message as soon as the job starts executing.

The example above is for logs on a UNIX system. Public logs are accessible from the following directories, depending on operating system:

- UNIX
  
  ```bash
cd $TWB_ROOT/logs
  ```

- Windows
  
  ```bash
  chdir %TWB_ROOT%\logs
  ```

- Linux
  
  ```bash
cd $TWB_ROOT/logs
  ```

- z/OS
  
  On z/OS platforms you must run a batch job to print out the public log. For information, see the section on “JCL Examples” in *Teradata Parallel Transporter Reference*. 


Private Logs

Private logs are automatically generated and filed by the name specified in the PrivateLogName attribute of all operator definitions (as used in the job) or by a system-generated name based on the user Id. Private logs contain more detail about job activity than public logs, and they separate activity by operator.

Private Log Contents

The private log contains the following categories of information about a job:

<table>
<thead>
<tr>
<th>Log Section Heading</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private log PXCRM</td>
<td>The checkpoint log for the job</td>
</tr>
<tr>
<td>Private log TWB_STATUS</td>
<td>The log of statistical performance metadata for operations carried out by the job.</td>
</tr>
<tr>
<td>Private log TWB_SRCTGT</td>
<td>The log of metadata for operations on the data source and data target carried out by the job.</td>
</tr>
<tr>
<td>Private log &lt;PrivateLogName&gt;</td>
<td>The private log contains a log for the activity of each operator for which the PrivateLogName attribute has been specified and a name assigned as the attribute value in the operator definition. The individual operator logs contain such informations as:</td>
</tr>
<tr>
<td></td>
<td>• Operator name</td>
</tr>
<tr>
<td></td>
<td>• Operator version information</td>
</tr>
<tr>
<td></td>
<td>• Separate sections for each stage of operator task execution, including data/time stamps, SQL submitted, and errors encountered.</td>
</tr>
</tbody>
</table>

Example Private Log

For an example of a private log, see “Private Log for a Successful Job” on page 486.

Accessing All Private Logs

To access all public and private logs for a particular Teradata PT job, enter the following tlogview command:

tlogview -j <job id>-<job sequence number> "*" -g

where:

• <job id> is the job name, if one was supplied with the tbuild command, else the userid of the user who executed the tbuild command.

• <job sequence number> is a number generated by the Teradata PT that enumerates the Teradata PT jobs submitted under the current userid since its installation, and which appears in a console message as soon as the job starts executing.

• “*” requests all log files. This option removes the need to request the files separately. However, it does not include the special options covered in the -v option.

• -g requests that the private log sections be shown separately rather than interspersed.
**Accessing an Individual Private Log**

To access an individual private log, enter the following `tlogview` command:

```bash
tlogview -j <job id>-<job sequence number> -f <private log file name>
```

where:

- `<job id>` is the job name, if one was supplied with the `tbuild` command, else the userid of the user who executed the `tbuid` command.
- `<job sequence number>` is a number generated by the Teradata PT that enumerates the Teradata PT jobs submitted under the current userid since its installation, and which appears in a console message as soon as the job starts executing.
- `<private log file name>` is the script-specified value of the `PrivateLogName` attribute for the operator whose log file is being accessed.

**Other Important `tlogview` Options for Viewing Logs**

In addition to using `tlogview` to access the private logs, you can also add the following specifications to the end of the `tlogview` command string (in any order) to customize the log output:

- Use `-v %<option>` to specify the fields in the log records that will be displayed.
- Use `-w <filter criteria>` to perform filtering on log messages. Only those log messages satisfying the filter criteria will be output by `tlogview`. Without the `-w` option, `tlogview` will select all messages in the public or private logs.

For detailed descriptions of all `tlogview` options and the associated syntax requirements, see the chapter on `tlogview` in *Teradata Parallel Transporter Reference*.

**Log Directory Locations by Operating System**

Output messages are stored in log files, which differ by operating system, as follows:

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX (including AIX, HP-UX, Solaris SPARC, and Linux)</td>
<td>The default directory is: <code>/opt/teradata/client/&lt;version_number&gt;/tbuild/logs</code>&lt;br&gt;The log directory cannot be a relative directory path and the directory. Log directory can be user-specified by modifying the &quot;LogDirectory&quot; entry in the local or global <code>twbcfg.ini</code> file</td>
</tr>
<tr>
<td>z/OS</td>
<td>Logs are user-specified in the Teradata PT job JCL&lt;br&gt;&lt;strong&gt;Note:&lt;/strong&gt; The <code>tlogview</code> command should be run in the batch environment by the appropriate JCL. The <code>tlogview</code> command is packaged as a z/OS load module in a single library, a required PDS/E, as part of the SMP/E installation procedure.&lt;br&gt;For JCL examples, see Appendix C: “z/OS Sample Files.”</td>
</tr>
<tr>
<td>Windows</td>
<td>Default directory is:&lt;br&gt;%ProgramFiles%\Teradata\Client&lt;version_number&gt;\Teradata Parallel Transporter\logs</td>
</tr>
</tbody>
</table>
For Windows and UNIX platforms the log files are located in the logs directory, which is created during the installation in the directory where Teradata PT is installed. For example, if Teradata PT is installed under the /opt/teradata/client/<version_number>/tbuild directory, then the Teradata PT logs are stored under the /opt/Teradata/client/<version_number>/tbuild/logs directory.

The selected part of a log can be written to standard output or to an output file following a defined format.

**Viewing Logs in UTF16 Format**

*Note:* The UTF16 session character set can only be specified on network-attached platforms.

Both private and public logs can be viewed in UTF16 format. Use the -e option with UTF16 as its value in the `tlogview` command line to display the log in UTF-16 characters. For example:

```
tlogview -l <job id>-<job sequence number>.out -e utf16
```

This `tlogview` command displays a public log named `<job id>.<job sequence number>.out` in UTF-16 format. Note that UTF16 is the only supported value of the -e option and is case insensitive.

**Directing Log Output on z/OS Systems**

For directing both private and public log output on z/OS systems, use the `tbuild` command’s -S option. Specify one of three parameters:

- To specify a dsname, where `dsname` is the target dataset name for the logfile:
  
  `-S <dsname>

  *Note:* A fully qualified `dsname` can be specified by enclosing the dsname in single quote marks.

- The DD statement directs the log output to a dataset, where `ddname` is the name for the log file:
  
  `-S DD:<ddname>

- To specify a SYSOUT class, where `class` is the SYSOUT class for the log file:
  
  `-S <class>

**Directing Log Output on non z/OS systems**

Use the `tbuild` command’s -L option to redirect log files to a specific location on a job-by-job basis.

```
-L <jobLogDirectory>
```

where `jobLogDirectory` is the full path name of the directory in which the log file is to be stored.
Accessing and Using Error Tables

Error tables are automatically generated for the Load, Stream, and Update operators in a Teradata PT job, to provide information on Teradata Database errors encountered while writing data to the Teradata Database. Error tables provide more detailed information about errors initially presented in the job logs. Error tables segregate errors into two groups:

- **Error Table1** (Acquisition Error Table) - Reports constraint violations, bad data, and data conversion errors.
- **Error Table2** (Application Error Table) - Contains any rows that cause violations of the unique primary index, for instance duplicate rows. This error table is not used when the target table has a non-unique primary index.

The following operators support error tables:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Generates acquisition and application error tables (ErrorTable1 and ErrorTable2). Error tables are named in one of the following two ways:</td>
</tr>
<tr>
<td></td>
<td>- The operator automatically names the table in terms of the target table, as follows:</td>
</tr>
<tr>
<td></td>
<td>- For ErrorTable1: <code>TargetTableName_ET</code></td>
</tr>
<tr>
<td></td>
<td>- For ErrorTable2: <code>TargetTableName_UV</code></td>
</tr>
<tr>
<td></td>
<td>- The ErrorTable1 and ErrorTable2 attributes name error tables. For information, see the Load and Update operators in <em>Teradata Parallel Transporter Reference</em>.</td>
</tr>
<tr>
<td>Update</td>
<td>Generates only an acquisition error table (ErrorTable), which is equivalent to ErrorTable1. The Stream operator places the table in the database associated with the job script user logon. The error table is named in one of the following two ways:</td>
</tr>
<tr>
<td></td>
<td>- Use the ErrorTable attribute to name the error table. You can also prefix the name with a database name if the table is to be stored in a different database than the one that contains the target table, using the form: <code>DatabaseName.ErrorTableName</code></td>
</tr>
<tr>
<td></td>
<td>For information, see chapters on Load and Update operators in <em>Teradata Parallel Transporter Reference</em>.</td>
</tr>
<tr>
<td>Stream</td>
<td>Generates only an acquisition error table (ErrorTable), which is equivalent to ErrorTable1. The Stream operator places the table in the database associated with the job script user logon. The error table is named in one of the following two ways:</td>
</tr>
<tr>
<td></td>
<td>- Use the ErrorTable attribute to name the error table. You can also prefix the name with a database name if the table is to be stored in a different database than the one that contains the target table, using the form: <code>DatabaseName.ErrorTableName</code></td>
</tr>
<tr>
<td></td>
<td>For information, see chapters on Load and Update operators in <em>Teradata Parallel Transporter Reference</em>.</td>
</tr>
</tbody>
</table>

*Note:* Teradata recommends that you do not use this default error table name. It is a large character string that may lead to data entry errors when accessing the table.
The content and format of error tables is different for each of these operators. For detailed information, see the sections beginning with “Load Operator Errors” on page 304.

Consider the following facts about error tables:

- If a job generates no errors, the error tables are empty. They are automatically dropped at the end of the job, unless the DropTable attribute is set to **No**.
- If errors are generated, error tables are retained at the end of a job.
- To rerun jobs from the beginning, either delete the associated error tables or rename them, otherwise an error message results, stating that the error tables already exist.
- Conversely, to restart a job from a step or checkpoint, an error table must already exist. Do not delete error tables until you are sure you will not have to restart the job.
- To reuse names specified for error tables, use the DROP TABLE statement in the BTEQ utility or the DDL operator to remove the tables from the Teradata Database.

### Mark/Ignore Options for Error Tables

The Stream and Update operators allow you to MARK or IGNORE various types of errors generated during execution of a job. Rows for error types designated as IGNORE will thrown away. Rows for error types designated as MARK are retained in the error table.

**Note:** For Update operator, MARKed rows will only appear in Error Table 2, the application error table. Stream operator has only a single Error Table and MARKed rows will appear there.

For Stream and Update operators:

- DUPLICATE ROWS (for both insert and update operations)
- DUPLICATE INSERT ROWS (for insert operations)
- DUPLICATE UPDATE ROWS (for delete operations)
- MISSING ROWS (both update and delete operations)
- MISSING UPDATE ROWS (for update operations)
- MISSING DELETE ROWS (for delete operations)

For Stream operator only:

- EXTRA ROWS (for both update and delete operations) [default]
- EXTRA UPDATE ROWS (for update operations)
- EXTRA DELETE ROWS (for delete operations)

Enter MARK or IGNORE and the affected row type from the list above immediately following the INSERT, UPDATE, or DELETE statement in the APPLY statement. MARKed items are added to the error tables.

**Note:** If neither option is specified in the APPLY statement, MARK is the default condition.

For details, see the section on the APPLY statement in *Teradata Parallel Transporter Reference.*
Strategy
Consider the following when deciding whether to MARK or IGNORE a particular error type.

- If you need to know about each duplicate, missing or extra row that is encountered during the job, use MARK to send them to the error tables.
- Saving row data and storing it in the error table may slightly degrade overall Teradata PT job performance. When job performance is important and the data is likely to include a high percentage of duplicate, missing, or extra rows, it may be best to IGNORE them.
- Even if minor job performance degradation is not a concern, using MARK to save all of the duplicate, missing, or extra rows may create so much clutter in the error table that it is difficult to read.
- You may need to run a job several times before you can determine the best use of MARK and IGNORE.

Accessing Error Tables
Error tables are stored in the Teradata Database. The following SQL requests access to the error tables shown in the in the examples in “Reading Error Tables” on page 284:

**ErrorTable1**

```sql
SELECT errorcode, errorfield, sourceseq, dmlseq from t2_e1;
```

If the operator definition does not specify a name for the ErrorTable1 attribute, the error table will be named `<TargetTableName>_ET` by default.

**Note:** If the Stream operator AppendErrorTable attribute is set to Yes, the Stream errors for the current job may be found in a table with errors from one or more previous jobs.

**ErrorTable2**

```sql
SELECT dbcerrcode, sourceseq, dmlseq FROM t3_e2;
```

If the operator definition does not specify a name for the ErrorTable1 attribute, the error table will be named `<TargetTableName>_UV` by default.

**Note:** When accessing error tables, you may find it useful to add the expression ORDER BY ErrorCode.

Reading Error Tables
The following are examples of Error Table 1 and Error Table 2, and are associated with the log examples shown in “Example Logs for an Unsuccessful Job” on page 499:

**Example of Error Table 1**

Errors found in Error Table 1 are detected in the data acquisition phase, while the consumer operator is acquiring data from the producer.

```sql
SELECT errorcode, errorfield, sourceseq, dmlseq From t2_e1;
```

*** Query completed. One row found. 4 columns returned.
*** Total elapsed time was 1 second.
Chapter 23: Post-Job Considerations
Accessing and Using Error Tables

<table>
<thead>
<tr>
<th>ErrorCode</th>
<th>ErrorField</th>
<th>SourceSeq</th>
<th>DMLSeq</th>
</tr>
</thead>
<tbody>
<tr>
<td>2679</td>
<td>A_IN_C1</td>
<td>49</td>
<td>1</td>
</tr>
</tbody>
</table>

The following explains the error table entry above:

<table>
<thead>
<tr>
<th>Error table Column</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorCode</td>
<td>2679</td>
<td>The Teradata Database error code associated with the error. In this case, message 2679 indicates: The format or data contains a bad character. <strong>Note:</strong> This error code also appears in the job logs.</td>
</tr>
<tr>
<td>ErrorField</td>
<td>A_IN_C1</td>
<td>Indicates where the error was generated. In this case: In column A_IN_C1, as defined in the INSERT INTO statement for the Stream operator.</td>
</tr>
<tr>
<td>SourceSeq</td>
<td>49</td>
<td>The sequence number of the data row that caused the error.</td>
</tr>
<tr>
<td>DMLSeq</td>
<td>1</td>
<td>The sequence number of the DML statement (within its DML Group) that caused the error.</td>
</tr>
</tbody>
</table>

**Example of Error Table 2**

Errors found in Error Table 2 are detected in the data application phase, by the consumer operator while it writes the data to the Teradata Database; in this case, Update operator.

```sql
select dbcerrorcode, sourceseq, dmlseq from t3_e2;
```

*** Query completed. 2 rows found. 3 columns returned. ***
*** Total elapsed time was 1 second. ***

<table>
<thead>
<tr>
<th>DBCErrorCode</th>
<th>SourceSeq</th>
<th>DMLSeq</th>
</tr>
</thead>
<tbody>
<tr>
<td>2793</td>
<td>50</td>
<td>2</td>
</tr>
<tr>
<td>2793</td>
<td>49</td>
<td>2</td>
</tr>
</tbody>
</table>

The following explains the error table entry above:

<table>
<thead>
<tr>
<th>Error table Column</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ErrorCode</td>
<td>2793</td>
<td>The Teradata Database error code associated with the error. In this case, message 2679 indicates: The format or data contains a bad character. <strong>Note:</strong> This error code also appears</td>
</tr>
<tr>
<td>SourceSeq</td>
<td>49, 50</td>
<td>The sequence numbers of the data rows that caused the error.</td>
</tr>
<tr>
<td>DMLSeq</td>
<td>2</td>
<td>The sequence number of the DML statement (within its DML Group) that caused the error.</td>
</tr>
</tbody>
</table>
Additional Information on Evaluating Error Tables

For additional information on using error tables to evaluate and correct operator errors, see the following sections in Chapter 24:

- “Load Operator Errors” on page 304
- “Stream Operator Errors” on page 307
- “Update Operator Errors” on page 311

Effects of Error Limits

When loading large amounts of data, it may be desirable to allow a small number of errors to occur without causing the job to terminate. You can set the number of allowable errors using the ErrorLimit attribute of the Load, Stream, and Update operators. The meaning of the error limit number differs depending on the operator and job situation. Note that error limits apply only to Error Table 1, that is, acquisition phase errors.

Error Limits For Load and Update Operators

The following example shows this variation for an operator with two instances and an ErrorLimit attribute value of 1000:

- If either operator instance reaches 1000, it will terminate the job with a fatal error. In this case, the error limit is calculated per instance.
- If instance #1 processes 500 error rows and instance #2 processes 500 error rows the job will do the following:
  - If the job has already passed the final checkpoint (the transaction is fully committed), the job will complete. In this case, the error limit is calculated per instance.
  - If the job reaches a checkpoint after logging the total of 1000 (500 + 500) errors, the job will terminate. In this case, the error limit is calculated based on the total errors among all instances.

Error Limits for Stream Operator

The Stream operator error limit determines the approximate number of rows that can be stored in the Error Table before the Stream operator job is terminated. This number is approximate because the Stream operator sends multiple rows of data at a time to Teradata. By the time Teradata PT processes the message indicating that the error limit has been exceeded, it may have loaded more rows into the error table than the actual number specified in the Error Limit.

The ErrorLimit specification is not cumulative, but applies to each instance of the Stream operator. Therefore a job with two instances of Stream operator and an ErrorLimit attribute value of 1000 will terminate only when one of the instances reaches 1000. Otherwise the job will continue.
Dropping Error Tables

Teradata PT automatically creates error tables for the Load, Stream, and Update operators each time a job script runs. In most cases, error tables are also automatically dropped.

Automatic Dropping of Error Tables

Teradata PT applies the following rules to error tables:

- Since Teradata PT automatically creates error tables each time a job runs, the error tables from the previous job run must be dropped before the next time the job runs.
- Teradata PT automatically drops error tables for successful job runs, that is, job runs with an exit code of 0 or 4. This includes jobs that succeed on the first run, as well as those that succeed after being repaired and rerun.
- Teradata PT does not automatically drop error tables for jobs that terminate with an exit code of 8 or 12 (to allow use of the error tables for debugging the job), or if the DropErrorTable attribute is set to No.

Strategy for Dropping Error Tables

Use the default Teradata PT behavior, that is, the automatic creation and dropping of error tables, except as follows:

- Set the DropErrorTable attribute to No, to retain the error tables if:
  - The job is new and you are not sure it will run correctly. Then you can use the retained error tables, even if the job completes with an exit code of 0 or 4, to evaluate how the job ran and whether or not you need to revise the job script to make it run better. Once the job runs successfully several times you can reset DropErrorTable to Yes.
  - The operator ErrorLimit attribute is set to a value greater than 0. This setting means that any errors that the job encounters will be loaded into the error table. Especially for jobs with high error limits, the job is not really complete until you can examine the errors and determine whether or not further action is required, so the error tables should be retained.
  - The job is a batch job run repeatedly at close intervals, such as “Job Example 8: Batch Directory Scan” on page 122. The job may run several times before you have time to evaluate the error tables, so they should be retained for evaluation.
    
    **Note:** Set the Stream operator AppendErrorTable attribute to allow successive runs of the job to write to the same error table.

- If error tables are retained and the Stream operator AppendErrorTable attribute is not in force, you must manually drop the error tables before the next run of the job, using a DROP TABLE statement.
- Some jobs, such as “Job Example 9: Active Directory Scan” on page 124, run continuously for the duration of the value of the VigilElapsedTime attribute, and will not drop error tables until the end of that elapsed time. This may result in the following problems, for which you must prepare:
Restart Log Tables

Teradata PT maintains a restart log table for the Load, Stream, and Update operators, to store checkpoint data for the job. The information in the restart log table is normally not accessed directly by Teradata PT users, but is automatically used by the Teradata PT infrastructure when a job is restarted. Once the job completes successfully, the restart log is automatically dropped.

The restart log for a particular operator is stored under the name specified in the LogTable attribute for the operator.

For required syntax and rules for specifying the name of the LogTable, see the chapter on the Load, Stream, or Update operator in Teradata Parallel Transporter Reference.

For information on restarting a job, see “Restarting A Job” on page 318.

Strategies for Evaluating a Successful Job

Even when a job runs successfully, the job logs may contain useful information that should be reviewed before the job runs again.

Evaluating Jobs with Exit Code=0

The job logs may contain the following important information, which is of value and may warrant further action.

Review the Metadata

Teradata PT provides two types of metadata.

- TWB_STATUS private log captures job performance metadata
- TWB_SRCTGT private log captures source and target metadata

TWB_STATUS

TWB_STATUS private log captures job performance data at different stages of the job. Teradata PT also provides a tbuild command option for specifying the interval (in seconds).
for collecting performance data. For details about all tbuild options, see *Teradata Parallel Transporter Reference*.

This information is useful for evaluating the performance of a job in terms of throughput and the cost of exporting and loading of data by each operator. It is also useful for capacity planning by collecting the performance data for a period of time, summarizing the CPU utilization and elapsed time for each job, and then determining the trend of performance for the overall loading and exporting processes for a specific system configuration.

**Action:**

Here are some tips for performance evaluations and tuning:

- Determine the difference in CPU utilization between the producer and consumer operators. For example, if the CPU utilization of the producer operator is 2 times greater than that of the consumer operator, increasing the number of producer instances by a factor of 2 might improve the throughput of the job.

- Determine the difference between the CPU utilization and the elapsed time for performing the exporting and loading of data (i.e. the EXECUTE method). If the elapsed time is much higher than the CPU time, this could mean that some bottlenecks might have occurred either on the network, I/O system, or the Teradata Database server.

- Find out how many rows were sent by the producer operator (or received by the consumer operator) with the above CPU utilization. Dividing the numbers of rows by the CPU seconds spent on processing these rows would give you the number of rows per CPU second.

- The difference between the "start time" of two successive methods would indicate how long the job spent on a method.

- Find out how much time being spent on each checkpoint. Note checkpoint takes time and resources to process. Tuning the number of checkpoints to be taken by changing the checkpoint interval is necessary.

**TWB_SRCTGT**

The source and target data shown in this log is for reference only, and requires no specific usage strategy.

**Review the Warnings**

Check for any minor warnings that may appear in the logs to see if further action is required, as shown in the following examples:

- The DDL operator may encounter database errors that the ErrorList attribute is set to ignore and will return a warning instead of an error, while allowing the job to continue executing.

  **Action:** Review the warnings and associated errors. Determine whether or not ignoring the error is achieving the results you expected. Reset the ErrorList attribute if required.

- The OS Command operator may not have been able to execute one or more of the commands requested of it.
**Action**: Review the error message output and correct the problems as you would any normal operating system error messages. If the OS Command operator IgnoreError attribute value was set to Yes, then any command errors would not have terminated the job. In these cases, look at the logs for any OS Command operator error messages and if any are present, determine whether or not later job steps were adversely affected by any commands that were not successfully executed.

**Allowed Errors**
When data is being written to the Teradata Database, consumer operators can be set to allow the job to proceed even if some data cannot be loaded, using the ErrorLimit attribute. This attribute applies only to the following operators:

- Load
- Stream
- Update

**Cause:**
There may be various reasons why the data did not load, but it is often due to violations of the schema or data type requirements when the data was originally entered into the source files.

For more information, refer to the sections on Load, Stream, and Update operator errors in Chapter 24: “Troubleshooting a Failed Job.”

**Corrective Action:**
- Examine the error tables for the operators to determine whether or not they contain any unprocessed data.
- Determine the reason the data did not load.
- Consider whether or not to correct the data errors in the source.
- In most cases, you will need to clean up the bad data and load it into Teradata Database with a separate job.
- Consider whether or not to reset the ErrorLimit attribute to a lower value.

### Evaluating Jobs with Exit Code=4
When the job exit code=4, all the observables for Exit Code=0 still apply. In addition, the job may have encountered one or more serious warnings of the following types:

- `tbuild`-based warnings
- `job` script-based warnings

#### Unnecessary `tbuild` Command Option
Teradata PT returns a warning message when the `tbuild -s` option, normally used to start a job from an intermediate step, directs the job to start at the first step (where the job would start without the `--s` option). The message is issued in case you had not meant to start at step one.

**Cause**: Unnecessary `tbuild` command option
Corrective Action:

- Do not use `tbuild -s` unless you intend to start a job from an intermediate step.
- If the job was supposed to start at an intermediate step, and starting at step one was an accident, examine the job logs to see if starting at step one caused any problems.

## Invalid Value for a `tbuild` Command Option

If the `tbuild -h` option specifies an invalid value for shared memory size, Teradata PT issues a warning identifying the size of shared memory it will actually use.

**Cause:** The `tbuild` command specified an invalid value for `-h`.

**Corrective Action:** For suggestions on how to correctly specify shared memory size, see *Teradata Tools and Utilities Installation Guide*.

## Truncated Data

If the values of source CHAR or VARCHAR columns could be, or will be, truncated when applied to the corresponding target columns, Teradata PT returns a warning message for each such column.

**Cause:** Possible mismatched source and target schema definitions in the job script.

**Corrective Action:** If the truncation is not acceptable, examine and adjust the source and target schemas to eliminate the mismatch that caused the truncation(s). Normally this requires ensuring that the schema for the target column (maximum) length is at least as large as the schema for the source column (maximum) length.

## Bad Source Data

The DataConnector operator may encounter bad data in the source file. If the RowErrFileName attribute specifies a file name, the bad data will be sent to the file and the job allowed to proceed.

**Cause:** Bad Source data.

**Corrective Action:** Clean up the data and enter it manually in target table to complete the job.

## Delete Task with More than One Row

This error depends on the following job scenario: The Update operator is set to delete data from a Teradata Database, with the DeleteTask attribute is set to a Y[es] value, but the deletion is triggered by a WHERE clause in the DELETE statement that is incomplete without some external data. The source of this data must be a single row on the source data stream.

In this case the Update operator is only looking for one row as the trigger. If it sees more than one row in the data stream, Teradata PT will issue a warning.

**Cause:** More than one row exists in the data stream.

**Corrective Action:** Check to see why the Producer operator is sending more than one row to the data stream, as it may result in a more serious problem in a later run of the job.
### Ignore Unsupported Large Decimal in Teradata Database or CLI

When you specify a valid value for the MaxDecimalDigits attribute and the IgnoreMaxDecimalDigits attribute is set to Yes, the job will proceed even if Teradata Database or CLI does not support the Large Decimal feature, but it will issue a warning that indicates the mismatch in decimal support.

**Note:** If the IgnoreMaxDecimalDigits is set to No, the job example above will abort with a fatal error.

The MaxDecimalDigits and IgnoreMaxDecimalDigits attributes apply only to the following operators:

- Export
- SQL Selector

**Cause:** The user requested to continue the job when the Teradata Database or CLI does not support the Large Decimal feature.

**Corrective Action:** None

### Paused Job

When the PauseAcq attribute for the Load or Update operator is set to a Yes value, the job is paused after the completion of the acquisition phase, that is, when all the data in the file has been read. This protocol is often used for scripts that empty a data file periodically, for instance, once a day. The warning is only a reminder that the job has paused. Re-launch the job again when the file contains more data.

**Cause:** User requested the job to pause after the completion of the acquisition phase.

**Corrective Action:** None.

### Unsupported Query Band in Teradata Database

When a value is specified for the QueryBandSessInfo attribute, the job proceeds even if Teradata Database does not support the Query Band feature.

The QueryBandSessInfo attribute applies only to the following operators:

- DDL
- Export
- Load
- SQL Inserter
- SQL Selector
- Stream
- Update

**Cause:** The version of the Teradata Database does not support the Query Band feature.

**Corrective Action:** If you want to use the Query Band feature, change the TdpId attribute value to a Teradata Database that supports the Query Band feature.
This chapter describes the procedure for troubleshooting a failed Teradata PT job.

Topics include:

- Detecting and Correcting the Cause of Failure
- Common Job Failures and Remedies
- Operator-Specific Error Handling
- Additional Debugging Strategies for Complex Job Failures
- Restarting A Job
- Removing Checkpoint Files

**Detecting and Correcting the Cause of Failure**

Use the following procedure to detect and correct the errors that caused a job to fail:

1. Access the logs and error tables for the job.
   
   For information on the content of the public and private logs and how to access them, see “Accessing and Using Job Logs” on page 277.
   
   For information on the content of error tables and how to access them, see “Accessing and Using Error Tables” on page 282.
   
2. Evaluate the logs and any errors they contain.
   
   - If the job fails before attempting the first job step, the associated errors and warnings will be in the public log. Evaluate the log entries, and take the needed corrective action.
     
     For a list of common errors for this type of failure, including causes and corrective actions, see “When the Job Fails to Begin Running” on page 294.
     
     If the job runs but fails to complete, errors will be found in the public and private logs.
     
     For a list of common errors for this type of failure, including causes and corrective actions, see “When the Job Fails to Complete” on page 301.
   
3. If operator errors are detected in the private log, evaluate the corresponding information in the error tables to provide more detailed information on the errors.
   
   For detailed information on using error tables, see “Accessing and Using Error Tables” on page 282.
   
4. Once the causes of errors have been corrected in the job script, re-launch the job.
   
   For information, see “Restarting A Job” on page 318.
Common Job Failures and Remedies

There are two categories of job failures. The evaluation and correction of each type of failure must be handled differently:

- Some jobs fail at launch, during execution of the tbuild statement, but before the initial job step has run.
- Some jobs launch successfully, and one or more job steps may execute successfully, but the job fails to run to completion.

The following sections describe common errors encountered by Teradata PT jobs.

When the Job Fails to Begin Running

When a job is launched but fails to begin execution, the associated errors appear in the public log. Errors are detected according to the launch sequence:

1. Teradata PT first processes the options specified in the tbuild command. If it detects tbuild command errors, the job stops.
   
   **Error types encountered:** tbuild command errors

2. If Teradata PT encounters no tbuild command errors, it then parses the job script and creates a parallel job execution plan that will perform the operations specified in the APPLY statement(s) in the job script.

   **Errors types encountered:**
   - Preprocessor errors -- Incorrect use of job variables or the INCLUDE directive.
   - Job script compilation errors -- Syntactic and semantic errors.

3. Only when script compilation is successful and the execution plan has been generated does the Teradata PT allocate resources for and launch the various internal tasks required to execute the job plan.

   **Errors types encountered:** System resource errors

The following common types of tbuild errors may occur at job launch:

- User errors
  - executing the `tbuild` command
  - script compiler errors
- System resource errors
  - semaphore errors
  - socket errors
  - shared memory errors
  - disk space errors
tbuild Command Errors

This type of user error occurs in the construction or execution of the `tbuild` command used to launch the job.

**Cause:** The user violated tbuild syntax rules or incorrectly entered the tbuild command statement. For instance, the tbuild command:

- did not specify a job file (-f)
- specified an invalid option identifier
- specified an invalid value for a valid option identifier

**Corrective Action:** Examine the tbuild statement for errors, correct the errors, and use the revised tbuild statement to re-launch the job.

Pre-processor Errors

Before the job script is parsed and compiled into an execution plan, Teradata PT does the following:

- If the script contains any INCLUDE directives, the script text from the file identified in each INCLUDE directive is imported into the job script text at the location of the INCLUDE directive. For information on INCLUDE, see “Reusing Definitions with the INCLUDE Directive” on page 380.
- If the job script contains references to any job variables, each such reference is replaced in the job script by the corresponding job variable value, taken from the job variable value source with the highest precedence. For information on job variables, see “Using Job Variables” on page 46.

**Error Case 1: INCLUDE Error**

**Error Message:** The file identified in the INCLUDE directive cannot be found.

**Cause:** The INCLUDE directive file reference is not the name of a file in your tbuild execution directory or the correct path of an existing file.

**Corrective Action:** Correct the file reference and resubmit the job.

**Error Case 2: Job Variable Error**

**Error Message:** Undefined job variable.

**Cause:** The sources for job variable values available to your job do not contain either the identified variable, or a value for the variable.

**Corrective Action:** Add a job variable assignment to at least one of the sources of job variable values, and resubmit the job.
Job Script Compilation Errors

Job script compilation errors are generally syntactic or semantic errors.

Syntactic errors include the following:

- Use of a keyword not known to Teradata PT.
- Absence of a keyword, identifier, or other script item, such as a punctuation mark expected in a particular part of the script.
- Out of order or missing DEFINE statement; or a DEFINE statement typographical error.

Error Case 3: Extra Comma

Extra comma errors include such things as erroneously coding a comma (,) after the last column definition in a DEFINE SCHEMA statement, or after the last attribute declaration in an ATTRIBUTES list.

The following script example:

```plaintext
DEFINE SCHEMA DAILY_SALES
(
    Store_Number     INTEGER,
    :             :
    Sales_Date          ANSI_DATE,
    Gross_Amount     DECIMAL(10,2),
);
```

results in the following console error:

```
line 37: syntax error at "}" missing { REGULAR_IDENTIFIER_ EXTENDED_IDENTIFIER EXTENDED_IDENTIFIER_NO_N_ } in Rule: Regular Identifier.
TPT_INFRA: TPT03022: Error: Syntax error occurred in parse rule Column Definition
Compilation failed due to errors. Execution Plan was not generated.
Job script compilation failed.
Job terminated with status 8.
```

**Note:** The reported line number in syntactic error messages is usually accurate, but occasionally the problem may actually appear on the previous line.

**Cause:** The extraneous comma after DECIMAL(10,2) is not recognized as being a syntax error until the parser encounters the closing '}' on line 37.

**Corrective Action:** Correct the error and resubmit the job.

Error Case 4: Omitted Semicolon

For the script example shown in Error Case 3: Extra Comma, suppose the DEFINES SCHEMA statement was missing the final semicolon (;). An error similar to the following would result:

```
line 38: syntax error at "DEFINE" missing SEMICOL_ in Rule: Job Definition Body
Compilation failed due to errors. Execution Plan was not generated.
Job script compilation failed.
Job terminated with status 8.
```

**Corrective Action:** Correct the error and resubmit the job.
Chapter 24: Troubleshooting a Failed Job

When the Job Fails to Begin Running

Error Case 5: Omitted Keyword
Omitting a required keyword, for example leaving out 'TO' in the APPLY ... TO OPERATOR portion of an APPLY statement, results in the following console message:

line 106: syntax error at "OPERATOR" missing TO_ in Rule: Restricted APPLY Statement
Compilation failed due to errors. Execution Plan was not generated.
Job script compilation failed.
Job terminated with status 8.

Corrective Action: Review the script line indicated in the error message against required syntax, then correct the error, and resubmit the job.

Error Case 6: Semantic Error
Semantic errors occur in script syntax that is correct, but not meaningful in some important way, including such common errors as:

- using the name of a CHAR column in an arithmetic expression
- using the name of a consumer operator where the Teradata PT script requires the name of a producer operator
- mismatch between the schemas for the producer and the consumer operator in a job step

For example, if a script references the name of a TYPE LOAD operator where the script requires the name of a producer operator, as shown in the following:

```
SELECT FROM OPERATOR( LOAD_OPERATOR [2] )
```

then the console would return the following error:

```
TPT_INFRA: TPT03168: Error: Semantic error at or near job script line 138:
Operator 'LOAD_OPERATOR' is not of type 'Producer'.
Operator is rejected as data source for SELECT operation.
Compilation failed due to errors. Execution Plan was not generated.
Job script compilation failed.
Job terminated with status 8.
```

Corrective Action: Modify the job script to correct the semantic error and resubmit the job.

System Resource Errors

The following section presents common system resource errors.

Error Case 7: Insufficient Semaphores
Teradata PT console message:

```
Teradata Parallel Transporter Version <version>
Execution Plan generation started
Execution Plan generation successfully completed
Job log: /opt/Teradata/Client/<version>/logs/udd014-18.out
OS_SemInit: semget() failed, System errno: 28 (No space left on device)
1008: Failed to Initialize necessary IPC resources to run this job
1155: Infrastructure for the Parallel Task Manager failed
1006: Failed to set up Parallel Task Manager infrastructure to run job
```

Teradata Parallel Transporter User Guide
Chapter 24: Troubleshooting a Failed Job
When the Job Fails to Begin Running

Cause:

Error 28, ENOSPC, on segment() indicates that the system limit on the maximum number of semaphores would be exceeded if the semget() request was honored.

Corrective Actions:

1. Use the `ipcs` command to check the computer from which the Teradata PT job was launched to see if there are semaphores that have been orphaned.
2. Use the `ipcm` command to free up any unused semaphores that may be available.
3. Use the `sysdef` command to find out the number of semaphores (SEMMNS) defined on the system. Increase the number and then reboot the system.
4. Re-launch the job.

Error Case 8: Insufficient Semaphore Undo Structures

Teradata PT console message:

Teradata Parallel Transporter Version <version>
Execution Plan generation started.
Execution Plan generation successfully completed.
Job log: /opt/Teradata/Client/<version>/logs/root-2.out
OS_SemOp: semop() failed. System errno: 28 (No space left on device)
OS_AllocSem: OS_SemOp failed

Cause:

Errno 28, ENOSPC, on a semop() means that the system has run out of undo structures for semaphores.

Corrective Action:

1. Use the `sysdef` command or something similar to find out the value of the semaphore undo structures (SEMMNU) defined on the system.
2. Increase the value of SEMMNU.
3. Reboot the system.
4. Re-launch the job.

Error Case 9: Socket Handle Error

Teradata PT console message:

Teradata Parallel Transporter Version <version>
Execution Plan generation started.
Execution Plan generation successfully completed.
Job id is load_dpforecast-1, running on WUSSL185013-V02
Job log: C:\Program Files\Teradata\Client\<version>\Teradata Parallel Transporter\logs\load_dpforecast-1.out
1405: Error occured while polling for any ready socket, System errno: 10038
   (An operation was attempted on something that is not a socket.)
PX_Node::Bind() [Node WUSSL185013-V02] - Failed with status 15
1113: Failed to read 8 bytes from socket 3872, System errno: 10054
   (An existing connection was forcibly closed by the remote host.)
1141: Failed to receive config response from the Job Logger
Chapter 24: Troubleshooting a Failed Job

When the Job Fails to Begin Running

Teradata Parallel Transporter User Guide

WUSSL185013-V02 - PTM status 15: the Job Logger facility could not be set up

Cause:

On some Windows XP machines the socket handle is not inherited correctly by Teradata PT, preventing the setup of job logging. This problem hasn't been found on UNIX or z/OS platforms.

Corrective Actions: Teradata PT Efix available.

Error Case 10: Insufficient Allocation of Shared Memory

Teradata PT console message:

Teradata Parallel Transporter Version <version>
Execution Plan generation started.
Execution Plan generation successfully completed.
Job log: /opt/Teradata/Client/<version>/tbuild/logs/root-2.out
OS_ShmInit: shmget(1048576) failed, System errno: 22 (Invalid argument)
1008: Failed to Initialize necessary IPC resources to run this job
1155: Infrastructure setup for the Parallel Task Manager failed
1006: Failed to setup Parallel Task Manager Infrastructure to run this job

Cause:

Teradata PT requested one meg (1024*1024) of shared memory (the minimum). The OS returned EINVAL, meaning that the requested size is less than SHMMIN, greater than SHMMAX, or greater than the size of any available segment.

Corrective Actions:

1. Use the sysdef command or something similar to find out the values of the shared memory parameters, SHMMIN, SHMMAX, and SHMSEG, defined on the system.
2. Use the sysdef command or something similar to find out the values of the shared memory parameters, SHMMIN, SHMMAX, and SHMSEG, defined on the system.
3. Decrease the value of SHMMIN, or increase the values for SHMMAX and SHMSEG, as required to provide adequate shared memory.
4. Reboot the system.
5. Re-launch the job.

Error Case 11: Shared Memory Overflow Due to Excessive Operator Instances

Teradata PT console message:

Teradata Parallel Transporter Version <version>
Execution Plan generation started.
Execution Plan generation successfully completed.
Job log: /opt/Teradata/Client/<version>/tbuild/logs/infomatc-66241.out
Job id is load_files-66241, running on system02-ib
Teradata Parallel Transporter DataConnector Version 08.02.00.01
Teradata Parallel Transporter Stream Operator Version 08.02.00.00
READ_DATA: Operator instance 1 processing file 'File00006'.
READ_DATA: Operator instance 1 processing file 'File00001'.
READ_DATA: Operator instance 1 processing file 'File00003'.
Chapter 24: Troubleshooting a Failed Job
When the Job Fails to Begin Running

READ_DATA: Operator instance 1 processing file 'File00005'.
READ_DATA: Operator instance 1 processing file 'File00016'.
READ_DATA: Operator instance 1 processing file 'File00015'.
READ_DATA: Operator instance 1 processing file 'File00007'.
READ_DATA: Operator instance 1 processing file 'File00008'.
READ_DATA: Operator instance 1 processing file 'File00013'.
READ_DATA: Operator instance 1 processing file 'File00009'.
READ_DATA: Operator instance 1 processing file 'File00014'.
READ_DATA: Operator instance 1 processing file 'File00012'.
READ_DATA: Operator instance 1 processing file 'File0002'.
READ_DATA: Operator instance 1 processing file 'File00010'.
READ_DATA: Operator instance 1 processing file 'File00004'.
STREAM_OPERATOR: connecting sessions
FXTB_AllocateMessage: Cannot create data buffer, Data Stream status = 3
1104: Insufficient main storage for attempted allocation

Cause:

Data moves from the producer operator instances to the consumer operator instances in *data streams*. Teradata PT allows allocation of up to 10MB of shared memory for use in servicing data streams, which imposes a limit of approximately 75 data streams for a job. When this limit is exceeded, the job can no longer allocate more buffers in the Data Stream, which causes the job to terminate.

For more detailed information about the relationship between instance usage and shared memory, see “Calculating Shared Memory Usage Based on Instances” on page 82.

Corrective Action:

1. Do one of the following
   - Decrease the number of consumer or producer instances.
   - Use the `tbuild` “-h” option to increase the shared memory size for the job. For details see the following section on “Allocating Shared Memory” on page 300.

2. Relaunch the job.

For required syntax and a description of `tbuild -h`, see the section on `tbuild` in Teradata Parallel Transporter Reference.

Allocating Shared Memory

By default, Teradata PT provides 10MB of shared memory for the execution of a job script. The `tbuild` `-h` option allows you to adjust the shared memory to more accurately reflect the needs of the job, as follows:

- Use `-h value` to specify a *value* in bytes ranging from 1,048,576 (that is, 1 MB) to 134,217,728 (that is, 128 MB).
- Use `-h valueK` to specify a *value* in kilobytes ranging from 1024 K (that is, 1,048,576 bytes) to 131,072 K (that is, 134,217,728 bytes).
- Use `-h valueM` to specify a *value* in megabytes ranging from 1 MB (that is, 1,048,576 bytes) to 128 MB (that is, 134,217,728 bytes).
For information on how to calculate shared memory usage, see “Calculating Shared Memory Usage Based on Instances” on page 82.

**Error Case 12: Log File is Full**

Teradata PT console message:

Teradata Parallel Transporter Version <version>
Execution Plan generation started.
Execution Plan generation successfully completed.
Job log: /opt/Teradata/Client/<version>/tbuild/logs/root-2.out
1403: Unable to Write data to the file, System errno: 113
(EDC5113I Bad file descriptor CEE5213S The signal SIGPIPE was received.)

**Cause:**

The 113 error occurs because the log file is full. The job directory has run out of disk space.

**Corrective Action:**

Delete unused log files from the directory.

---

**When the Job Fails to Complete**

When a job launches but fails to complete, the following type of errors appear in the private log.

- Initialization Errors
  - Invalid attribute specification
  - Invalid attribute value
  - Schema mismatch
- Data acquisition errors
  - Extra column does not match schema
- Data application errors
- SQL Errors

**Initialization Errors**

Initialization errors occur when the Teradata PT infrastructure processes the schemas and operator definitions prior to executing the APPLY statement in a job step.

**Error Case 13: Mismatched Schema**

**Cause:**

The schema in the DEFINE SCHEMA statement does not match the schema defined in the SQL statement in the APPLY statement.
Corrective Action:

1. Compare the schema definition, the schema called by each operator, and the schema of the data source/target schema to determine the cause of the mismatch.
2. Correct the schema definition and/or the schemas specified by the operators, as required to correct the problem.
3. Re-launch the job.

Error Case 14: Invalid Attribute Specification

One or more attributes in a DEFINE OPERATOR statement are specified incorrectly or have invalid values.

Cause:
Scripting error.

Corrective Action:
Review the job script operator definitions and check attribute specifications against the related chapter in Teradata Parallel Transporter Reference.

Data Acquisition Errors

Data acquisition errors occur while the consumer is receiving data from the producer. They include the following common error types:

- Unexpected extra column
- Delimited data error
- Data type error
- Data size error

Error Case 15: Unexpected Extra Column

Cause:
The schema in the DEFINE SCHEMA statement does not match the schema of the data source/target.

Corrective Action:

1. Compare the schema definition, the schema called by each operator, and the schema of the data source/target schema to determine the cause of the mismatch.
2. Correct the schema definition and/or the schemas specified by the operators, as required to correct the problem.
3. Re-launch the job.

Error Case 16: Delimited Data Errors

When using the DataConnector operator to extract delimited data, errors may occur if the escape character is not defined. Since there is no default escape character, use the DataConnector operator EscapeTextDelimiter optional attribute to define the escape character. If not provided, the TextDelimiter attribute defaults to the pipe character ( | ).
Data Application Errors

Data application errors occur while the consumer operator is writing data to the Teradata Database, and include the following common error types.

Cause:
- Row to be INSERTed duplicates an existing row in the target table.
- Row to be UPDATED or DELETED does not exist in the target table.
- A DML statement, intended to UPDATE or DELETE one specific row in the target table, actually applies to more than one row in the target table.

When the consumer operator is Load, Update or Stream, source rows causing data application errors are written to standard error tables.

Corrective Action:
For information on assessing row errors, see “Accessing and Using Error Tables” on page 282.

SQL Errors

SQL errors occur as a result of the job script executing SQL statements in the database. SQL errors are returned by Teradata Database. SQL errors show up in the job logs, but not in the error tables. For detailed information on SQL errors, see Messages.

Corrective Action: Review the error and correct the SQL statement

Operator-Specific Error Handling

Teradata PT handles errors differently depending on the operator that detects the error and whether or not the operator has been directed, through one of its attributes, to ignore the error just detected.

The Load, Update and Stream operators, which typically process large numbers of data rows, have built-in tolerances for data application errors, which are specifiable through operator attributes. For detailed information, see the sections on these operators later in this chapter.

Other operators generally terminate for any error that occurs during their execution. The exceptions are:
- DDL operator: Use the ErrorList attribute to specify one or more Teradata Database error codes that the operator will ignore, instead of causing the job to terminate, as would normally be the case.
- DataConnector operator:
  - Use the RowErrFileName attribute to write erroneous source rows to a named file instead of terminating.
  - Use the AcceptExcessColumns attribute to ignore extra columns in its source rows instead of terminating.
Load Operator Errors

The Teradata Database tracks and records information about various types of error conditions that cause an input data record to be rejected during a load operation. The following error conditions can occur:

Table 15: Load Errors

<table>
<thead>
<tr>
<th>Error Condition</th>
<th>Cause of Rejection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint violation</td>
<td>Records do not comply with the range constraints you defined when creating the table.</td>
</tr>
<tr>
<td>Unavailable AMP condition</td>
<td>Records are destined for a non-fallback table on an AMP that is down.</td>
</tr>
<tr>
<td>Data conversion errors</td>
<td>Refers to records from the input file that fail a specific data type conversion.</td>
</tr>
<tr>
<td>Unique primary index violation</td>
<td>Records contain a value for the unique primary index field that already exists, but is not a duplicate row.</td>
</tr>
<tr>
<td>Duplicate row</td>
<td>Records are exact duplicates of existing rows.</td>
</tr>
</tbody>
</table>

Error Recording

The Load operator automatically creates two error tables that capture errors during job execution, ErrorTable1 and ErrorTable2, which separate information as follows:

- **Error Table 1**: The Acquisition Error Table. Contains most of the errors relating to data and the data environment. The following types of errors are captured:
  - Constraint violations - Records that violate a range or value constraint defined for specific columns of a table.
  - Unavailable AMP - Records to be written to a non-fallback table about an offline AMP.
  - Data conversion errors - Records that fail to convert to a specified data type.

- **Error Table 2**: The Application Error Table contains all of the rows that have violations of the unique primary index. This error table is not used when the target table has a non-unique primary index.

Jobs can use the default names of the error tables, or can specify an alternate table names using the ErrorTable1 and ErrorTable2 attributes in the operator definition.

The Teradata Database discards all records that produce a duplicate row error, but reports the total number of duplicate rows encountered and the total records in each error table, in the end-of-operation status report.

Error Table Format

The Load operator error tables have specific formats:

- The acquisition error table contains the following columns:
Correcting Load Errors

Though the procedures are somewhat different depending on the error table in question, use the following procedure to correct load errors:

1  Retrieve the error information from the error tables on the Teradata Database.
2  Evaluate and correct the errors.
3  Insert the corrected records into the Load TargetTable.

Because the Load operator accesses only an empty table, after the job is complete you must use a utility, such as BTEQ, to access the Teradata Database. The following procedures and examples assume that BTEQ is running and that you are logged on to the Teradata Database.

For more information about using BTEQ, see Basic Teradata Query Reference.

Acquisition Error Table

Use the following procedure to correct errors recorded in the acquisition error table, which is defined by the ErrorTable1 attribute:

1  Use the following Teradata SQL statement to retrieve the error code and field name for each error in the first error table, where etname1 is the name you specified for the ErrorTable1 error table:

   SELECT ErrorCode, ErrorFieldName FROM etname1 ORDER BY ErrorCode ;

   Note: If the operator definition does not specify a name for the ErrorTable1 attribute, the error table will be named <TargetTableName>_ET by default. For details, see the chapter on Load operator in Teradata Parallel Transporter Reference.

   The BTEQ response is a list of the error codes and the associated field names, formatted as follows:

   ***Query completed. 2 rows found. 2 columns returned.
   ***Total elapsed time was 1 second.
   ErrorCode  ErrorFieldName
   --------    ------------------
   2679        A
   2679        A

• The application error table is formatted to match the target table.
Chapter 24: Troubleshooting a Failed Job
Load Operator Errors

- The values listed in the ErrorCode column are the Teradata Database return codes for each error condition, as specified in the messages reference documentation for your operating system environment.

- The values listed in the ErrorFieldName column are the names of the fields that caused each error.

2 Use the following BTEQ commands and Teradata SQL statements to retrieve the data records for each error in the first error table and store them in the specified err.out file on your client system:

   If the values in the ErrorCode column indicate that a constraint violation error occurred, retrieve the DataParcel information in record mode:
   
   ```
   .SET RECORDMODE ON
   .EXPORT DATA FILE=err.out
   SELECT DataParcel FROM etname1
   ```

   Otherwise, if the values in the ErrorCode column indicate that the errors were all caused by unavailable AMP conditions, do not use the RECORDMODE command:

   ```
   .EXPORT DATA FILE=err.out
   SELECT DataParcel FROM etname1
   ```

3 Use the ErrorCode and ErrorFieldName information returned in step 1 and the DataParcel information returned in step 2 to determine which records you want to correct and reload to the Teradata Database.

   The methods that you can use to correct the individual error conditions will vary depending on the number and types of errors encountered.

4 After correcting the errors, use the following BTEQ commands and Teradata SQL statements to insert the corrected records into the Load table on the Teradata Database:

   - BTEQ IMPORT command to transfer the data to the Teradata Database
   - Teradata SQL USING modifier to define the fields in each record
   - Teradata SQL INSERT statement to insert a record into the Load table

   **Caution:** Do not reference the first two bytes in the INSERT statement for data records exported from the Teradata Database in record mode. Instead, make the first field (variable parameter) in the USING modifier a dummy SMALLINT field. When selecting data in record mode, the variable-length columns are all preceded by a two-byte field whose value indicates the length of the data field. But, because the DataParcel column of the ErrorTable1 table is defined as a variable-length field, the first two bytes always indicate the length. If you do not reference this field in the INSERT statement, the Teradata Database ignores this portion of each record in the input data.

5 Repeat steps 2 through 4 as required to resolve all of the ErrorTable1 error conditions.

6 After you resolve all errors, drop the ErrorTable1 table from the Teradata Database. 
Application Error Table

Use the following procedure to correct errors recorded in the application error table, which is defined by the ErrorTable2 attribute:

1. Use the following Teradata SQL statement to retrieve all rows from the second error table, where `ttname_UV` is the name of the second error table and `cname` is the unique primary index for the table:
   
   ```sql
   SELECT * FROM ttname_UV ORDER BY cname;
   
   Note: Use `ttname_UV` for the default name of ErrorTable2. If the operator definition specifies a name for the ErrorTable2 attribute, the SELECT statement shown above must contain use the name specified.
   
   The BTEQ response is a list of the contents of the second error table, ordered by the values in the primary index column.
   
2. Use the following Teradata SQL statement to retrieve each row from the Load TargetTable that has a primary index value identical to a row retrieved from the second error table, where `ttname` is the name of the Load TargetTable, `cname` is the index of the Load TargetTable, and `errorvalue` is the index value retrieved from the second error table:
   
   ```sql
   SELECT * FROM ttname WHERE cname = errorvalue
   
   3. Compare the rows selected from the ErrorTable2 with the rows selected from the TargetTable and determine which is correct:
      
      • If ErrorTable2 is correct, use one of the following:
         • DELETE statement to delete the incorrect row from the TargetTable.
         • INSERT statement to insert the correct row.
      
      • If TargetTable is correct, use the DELETE statement to delete the corresponding row from the ErrorTable2 table.

4. Repeat steps 2 and 3 until all rows in the ErrorTable2 table are accounted for.

5. Using BTEQ, drop the ErrorTable2 table from the Teradata Database after you resolve all of the errors, where `etname2` is the name of the second error table:
   
   ```sql
   DROP TABLE etname2
   ```

Stream Operator Errors

The Stream operator uses a single error table that contains records rejected because of data conversion, constraint, or other errors.

Error Capture

The APPLY statement that invokes the Stream operator provides DML error options that tell the Stream operator what to do with errors. These options allow you to mark or ignore error conditions, such as duplicate rows and missing rows. Marked error conditions are directed to the error table.
Chapter 24: Troubleshooting a Failed Job
Stream Operator Errors

These MARK/IGNORE options are:

- **DUPLICATE ROWS** (for both insert and update operations)
- **DUPLICATE INSERT ROWS** (for insert operations)
- **DUPLICATE UPDATE ROWS** (for update operations)
- **MISSING ROWS** (both update and delete operations)
- **MISSING UPDATE ROWS** (for update operations)
- **MISSING DELETE ROWS** (for delete operations)
- **EXTRA ROWS** (for both update and delete operations)
- **EXTRA UPDATE ROWS** (for update operations)
- **EXTRA DELETE ROWS** (for delete operations)

These options take effect when they are entered immediately following INSERT, UPDATE, or DELETE statements in the APPLY statement.

**Note:** If neither option is specified in the APPLY statement, MARK is the default condition.

For more information, see the MARK/IGNORE options in the section on APPLY in *Teradata Parallel Transporter Reference*.

### Error Table

Each error table row can include up to ten columns of information that you can use to help determine the cause of the error. You can specify any or all of these columns, in any order, in the SELECT statement used to access the error table.

The following table lists the Stream error table columns that can be specified.

**Table 17: Error Table Columns**

<table>
<thead>
<tr>
<th>Column</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSeq</td>
<td>Sequence number assigned to the input source in which the error occurred.</td>
</tr>
<tr>
<td>DMLSeq</td>
<td>Sequence number assigned to the DML group in which the error occurred.</td>
</tr>
<tr>
<td>ErrorCode</td>
<td>Code for the error.</td>
</tr>
<tr>
<td>ErrorField</td>
<td>This field is zero for the Stream operator.</td>
</tr>
<tr>
<td>ErrorMsg</td>
<td>The Teradata Database error message for the error.</td>
</tr>
<tr>
<td>HostData</td>
<td>Client data being processed when the error occurred.</td>
</tr>
<tr>
<td>LoadStartTime</td>
<td>Queue Insertion TimeStamp (QITS) value indicates when the job started. On restart, it indicates when the job restarted.</td>
</tr>
<tr>
<td>RowInsertTime</td>
<td>Indicates when the row was inserted into the Stream operator error table.</td>
</tr>
<tr>
<td>SourceSeq</td>
<td>Sequence number assigned to the row from the input source (the DataSeq number) in which the error occurred.</td>
</tr>
<tr>
<td>STMTSeq</td>
<td>Sequence number of the DML statement within the DML group (as indicated by the previous column DMLSeq) being executed when the error occurred.</td>
</tr>
</tbody>
</table>
Reusing Error 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 operator job. To reuse the names specified for the error tables, use the DROP TABLE statement via the BTEQ utility or the DDL operator to remove the Stream operator error tables from the Teradata Database.

Allowable Errors

The Stream operator definition can employ the ErrorLimit attribute to specify the approximate number of records that can be stored in the error table before the Stream operator job is terminated. This number is approximate because the Stream operator sends multiple rows of data simultaneously to the Teradata Database. By the time Teradata PT processes the message indicating that the error limit has been exceeded, it may have loaded more records into the error table than the number specified in the error limit.

When the Stream operator encounters a data row that cannot be processed properly, it creates a row in the error table. Such errors are added to the error table until it reaches the limit. Specify these options in the APPLY statement, immediately following the DML statements to which they apply, to control error handling for those DML statements by the Stream operator.

Note: The application of the error limit may apply either per operator instance or per operator depending on the stage of the load task when the limit is reached. For details, see “Effects of Error Limits” on page 286.

Strategy

Consider the following when setting the ErrorLimit value:

- The ErrorLimit is valuable because it will allow the job to continue when errors are encountered, instead of allowing the errors to terminate the job. However, you must manually clean up the accumulated errors after the job has completed, so do not let more errors accumulate than you have time to process.
- The errors encountered are mostly the result of bad data. If you need to keep the error limit set very high, it may be useful to look for ways to improve the data.
- The value should be set empirically, based on how the job runs. The actual setting must be based on the amount of data that will be processed by the job.

For a detailed description and required syntax, see the chapter on Stream operator in Teradata Parallel Transporter Reference.

Correcting Stream Errors

Following is an abbreviated Stream operator task, an error table listing, and a procedure for determining the cause of the error. The task example includes only one DML group consisting of two DML statements, an INSERT statement, and an UPDATE statement, for a complete Stream operator job.

The example procedure, below, uses all of the error information from the error table. In most cases, you can determine the cause by evaluating only one or two columns of the error table entry. The example uses the following APPLY statement to create error tables:
Task Example

In the following task example, the Sequence Type and Number columns are the type and number assignments for each statement. The Statement column shows the actual Stream operator job statements.

Table 18: Task Example

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Type</th>
<th>Number</th>
<th>Statement</th>
</tr>
</thead>
</table>
|          | DML  | 001    | 'INSERT INTO table1 VALUES (:FIELD1,:FIELD2 );  
            UPDATE table2 SET field3 = :FIELD3 WHERE field4 = :FIELD4;' |
|          | STMT | 001    | INSERT INTO table1 VALUES (:FIELD1, :FIELD2 ); |
|          | STMT | 002    | UPDATE table2 SET field3 = :FIELD3 WHERE field4 = :FIELD4; |

The following shows an error in the first error table created by the above task.

<table>
<thead>
<tr>
<th>DataSeq</th>
<th>DMLSeq</th>
<th>SMTSeq</th>
<th>SourceSeq</th>
<th>ErrorCode</th>
<th>ErrorField</th>
</tr>
</thead>
<tbody>
<tr>
<td>002</td>
<td>001</td>
<td>002</td>
<td>20456</td>
<td>2679</td>
<td>000</td>
</tr>
</tbody>
</table>

Use the following procedure to evaluate the error information and isolate the problem:

1. Check the DMLSeq field to find the DML group. It contains the sequence number 001.
2. Check the STMTSeq field. The sequence number 002 in this field means that the error occurred while executing the second DML statement, which is the UPDATE statement in the example task.
3. Verify that the Stream operator job script uses two DML statements in the first DML group (because DMLSeq was 001).
4. Check the DataSeq field. The value of 002 indicates that the error occurred while processing a row from the second input data source. (The input data source sequence is determined by Teradata PT.)
5. Check the meaning of the ErrorCode field. Error 2679, “The format or data contains a bad character,” indicates a problem with the data from your client system.
6. Because the script shows that the UPDATE statement was loading table2, you now know:
   - What error occurred
   - Which statement detected the error
   - Which input data source has the error
Chapter 24: Troubleshooting a Failed Job

Update Operator Errors

7 Check the SourceSeq field. The value of 20456 indicates that the problem is with the 20,456th record of the input data source.
8 Fix the problem.

Using the Error Table as a Queue Table

Setting the QueueErrorTable attribute to Yes causes the Stream Operator to create the error table as a queue table. If the error table contains one or more rows, use a single SELECT AND CONSUME request on the error table to retrieve and delete a row from the error table.

The benefit of using a SELECT AND CONSUME request is that it returns a row from the error table for you to fix, then deletes the row in a single operation, eliminating the need to send a separate request for deletions. For example, in an error table that contains five rows, you can issue the following request five times to retrieve and delete all five rows in the error table:

"SELECT AND CONSUME TOP 1 * FROM <error table name>;
"

To submit a SELECT AND CONSUME request, either use BTEQ to submit request directly to the Teradata Database, or use a software application that submits SQL requests to the Teradata Database. For more information, see the Teradata SQL SELECT statement in SQL Data Manipulation Language.

Changing the QueueErrorTable Value on Restart

The Teradata Database does not allow a queue table to change into a non-queue table, or vice-versa. Therefore, if you change the value for the QueueErrorTable attribute on a restart, the Stream operator ignores the value and continues with the job. For example, even if the value of the QueueErrorTable attribute is changed from No to Yes at restart, the operator still ignores the value and continues with the job.

Update Operator Errors

When the Update operator encounters a data row that cannot be processed properly, it creates a row in one of the two error tables that are created for each target table in the Update operator job:

- Acquisition Error Table
- Application Error Table

These error tables are similar to those used for the Load operator, but the Update error tables are typically named with the following suffixes to distinguish them.

- ErrorTable1 uses the suffix ET
- ErrorTable2 uses the suffix UV

Consider the following facts about error tables:

- If a job generates no errors, the error tables will be empty. They are automatically dropped at the end of the job.
If errors are generated, the tables are retained at the end of the job so error conditions can be analyzed.

To rerun a job from the beginning, either delete the error tables, or rename them, otherwise an error message results, stating that error tables already exist.

Conversely, if you restart a job (not from the beginning), an error tables must already exist. In other words, do not delete error tables to restart an update job.

Names for error tables can be defaulted or they can be explicitly named using the VARCHAR ErrorTable attribute.

Errors are separated into two tables, as follows:

- **Error Table (ET)** contains most of the errors relating to data and the data environment.
  
The following types of errors are captured:
  
  - Constraint violations records that violate a range constraint defined for the table.
  - Unavailable AMP records that are written to a non-fallback table on an offline AMP.
  - Data conversion errors records that fail to convert to a specified data type.

  By default, this error table is assigned a name using the convention:

  Target_Tablename_ET

- **Uniqueness Violations (UV)** contains all of the rows that have violations of a unique primary index.

  By default, this error table is assigned a name using the following convention:

  Target_Tablename_UV

Each error table generates eight columns of information that you can use to help determine the cause of the problem. You can specify that the error tables return any or all of these columns, in any order, using an SQL SELECT statement in a BTEQ job.

For details on accessing error tables, see “Accessing and Using Error Tables” on page 282.

In addition, the acquisition error table includes the faulty record, and the application error table includes a mirror image of the target table columns.

**Note:** Because the application error table includes a mirror image of the target table, preceded by the error information, the target tables for the Update operator job cannot contain column names that are the same as the error table columns, or the Update job terminates and returns a 3861 error message.

For the names of the error table columns, see “Acquisition Error Table” on page 313 and “Application Error Table” on page 314.

**Error Capture**

The APPLY statement that invokes the Stream operator provides DML error option attributes that tell the Update operator what to do with errors. These options allow you to mark or ignore error conditions, such as duplicate rows and missing rows. Marked error conditions are directed to the error table.

These MARK/IGNORE options are:
• DUPLICATE ROWS (for both insert and update operations)
• DUPLICATE INSERT ROWS (for insert operations)
• DUPLICATE UPDATE ROWS (for update operations)
• MISSING ROWS (both update and delete operations)
• MISSING UPDATE ROWS (for update operations)
• MISSING DELETE ROWS (for delete operations)

Specify these options in the APPLY statement, immediately following the DML statements to which they apply, to control error handling for those DML statements by the Update operator.

Note: If neither option is specified in the APPLY statement, MARK is the default condition.

For more information, see APPLY in Teradata Parallel Transporter Reference.

Acquisition Error Table

The first error table, called the acquisition error table, is specified with the ErrorTable1 attribute. It provides information about:

• All errors that occur during the acquisition phase of the Update operator job.
• Some errors that occur during the application phase if the Teradata Database cannot build a valid primary index.

The following table lists, in alphabetical order, the acquisition error table columns that can be specified.

Table 19: Acquisition Error Table Format for the Update Operator

<table>
<thead>
<tr>
<th>Column</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplySeq</td>
<td>Sequence number assigned to the DML group in which the error occurred (the same as DMLSeq). It can be ignored in error handling.</td>
</tr>
<tr>
<td>DMLSeq</td>
<td>Sequence number assigned to the DML statement within the DML group in which the error occurred.</td>
</tr>
<tr>
<td>ErrorCode</td>
<td>Code for the error.</td>
</tr>
<tr>
<td>ErrorField</td>
<td>Field name of the target table in which the error occurred. Note: This field may be blank if the system cannot determine which field caused the problem. Error 2677 (stack overflow) is an example of such a condition.</td>
</tr>
<tr>
<td>HostData</td>
<td>Client data being processed when the error occurred.</td>
</tr>
<tr>
<td>ImportSeq</td>
<td>Sequence number assigned to the input source in which the error occurred.</td>
</tr>
<tr>
<td>SourceSeq</td>
<td>Sequence number assigned to the row from the input source (the ImportSeq number) in which the error occurred.</td>
</tr>
<tr>
<td>STMTSeq</td>
<td>Sequence number of the DML statement within the DML group (as indicated by the previous column DMLSeq) being executed when the error occurred.</td>
</tr>
</tbody>
</table>
**Application Error Table**

The second error table, called the application error table, is the one specified from the ErrorTable2 attribute. It provides information about:

- Uniqueness violations
- Field overflow on columns other than primary index fields
- Constraint errors

The following table lists, in alphabetical order, the application error table columns that can be specified.

**Note:** A copy (or mirror) of the target table columns follows the DBCErrorField column in the application error table.

<table>
<thead>
<tr>
<th>Column</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplySeq</td>
<td>Sequence number assigned to the DML group in which the error occurred (the same as DMLSeq). It can be ignored in error handling.</td>
</tr>
<tr>
<td>DBCErrorCode</td>
<td>Code for the error.</td>
</tr>
<tr>
<td>DBCErrorField</td>
<td>Field name of the target table in which the error occurred.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This field may be blank if the system cannot determine which field caused the problem. Error 2677 (stack overflow) is an example of such a condition.</td>
</tr>
<tr>
<td>DMLSeq</td>
<td>Sequence number assigned to the DML statement within the DML group in which the error occurred.</td>
</tr>
<tr>
<td>ImportSeq</td>
<td>Sequence number assigned to the input source in which the error occurred.</td>
</tr>
<tr>
<td>SourceSeq</td>
<td>Sequence number assigned to the row from the input source (the ImportSeq number) in which the error occurred.</td>
</tr>
<tr>
<td>STMTSeq</td>
<td>Sequence number of the DML statement within the DML group (as indicated by the previous column DMLSeq) that is executed when the error occurred.</td>
</tr>
<tr>
<td>Uniqueness</td>
<td>Value that prevents duplicate row errors in the error table. It can be ignored in error handling.</td>
</tr>
</tbody>
</table>

**Correcting Update Errors**

Following is an abbreviated Update operator task, an error table listing, and a procedure for determining the cause of the error. This task example includes only one DML group consisting of two DML statements, an INSERT statement, and an UPDATE statement for a complete Update operator job.

The example uses all of the error information from the error table. In most cases, you can determine the cause by evaluating only one or two columns of the error table entry.

This example uses the following APPLY statement to create the error tables in this section:

```
APPLY
```
'INSERT INTO table1 VALUES (:FIELD1,:FIELD2);
UPDATE table2 SET field3 = :FIELD3 WHERE field4 = :FIELD4;'

**Task Example**

In the following example, the Sequence Type and Number columns are the type and number assignments for each statement. The Statement column shows the actual job statements.

Table 21: Task Example

<table>
<thead>
<tr>
<th>Sequence Type</th>
<th>Type</th>
<th>Number</th>
<th>Statement</th>
</tr>
</thead>
</table>
| DML           | 001  |        | 'INSERT INTO table1 VALUES (:FIELD1,:FIELD2);
              |      |        | UPDATE table2 SET field3 = :FIELD3 WHERE field4 = :FIELD4;'              |
| STMT          | 001  |        | INSERT INTO table1 VALUES (:FIELD1,:FIELD2);                             |
| STMT          | 002  |        | UPDATE table2 SET field3 = :FIELD3 WHERE field4 = :FIELD4;               |

Following is the first error table created by the above task. The information indicates a problem with the example task:

<table>
<thead>
<tr>
<th>ImportSeq</th>
<th>DMLSeq</th>
<th>SMTSeq</th>
<th>ApplySeq</th>
<th>Source Seq</th>
<th>ErrorCode</th>
<th>ErrorField</th>
</tr>
</thead>
<tbody>
<tr>
<td>002</td>
<td>001</td>
<td>002</td>
<td>001</td>
<td>20456</td>
<td>2679</td>
<td>field3</td>
</tr>
</tbody>
</table>

Use the following procedure to evaluate error table information to isolate the problem:

1. Check the DMLSeq field to find the DML group. It contains the sequence number 001.
2. Check the STMTSeq field. The sequence number 002 in this field means that the error occurred while executing the second DML statement, which is the UPDATE statement in the example task.
3. Verify that the Update operator job script uses two DML statements in the first DML group (because DMLSeq was 001).
4. Check the ImportSeq field. The value of 002 indicates that the error occurred while processing a row from the second input data source.
5. Check the meaning of the ErrorCode field. Error 2679, “The format or data contains a bad character” indicates a problem with the data from your client system.
6. Check the ErrorField field. The field3 indicates that the error occurred while building field3 of the target table. The name refers to the field in the input schema from the Update operator job script.
7. Because the script shows that the UPDATE statement is loading table2, you now know:
   - What error occurred
   - Which statement detected the error
Chapter 24: Troubleshooting a Failed Job

SQL Selector Operator Errors

- Which input data source has the error
- Which field in table2 has the error

8 Check the SourceSeq field. The value of 20456 indicates that the problem is with the 20,456th record of the input source.

The problem is isolated, and it can now be fixed.

**SQL Selector Operator Errors**

If Teradata Database encounters any errors while the SQL Selector operator is retrieving LOB data, the job will be terminated with error messages.

If any errors occur on the client side (for example, if there is an I/O error while writing LOB data to a file), the SQL Selector operator will issue an explanatory error message and terminate the job.

When a client-side failure causes the SQL Selector operator to terminate the job, the temporary work files that it creates to transport LOB data from the source table to the target may not get deleted.

- If the target is another Teradata table, which means the consumer operator is the SQL Inserter operator, then that operator deletes these temporary work files upon error termination.
- If the target is a flat file, which means the consumer operator is the DataConnector operator, the files are not deleted; they remain after the job terminates. Users need to delete these files manually.
Additional Debugging Strategies for Complex Job Failures

In some cases, simply evaluating the job logs and error tables does not provide enough information to adequately define the required corrective action. In other cases, the corrective action is in place, but the job still doesn’t run correctly. In these cases, Teradata PT provides two additional levels of debugging:

- Check the values of system resources such as shared memory, processes, semaphores, memory, and so on.
  For example, on Solaris SPARC platform, use the following commands to get the values:
  - `/usr/sbin/sysdef -i | grep SHMMAX`
  - `/usr/sbin/sysdef -i | grep SHMSEG`
  - `/usr/sbin/sysdef -i | grep SEMMNI`
  - `/usr/sbin/sysdef -i | grep SEMMNS`
  - `ulimit -a`
- Run the job in trace mode
  - `tbuild -t -f <job file>`
  - Run the operators in your Teradata PT job in trace mode using the TraceLevel attribute.
    ```
    TraceLevel = 'all'
    ```
- Provide truss output (UNIX only) from the Teradata PT problem component if any of the following errors occurs:
  - IPC Initialization Error (Inter-Process Communication problem)
  - Failed to create Coordinator task
  - Unexpected hanging
- Use the following steps to get the truss output of the problem component:
  a  `ps -ef | grep tbuild` (if Coordinator, or Executor).
  b  Find the processid for the problem component.
  c  `truss -f -o /tmp/trussout -p <processid>`.
Teradata PT provides fault tolerance by allowing a stopped job to restart from an internal checkpoint rather than requiring that the job be rerun from the beginning.

For information on using `tbuild` to setup job-specific checkpoint and restart options, see “Setting `tbuild` Options” on page 261.

Consider the following Teradata PT job stop/restart scenarios:

- If the Teradata Database restarts, the job waits until the database is back online and then automatically resumes the job from the last known checkpoint.
- If the job was held up by a Teradata Database lock and the lock is resolved, the job automatically resumes the from the last known checkpoint.
- If you pause a job using the `twbcmd` JOB PAUSE option, you can restart it from the same point it was paused using the `twbcmd` JOB RESUME option. The JOB PAUSE command automatically takes a checkpoint.
- If you terminate a job using the `twbcmd` JOB TERMINATE option, it takes a checkpoint and is restartable from that point.
- If a job fails due to a fatal error, you can manually restart the job from the last recorded checkpoint before the error occurred, resubmitting the job.

**Checkpoint Functionality**

When a Teradata PT job logs a checkpoint, the producer operator in the currently-executing job step stops putting rows into the output data stream, and the consumer operator processes all the rows in the input data stream. All executing operators write records to the job checkpoint files with the information that would allow them to resume processing with no loss or duplication of data at the point the checkpoint was completed.

Teradata PT automatically creates a start-of-data and an end-of-data checkpoint. In addition, you can use the `tbuild` command to specify a user-defined checkpoint interval (in seconds).

**Handling Data Processed After the Checkpoint**

If rows are already in the data streams or loaded when a job fails, the restarting of the job could cause the same rows to be sent again. Here is how the operators handle duplicate rows on restart:

- **Load Operator:** Duplicate rows are always thrown away, in the Application Phase.
- **Update Operator:** While duplicate rows are valid for multiset tables, rows that are sent again during restart would be identified by DBS as “duplicate” and would be ignored or sent to the error table based on user-specified DML options.
- **Stream Operator:** If the Stream Operator has not sent the rows to the DBS, then there will be no duplicates on the target table. If the Stream operator has sent rows to the DBS:
  - If ROBUST recovery is on, then Stream Operator will not re-send the rows when the job is restarted. ROBUST recovery is the default.
  - If ROBUST recovery is off, then Stream Operator will re-send the rows to the DBS.
Automatic Restarts

Teradata PT automatically restarts a job when a “retryable” error, such as a database restart or deadlock occurs before, during, or after the loading of data. The job will restart on its own without a manual resubmission of the `tbuild` command.

Jobs will automatically restart as many times as specified at the original job launch with the `tbuild -R` (*not* the lowercase `-r`) option. If `-R` is not specified in the `tbuild` command that launches the job, the default limit of up to five restarts will apply.

Automatic restarts will use the last interval checkpoint taken, if interval checkpointing is specified for the job. If not the automatic restart will use the two standard default checkpoints.

Restarting from a Job Step

The Teradata PT has a facility to start a job at the job step specified with `tbuild` command option `-s`:

```
tbuild -f <script file name> -s <job step identifier>
```

where `<job step identifier>` is the job step name in the job script, or the implicit step number, 1, 2, ..., corresponding to the top-to-bottom order in which the steps are defined in the script.

**Note:** This command is not intended for use in normal job restarts. Use it only if you do not want to finish the work in the job step that was executing at the time the job was interrupted.

There are two ways to restart from a job step:

- If you specify a job step *before* the step that was interrupted, or the interrupted step itself, the job will restart at the interrupted step, using either of the following: the default Start-of-Data checkpoint (if no checkpoint interval was originally specified) or the last interval-driven checkpoint taken during the step. In these cases, the result is the same as if the `tbuild` command option `-s` had not been specified
- If interval checkpointing was not specified in the `tbuild` statement that launched the job, the job will restart from the default Start-of-Data checkpoint for the step.
- If interval checkpointing was specified in the `tbuild` command that launched the job, the job will restart from the last interval checkpoint before the failure.
- If you specify a job step *beyond* the step that was interrupted, then the job will restart at the specified step; any unfinished work in the interrupted step will not be completed, and any other job steps between the interrupted step and the specified step will not be executed. This approach would likely produce bad results and is not recommended.

Teradata recommends that you do not use the `tbuild -s` option to restart a job from a job step unless you are fully aware of the how it will affect the job.
Restarting a Job From the Last Checkpoint Taken

To restart a job from the last checkpoint taken, do the following:

1. Determine whether the error that caused the failure is associated with an operator that offers full or limited support of checkpoint restarts.
2. Determine the identity and location of the checkpoint file `tbuild` will use for the restart and whether or not you need to specify a checkpoint interval.
3. Run the `tbuild` restart command.
4. Once the job restarts and runs correctly, Teradata PT will delete the checkpoint files automatically.

Support for Checkpoint Restarts

Support for checkpoint restartability varies by operator:

• The following operators fully support checkpoint restartability:
  • Load
  • Update
  • Stream
  • DataConnector
  • FastLoad INMOD Adapter
  • FastExport OUTMOD Adapter
  • MultiLoad INMOD Adapter

• These operators support limited checkpoint restartability:
  • DDL is restartable from the SQL statement that was being executed, but had not completed, at the time the original run of the job terminated.
  • Export and SQL Selector operators are restartable, but not during the exporting of data, as these operators take a checkpoint only when all of the data has been sent to the Teradata PT data stream. Restarting from this checkpoint prevents the operators from having to resend the data.

• The following operators do not support checkpoint restartability:
  • SQL Inserter
  • The ODBC operator does not support checkpoint and restart operations because it is unknown how the databases to which it can connect will handle restarts.

Locating Checkpoint Files

Checkpoint files must be specified in the `tbuild` command that restarts the job. Checkpoint files can be found in the following options locations, depending on how your site and the job are set up.

• In the Global Configuration File -- `twbcfg.ini`
  The Teradata PT installation automatically creates a directory named checkpoint (in italics) as the default checkpoint directory under the directory in which the Teradata PT
software is installed. This checkpoint directory name is automatically recorded in the Global Configuration File (twbcfg.ini) during the installation of the Teradata PT software.

- In the Local Configuration File -- $HOME/.twbcfg.ini (UNIX only)

On UNIX, the checkpoint directory can be set up through the Local Configuration File -- file twbcfg.ini (in italics) in your home directory. The Local Configuration File takes precedence if the CheckpointDirectory entry is defined in both the Global Configuration File and the Local Configuration File. Any changes made to the Local Configuration File affect only the individual user. On Windows there is no Local Configuration File.

- As defined by the `tbuild` -r option

```
tbuild -f <script file name> -r <checkpoint directory name>
```

The -r option of the `tbuild` command sets up the checkpoint directory with the specified name. This option overrides -- only for the job being submitted -- any default checkpoint directory that is specified in the Teradata PT configuration files.

For more information about setting up checkpoint directories, see *Teradata Tools and Utilities Installation Guide* for UNIX and Linux.

If the entry CheckpointDirectory is defined in both configuration files, the one defined in the local configuration file takes precedence. Note that whatever is specified in the local configuration file affects only its owner, not other users.

**Note:** On the z/OS platform, checkpoint datasets are defined in the Job Control Language for a Teradata PT job.

For information on setting up the configuration files for the checkpoint directories, see “Setting Up Configuration Files” on page 67.

**Default Checkpoint File Names**

Each Teradata PT job automatically creates three associated checkpoint files during job execution and places them in the specified checkpoint directories. These files extend across multiple job steps, if the job has more than one step. They are automatically deleted after a job runs all the way to completion without errors, but if any step fails to finish successfully, the checkpoint files are *not* deleted and remain in the checkpoint directory.

Default name formulas for the standard job checkpoint files vary by operating system as follows:

On UNIX and Windows platforms:

- `<job identifier>CPD1`
- `<job identifier>CPD2`
- `<job identifier>LVCP`

where `<job identifier>` is the job name from the tbuild command line, if a jobname was specified, or the userid in the job logon, if a job name was not specified.

On z/OS platforms, the checkpoint datasets have the following DDNAMEs:

- `<high-level qualifier>.CPD1`
- `<high-level qualifier>.CPD2`
• \(<\text{high-level qualifier}>\).LVCP

where \(<\text{high-level qualifier}>\) is a user-supplied parameter of the Job Control Language
procedure TBUILD for running Teradata PT jobs.

**Use tbuild to Restart the Job**

Use one of the following variations to restart a failed job. To restart any job that terminated
abnormally, use the same `tbuild` command that you used to submit the job the first time. The
job will then be automatically restarted at the point where the last checkpoint was taken.

**Restarting with a Default Job Name**

When no job name is specified in the `tbuild` statement at job launch, Teradata PT assigns a
default name to the job that is based on the login name, and creates a checkpoint file called
\(<\text{username}>\).LVCP.

Jobs executed under the same login name, therefore, use the same \(<\text{username}>\).LVCP file,
which can be a problem if a job fails because the checkpoint file associated with a failed job
remains in the checkpoint directory.

Starting a new job before restarting the failed job results in unpredictable errors because the
new job will use the checkpoint file of the failed job. To avoid such errors, do the following:

• Restart failed jobs and run them to completion before starting any new jobs.
• Always delete the checkpoint file of failed jobs before starting a new job. Restarting a failed
  job after deleting its checkpoint file will cause it to restart from its beginning.
• Always specify the `jobname` parameter for all jobs so every job has a unique checkpoint file.

**Restart Failures Due to Checkpoint Files**

**Error message:** Cannot get current job step from the Checkpoint file.

This type of job termination occurs when a restarted job uses a checkpoint file that is either
out-of-date or that was created by another job.

**Solution:**

• If the checkpoint file is out-of-date, manually delete the file from the `TWB_ROOT/Checkpoint` directory.
• If the checkpoint file was created by another job, this means that the job does not have a
  unique job name. Specify a unique job name in the tbuild command using the `jobname`
  parameter so Teradata PT can create a unique checkpoint file for the job.

To avoid this problem, only submit jobs with unique, specified job names.

For more information about checkpoint restarting, see “Teradata PT Features” in *Teradata Parallel Transporter Reference*. 
Removing Checkpoint Files

Job checkpoint files are automatically deleted if the job completes without an error. However, you will need to remove checkpoint files before they are automatically deleted if you want to do either of the following:

- Rerun an interrupted job from the beginning, rather than restart it from the last checkpoint taken before the interruption. Delete the checkpoint files before job restart, so the job can start from the beginning.
- Abandon an interrupted job and run another job, but the new job checkpoint files will have the same names as the existing checkpoint files.

Use the methods shown in the following sections to remove checkpoint files for a specified user ID or jobname.

Using twbrmcp to Remove Checkpoint Files

Use the `twbrmcp` command to remove checkpoint files for a specified user ID or jobname, on Windows or Unix only, as follows:

If the job script specifies a job name:
```
twbrmcp <job name>
```

If the job does not specify a job name
```
twbrmcp <userid>
```

For detailed syntax and options, see the section on `twbrmcp` in *Teradata Parallel Transporter Reference*.

Manually Deleting Checkpoint Files

Instead of using `twbrmcp`, you can delete the files manually. The procedure for manually deleting files varies depending on the operating system.

On Unix or Windows

- To delete checkpoint files from a directory defined by Teradata PT, enter the following command:
  - On Unix:
    ```
    rm $TWB_ROOT/checkpoint/*
    ```
  - On Windows:
    ```
    del %TWB_ROOT%\checkpoint\*. *
    ```
- To delete checkpoint files from a user-defined directory, enter the following command:
  - On Unix:
    ```
    rm <user-defined directory>/*
    ```
  - On Windows:
    ```
    del <user-defined directory>\*. *
    ```
On Z/OS

On z/OS, you can remove checkpoint files with either of the following two methods:

Method 1:

1. Go to the Data Set Utility panel (panel 3.2) in the Primary Options Menu of the TSO System Productivity Facility.
2. Enter the name of each checkpoint file in the name entry fields provided on this panel.
3. Type D (for "delete") for the requested data set option.
4. Hit Enter

Method 2:

Add a step to the beginning of your next Teradata PT job, with the following Job Control Language statements:

```plaintext
//DELETE PGM=IEFBR14
//CPD1 DD DISP=(OLD,DELETE),DSNAME=<high-level qualifier>.CPD1
//CPD2 DD DISP=(OLD,DELETE),DSNAME=<high-level qualifier>.CPD2
//LVCP DD DISP=(OLD,DELETE),DSNAME=<high-level qualifier>.LVCP
```

where `<high-level qualifier>` is the high-level qualifier you supplied to the TBUILD JCL PROC when you submitted the job that created these checkpoint files. Or substitute the names of your checkpoint datasets for everything to the right of DSNAME= above, if you have a different convention for naming them.

For examples of naming and using checkpoint datasets on z/OS, see the section on JCL Examples in Attachment C: “z/OS Sample Files.”
This chapter describes notify exit routines.

Topics include:
- Notify Exit Routines
- Using Notify Exit Routines to Monitor Events
- Compiling and Linking Notify Exit Routines
- Examples

## Notify Exit Routines

The Load, Export, Update, and Stream operators support notify exit routines. A notify exit routine specifies a predefined action to be performed whenever certain significant events occur during a job, for example, whether a load job succeeds or fails, how many records are loaded, what the return code is for the failed job, and so on. Only the main instance sends a notify event.

Jobs accumulate operational information about specific events that occur. If the NotifyMethod attribute specifies the Exit method, when the specific events occur the operator calls the named notify exit routine and passes the following to it:
- An event code to identify the event
- Specific information about the event

## Export Operator Events

The Export operator supports notify exit routines. A notify exit routine specifies a predefined action to be performed whenever certain significant events occur during a job, for example, whether a job succeeds or fails, how many records are exported, what the return code is for the failed job, and so on. Only the main instance sends a notify event.

The Export operator job accumulates operational information about specific events that occur during a job. If the Export operator job script includes the NotifyMethod attribute with the Exit method specification, then when the specific events occur, the Export operator calls the named notify exit routine and passes to it:
- An event code to identify the event
- Specific information about the event
Table 22 lists the event codes and describes the data that the Export operator passes to the notify exit routine for each event. The information in this table is also sent to the system log.

**Note:** Ensure that notify exit routines *ignore* invalid or undefined event codes, and that they *do not* cause the operator to terminate abnormally.

### Table 22: Export Operator Notify Event Codes

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Code</th>
<th>Event Description and Data Passed to the Notify Exit Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>0</td>
<td>Signifies successful processing of the notify feature:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Version ID length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Version ID string—32-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Operator ID—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Operator name length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Operator name string—32-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• User name length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• User name string—64-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Optional string length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Optional string—80-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Operator handle—4-byte unsigned integer</td>
</tr>
<tr>
<td>Teradata Database Restart</td>
<td>9</td>
<td>Signifies that the Export operator received a crash message from the Teradata Database or from the CLIv2:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No data accompanies the Teradata Database restart event code</td>
</tr>
<tr>
<td>CLIv2 Error</td>
<td>10</td>
<td>Signifies that the Export operator received a CLIv2 error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Error code—4-byte unsigned integer</td>
</tr>
<tr>
<td>Teradata Database Error</td>
<td>11</td>
<td>Signifies that the Export operator received a Teradata Database error that will produce an exit code of 12:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Error code—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Note:</strong> Not all Teradata Database errors cause this event. A 3807 error, for example, while trying to drop or create a table does not terminate the Export operator.</td>
</tr>
<tr>
<td>Exit</td>
<td>12</td>
<td>Signifies that the Export operator is terminating:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Exit code—4-byte unsigned integer</td>
</tr>
<tr>
<td>Export Begin</td>
<td>31</td>
<td>Signifies that the Export operator is about to begin the export task by opening the export file:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No data accompanies the export begin event code</td>
</tr>
<tr>
<td>Request Submit Begin</td>
<td>32</td>
<td>Signifies that the Export operator is about to submit the SELECT request to the Teradata Database:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Request length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Request text—32,000-character (maximum) array</td>
</tr>
</tbody>
</table>
Table 22: Export Operator Notify Event Codes (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Code</th>
<th>Event Description and Data Passed to the Notify Exit Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request Submit End</td>
<td>33</td>
<td>Signifies that the Export operator has received the response to the SELECT request:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Statement count—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Block count—4-byte unsigned integer</td>
</tr>
<tr>
<td>Request Fetch Begin</td>
<td>34</td>
<td>Signifies that the Export operator is about to fetch the results of the SELECT request</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• No data accompanies the request fetch begin event code</td>
</tr>
<tr>
<td>File or OUTMOD Open</td>
<td>35</td>
<td>Signifies that the Export operator is about to open an output or OUTMOD routine file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• File name length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• File name—256-character (maximum) array</td>
</tr>
<tr>
<td>Statement Fetch Begin</td>
<td>36</td>
<td>Signifies that the Export operator is about to fetch the current statement in a request:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Statement number—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Block count—4-byte unsigned integer</td>
</tr>
<tr>
<td>Statement Fetch End</td>
<td>37</td>
<td>Signifies that the Export operator has fetched all of the records for the current statement:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Record count—4-byte unsigned integer</td>
</tr>
<tr>
<td>Request Fetch End</td>
<td>38</td>
<td>Signifies that the Export operator has fetched all of the records for the current request:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Records exported—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Records rejected—4-byte unsigned integer</td>
</tr>
<tr>
<td>Export End</td>
<td>39</td>
<td>Signifies that the Export operator has completed the export operation and displayed the number of exported records:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Records exported—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Records rejected—4-byte unsigned integer</td>
</tr>
</tbody>
</table>

Table 23 lists events that create notifications.

Table 23: Export Operator Events That Create Notifications

<table>
<thead>
<tr>
<th>Event</th>
<th>Notification Level</th>
<th>Signifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Teradata Database</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Restart</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>CLIv2 Error</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Successful processing of the notify option

A crash error from the Teradata Database or the CLIv2

A CLIv2 error was encountered
Table 23: Export Operator Events That Create Notifications (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Notification Level</th>
<th>Signifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teradata Database Error</td>
<td>Yes</td>
<td>A Teradata Database error was encountered that terminates the Export operator</td>
</tr>
<tr>
<td>Exit</td>
<td>Yes</td>
<td>The Export operator is terminating</td>
</tr>
<tr>
<td>Export Begin</td>
<td>No</td>
<td>Opening the export file</td>
</tr>
<tr>
<td>Request Submit Begin</td>
<td>No</td>
<td>Submitting the SELECT request</td>
</tr>
<tr>
<td>Request Submit End</td>
<td>No</td>
<td>Received SELECT request response</td>
</tr>
<tr>
<td>Request Fetch Begin</td>
<td>No</td>
<td>Fetching SELECT request results</td>
</tr>
<tr>
<td>File or OUTMOD Open</td>
<td>No</td>
<td>Opening output file or OUTMOD</td>
</tr>
<tr>
<td>Statement Fetch Begin</td>
<td>No</td>
<td>Fetching current statement</td>
</tr>
<tr>
<td>Statement Fetch End</td>
<td>No</td>
<td>Last record fetched for current statement</td>
</tr>
<tr>
<td>Request Fetch End</td>
<td>No</td>
<td>Last record fetched for current request</td>
</tr>
<tr>
<td>Export End</td>
<td>No</td>
<td>Export task completed</td>
</tr>
</tbody>
</table>

**Load Operator Events**

Table 24 lists the event codes and describes the data that operators pass to the notify exit routine and the system log for each event.

**Note:** To support future enhancements, always verify that notify exit routines *ignore* invalid or undefined event codes, and that they *do not* cause the operator to terminate abnormally.
## Table 24: Load Operator Notify Event Codes

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Code</th>
<th>Event Description and Data Passed To The Notify Exit Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>0</td>
<td>Signifies successful processing of the notify feature:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Version ID length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Version ID string—32-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Operator ID—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Operator name length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Operator name string—32-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* User name length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* User name string—64-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Optional string length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Optional string—80-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Operator handle—4-byte unsigned integer</td>
</tr>
<tr>
<td>Phase 1 Begin</td>
<td>2</td>
<td>Signifies the beginning of the insert phase, where the table name is specified by the INSERT statement:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Table name length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Table name—128-character (maximum) array</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>3</td>
<td>Signifies that checkpoint information is written to the restart log table:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Record number—4-byte unsigned integer</td>
</tr>
<tr>
<td>Phase 1 End</td>
<td>4</td>
<td>Signifies the CHECKPOINT LOADING END request has successfully completed after the end of the insert phase:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Records read—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Records sent to the Teradata Database—4-byte unsigned integer</td>
</tr>
<tr>
<td>Phase 2 Begin</td>
<td>5</td>
<td>Signifies that the application phase is beginning:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* No data accompanies the phase 2 begin event code</td>
</tr>
<tr>
<td>Phase 2 End</td>
<td>6</td>
<td>Signifies that the application phase is complete:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Records loaded—4-byte unsigned integer</td>
</tr>
<tr>
<td>Error Table 1</td>
<td>7</td>
<td>Signifies that processing of the SEL COUNT(*) request completed successfully for the first error table:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Number of rows—4-byte unsigned integer</td>
</tr>
<tr>
<td>Error Table 2</td>
<td>8</td>
<td>Signifies that processing of the SEL COUNT(*) request completed successfully for the second error table:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Number of rows—4-byte unsigned integer</td>
</tr>
<tr>
<td>Teradata Database Restart</td>
<td>9</td>
<td>Signifies that the Load operator received a crash message from the Teradata Database or from the CLIv2:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* No data accompanies the Teradata Database restart event code</td>
</tr>
<tr>
<td>CLIv2 Error</td>
<td>10</td>
<td>Signifies that the Load operator received a CLIv2 error:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* Error code—4-byte unsigned integer</td>
</tr>
</tbody>
</table>
Chapter 25: Notify Exit Routines

Table 24: Load Operator Notify Event Codes (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Code</th>
<th>Event Description and Data Passed To The Notify Exit Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teradata Database Error</td>
<td>11</td>
<td>Signifies that the Load operator received a Teradata Database error that will produce an Exit code of 12:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Error code—4-byte unsigned integer</td>
</tr>
<tr>
<td>Exit</td>
<td>12</td>
<td>Signifies that the Load operator is terminating:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Exit code—4-byte unsigned integer</td>
</tr>
</tbody>
</table>

Table 25 lists events that create notifications.

Table 25: Load Operator Events That Create Notifications

<table>
<thead>
<tr>
<th>Event</th>
<th>Notification Level</th>
<th>Signifies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
<td>Med</td>
</tr>
<tr>
<td>Initialize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Phase 1 Begin</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Phase 1 End</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Phase 2 Begin</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Phase 2 End</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Error Table 1</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Error Table 2</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Teradata Database Restart</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>CLIv2 Error</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Teradata Database Error</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Exit</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**Update Operator Events**

Table 26 lists the event codes and describes the data that the Update operator passes to the notify exit routine for each event. The information in this table is also sent to the system log.

**Note:** To support future enhancements, always verify that your notify exit routines ignore invalid or undefined event codes, and that they do not cause the operator to terminate abnormally.
### Table 26: Update Operator Notify Event Codes

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Code</th>
<th>Event Description and Data Passed To The Notify Exit Routine</th>
</tr>
</thead>
</table>
| Initialize   | 0          | Signifies successful processing of the notify feature:  
  • Version ID length—4-byte unsigned integer  
  • Version ID string—32-character (maximum) array  
  • Operator ID—4-byte unsigned integer  
  • Operator name length—4-byte unsigned integer  
  • Operator name string—32-character (maximum) array  
  • User name length—4-byte unsigned integer  
  • User name string—64-character (maximum) array  
  • Optional string length—4-byte unsigned integer  
  • Optional string—80-character (maximum) array  
  • Operator handle—4-byte unsigned integer |
| Phase 1 Begin| 2          | Signifies the beginning of the insert phase, where table name is specified by the INSERT statement:  
  • Table name length—4-byte unsigned integer  
  • Table name—128-character (maximum) array  
  • Table number—4-byte unsigned integer |
| Checkpoint   | 3          | Signifies that checkpoint information is written to the restart log table:  
  • Record number—4-byte unsigned integer |
| Phase 1 End  | 4          | Signifies the CHECKPOINT LOADING END request has successfully completed after the end of the acquisition phase:  
  • Records read—4-byte unsigned integer  
  • Records skipped—4-byte unsigned integer  
  • Records rejected—4-byte unsigned integer  
  • Records sent to the Teradata Database—4-byte unsigned integer |
| Phase 2 Begin| 5          | Signifies that the application phase is beginning:  
  • No data accompanies the phase 2 begin event code |
| Phase 2 End  | 6          | Signifies that the application phase is complete. For each table in the request:  
  • Records inserted—4-byte unsigned integer  
  • Records updated—4-byte unsigned integer  
  • Records deleted—4-byte unsigned integer  
  • Table number—4-byte unsigned integer |
| Error Table 1| 7          | Signifies that processing of the SEL COUNT(*) request completed successfully for the first error table.  
  • Number of rows—4-byte unsigned integer  
  • Table number—4-byte unsigned integer |
Table 26: Update Operator Notify Event Codes (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Code</th>
<th>Event Description and Data Passed To The Notify Exit Routine</th>
</tr>
</thead>
</table>
| Error Table 2                  | 8          | Signifies that processing of the SEL COUNT(*) request completed successfully for the second error table.  
|                                |            | • Number of rows—4-byte unsigned integer  
|                                |            | • Table number—4-byte unsigned integer  
| Teradata Database Restart      | 9          | Signifies that the Update operator received a crash message from the Teradata Database or from the CLIV2:  
|                                |            | • No data accompanies the Teradata Database restart event code  
| CLIV2 Error                    | 10         | Signifies that the Update operator received a CLIV2 error:  
|                                |            | • Error code—4-byte unsigned integer  
| Teradata Database Error        | 11         | Signifies that the Update operator received a Teradata Database error that will produce an Exit code of 12:  
|                                |            | • Error code—4-byte unsigned integer  
|                                |            | **Note:** Not all Teradata Database errors cause this event. A 3807 error, for example, while trying to drop or create a table does not terminate the Update operator.  
| Exit                           | 12         | Signifies that the Update operator is terminating:  
|                                |            | • Exit code—4-byte unsigned integer  
| AMPs Down                      | 21         | Signifies that the Teradata Database has one or more non-operational AMPs, just prior to the acquisition phase:  
|                                |            | • No data accompanies the AMPs Down event code  
| Import Begin                   | 22         | Signifies that the first record is about to be read for each import task:  
|                                |            | • Import number—4-byte unsigned integer  
| Import End                     | 23         | Signifies that the last record is read for each import task. The returned data is the record statistics for the import task:  
|                                |            | • Records read—4-byte unsigned integer  
|                                |            | • Records skipped—4-byte unsigned integer  
|                                |            | • Records rejected—4-byte unsigned integer  
|                                |            | • Records sent to the Teradata Database—4-byte unsigned integer  
|                                |            | • Import number—4-byte unsigned integer  
| Delete Init                    | 24         | Signifies successful processing of a DELETE statement:  
|                                |            | • No data accompanies the init. event code.  
| Delete Begin                   | 25         | Signifies that a DELETE statement is about to be sent to the Teradata Database:  
|                                |            | • Table name length—4-byte unsigned integer  
|                                |            | • Table name—128-character (maximum) array  
|                                |            | • Table number—4-byte unsigned integer  

Chapter 25: Notify Exit Routines
Notify Exit Routines
Table 26: Update Operator Notify Event Codes (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Code</th>
<th>Event Description and Data Passed To The Notify Exit Routine</th>
</tr>
</thead>
</table>
| Delete End             | 26         | Signifies successful processing of a delete task:  
|                        |            | • Records deleted—4-byte unsigned integer  
|                        |            | • Table number—4-byte unsigned integer                                                                                     |
| Delete Exit            | 27         | Signifies the end of a delete task:  
|                        |            | • Exit code—4-byte unsigned integer                                                                                         |

The following table lists events that create notifications. Some events create notifications only for import tasks, some only for delete tasks, and some for both.

Table 27: Update Operator Events That Create Notifications

<table>
<thead>
<tr>
<th>Event</th>
<th>Import Task</th>
<th>Delete Task</th>
<th>Notification Level</th>
<th>Signifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>x</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Successful processing of the notify option</td>
</tr>
<tr>
<td>Phase 1 Begin</td>
<td>x</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The acquisition phase is beginning</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>x</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Checkpoint information is written to the restart log table</td>
</tr>
<tr>
<td>Phase 1 End</td>
<td>x</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Successful completion of the acquisition phase</td>
</tr>
<tr>
<td>Phase 2 Begin</td>
<td>x</td>
<td>x</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The application phase is beginning</td>
</tr>
<tr>
<td>Phase 2 End</td>
<td>x</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Successful completion of the application phase</td>
</tr>
<tr>
<td>Error Table 1</td>
<td>x</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Successful processing of the SEL COUNT(*) request for the first error table</td>
</tr>
<tr>
<td>Error Table 2</td>
<td>x</td>
<td>x</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Successful processing of the SEL COUNT(*) request for the second error table</td>
</tr>
<tr>
<td>Teradata Database</td>
<td>x</td>
<td>x</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Restart</td>
<td></td>
<td></td>
<td></td>
<td>A crash error was encountered from the Teradata Database or CLIV2 that will terminate the load operation</td>
</tr>
<tr>
<td>CLIV2 Error</td>
<td>x</td>
<td>x</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A CLIV2 error was encountered</td>
</tr>
<tr>
<td>Teradata Database</td>
<td>x</td>
<td>x</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Error</td>
<td></td>
<td></td>
<td></td>
<td>A Teradata Database error was encountered that will terminate the load operation</td>
</tr>
<tr>
<td>Exit</td>
<td>x</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The Update operator is terminating</td>
</tr>
</tbody>
</table>
Stream Operator Events

Table 28 lists the event codes and describes the data that the Stream operator passes to the notify exit routine or the system log.

Note: To support future enhancements, always verify that your notify exit routines ignore invalid or undefined event codes, and that they do not cause the operator to terminate abnormally.

Table 28: Stream Operator Notify Event Codes

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Code</th>
<th>Event Description and Data Passed To The Notify Exit Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize</td>
<td>0</td>
<td>Signifies successful processing of the notify option:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Version ID length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Version ID string—32-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Operator ID—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Operator name length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Operator name string—32-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• User Name length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• User Name string—64-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Optional string length—4-byte unsigned integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Optional string—80-character (maximum) array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Operator handle—4-byte unsigned integer</td>
</tr>
<tr>
<td>Checkpoint</td>
<td>2</td>
<td>Signifies that the Stream operator is about to perform a checkpoint operation</td>
</tr>
<tr>
<td>Begin</td>
<td></td>
<td>Record number—4-byte unsigned integer</td>
</tr>
</tbody>
</table>

Table 27: Update Operator Events That Create Notifications (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Import Task</th>
<th>Delete Task</th>
<th>Notification Level</th>
<th>Signifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMPs Down</td>
<td>x</td>
<td>x</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Import Begin</td>
<td>x</td>
<td></td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Import End</td>
<td></td>
<td></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Delete Init</td>
<td>x</td>
<td></td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Delete Begin</td>
<td></td>
<td>x</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Delete End</td>
<td>x</td>
<td></td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Delete Exit</td>
<td></td>
<td>x</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Table 28: Stream Operator Notify Event Codes (continued)

<table>
<thead>
<tr>
<th>Event</th>
<th>Event Code</th>
<th>Event Description and Data Passed To The Notify Exit Routine</th>
</tr>
</thead>
</table>
| Import Begin         | 3          | Signifies that the first record is about to be read for each import task:  
|                      |            |   • Import number—4-byte unsigned integer                   |
| Import End           | 4          | Signifies that the last record is read for each import task. The returned data is the record statistics for the import task:  
|                      |            |   • Import number—4-byte unsigned integer                   |
|                      |            |   • Records read—4-byte unsigned integer                    |
|                      |            |   • Records skipped—4-byte unsigned integer                 |
|                      |            |   • Records rejected—4-byte unsigned integer                |
|                      |            |   • Records sent to the Teradata Database—4-byte unsigned integer |
|                      |            |   • Data Errors—4-byte unsigned integer.                    |
| Error Table          | 5          | Signifies that processing of the SEL COUNT(*) request completed successfully for the error table:  
|                      |            |   • Table Name—128-byte character (maximum) array.          |
|                      |            |   • Number of Rows—4-byte unsigned integer.                 |
| Teradata Database    | 6          | Signifies that the Stream operator received a crash message from the Teradata Database or from the CLIv2:  
| Database Restart     |            |   • No data accompanies the Teradata Database restart event code |
| CLIv2 Error          | 7          | Signifies that the Stream operator received a CLIv2 error:  
|                      |            |   • Error code—4-byte unsigned integer.                     |
| Teradata Database    | 8          | Signifies that the Stream operator received a Teradata Database error that will produce an Exit code of 12:  
| Database Error       |            |   • Error code—4-byte unsigned integer.                      |
|                      |            | **Note:** Not all Teradata Database errors cause this event. A 3807 error, for example, while trying to drop or create a table, does not terminate the Stream operator. |
| Exit                 | 9          | Signifies that the Stream operator completed a load task:  
|                      |            |   • Exit code—4-byte unsigned integer.                      |
| Table Statistics     | 10         | Signifies that the Stream operator has successfully written the table statistics:  
|                      |            |   • Type (I = Insert, U = Update, D = Delete, or M = Merge) — 1-byte character variable. |
|                      |            |   • Database Name — 64-character (maximum) array.           |
|                      |            |   • Table/Macro Name — 64-character (maximum) array         |
|                      |            |   • Activity count—4-byte unsigned integer.                 |
| Checkpoint End       | 11         | Signifies that the Stream operator has successfully completed the checkpoint operation:  
|                      |            |   • Record number—4-byte unsigned integer.                  |
Interim Run Statistics

Event Code: 12

Signifies that the Stream operator is flushing the stale buffers (because a latency interval has expired), has just completed a checkpoint, or has read the last record for an import task. The returned data is the statistics for the current load.

- Import number—4-byte unsigned integer
- Statements sent to the Teradata Database—4-byte unsigned integer
- Requests sent to the Teradata Database—4-byte unsigned integer
- Records read—4-byte unsigned integer
- Records skipped—4-byte unsigned integer
- Records rejected—4-byte unsigned integer
- Records sent to the Teradata Database—4-byte unsigned integer
- Data errors—4-byte unsigned integer

DML Error

Event Code: 13

Signifies that the Stream operator received a Teradata Database error that was caused by DML and will introduce an error-row insert to the error table.

- Import number - 4-byte unsigned integer
- Error code - 4-byte unsigned integer
- Error message - 256-character (maximum) array
- Record number - 4-byte unsigned integer
- Data Input number - 1-byte unsigned char
- DML number - 1-byte unsigned char
- Statement number - 1-byte unsigned char
- Record data - 6,004-character (maximum) array
- Record data length - 4-byte unsigned integer
- Feedback - a pointer to 4-byte unsigned integer

"Feedback" always points to integer 0 when it is passed to the notify exit routine. You may change the value of this integer to 1 to instruct the Stream operator not to log the error to the error table. In this case, the Stream operator will not log the error, but will continue other regular processes on this error.

Table 29 lists events that create notifications.

Table 29: Stream Operator Events that Create Notifications

<table>
<thead>
<tr>
<th>Event</th>
<th>Notification Level</th>
<th>Signifies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Initialize</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Checkpoint Begin</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Using Notify Exit Routines to Monitor Events

A notify exit routine (sometimes called an INMOD) specifies a predefined action to be performed whenever certain significant events occur during a Teradata PT job. Notify exit routines are especially useful in environments where job scheduling relies heavily on automation to optimize system performance.

For example, by writing an exit routine in C (without using CLIv2) and using the NotifyMethod and NotifyExit attributes, you can provide a routine to detect whether a Teradata PT job succeeds or fails, how many records were loaded, what the return code is for a failed job, and so on. In all cases, notify exit routines are dynamically loaded at run time, rather than link edited into the Teradata PT module.
The entry point for notify exit routines for all notify exits must be \_dynamn.

**Interface**

Teradata PT accumulates operational information about specific events that occur during a job. If the script includes a notify attribute with an exit option specification, then, when the specific events occur, Teradata PT calls the named notify exit routine and passes to it:

- An event code to identify the event
- Specific information about the event

The event codes and the data that Teradata PT passes to the notify exit routine for each event encountered are described in the individual operators. The events associated with each level of notification (low, medium and high) are also described in the individual operators.

**Note:** To support future enhancements, always make sure that your notify exit routines ignore invalid or undefined event codes, and that they do not cause Teradata PT to terminate abnormally.

**Samples**

Teradata PT includes sample notify exit routines that show you how to write a notify exit routine using the C programming language.

Sample notify exit routines for the following operators are installed in the sample folder:

- Load operator (ldnfyext.c)
- Update operator (updnfyxt.c)
- Export operator (expnfyxt.c)
- Stream operator (stmnfyxt.c)

**Compiling and Linking Notify Exit Routines**

To meet your particular system needs, you can either write your own notify exit routine, or you can modify the sample notify exit routine provided with your Teradata PT software.

Whenever you create or modify exit routines, you must compile the routine and link it into a shared object for use by Teradata PT.

**UNIX Systems**

**Solaris-SPARC**

Use the following syntax to compile source files into a shared object module for notify exit routines on Solaris-SPARC client systems:
Chapter 25: Notify Exit Routines

Compiling and Linking Notify Exit Routines

Teradata Parallel Transporter User Guide 339

where:

Table 30: UNIX Syntax for Notify Exit Routines

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cc</code></td>
<td>Call to the program that invokes the native UNIX C compiler.</td>
</tr>
<tr>
<td><code>-G</code></td>
<td>Linker option that generates a shared object file.</td>
</tr>
<tr>
<td><code>-KPIC</code></td>
<td>Compiler option that generates Position Independent Code (PIC) for all notify exit routines.</td>
</tr>
<tr>
<td><code>-o</code></td>
<td>Switch to the linker.</td>
</tr>
<tr>
<td><code>shared-object-name</code></td>
<td>Name of your shared object file. The <code>shared-object-name</code> can be any valid UNIX file name. This is the name you specify in the NotifyExit attribute value supplied in the operator definition section of a job script.</td>
</tr>
<tr>
<td><code>sourcefile</code></td>
<td>UNIX file name(s) of the source file(s) for your notify exit routine.</td>
</tr>
</tbody>
</table>

**HP-UX**

Use the following syntax to compile and link source files into a shared object module for notify exit routines on HP-UX client systems:

**Compile Syntax**

```
cc  -Aa  -D_HPUX_SOURCE  +z  +ul  -c  -o sourcefile.c
```

**Link Syntax**

```
ld  -b  objectfile.o  -o shared-object-name
```

where:
Chapter 25: Notify Exit Routines
Compiling and Linking Notify Exit Routines

Table 31: HP-UX Syntax for Notify Exit Routines

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Aa</td>
<td>Compiler option which enables compiler to conform to the ANSI standard.</td>
</tr>
<tr>
<td>-b</td>
<td>Linker option that generates a shared object file.</td>
</tr>
<tr>
<td>-c</td>
<td>Compile-only option (does not link).</td>
</tr>
<tr>
<td>cc</td>
<td>Call to the program that invokes the native UNIX C compiler.</td>
</tr>
<tr>
<td>-D_HPUX_SOURCE</td>
<td>Symbol that enables the compiler to access macros and typedefs that are not defined by the ANSI Standard but are provided by the HPUX Operating System.</td>
</tr>
<tr>
<td>ld</td>
<td>Call to the program that invokes the native UNIX linker.</td>
</tr>
<tr>
<td>objectfile</td>
<td>File that the compiler generates and linker uses to generate shared-object-name.</td>
</tr>
<tr>
<td>-o</td>
<td>Switch to the linker.</td>
</tr>
<tr>
<td>shared-object-name</td>
<td>Name of your shared object file. The shared-object-name can be any valid UNIX file name. This is the name you specify in the NotifyExit attribute value supplied in the operator definition section of a job script.</td>
</tr>
<tr>
<td>+z</td>
<td>Compiler option that generates Position Independent Code for all notify exit routines.</td>
</tr>
<tr>
<td>sourcefile</td>
<td>UNIX file name(s) of the source file(s) for your notify exit routine.</td>
</tr>
<tr>
<td>+ul</td>
<td>Compiler option that allows pointers to access non-natively aligned data.</td>
</tr>
</tbody>
</table>

**Linux**

Use the following syntax to compile source files into a shared object module for notify exit routines on Linux client systems:

```
gcc -shared sourcefile -o shared-object-name
```

where:

Table 32: Linux Syntax for Notify Exit Routines

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>Call to the program that invokes the gcc compiler.</td>
</tr>
<tr>
<td>-shared</td>
<td>Link option that generates a shared object file.</td>
</tr>
</tbody>
</table>
IBM AIX

Use the following syntax to compile and link source files into a shared object module for notify exit routines on IBM AIX client systems:

**Compile Syntax**

```
cc  -c  -brtl  -fPIC  sourcefile.c
```

**Link Syntax**

```
ld  -G  -e_dynamn  -bE:export_dynamn.txt  
```

where:

Table 33: IBM AIX Syntax for Notify Exit Routines

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-be:</td>
<td>Linker option that exports the symbol &quot;_dynamn&quot; explicitly and the file <code>export-dynamn.txt</code> contains the symbol.</td>
</tr>
<tr>
<td><code>export_dynamn.txt</code></td>
<td></td>
</tr>
<tr>
<td>-bexpall</td>
<td>Option that exports all global symbols, except imported symbols, unf-referenced symbols defined in archive members, and symbols beginning with an underscore (_).</td>
</tr>
<tr>
<td>-brtl</td>
<td>Option that tells the linkage editor to accept both .so and .a library file types.</td>
</tr>
<tr>
<td>-c</td>
<td>Compiler option specifying to not send object files to the linkage editor.</td>
</tr>
</tbody>
</table>
IBM OS z/OS

Use the following syntax from the USS environment (due to its ease of use) to compile and link source files to a DLL:

```
cc -o "//MVS.PDSE.LOAD(module_name)"

A -W c,dll,expo -W l,dll
```

where:

- `cc`: Call to the program that invokes the native UNIX C compiler.
- `-o`: Switch to the linker.
- `"//MVS.PDSE.LOAD(module_name)"`: File that the compiler generates and linker uses to generate shared-object-name.
- `shared-object-name`: The shared-object-name can be any valid UNIX file name. This is the name you specify in the NotifyExit attribute value supplied in the operator definition section of a job script.
- `sourcefile`: UNIX file name(s) of the source file(s) for your notify exit routine.
- `-W c,dll,expo`: Link with the /lib/libc.a library.
- `-W l,dll`: Link with the /lib/libc.a library.

**Table 33: IBM AIX Syntax for Notify Exit Routines (continued)**

<table>
<thead>
<tr>
<th>Syntax Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc</td>
<td>Call to the program that invokes the native UNIX C compiler.</td>
</tr>
<tr>
<td>-e_dynamn</td>
<td>Option that sets the entry point of the notify exit routine to _dynamn.</td>
</tr>
<tr>
<td>-fPIC</td>
<td>Compiler option that generates Position Independent Code for all notify exit routines.</td>
</tr>
<tr>
<td>-G</td>
<td>Option that produces a shared object-enabled for use with the runtime linker.</td>
</tr>
<tr>
<td>-lc</td>
<td>Link with the /lib/libc.a library.</td>
</tr>
<tr>
<td>ld</td>
<td>Call to the program that invokes the native UNIX linker.</td>
</tr>
<tr>
<td>-lm</td>
<td>Link with the /lib/libc.a library.</td>
</tr>
<tr>
<td>-o</td>
<td>Switch to the linker.</td>
</tr>
<tr>
<td>objectfile</td>
<td>File that the compiler generates and linker uses to generate shared-object-name.</td>
</tr>
<tr>
<td>shared-object-name</td>
<td>The shared-object-name can be any valid UNIX file name. This is the name you specify in the NotifyExit attribute value supplied in the operator definition section of a job script. When creating a shared object module for a notify exit routine, if the notify exit routine uses functions from an external library, then that library must be statically linked with the notify exit routine so that Teradata PT can resolve the external references.</td>
</tr>
<tr>
<td>sourcefile</td>
<td>UNIX file name(s) of the source file(s) for your notify exit routine.</td>
</tr>
</tbody>
</table>
Chapter 25: Notify Exit Routines

Windows Platforms

Consider the following

- To generate and use a notify exit routine with Teradata PT on a Windows client system, the routine must meet the following requirements:
  - Written in C
  - The _dynamn entry point is __declspec
  - Saved as a dynamic-link library (DLL) file
- To generate an INMOD or notify exit routine on a network-attached Windows client, do the following:
  a. Edit the routine source file and ensure that the dynamn is named "__declspec."
  b. Create a dynamic link library by entering the following command (in Microsoft C
     compiler syntax) where sourcefilename is the name of your INMOD or notify exit
     routine source file:

    cl /DWIN32 /LD sourcefilename

    This command produces a file with the same name as the source file, but with a .dll file
    extension.
- To use a compiler other than the Microsoft C compiler, see the documentation for that
  compiler for instructions on creating .dll files.

For sample scripts that generate INMOD and notify exit routines, see the following examples.

Examples

The following examples are suitable for use on all platforms.
# Load Operator Example

************************************************************************
* TITLE: ldnfyext.c .... an example notify exit...
* COPYRIGHT (C) Teradata Corporation. 1999-2008
* Description This file is a sample user exit routine for
  processing NOTIFY events
* History Information
* Revision    Date       DCR   DID     Comments
* ----------- ---------- ----- ------- --------------------------------
* 08.01.00.01 03/01/2005 67847 XXX Initial Version
* Notes         The entry point to this User Exit must
*               be called "_dynamn"
************************************************************************/

#include <stdio.h>

typedef int Int32;
typedef unsigned int UInt32;
typedef enum { NFEventInitialize = 0,
  NFEventInmodOpen = 1,
  NFEventPhaseIBegin = 2,
  NFEventCheckPoint = 3,
  NFEventPhaseIEnd = 4,
  NFEventPhaseIIBegin = 5,
  NFEventPhaseIIEnd = 6,
  NFEventDropErrTableI = 7,
  NFEventDropErrTableII = 8,
  NFEventDBSRestart = 9,
  NFEventCLIError = 10,
  NFEventDBSError = 11,
  NFEventExit = 12
} NfyLDEvent;

#define NOTIFYID_LOAD          1
#define NOTIFYID_UPDATE        2
#define NOTIFYID_EXPORT        3
#define NOTIFYID_INSERT        4
#define NOTIFYID_STREAM        5
#define MAXVERSIONIDLEN       32
#define MAXUTILITYNAMELEN     32
#define MAXUSERNAMELEN        64
#define MAXUSERSTRLEN         80
#define MAXTABLENAMELEN      128
#define MAXFILENAMELEN       256

typedef struct _LDNotifyExitParm {
  Int32 Event;
  union {
    struct {
      UInt32 VersionLen;
      char VersionId[MAXVERSIONIDLEN];
      UInt32 UtilityId;
      UInt32 UtilityNameLen;
      char UtilityName[MAXUTILITYNAMELEN];
      UInt32 UserNameLen;
      char UserName[MAXUSERNAMELEN];
      UInt32 UserStringLen;
      char UserString[MAXUSERSTRLEN];
    } Initialize;
} NfyLDEvent;
struct {
    UInt32 InmodNameLen;
    char InmodName[MAXFILENAMELEN];
} InmodOpen;
struct {
    UInt32 TableNameLen;
    char TableName[MAXTABLENAMELEN];
    UInt32 dummy;
} PhaseIBegin;
struct {
    UInt32 RecordCount;
} CheckPoint;
struct {
    UInt32 RecsRead;
    UInt32 RecsSkipped;
    UInt32 RecsRejected;
    UInt32 RecsSent;
} PhaseIEnd;
struct {
    UInt32 dummy;
} PhaseIIBegin;
struct {
    UInt32 Inserts;
    UInt32 dummy1;
    UInt32 dummy2;
    UInt32 dummy3;
} PhaseIIEnd;
struct {
    UInt32 Rows;
    UInt32 dummy;
} DropErrTableI;
struct {
    UInt32 Rows;
    UInt32 dummy;
} DropErrTableII;
struct {
    UInt32 dummy;
} DBSRestart;
struct {
    UInt32 ErrorCode;
} CLIError;
struct {
    UInt32 ErrorCode;
} DBSError;
struct {
    UInt32 ReturnCode;
} Exit;
} Vals;
} LDNotifyExitParm;
Chapter 25: Notify Exit Routines
Examples

************************************************************************
* CODE STARTS HERE
************************************************************************/
#ifdef WIN32
__declspec(dllexport) Int32 _dynamn(LDNotifyExitParm *P)
#else
Int32 _dynamn(LDNotifyExitParm *P)
#endif
{
  static FILE *fp;
  if (!fp) {
    #ifdef I370
    if (!(fp = fopen("NFYEXIT", "w")))
      return(1);
    #else
    if (!(fp = fopen("NFYEXIT.OUT", "w")))
      return(1);
    #endif
  }
  switch(P->Event) {
    case NFEventInitialize :
      fprintf(fp,
              "exit called @ Load Operator Notify initialization.
              Version Id:   %s.
              Utility Id:   %d.
              Utility Name: %s.
              User Name:    %s.
              ");
      fprintf(fp,
              "                  %s.
              ");
      fprintf(fp,
              "                  %s.
              ");
      fprintf(fp,
              "                  %s.
              ");
      fprintf(fp,
              "                  %s.
              ");
      break;
    case NFEventInmodOpen :
      fprintf(fp,
              "exit called @ Load Operator file/inmod open: %s.
              ");
      break;
    case NFEventPhaseIBegin :
      fprintf(fp,
              "exit called @ Load Operator phase I (start) for table
              ");
      break;
    case NFEventCheckPoint :
      fprintf(fp,
              "exit called @ Load Operator checkpoint : %d records
              ");
      break;
  }
}
case NFEEventPhaseIEnd :
    fprintf(fp, "exit called @ Load Operator phase I (end).\n") ;
    fprintf(fp, "Records read: %d\n", P->Vals.PhaseIEnd.RecsRead);
    fprintf(fp, "Records skipped: %d\n", P->Vals.PhaseIEnd.RecsSkipped);
    fprintf(fp, "Records rejected: %d\n", P->Vals.PhaseIEnd.RecsRejected);
    fprintf(fp, "Records sent: %d\n", P->Vals.PhaseIEnd.RecsSent);
    break;
  case NFEEventPhaseIIBegin :
    fprintf(fp, "exit called @ Load Operator phase II (start).\n") ;
    break;
  case NFEEventPhaseIIEnd :
    fprintf(fp, "exit called @ Load Operator phase II (end): %d records loaded.\n", P->Vals.PhaseIIEnd.Inserts);
    break;
  case NFEEventDropErrTableI :
    fprintf(fp, "exit called @ Load Operator ET 1 Drop : %d records in table.\n", P->Vals.DropErrTableI.Rows);
    break;
  case NFEEventDropErrTableII :
    fprintf(fp, "exit called @ Load Operator ET 2 Drop : %d records in table.\n", P->Vals.DropErrTableII.Rows);
    break;
  case NFEEventDBSRestart :
    fprintf(fp, "exit called @ Load Operator detects DBS restart.\n") ;
    break;
  case NFEEventCLIError :
    fprintf(fp, "exit called @ Load Operator detects CLI error: %d.\n", P->Vals.CLIError.ErrorCode);
    break;
  case NFEEventDBSError :
    fprintf(fp, "exit called @ Load Operator detects DBS error: %d.\n", P->Vals.DBSError.ErrorCode);
    break;
case NFEventExit:
    fprintf(fp,
            "exit called @ Load Operator exiting. Return code: %d\n",
            P->Vals.Exit.ReturnCode);
    fclose(fp);
    break;
return(0);
/
****************************************************************************
* End of ldnfyext.c
**************************************************************************/

Update Operator Example
**************************************************************************
* TITLE: updnfyxt.c .... an example notify exit...
* Purpose As a test and an example for users.
* COPYRIGHT (C) Teradata Corporation. 1999-2008
* Description stuff for notify.
* Revision    Date       DR    DID     Comments
* ----------- ---------- ----- ------- ------------------------------
* 08.01.00.00 03/01/2005 67848 XXX Notify Feature Implementation
* *
* Notes The entry point to this User Exit must
*        be called "dynmxn"
**************************************************************************
#include <stdio.h>
typedef int Int32;
typedef unsigned int UInt32;
typedef enum {
    NMEventInitialize = 0,
    NMEventFileInmodOpen = 1,
    NMEventPhaseIBegin = 2,
    NMEventCheckPoint = 3,
    NMEventPhaseIEnd = 4,
    NMEventPhaseIIBegin = 5,
    NMEventPhaseIIEnd = 6,
    NMEventErrorTableI = 7,
    NMEventErrorTableII = 8,
    NMEventDBSRestart = 9,
    NMEventCLIError = 10,
    NMEventDBSError = 11,
    NMEventExit = 12,
    NMEventAmpsDown = 21,
    NMEventImportBegin = 22,
    NMEventImportEnd = 23,
    NMEventDeleteInit = 24,
    NMEventDeleteBegin = 25,
    NMEventDeleteEnd = 26,
    NMEventDeleteExit = 27
} NfyUPDEvent;
/**************************************/
/* Structure for User Exit Interface  */
/**************************************/
#define NOTIFYID_LOAD          1
#define NOTIFYID_UPDATE        2
#define NOTIFYID_EXPORT        3
#define NOTIFYID_INSERT        4
#define NOTIFYID_STREAM        5
#define MAXVERSIONIDLEN       32
#define MAXUTILITYNAMELEN     32
#define MAXUSERNAMELEN        64
#define MAXUSERSTRLEN         80
#define MAXTABLENAMELEN      128
#define MAXFILENAMELEN       256

typedef struct _UPNotifyExitParm {
    Uint32 Event; /* should be NfyUPDEvent values */
    union {
        struct {
            Uint32 VersionLen;
            char   VersionId[MAXVERSIONIDLEN];
            Uint32 UtilityId;
            Uint32 UtilityNameLen;
            char   UtilityName[MAXUTILITYNAMELEN];
            Uint32 UserNameLen;
            char   UserName[MAXUSERNAMELEN];
            Uint32 UserStringLen;
            char   UserString[MAXUSERSTRLEN];
        } Initialize;
        struct {
            Uint32 FileNameLen;
            char   FileOrInmodName[MAXFILENAMELEN];
            Uint32 ImportNo;
        } FileInmodOpen ;
        struct {
            Uint32 TableNameLen;
            char   TableName[MAXTABLENAMELEN];
            Uint32 TableNo;
        } PhaseIBegin;
        struct {
            Uint32 RecordCount;
        } CheckPoint;
        struct {
            Uint32 RecsRead;
            Uint32 RecsSkipped;
            Uint32 RecsRejected;
            Uint32 RecsSent;
        } PhaseIEnd ;
        struct {
            Uint32 dummy;
        } PhaseIIBegin;
        struct {
            Uint32 Inserts;
            Uint32 Updates;
            Uint32 Deletes;
            Uint32 TableNo;
        } PhaseIIEnd;
};
struct {
    UInt32 Rows;
    UInt32 TableNo;
} ErrorTableI;
struct {
    UInt32 Rows;
    UInt32 TableNo;
} ErrorTableII;
struct {
    UInt32 dummy;
} DBSRestart;
struct {
    UInt32 ErrorCode;
} CLIError;
struct {
    UInt32 ErrorCode;
} DBSError;
struct {
    UInt32 ReturnCode;
} Exit;
struct {
    UInt32 dummy;
} AmpsDown;
struct {
    UInt32 ImportNo;
} ImportBegin;
struct {
    UInt32 RecsRead;
    UInt32 RecsSkipped;
    UInt32 RecsRejected;
    UInt32 RecsSent;
    UInt32 ImportNo;
} ImportEnd;
struct {
    UInt32 dummy;
} DeleteInit;
struct {
    UInt32 TableNameLen;
    char TableName[MAXTABLENAMELEN];
    UInt32 TableNo;
} DeleteBegin;
struct {
    UInt32 Deletes;
    UInt32 TableNo;
} DeleteEnd;
struct {
    UInt32 ReturnCode;
} DeleteExit;
} Vals;
} UPNotifyExitParm;
/***********************************************
/* CODE STARTS HERE*/
***********************************************/

#ifdef WIN32
__declspec(dllexport) Int32 _dynamn(UPNotifyExitParm *P)
#else
Int32 _dynamn(UPNotifyExitParm *P)
#endif
{
    static FILE *fp;
    if (!fp) {
#ifdef I370
        if (!(fp = fopen("NFYEXIT", "w")))
            return(1);
#else
        if (!(fp = fopen("NFYEXIT.OUT", "w")))
            return(1);
#endif
        switch(P->Event) {
            case NMEventInitialize :
                fprintf(fp, "exit called @ Update Operator Notify init.\n"");
                fprintf(fp, "              Version: %s\n", 
                    P->Vals.Initialize.VersionId);
                fprintf(fp, "              Utility: %s\n", 
                    P->Vals.Initialize.UtilityName);
                fprintf(fp, "              User: %s\n", 
                    P->Vals.Initialize.UserName);
                if (P->Vals.Initialize.UserStringLen)
                    fprintf(fp, "              UserString: %s\n", 
                    P->Vals.Initialize.UserString);
                break;
            case NMEventFileInmodOpen:
                fprintf(fp, "exit called @ File open: import[%d]: %s\n", 
                    P->Vals.FileInmodOpen.ImportNo,
                    P->Vals.FileInmodOpen.FileOrInmodName);
                break;
            case NMEventPhaseIBegin :
                fprintf(fp, "exit called @ Acquisition start: tablename[%d] : 
                    %s.\n", 
                    P->Vals.PhaseIBegin.TableNo,
                    P->Vals.PhaseIBegin.TableName);
                break;
            case NMEventCheckPoint :
                fprintf(fp, "exit called @ Checkpoint : %d records loaded.\n", 
                    P->Vals.CheckPoint.RecordCount);
                break;
            case NMEventPhaseIEnd :
                fprintf(fp, "exit called @ Acquisition end.\n"");
                fprintf(fp, "   Records Read:     %d\n", 
                    P->Vals.PhaseIEnd.RecsRead);
                fprintf(fp, "   Records Skipped:  %d\n", 
                    P->Vals.PhaseIEnd.RecsSkipped);
                fprintf(fp, "   Records Rejected: %d\n", 
                    P->Vals.PhaseIEnd.RecsRejected);
                fprintf(fp, "   Records Sent:     %d\n", 
                    P->Vals.PhaseIEnd.RecsSent);
                break;
        }
    }
#endif
case NMEventPhaseIIBegin :
    fprintf(fp, "exit called @ Application start\n");
    break;

case NMEventPhaseIIEnd :
    fprintf(fp, "exit called @ Application complete for table %d.\n",
            P->Vals.PhaseIIEnd.TableNo);
    fprintf(fp, " %d updates, %d inserts, %d deletes\n", 
            P->Vals.PhaseIIEnd.Updates,
            P->Vals.PhaseIIEnd-inserts,
            P->Vals.PhaseIIEnd.Deletes);
    break;

case NMEventErrorTableI :
    fprintf(fp, 
            "exit called @ ET Table[%d] Drop : %d records in table.\n",
            P->Vals.ErrorTableI.TableNo, P->Vals.ErrorTableI.Rows);
    break;

case NMEventErrorTableII :
    fprintf(fp, 
            "exit called @ UV Table[%d] Drop : %d records in table.\n",
            P->Vals.ErrorTableII.TableNo, P-
>Vals.ErrorTableII.Rows);
    break;

case NMEventDBSRestart :
    fprintf(fp, "exit called @ RDBMS restarted\n");
    break;

case NMEventCLIError :
    fprintf(fp, "exit called @ CLI error %d\n", 
            P->Vals.CLIError.ErrorCode);
    break;

case NMEventDBSError :
    fprintf(fp, "exit called @ DBS error %d\n", 
            P->Vals.DBSError.ErrorCode);
    break;

case NMEventExit :
    fprintf(fp, "exit called @ Notify out of scope: return code %d.\n",
            P->Vals.Exit.ReturnCode);
    fclose(fp);
    break;

case NMEventAmpsDown :
    fprintf(fp, "exit called @ Down amps have been detected\n");
    break;

case NMEventImportBegin :
    fprintf(fp, "exit called @ Import %d starting\n", 
            P->Vals.ImportBegin.ImportNo);
    break;
case NMEventImportEnd :
    fprintf(fp, "exit called @ Import %d ending.
", 
P->Vals.ImportEnd.ImportNo);
    fprintf(fp, " Records Read: %d\n", 
P->Vals.ImportEnd.RecsRead);  
    fprintf(fp, " Records Skipped: %d\n", 
P->Vals.ImportEnd.RecsSkipped);  
    fprintf(fp, " Records Rejected: %d\n", 
P->Vals.ImportEnd.RecsRejected);  
    fprintf(fp, " Records Sent: %d\n", 
P->Vals.ImportEnd.RecsSent);  
    break;

case NMEventDeleteInit : /* nothing */
    fprintf(fp, "exit called @ Delete task init.\n");
    break;

case NMEventDeleteBegin :
    fprintf(fp, "exit called @ Delete app start for table[%d]:
" ,
P->Vals.DeleteBegin.TableNo, P-
>Vals.DeleteBegin.TableName);
    break;

case NMEventDeleteEnd :
    fprintf(fp, "exit called @ Delete app done for table[%d]: %d
" ,
P->Vals.DeleteEnd.TableNo, P->Vals.DeleteEnd.Deletes);
    break;

case NMEventDeleteExit :
    fprintf(fp, "exit called @ Notify out of scope: return code
" ,
P->Vals.DeleteExit.ReturnCode);  
    fclose(fp);
    break;
return(0);
/
************************************************************************
* End of updnfyxt.c
************************************************************************/
NMEventImportBegin = 3,
NMEventImportEnd = 4,
NMEventErrorTable = 5,
NMEventDBSRestart = 6,
NMEventCLLError = 7,
NMEventDBSError = 8,
NMEventExit = 9,
NMEventTableStats = 10,
NMEventCkptEnd = 11,
NMEventRunStats = 12,
NMEventDMLError = 13
} NfySTMEvent;

/******************************************************/
/* Structure for User Exit Interface */
/******************************************************/
#define NOTIFYID_LOAD          1
#define NOTIFYID_UPDATE        2
#define NOTIFYID_EXPORT        3
#define NOTIFYID_INSERT        4
#define NOTIFYID_STREAM        5
#define MAXVERSIONIDLEN       32
#define MAXUTILITYNAMELEN     32
#define MAXUSERNAMELEN        64
#define MAXUSERSTRLEN         80
#define MAXTABLENAMELEN      128
#define MAXFILENAMELEN       256

/******************************************************/
/* Stream Operator Specific Constants */
/******************************************************/
#define STMDUPROW       9901
#define STMMISROW       9903
typedef enum {
    DEFeedbackDefault   = 0,
    DEFeedbackNoLogging = 1
} DMLErrorFeedbackType;
typedef struct _STNotifyExitParm {
    UInt32 Event; /* should be NfySTMEvent values */
    union {
        struct {
            UInt32 VersionLen;
            char   VersionId[MAXVERSIONIDLEN];
            UInt32 UtilityId;
            UInt32 UtilityNameLen;
            char   UtilityName[MAXUTILITYNAMELEN];
            UInt32 UserNameLen;
            char   UserName[MAXUSERNAMELEN];
            UInt32 UserStringLen;
            char   UserString[MAXUSERSTRLEN];
        } Initialize;
        struct {
            UInt32 FileNameLen;
            char   FileOrInmodName[MAXFILENAMELEN];
            UInt32 ImportNo;
        } FileInmodOpen;
        struct {
            UInt32 RecordCount;
        } CheckPoint;
    }
}
struct {
    char  *TableName;
    UInt32 Rows;
} ErrorTable;
struct {
    UInt32 dummy;
} DBSRestart;
struct {
    UInt32 errorCode;
} CLIError;
struct {
    UInt32 errorCode;
} DBSError;
struct {
    UInt32 returnCode;
} Exit;
struct {
    UInt32 ImportNo;
} ImportBegin;
struct {
    UInt32 ImportNo;
    UInt32 RecsRead;
    UInt32 RecsSkipped;
    UInt32 RecsRejected;
    UInt32 RecsSent;
    UInt32 RecsCausedDBSError;
} ImportEnd;
struct {
    char   StmtType;
    char  *DatabaseName;
    char  *TableOrMacroName;
    UInt32 ActivityCount;
} TableStats;
struct {
    UInt32 ImportNo;
    UInt32 SQLStmtSent;
    UInt32 RequestSent;
    UInt32 RecsRead;
    UInt32 RecsSkipped;
    UInt32 RecsRejected;
    UInt32 RecsSent;
    UInt32 RecsCausedDBSError;
} RunStats;
struct {
    UInt32 ImportNo;
    UInt32 ErrorCode;
    char   *ErrorMsg;
    UInt32 RecordNo;
    unsigned char InputSeqNo;
    unsigned char DMLSeqNo;
    unsigned char SMTSeqNo;
    char   *ErrorData;
    UInt32 ErrorDataLen;
    UInt32 *feedback;
} DMLError;
} Vals;
} STNotifyExitParm;
/*******************************************************************************/
* CODE STARTS HERE
*******************************************************************************/
#ifdef WIN32
__declspec(dllexport) Int32 _dynamn(STNotifyExitParm *P)
#else
Int32 _dynamn(STNotifyExitParm *P)
#endif
{
    int i;
    FILE *fp;
    if (!(fp = fopen("NFYEXIT.OUT", "a")))
        return(1);
    switch(P->Event) {
    case NMEventInitialize :
        fprintf(fp, "exit called @ Stream Operator Notify init.\n");
        fprintf(fp, "      Version: %s\n", P->Vals.Initialize.VersionId);
        fprintf(fp, "      Utility: %s\n", P->Vals.Initialize.UtilityName);
        fprintf(fp, "      User: %s\n", P->Vals.Initialize.UserName);
        if (P->Vals.Initialize.UserStringLen)
            fprintf(fp, "      UserString: %s\n", P->Vals.Initialize.UserString);
        break;
    case NMEventFileInmodOpen :
        fprintf(fp, "exit called @ File open: import[%u]: %s\n",
            P->Vals.FileInmodOpen.ImportNo, P->Vals.FileInmodOpen.FileOrInmodName);
        break;
    case NMEventCkptBeg :
        fprintf(fp, "exit called @ Checkpoint Begin: %u records loaded.\n",
            P->Vals.CheckPoint.RecordCount);
        break;
    case NMEventCkptEnd :
        fprintf(fp, "exit called @ Checkpoint End: %u records loaded.\n",
            P->Vals.CheckPoint.RecordCount);
        break;
    case NMEventErrorTable :
        fprintf(fp, "exit called @ Error Table : %s. %u record(s) in table.\n",
            P->Vals.ErrorTable.TableName, P->Vals.ErrorTable.Rows);
        break;
    case NMEventDBSRestart :
        fprintf(fp, "exit called @ RDBMS restarted\n");
        break;
    case NMEventCLIError :
        fprintf(fp, "exit called @ CLI error %u\n",
            P->Vals.CLIError.ErrorCode);
        break;
    case NMEventDBSError :
        fprintf(fp, "exit called @ DBS error %u\n",
            P->Vals.DBSError.ErrorCode);
        break;
    }
case NMEventExit:
    fprintf(fp, "exit called @ Notify out of scope: return code %u
", P->Vals.Exit.ReturnCode);
    fclose(fp);
    break;
  case NMEventImportBegin:
    fprintf(fp, "exit called @ Import %u starting
", P->Vals.ImportBegin.ImportNo);
    break;
  case NMEventImportEnd:
    fprintf(fp, "exit called @ Import %u ending.
", P->Vals.ImportEnd.ImportNo);
    fprintf(fp, "Records Read:               %u
", P->Vals.ImportEnd.RecsRead);
    fprintf(fp, "Records Skipped:            %u
", P->Vals.ImportEnd.RecsSkipped);
    fprintf(fp, "Records Rejected:           %u
", P->Vals.ImportEnd.RecsRejected);
    fprintf(fp, "Records Sent:               %u
", P->Vals.ImportEnd.RecsSent);
    fprintf(fp, "Records resulting in RDBMS errors %u
", P->Vals.ImportEnd.RecsCausedDBSError);
    break;
  case NMEventTableStats:
    fprintf(fp, "exit called @ Table Stats: 
");
    if(P->Vals.TableStats.StmtType == 'I') {
        fprintf(fp,"Rows Inserted :      
" "%u
" "Table/Macro Name :  %s
" "Database Name :     %s
", P->Vals.TableStats.ActivityCount,
        P->Vals.TableStats.TableOrMacroName,
        P->Vals.TableStats.DatabaseName);
    }
    if(P->Vals.TableStats.StmtType == 'U') {
        fprintf(fp,"Rows Updated :      
" "%u
" "Table/Macro Name :  %s
" "Database Name :     %s
", P->Vals.TableStats.ActivityCount,
        P->Vals.TableStats.TableOrMacroName,
        P->Vals.TableStats.DatabaseName);
    }
    if(P->Vals.TableStats.StmtType == 'D') {
        fprintf(fp,"Rows Deleted :      
" "%u
" "Table/Macro Name :  %s
" "Database Name :     %s
", P->Vals.TableStats.ActivityCount,
        P->Vals.TableStats.TableOrMacroName,
        P->Vals.TableStats.DatabaseName);
    }
    if(P->Vals.TableStats.StmtType == 'M') {
        fprintf(fp,"Rows Merged :      
" "%u
" "Table/Macro Name :  %s
" "Database Name :     %s
", P->Vals.TableStats.ActivityCount,
        P->Vals.TableStats.TableOrMacroName,
        P->Vals.TableStats.DatabaseName);
    }
    break;
case NMEventRunStats:
    fprintf(fp, "exit called @ Run States \n");
    fprintf(fp, "Import %u \n", P->Vals.RunStats.ImportNo);
    fprintf(fp, "Statement Sent : %u \n"
    "Request Sent : %u \n"
    "Records Read : %u \n"
    "Records Skipped : %u \n"
    "Records Rejected : %u \n"
    "Records Sent : %u \n"
    "Records resulting in RDBMS errors : %u \n",
    P->Vals.RunStats.SQLStmtSent,
    P->Vals.RunStats.RequestSent,
    P->Vals.RunStats.RecsRead,
    P->Vals.RunStats.RecsSkipped,
    P->Vals.RunStats.RecsRejected,
    P->Vals.RunStats.RecsSent,
    P->Vals.RunStats.RecsCausedDBSError);
    break;

case NMEventDMLError :
    fprintf(fp, "exit called @ DML Error \n");
    fprintf(fp, "Import %u \n", P->Vals.DMLError.ImportNo);
    fprintf(fp, "Error code: %u \n"
    "Error text: %s \n"
    "Record number: %u \n"
    "Input Sequence number: %d \n"
    "DML number: %d \n"
    "Statement number: %d \n"
    "Error data length : %u \n"
    "Received feedback : %u \n",
    P->Vals.DMLError.ErrorCode,
    P->Vals.DMLError.ErrorMsg,
    P->Vals.DMLError.RecordNo,
    P->Vals.DMLError.InputSeqNo,
    P->Vals.DMLError.DMLSeqNo,
    P->Vals.DMLError.SMTSeqNo,
    P->Vals.DMLError.ErrorDataLen,
    *(P->Vals.DMLError.feedback));

    fprintf(fp, "Error data: ");
    for (i=0 ; i<P->Vals.DMLError.ErrorDataLen; i++) {
        fprintf(fp, "%c", P->Vals.DMLError.ErrorData[i]);
    }
    fprintf(fp, "\n");
    if (P->Vals.DMLError.ErrorCode == STMDUPROW)
        *(P->Vals.DMLError.feedback) = DEFeedbackNoLogging;
    fprintf(fp, "STMDUPROW: feedback = %u \n", DEFeedbackNoLogging);
    if (P->Vals.DMLError.ErrorCode == STMMISROW)
        *(P->Vals.DMLError.feedback) = DEFeedbackNoLogging;
    fprintf(fp, "STMMISROW: feedback = %u \n", DEFeedbackNoLogging);
    break;
    default :
        fprintf(fp, "An Invalid Event Passed to the Exit Routine\n");
        break;
    }
    return(0);
}
**Export Operator Example**

******************************************************************************
* TITLE: expnfyxt.c .... an example notify exit...
* Purpose As a test and an example for users.
* COPYRIGHT (C) Teradata Corporation. 1999-2008
* Description Sample user exit routine for processing NOTIFY events
* History Information

* Revision Date DR DID Comments
* 02.04.00.01 04/02/2003 67849 XXX Initial implementation
* Notes The entry point to this User Exit must be called "_dynamn"
******************************************************************************

#include <stdio.h>
typedef int Int32;
typedef unsigned int UInt32;
#define NOTIFYID_LOAD          1
#define NOTIFYID_UPDATE        2
#define NOTIFYID_EXPORT        3
#define NOTIFYID_INSERT        4
#define NOTIFYID_STREAM        5
#define MAXVERSIONIDLEN       32
#define MAXUTILITYNAMELEN     32
#define MAXUSERNAMELEN        64
#define MAXUSERSTRLEN         80
#define MAXFILENAMELEN       256
#define MAXREQUESTLEN      32000

typedef enum {
    NXEventInitialize     = 0,
    NXEventFileInmodOpen  = 1,
    NXEventDBSRestart     = 9,
    NXEventCLIError       = 10,
    NXEventDBSError       = 11,
    NXEventexit           = 12,
    NXEventExportBegin    = 31,
    NXEventReqSubmitBegin = 32,
    NXEventReqSubmitEnd   = 33,
    NXEventReqFetchBegin  = 34,
    NXEventFileOutmodOpen = 35,
    NXEventStmtFetchBegin = 36,
    NXEventStmtFetchEnd   = 37,
    NXEventReqFetchEnd    = 38,
    NXEventExportEnd      = 39
} NfyExpEvent;

typedef struct _EXNotifyexitParm {
    UInt32 Event;
    union {
        struct {
            UInt32 VersionLen;
            char VersionId[MAXVERSIONIDLEN];
            UInt32 UtilityId;
            UInt32 UtilityNameLen;
            char UtilityName[MAXUTILITYNAMELEN];
            UInt32 UserNameLen;
            char UserName[MAXUSERNAMELEN];
            UInt32 UserStringLen;
            char UserString[MAXUSERSTRLEN];
        } Initialize;
    } Initialize;
};
struct {
    UInt32 FileNameLen;
    char FileOrInmodName[MAXFILENAMELEN];
    UInt32 dummy;
} FileInmodOpen;

struct {
    UInt32 dummy;
} DBSRestart;

struct {
    UInt32 ErrorCode;
} CLIError;

struct {
    UInt32 ErrorCode;
} DBSError;

struct {
    UInt32 ReturnCode;
} exit;

struct {
    UInt32 dummy;
} ExportBegin;

struct {
    UInt32 RequestLen;
    char Request[MAXREQUESTLEN];
} ReqSubmitBegin;

struct {
    UInt32 StatementCnt;
    UInt32 BlockCnt;
} ReqSubmitEnd;

struct {
    UInt32 dummy;
} ReqFetchBegin;

struct {
    UInt32 FileNameLen;
    char FileOrOutmodName[MAXFILENAMELEN];
} FileOutmodOpen;

struct {
    UInt32 StatementNo;
    UInt32 BlockCnt;
} StmtFetchBegin;

struct {
    UInt32 Records;
} StmtFetchEnd;

struct {
    UInt32 RecsExported;
    UInt32 RecsRejected;
} ReqFetchEnd;

struct {
    UInt32 RecsExported;
    UInt32 RecsRejected;
} ExportEnd;

} Vals;

} EXNotifyexitParm;
/***********************************************************************
*   CODE STARTS HERE
***********************************************************************
#ifdef WIN32
__declspec(dllexport) Int32 _dynamn(EXNotifyexitParm *P)
#else
Int32 _dynamn(EXNotifyexitParm *P)
#endif
{
    static FILE *fp;
    if (!fp) {
        #ifdef I370
        if (!(fp = fopen("NFYEXIT", "w")))
            return(1);
        #else
        if (!(fp = fopen("NFYEXIT.OUT", "w")))
            return(1);
        #endif
        switch(P->Event) {
        case NXEventInitialize :   /* Nothing */
            fprintf(fp, "exit called @ Export Operator Initialization.\n");
            fprintf(fp, "          Version: %s\n",
                P->Vals.Initialize.VersionId);
            fprintf(fp, "          Utility: %s\n",
                P->Vals.Initialize.UtilityName);
            fprintf(fp, "          User: %s\n",
                P->Vals.Initialize.UserName);
            if (P->Vals.Initialize.UserStringLen)
                fprintf(fp, "          UserString: %s\n",
                    P->Vals.Initialize.UserString);
            break;
        case NXEventFileInmodOpen:
            fprintf(fp, "exit called @ input file open: %s\n",
                P->Vals.FileInmodOpen.FileOrInmodName);
            break;
        case NXEventDBSRestart :
            fprintf(fp, "exit called @ RDBMS restart detected\n");
            break;
        case NXEventCLIError :
            fprintf(fp, "exit called @ CLI error %d\n",
                P->Vals.CLIError.ErrorCode);
            break;
        case NXEventDBSError :
            fprintf(fp, "exit called @ DBS error %d\n",
                P->Vals.DBSError.ErrorCode);
            break;
        case NXEventexit :
            fprintf(fp,
                "exit called @ Export Operator exiting: return code %d.\n",
                P->Vals.exit.ReturnCode);
            fclose(fp);
            break;
        case NXEventExportBegin :
            fprintf(fp, "exit called @ Export beginning.\n");
            break;
        case NXEventReqSubmitBegin :
            fprintf(fp, "exit called @ request submitted: '%s'.\n",
                P->Vals.ReqSubmitBegin.Request);
            break;
        }
    }
    #endif

case NXEventReqSubmitEnd :
    fprintf(fp, "exit called @ request done: %d statement(s), \n %d blocks.\n",
        P->Vals.ReqSubmitEnd.StatementCnt,
        P->Vals.ReqSubmitEnd.BlockCnt);
    break;
case NXEventReqFetchBegin :
    fprintf(fp, "exit called @ request fetch beginning.\n");
    break;
case NXEventFileOutmodOpen:
    fprintf(fp, "exit called @ output file open: %s\n",
        P->Vals.FileOutmodOpen.FileOrOutmodName);
    break;
case NXEventStmtFetchBegin :
    fprintf(fp, "exit called @ statement fetch beginning: stmt #\d, \n %d blocks.\n",
        P->Vals.StmtFetchBegin.StatementNo,
        P->Vals.StmtFetchBegin.BlockCnt);
    break;
case NXEventStmtFetchEnd :
    fprintf(fp, "exit called @ statement fetch end: %d records.\n",
        P->Vals.StmtFetchEnd.Records);
    break;
case NXEventReqFetchEnd :
    fprintf(fp, "exit called @ request fetch ends: \ Records exported: %d, Records rejected: %d\n",
        P->Vals.ReqFetchEnd.RecsExported,
        P->Vals.ReqFetchEnd.RecsRejected);
    break;
case NXEventExportEnd :
    fprintf(fp, "exit called @ Export ends: \ Records exported: %d, Records rejected: %d\n",
        P->Vals.ExportEnd.RecsExported,
        P->Vals.ExportEnd.RecsRejected);
    break;
return(0);
}
CHAPTER 26

Advanced Teradata Database Considerations

This chapter describes some advanced Teradata Database considerations. Topics include:

- Query Banding Considerations
- Large Objects
- Data Conversions

Query Banding Considerations

Available through the Load, Update, Stream, Export, SQL Selector, SQL Inserter and DDL operators, use the `QueryBandSessInfo` attribute to set the query banding feature of the Teradata Database.

The `QueryBandSessInfo` attribute may be specified as an ARRAY attribute so that operators for which the `QueryBandSessInfo` attribute is available can send a separate SET QUERY_BAND request to the Teradata Database for each entry of query band data in the array attribute.

See Teradata Parallel Transporter Reference for more details on default values and setting the Query Band for the duration of a job session.

There are two interaction items to consider for systems that are using Teradata Dynamic Workload Manager (Teradata DWM) and Teradata PT:

- Teradata Dynamic Workload Manager classifies the first utility-specific command in for a Teradata PT operator. Classification is based on query bands if the `QueryBandSessInfo` attribute contains one or more values.

- When the `QueryBandSessInfo` attribute contains one or more values, the system sets a flag in the session context so that all subsequent operator commands are not classified. This ensures that all work for the operator runs at the same priority and workload definition.

- When the Delay option is specified, the TENACITY/TenacityHours and SLEEP/TenacitySleep attributes become ineffective for that job because Teradata DWM will automatically delay a logon until it is eligible to run.
Large Objects

Teradata PT supports loading Large Object (LOB) data into, and extracting LOB data from, Teradata Database tables.

Large Objects are data types that can be Binary Large Objects (BLOBs) or Character Large Objects (CLOBs).

Depending on how the LOB columns are defined in the Teradata PT schema, Teradata PT uses one of the following two methods:

- Inline
- Deferred

to load LOB data into, or extract LOB data from, Teradata Database.

Defining LOB Data in a Teradata PT Schema

The syntax for defining LOB data, either as BLOB or CLOB, in a Teradata PT schema is shown below:

- BLOB (lengthBytes) or CLOB (lengthBytes)
  This syntax indicates that the Teradata PT load or extract job will use the inline method to transfer the LOB data between the Teradata PT and Teradata Database.
- BLOB (lengthBytes) AS DEFERRED BY NAME or CLOB (lengthBytes) AS DEFERRED BY NAME
  This syntax indicates that the LOB data is transferred using the deferred method.

When LOB columns are defined as DEFERRED BY NAME in the schema, Teradata PT expects regular VARCHARs in place of deferred LOB columns in a data row. Each VARCHAR is a file name of a flat file that contains the actual data for a deferred LOB column. Teradata PT refers to these VARCHARs as LOB file locators.

Inline Method

In the inline method, data rows sent across the Teradata PT data stream must contain both non-LOB and LOB data. The inline BLOB and CLOB columns in the data rows are treated as similar as the VARBYTE and VARCHAR columns.

However, these BLOB and CLOB columns must have an 8-byte integer as their length indicator field instead of 2-byte length indicator as in the VARBYTE and VARCHAR columns.

One restriction in using the inline method to transfer LOB data is that the entire data row should not be larger than 64,000 bytes since that is the row-size limit imposed by the Teradata PT data stream.

Deferred Method

In the deferred method, the Teradata PT does not send the actual LOB data across its data stream for LOB columns defined as DEFERRED BY NAME.
Deferred LOB data is processed entirely separately from the non-LOB data. Data rows sent across the Teradata PT data stream contain only non-LOB data and LOB file locators. The locators point to the flat files containing actual data for the deferred LOB columns.

**Moving LOB Data from one Teradata Database Table to Another**

To move LOB data from one Teradata Database table to another, a Teradata PT script must use:

- The SQL Selector operator to extract LOB data from a source table, and
- The SQL Inserter operator to load the data to a target table.

**Using the Inline Method**

If you want to use the inline method for the job, you must define the LOB columns as BLOB (lengthBytes) or CLOB (lengthBytes), according to their data content, in the schema.

**Using the Deferred Method**

If you want to use the deferred method for the job, you must define the LOB columns as BLOB (lengthBytes) AS DEFERRED BY NAME or CLOB (lengthBytes) AS DEFERRED BY NAME, according to their data content, in the schema.

You should be aware of the following events during the job run:

- The SQL Selector operator retrieves data from deferred LOB columns and writes the data to external data files. These data files reside temporarily under the directory specified in LobDirectoryPath attribute or under the current working directory if the LobDirectoryPath attribute is not used.
- One data file represents data for one LOB column.
- These LOB data files have formulaic names as follows:
  
  `<file-basename>_<column-name>_<job-id>_p<#>_r<#>.<file-extension>`

  where

  - `<file-basename>` is the value specified in the LobFileBaseName attribute.
  - `<column-name>` is the column name defined in a source table.
  - `<job-id>` is the job name from the tbuild command line.
  - `<#>` is an integer number generated internally by Teradata PT
  - `<file-extension>` is the value specified in the LobFileExtension attribute.

  These LOB data files will be deleted immediately after their data has been inserted into the target table.

For a sample Teradata PT script, see “Script Example 18: Exporting BLOB/CLOB Data from One Teradata Database Table and Loading It into Another” on page 447.
Loading LOB Data into a Teradata Database Table from External Data Files

To load LOB data into a Teradata Database table from external data files, the Teradata PT script must use:

- The Data Connector producer operator to read LOB data from an external data file, and
- The Inserter operator to insert LOB data to a target table.

If the LOB columns are defined as BLOB (\textit{lengthBytes}) or CLOB (\textit{lengthBytes}) in the schema, data records in the external data file specified in the FileName attribute must have an 8-byte integer length indicator preceding the BLOB or CLOB data.

If the LOB columns are defined as BLOB (\textit{lengthBytes}) AS DEFERRED BY NAME or CLOB (\textit{lengthBytes}) AS DEFERRED BE NAME, data records in the external file specified in FileName attribute must have regular VARCHARS, known as LOB file locators, where the LOB columns are defined as deferred.

For a sample Teradata PT script, see “Script Example 3: Loading BLOB and CLOB Data into Teradata Database” on page 414.

Extracting LOB Data from Teradata Database Tables and Writing LOB Data to an External Target

To extract LOB data from Teradata Database table and write LOB data to an external target, the Teradata PT script must use:

- The SQL Selector operator to extract LOB data from a source table, and
- The Data Connector operator to write it an external target.

The LOB columns must be defined accordingly to their data content as BLOB (\textit{lengthBytes}) AS DEFERRED BY NAME or CLOB (\textit{lengthBytes}) AS DEFERRED BE NAME in the schema.

The job will produce the following:

- LOB data files located under the directory specified in the LobDirectoryPath attribute or under the current working directory if there is no value specified for the LobDirectoryPath attribute. For the file name formula, see “Using the Deferred Method” on page 367.
- An external data file whose name is specified in the FileName attribute. Data records written to the file contain non-LOB data and LOB file locators that identify the files containing the LOB data. These LOB file locators have the same format as VARCHAR columns and contain the file names of the LOB data files produced by the SQL Selector operator.

For a sample Teradata PT script, see “Script Example 14: Extract BLOB/CLOB Data and Write It to an External Target” on page 440.

User Considerations

Users should consider the follow:

- If possible, it is recommended to use the inline method to transfer LOB data between Teradata PT and Teradata Database because this method gives better performance, especially when multiple instances of the Inserter operator are used in the loading job.
However, there is a drawback in using the inline method, which is that the entire data row cannot be larger than 64000 bytes, which is the current row-size limit imposed by the Teradata PT data stream implementation.

- When a massive amount of data (may be 1, 000, 000 rows or more) is loaded in a Teradata Database table and the data row size is not larger than 64 000 bytes, the user may consider loading BLOBs as VARBYTEs and CLOBs as VARCHARs. Teradata Database can perform the trivial data conversion of VARBYTE to BLOB and VARCHAR to CLOB. That allows Teradata PT to use the Export operator to extract LOB data (defined to it as VARBYTE or VARCHAR columns) and the Load or Update operator to load LOB data (defined to it as VARBYTE or VARCHAR columns) at high speed.

- If you want to load BLOB or CLOB data into a table as VARBYTE or VARCHAR, the following needs to be done in a TPT job script:
  - BLOB columns must be defined as VARBYTE.
  - CLOB columns must be defined as VARCHAR.
  - Use the CAST function in the SQL SELECT statement specified in the SelectStmt attribute to explicitly convert BLOB to VARBYTE and CLOB to VARCHAR if data is extracted from a Teradata table.

- When the SQL Selector operator extracts deferred LOB data from a Teradata Database table and the SQL Inserter loads it into another Teradata Database table, the LOB data files associated with each row are deleted after the row has been inserted successfully. By the end of the loading job, all LOB data files that the SQL Selector operator created are deleted.

- When a Teradata PT job extracts LOB data from a Teradata Database table using the deferred method and writes it to an external target, the LOB data files associated with each record written to the external target will not be deleted at the end of the job.

- For a job loading LOB data into Teradata Database table from an external file (non-Teradata source), a user can define both inline and deferred LOB data types in the same schema. The job script uses the Data Connector operator as producer and Inserter operator as consumer. Thus, the following schema is valid for the job:

```sql
col1  CLOB(2000),
col2  BLOB(2000),
col3  CLOB(75000) AS DEFERRED BY NAME,
col4  BLOB(75000) AS DEFERRED BY NAME
```

**Note:** If the job script reads LOB data from a Teradata table, the above schema is not valid. A schema for this extraction scenario cannot contain both inline and deferred LOB columns.

**Limitations on LOB Processing**

Users should be aware of the following:

- Teradata PT does not support the use of multiple APPLY when deferred LOB data types are defined in a job schema and the LOB data is moved from one Teradata Database table to one or more target table simultaneously.

- Teradata PT does not support writing LOB data to a z/OS data set defined by a JCL DD statement. This means that the LOB exporting feature can only work with HFS files on
z/OS platforms. Users cannot specify a z/OS PDS name for the location where the LOB data files are written or for the file names.

- Teradata PT does not allow a LOB column to be used in any predicate evaluation specified in the WHERE clause or in either type of CASE expression.
- If LOB data sent to Teradata Database is larger than the size of the LOB column defined in the schema of the target table, Teradata Database truncates the data. This behavior is consistent with the behavior of VARCHAR, VARBYTE or VARGRAPHIC column types. But Teradata PT checks the size of a LOB data file against its schema to prevent Teradata Database from truncating the data with no user warning.

Teradata PT checks if the LOB data file is a flat file. If it is, The SQL Inserter operator compares the file size with the LOB column size defined in the Teradata PT schema. If the file size is bigger, then the job is terminated with a detailed error message.

If the LOB data source is not a file (it can be a named pipe, for instance), the SQL Inserter operator cannot make the file size check, and thus, if the data size exceeds the defined column size, Teradata Database truncates the data without any warning message.

Data Conversions

Teradata PT’s ability to convert data is limited to assigning a data type to a null value or changing data from a null value. For example:

```sql
CAST (NULL AS INTEGER)
```

Using the CAST clause, you can convert to an alternate data type prior to loading data into a table. The following APPLY statement illustrates this option.

```sql
APPLY
('INSERT INTO CUSTOMER (:CUST_NUM, :LAST_NAME, :FIRST_NAME, :SOC_SEC_NO);')
TO OPERATOR (LOAD_OPERATOR [1] )
SELECT * FROM OPERATOR
(EXPORT_OPERATOR [1]...)
```

Here, the use of SELECT * implies data is accepted as is from the Export operator; however, the data can also be converted.

If data is needed in a different table, create the following APPLY statement:

```sql
APPLY
('INSERT INTO CUSTOMER (:CUST_NUM, :LAST_NAME, :FIRST_NAME, :SOC_SEC_NO);')
TO OPERATOR (LOAD_OPERATOR [1] )
SELECT
  CAST (NULL AS CHAR(10)) AS CUST_NUM,
  LAST_NAME,
  FIRST_NAME,
  CASE
    WHEN (SOC_SEC_NO <> '000000000')
      THEN SOC_SEC_NO
    ELSE NULL
  END AS SOC_SEC_NO,
```
FROM OPERATOR
(EXPORT_OPERATOR [1]...)

Notice that:

- This example assumes that the schema of the source data is not the same as the schema of the target table.
- The first field is a derived column with a NULL value.
- The target Social Security number is assigned the NULL value if the source Social Security number is a string of all zero characters.
- This use of the CASE statement is comparable to the NULLIF function of the FastLoad utility.
- The functionality provided by the CASE expression is available to all Teradata PT operators because expressions are allowed in the APPLY statement.
Advanced Scripting Strategies

This chapter describes advanced techniques for use in Teradata PT job scripts.

Topics include:

- Data Acquisition and Loading Options
- Data Filtering and Conditioning Options
- Reusing Definitions with the INCLUDE Directive
- Using the Multiple APPLY Feature

Data Acquisition and Loading Options

The following data acquisition and loading options are available to augment basic scripting techniques, in order to facilitate the handling of more complex job requirements.

**UNION ALL: Combining Data from Multiple Sources**

UNION ALL takes data from two or more sources, obtained by different producer operators working in parallel, and combines the output data rows into a single logical data stream that is applied to the desired data target(s). Since all of the operators involved in a UNION ALL operation are working in parallel, the time to acquire source data is significantly reduced.

The following producer operators are typically used with UNION ALL:

- DataConnector
- ODBC

**Usage Requirements**

To be compatible with UNION ALL one of the following must be true:

- The rows put onto the output data streams by all UNION ALL producer operators must be identical in column structure. This is the case if the producer schemas are all UNION-compatible (see “Using Multiple Source Schemas” on page 49).

  or,

- A similar result can be achieved through column selection and/or the use of derived columns in the SELECT clauses that are combined with UNION ALL an the APPLY statement, even if all producer schemas are not UNION-compatible.
Code Example

```sql
SELECT * FROM OPERATOR (REGION_1_ACCOUNTS_READER)
UNION ALL
SELECT * FROM OPERATOR (REGION_2_ACCOUNTS_READER)
```

UNION ALL is used in a number of common job scenarios. For a typical application, see Example 1C in "Job Example 1: High Speed Bulk Loading into an Empty Table" on page 107.

For a complete example job script, see “Script Example 1C: High Speed Bulk Loading from Two Flat Files into an Empty Teradata Database Table Using UNION ALL” on page 408.

Intermediate File Logging

An Intermediate File Logging job reads transactional data from MQ or JMS and performs continuous INSERT, UPDATE, and DELETE operations in a Teradata Database table, while simultaneously loading a duplicate data stream into an external flat file that can serve as an archive or backup file of the data that has been loaded.

Strategy

Intermediate File Logging requires use of multiple APPLY clauses, one for the operator writing to Teradata Database and one for the operator writing to the external flat file.

The DataConnector operator is used twice in the job script:

- A DataConnector producer operator reads data from a transactional data source, either the JMS or MQ access module.

- A DataConnector consumer operator receives the data stream (a duplicate of what is being written to Teradata Database) from the DataConnector producer and writes it to an external flat file.

Note that the two DataConnector operator definitions differ in content, in addition to the common required attributes:

- The producer version requires specification of the following:
  - Use the AccessModuleName and AccessModuleInitStr attributes in order to interface with the access module providing the transactional data.
  - Set the OpenMode attribute to ‘read.’

- The consumer version requires specification of the following:
  - Use the DirectoryPath attribute to specify the destination directory.
  - Set the OpenMode attribute to ‘write.’

For a complete list of key DataConnector operator attributes, see “Chapter 8 DataConnector Operator” on page 151.

For a typical application of Intermediate File Logging, see Example 5C in “Job Example 5: Continuous Loading of Transactional Data from JMS or MQ” on page 116.

For a complete example job script, see “Script example 5C: Intermediate File Logging Using Multiple APPLY Clauses with Continuous Loading of Transactional Data” on page 423.
Mini-Batch Loading

Mini-Batch Loading reads data directly from one or more external flat files and writes it to a Teradata Database table. Use this job type when the destination table is already populated, or has join indexes or other restrictions that prevent it from being accessed by the Load operator.

**Strategy**

To circumvent the restrictions placed on use of the load operator by conditions in the target table, the job includes an intermediate step that temporarily loads the data into a staging table and then uses the DDL operator with INSERT...SELECT to move the data into the final destination table.

A mini-batch job requires three steps:

1. The DDL operator sets up the staging table using the CREATE TABLE statement specified in the APPLY statement.
2. The DataConnector reads the data from the flat files and the Load operator executes a high-speed load of the data into the staging table.
3. The DDL operator is used again to insert rows from the staging table into the target table using a different DML statement, this time an INSERT...SELECT, in the APPLY statement.

For a complete example job script, see “Script Example 7: Mini-Batch Loading into Teradata Database Tables” on page 427.

Batch Directory Scan

Batch Directory Scan uses multiple DataConnector operator instances to scan an external directory of flat files, searching for files that match the wildcard specification in the FileName attribute.

When the scan is complete, DataConnector places the data in the data stream for use by the consumer operator in the next job step. No further scanning is done, and any data added to the flat files after the scan will not be picked up until the next time the job is run.

**Strategy**

Use the following strategy when setting up the Batch directory scan:

- Specify the name of the directory to be scanned using the DataConnector operator DirectoryPath attribute.
- Use the wildcard character ( * ) for the FileName attribute, as follows:
  - Specify “*” to instruct the DataConnector operator to scan and load all files in the directory.
  - Specify “abc.*” to instruct the DataConnector operator to scan for all files in the directory having file names that begin with the specified character string.
- Use the ArchiveDirectoryPath attribute to specify an archive directory. When the scan is complete for a particular batch job, the scanned files will be moved to the archive.
directory. This prevents the build-up of old data in the “scanning” directory and prevents the job from seeing the old data the next time it runs.

- No limit exists to the number of files that can be used as input while appearing as a single source to Teradata PT. Multiple instances of the operator can be specified to speed the data acquisition process.

For a complete example job script, see “Script Example 8: Batch Directory Scan” on page 431.

**Active Directory Scan: Continuous Loading of Transactional Data**

Many companies monitor and store thousands—or hundreds of thousands—of transactions per day. Transactional data is collected and stored in client directories. You can use the "active directory scan" feature to continuously collect data from these directories based on a user-defined time interval for scanning the directory, and a start and stop time for the whole scan job, using the Data Connector operator.

All files present in the source directories that meet the user-specified file name criteria (which include "wildcard" specifications) are processed by the Data Connector operator. Whenever the defined scan interval expires, the Data Connector operator scans the directory and looks for new files that have entered the directory since the last scan. It then reads the rows from each of the files collected and sends them to the consumer operator, which is usually the Stream operator, for purposes of continuous loading. If no new files are found during the directory scan, the Data Connector operator waits for the defined interval to expire before scanning the directory again.

**Strategy**

Consider the following when setting up a job for Active Directory Scan:

- Specify the attribute names and values for the standard attributes required for the DataConnector operator; FileName, Format, IndicatorMode (where required), and TextDelimiter (required if format is “delimited”).

  For information on use of these standard attributes, see the section on the DataConnector operator in *Teradata Parallel Transporter Reference*.

- Use the wildcard character ( * ) for the FileName attribute according to one of the following strategies:
  - Specify “*” to instruct the DataConnector operator to scan and extract data from all files in the directory.
  - Specify “abc.*” to instruct the DataConnector operator to scan for all files in the directory having file names that begin with the specified character string.

- Specify the directory to be scanned using the DirectoryPath attribute, in the form:

  DirectoryPath=<PathName>

- Use the ArchiveDirectoryPath attribute to specify the path for the archive directory. Once files in the directory have been scanned and their data has been extracted, this specification will cause the files to be moved from the directory identified in the DirectoryPath attribute to that specified in ArchiveDirectoryPath attribute, in order to keep the files from being scanned again.
• Use the DataConnector Vigil attributes to set up the time constraints for the directory scan, as follows:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Setup Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>VigilStartTime</td>
<td>Required to specify the start time for the initial directory scan.</td>
</tr>
<tr>
<td>VigilStopTime</td>
<td>Specifies the time after which no more scans will begin. Any scan that begins before the stop time will run to completion. This attribute is interchangeable with the VigilElapsedTime attribute. Using one of these two attributes is required.</td>
</tr>
<tr>
<td>VigilWaitTime</td>
<td>Specifies the time in seconds between the beginning of one scan and the beginning of the next scan.</td>
</tr>
<tr>
<td>VigilElapsedTime</td>
<td>Specifies the total time in minutes the job will scan the directory for new files in intervals defined by VigilWaitTime. Any scan that starts before the end of the specified elapsed time will run to completion.</td>
</tr>
</tbody>
</table>

For required syntax and detailed descriptions for all DataConnector attributes, see Teradata Parallel Transporter Reference.

**Active Directory Scan Options**

The following options are available to further customize an Active Directory Scan.

• Use several DataConnectors operating in parallel to monitor multiple data sources.
• Use multiple instances of Stream operator to INSERT data into a Teradata Database table at an optimal rate.
• Important optional attributes:
  • Specify the VigilSortFile attribute and set it to TIME to sort files according to the time they were last modified.
  • Specify the VigilNoticeFileName attribute with a file name, so that when the scan file is updated with new data, a notification will be placed in that file.
  • Specify VigilMaxFiles to define the maximum number of files that can be scanned in one pass.
• Multiple schemas:
  When the data from the sources are not all described by UNION-compatible schemas, use column selection and/or derived columns in the Select clauses in the APPLY statement to put UNION-compatible data on the output data streams.

For a typical application of Active Directory Scan, see “Job Example 9: Active Directory Scan” on page 124.

For a complete example job script, see “Script Example 9: Active Directory Scan” on page 432.
Data Filtering and Conditioning Options

The following data filtering and conditioning options can be specified in the SELECT clause of an APPLY statement to filter or condition the data prior to loading.

**Simple Data Conditioning in the SELECT Statement**

Teradata PT scripts can accomplish a variety of simple data conditioning tasks. Conditioning tasks that do not require a filter operator are specified in the executable section of the job script.

Table 35 contains some examples of data conditioning using SELECT and the syntax for each.

Table 35: Data Conditioning Syntax Using SELECT

<table>
<thead>
<tr>
<th>Filtering Task</th>
<th>Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename a column from “price” to “original price”</td>
<td>SELECT price AS original_price</td>
</tr>
<tr>
<td>Assign new values to a column based on the value of</td>
<td>SELECT price*0.8 AS discounted_price</td>
</tr>
<tr>
<td>other columns, literals, and arbitrary expressions,</td>
<td></td>
</tr>
<tr>
<td>such as calculating a discounted price of 20% off the</td>
<td></td>
</tr>
<tr>
<td>base price.</td>
<td></td>
</tr>
<tr>
<td>Assign NULL values to a column.</td>
<td>SELECT NULL(VARCHAR(n)) AS product_name</td>
</tr>
<tr>
<td></td>
<td>where n is the size of the VARCHAR column as defined in the DEFINE SCHEMA</td>
</tr>
<tr>
<td></td>
<td>statement.</td>
</tr>
<tr>
<td>Concatenate columns using the concatenation operator</td>
<td>SELECT AREA_CODE</td>
</tr>
<tr>
<td>(</td>
<td></td>
</tr>
<tr>
<td>telephone number into the same column to create a</td>
<td></td>
</tr>
<tr>
<td>customer number</td>
<td></td>
</tr>
<tr>
<td>Load a value into a column that does not exist in the</td>
<td>SELECT '123' as JOB_ID</td>
</tr>
<tr>
<td>source, such as putting the value “123” in a column</td>
<td></td>
</tr>
<tr>
<td>called “JOB_ID”</td>
<td></td>
</tr>
<tr>
<td>Assign different values to a column in an output row</td>
<td>APPLY 'INSERT INTO SALES_TABLE (original_price, discounted_price,</td>
</tr>
<tr>
<td>based on conditions on columns in the corresponding</td>
<td>product_name) TO OPERATOR (UPDATE_OPERATOR [4])</td>
</tr>
<tr>
<td>input row, using a CASE value expression.</td>
<td>SELECT price AS original_price, price*0.8 AS discounted_price,</td>
</tr>
<tr>
<td></td>
<td>CASE WHEN product_name = '' THEN</td>
</tr>
<tr>
<td></td>
<td>NULL(VARCHAR) ELSE product_name</td>
</tr>
<tr>
<td></td>
<td>END AS product_name FROM OPERATOR (DATA_CONNECTOR [2])</td>
</tr>
</tbody>
</table>

For command syntax details, see the section on “APPLY” in *Teradata Parallel Transporter Reference*. 
CASE DML Expressions

CASE DML expressions allow Teradata PT jobs to require that each source row satisfy one of a number of conditions before being applied to any data targets, such that these conditions control which groups of DML statements are applied to the target table.

The following example shows typical CASE DML expressions structure:

```
CASE WHEN <condition 1> THEN <DML expression 1>
    WHEN <condition 2> THEN <DML expression 2>
    :        :         :           :
    WHEN <condition n> THEN <DML expression n>
    ELSE <DML expression n+1>
END
```

The conditions in a CASE DML expression are evaluated one by one from left to right; the first condition that is met for a given row causes the Teradata PT to apply the DML statement(s) in the corresponding DML expression to the row. The DML statements in the optional ELSE's DML expression will be applied by default, if none of the conditions are met.

The conditions can be simple predicates that reference column values in the source row, or they can be arbitrarily complex predicates that consist of simple predicates joined by logical ANDs and ORs. Any value in an expression can be specified as a CASE value expression.

The following is a typical use for the CASE DML expression:

**CASE DML Expression Example**

```c
CASE WHEN ( Expected_Arrival_Time = Scheduled_Arrival_Time )
    THEN 'UPDATE Flight_Status_Board
            SET Flight_Status   = ''On Time'',
                Gate_Number     = :Scheduled_Gate_Number,
                Carousel_Number = :Scheduled_Carousel_Number;'
    WHEN ( Expected_Arrival_Time > Scheduled_Arrival_Time )
    THEN ('UPDATE Flight_Status_Board
            SET Flight_Status = ''Delayed'',
                Arrival_Time  = :Expected_Arrival_Time,'
            'INSERT INTO LAX.AIRPORT_OPERATIONS(:Flight_Number,
                                                   ''Seat of the Pants Airlines'','
                                                   :Passenger_Count);')
    WHEN ( Expected_Arrival_Time = 0 )
    THEN ('UPDATE Flight_Status_Board
            SET Flight_Status = ''Cancelled'',
                Gate_Number     = NULL,
                Carousel_Number = NULL;',
            'DELETE FROM Pending_Arrivals
            WHERE Flight_Number = :Flight_Number
            AND Airline = ''Seat of the Pants'';')
ELSE 'UPDATE Flight_Status_Board
            SET Flight_Status = ''Early'',
                Arrival_Time  = :Expected_Arrival_Time;'
END
```
Case Value Expression

CASE value expressions allow derived column values in a target row to vary depending on which condition is satisfied by the corresponding source row. The CASE value expression has the same structure as the CASE DML expression, except that it associates a numeric value expression or string value expression with each condition rather than a DML group, as follows:

```
CASE  WHEN <condition 1> THEN <value expression 1>
WHEN <condition 2> THEN <value expression 2>
    :        :         :            :
WHEN <condition n> THEN <value expression n>
ELSE <value expression n+1>
END
```

The value of a CASE value expression is the value of the expression corresponding to the first condition that is met, else the value of the ELSE’s expression, if present, else NULL. The value expressions must all evaluate to data values of the same basic type, either all numeric or all string.

**CASE Value Expression Example**

```
SELECT COL1
CASE WHEN COL2 < 256    THEN COL4 * 16
     WHEN COL2 > 32767  THEN COL4 + COL5
     ELSE COL6
END AS COL2,
COL3 FROM...
```

Using the WHERE Clause

Use the WHERE clause in an APPLY statement to filter out source rows.

The WHERE clause is an optional part of the SELECT clause in an APPLY statement. It acts as a filter, determining which of the rows put into the data stream by the producer operator(s) will be kept in the data stream and delivered to the consumer operator(s). The decision whether or not to keep each row is based on condition(s) specified in the WHERE clause.

The conditional expression in a WHERE clause consists of simple predicates combined with logical ANDs and ORs. Any value in a predicate can be specified by a CASE value expression.

The following is an of a WHERE clause:

```
WHERE ( Years_In_Business > 5 AND Gross_Sales > 10000000 )
    OR Company_Name = 'Excellent Software, Inc.'
```

Reusing Definitions with the INCLUDE Directive

Teradata PT allows the inclusion of object definitions from external files into a job script, and thus the use of these definitions in more than one script. For example, you can define a schema in a text file called `customer.schema`:

```
DEFINE SCHEMA
```
Using the Multiple APPLY Feature

Using multiple APPLY clauses (operations within an APPLY statement), it is possible to extract data and simultaneously load it into as many as 32 targets in a single job step.

This read-once-apply-to-many approach allows source data to be loaded into multiple targets, ensuring that each target receives identical data. By using multiple APPLY clauses, multiple updates and loads can now be accomplished with fewer system resources compared to creating separate job steps for each load or update, thereby redundantly extracting data from a data source.

Scenarios

The following scenarios are examples of situations that benefit from using multiple targets.

- **Simultaneous loading of multiple warehouse targets** - Multiple warehouse targets can be loaded with a single input data source by using multiple APPLY, and the loading of each target can be done in a parallel, scalable manner by using multiple operator instances. The benefit of this method is that if a failure occurs, all load operations terminate, then restart in an orderly, coordinated manner.

  The use of multiple APPLY and multiple operator instances allows input data to be read and processed once, which minimizes I/O and system resource usage. Besides homogeneous loading, multiple kinds of consumer operators can also be used simultaneously. For example, warehouse A can be loaded using the Update operator while warehouse B is loaded using the Stream operator, and so on.

  This method also allows the use of the CASE DML expression in each APPLY clause so data that is applied to each of the targets can be handled by different CASE DML expressions.
Simultaneous loading and archiving (Intermediate file logging) - Maintaining archives that accurately reflect loaded data can be problematic when data is transformed between source and target, with only the transformed data being written to the target. Redundant extractions and redundant transformations are time-consuming and difficult. With the ability of Teradata PT to load multiple data targets, transformed data can simply be loaded into both the primary target and an archive in a single job step. For details, see “Intermediate File Logging” on page 374.

**Procedure**

Use this procedure to implement multiple data targets in an APPLY statement in the executable section of a script (after the DEFINE statements).

**To send data to multiple targets**

In all of the following syntax examples, `<DML spec x>` represents the DML statements to be applied to data target x. For more information, see “APPLY” in the Teradata Parallel Transporter Reference.

To send data to multiple targets, do the following:

1. Define an APPLY clause for the first target, specifying its consumer operator:
   
   ```
   APPLY <DML spec> TO OPERATOR <consumer_operator>
   ```

2. Repeat Step 1 for a maximum of 32 targets, separating each APPLY clause by a comma. Omit the comma after the last one.

3. Define one or more sources with any combination of the following:
   
   - Use a SELECT statement for each reference to a producer operator or database object.
   - Use a UNION ALL statement to combine multiple SELECT statements.

   Use the following syntax to define multiple sources:
   
   ```
   SELECT <column_list> FROM <producer_operator1>
   UNION ALL
   SELECT <column_list> FROM <producer_operator2>
   UNION ALL
   SELECT <column_list> FROM <producer_operator3>
   ```

   For more information about the required and optional attributes for the APPLY clause, see the Teradata Parallel Transporter Reference.

   For more information about the UNION ALL option, see “UNION ALL: Combining Data from Multiple Sources” on page 373.

**Example**

The following examples compare a single APPLY clause to multiple APPLY specifications. The examples use the syntax discussed in the previous procedure:

- Single APPLY target:
  
  ```
  APPLY ('INSERT INTO EMP_TARGET1 (:EMP_ID, :EMP_LNAME, :EMP_FNAME, :EMP_DEP);') TO OPERATOR (LOAD_OPERATOR_1)
  ```
SELECT * FROM OPERATOR (EXPORT_OPERATOR_1);

- Two APPLY targets:

  APPLY ( 'UPDATE table1 SET C2 = :col2 WHERE C1 = :col1;', 'INSERT INTO table2 ( :col1, :col2, ...)' ) TO OPERATOR ( UPDATE_OPERATOR () [2])

  APPLY ( 'INSERT INTO table3 ( :col1, :col2, ...)') TO OPERATOR ( LOAD_OPERATOR () [3] ATTR(....))

SELECT * FROM OPERATOR (EXPORT_OPERATOR_1);
This chapter provides information on extended character sets.

Topics include:

- Using Extended Character Sets
- Using LONG VARCHAR with Unicode Character Sets

Using Extended Character Sets

Teradata PT allows use of extended multibyte character sets, such as Japanese, Chinese, Korean, Latin, UTF8, and UTF16, as long as they are supported and defined for Teradata Database.

The character set must be common among the following elements when using Teradata PT:

- Data
- Session character set, as specified in the job script.
- Character set used to encode the job script.

The exception to the preceding rule is that for UTF8 and UTF16, the job script may be coded in one character set while the data is in the other, if the proper setup is executed.

For information about UTF8 / UTF16 compatibility and setup, see “Options for Specifying UTF8 and UTF16 in a Job Script” on page 388.

For a list of valid values for session character set, see International Character Set Support.

Note: UTF8 and UTF16 are only supported by Teradata PT on network-attached platforms.

Character Set Hierarchy

Teradata PT follows strict hierarchies for determining what character set is used:

Communication Sessions

- For communication sessions between Teradata PT and the Teradata Database, the following rules apply in the following order:
  - User-Determined: The character set specified in the USING CHARACTER SET clause placed immediately before the DEFINE JOB statement is used above all other character set definitions.
• **System Parameter Block (SPB):** If no user-specified character set is specified, the character set is defined in the HSHSPB file on channel-attached systems or in the clispb.dat file on network-attached systems.

• **Teradata Database:** If neither of the above is specified, Teradata PT uses the character set defined in the Teradata Database. This rule applies to the following operators that communicate with the Teradata Database:
  - Load
  - Update
  - Export
  - Stream
  - SQL Inserter
  - SQL Selector
  - DDL

• **Data Transmission:** If the character set is not specified in a job that uses the DataConnector operator, Teradata PT uses the platform default character set of ASCII or EBCDIC to export data to the Teradata Database.

### Specifying an Extended Character Set in a Teradata PT Job Script

**Session Character Set:** the session character set in a Teradata PT script is determined in one of the following ways:

- **Default:** It is not necessary to specify any session character set before defining the job if you are using the default session character set. The default session character set for all Teradata PT jobs are:
  - ASCII for network-attached client systems
  - EBCDIC for channel-attached client systems
- **Specified:** To use a session character set other than ASCII or EBCDIC, specify the session character set by including a session character set identification clause:
  ```plaintext
  USING CHARACTER SET <characterSet>
  ```

  The identification clause is specified right before the DEFINE JOB <jobname> statement.

  For example:
  ```plaintext
  DEFINE JOB LOAD_TABLES
  (...
   ...
   ...)
  ;
  ```

  In the above example, the session character set will be set to the default session character set: ASCII (network) and EBCDIC (mainframe).

  In the following example, the session character set, an extended session character set, will be set to KANJISJIS_0S:
  ```plaintext
  USING CHARACTER SET KANJISJIS_0S
  DEFINE JOB LOAD_TABLES
  (....
   ...)
Extended Identifiers: an identifier inside a Teradata PT script is non-keyword name that is used to uniquely identify an object. For example:

- DEFINE JOB JOB_NAME
  where JOB_NAME is the identifier.
- DEFINE OPERATOR FILE_READER
  where FILE_READER is the identifier.
- STEP Setup_Tables
  where Setup_Tables is the identifier.

Teradata PT allows identifiers inside a job script to contain extended characters, provided they match the extended session character set. However, when using extended identifiers in Teradata PT, delimit the extended identifier with double quotes. For example:

DEFINE JOB "JOB_NAME_WITH_KANJISJIS_0S_CHARACTERS"
DEFINE OPERATOR "FILE_READER_WITH_UTF8_CHARACTERS"
STEP "Setup_Tables_with_TCHBIG5_1R0_characters"

Extended Session Character Set Data

Data to be loaded and exported must be the same as the session character set. The same is true for extended character sets. Data must match the extended session character set.

UTF16 Support - Specifying Unicode in tbuild Command

As described above, the USING CHARACTER SET <characterSet> statement in the Teradata PT job script is used to define the session character set. The session character set must match the data and it must match the encoding of the job script.

When submitting a job script that is encoded in UTF16, however, you must also specify the -e command line option for the tbuild command.

tbuild -f <file name> [-v jobVariableFile] -e UTF16

-e UTF16 indicates to Teradata PT that the job script is encoded in UTF16. The file endianness is determined by the Byte Order Mark (BOM) at the beginning of the file.

The following -e options are now added to TPT to support the different encoding schemes:

1. UTF16 / UTF-16 and any upper/lower case variations. For UTF16 scripts. If the script is not UTF16, error is reported. If the script endianness differs from the platform encoding, the script is converted to the platform endianness before execution.

2. UTF16LE / UTF-16LE and any upper/lower case variations. For UTF16 little endian scripts. If the script is not little endian, an error is reported. If the platform is big endian, the script is converted to big endian before execution.
3. UTF16BE / UTF-16BE and any upper/lower case variations. For UTF16 big endian scripts. If the script is not big endian, an error is reported. If the platform is little endian, the script is converted to little endian before execution.

4. UTF8 / UTF-8 and any upper/lower case variations. For UTF8 scripts. If the script is not UTF8, error is reported. Specifying UTF8 is optional.

The job variable and include files in either big endian or little endian format can be executed on either kind of platform.

**Setting the Byte Order Mark**

Use this feature to write a BOM at the beginning of a data file. The actual BOM written is determined by the character set in use and, when using UTF16, the endian aspect of the active platform. BOMs are detected when data is read back to ensure correct processing.

The following conditions must be met:

- The operator must be a consumer
- A Unicode character set must be specified in the script (for example, USING CHARACTER SET UTF8)
- The data format must be either text or delimited (for example, VARCHAR Format = 'Delimited')
- The WriteBOM attribute must be set to 'Yes' (for example, VARCHAR WriteBOM = 'Yes')

**Note:** If WriteBOM = 'Yes' and any of the other necessary conditions are not met, the job fails.

**Usage Notes**

Consider the following when working with varying session character sets:

When using UTF16 character set in TPT scripts, the value of n in VARCHAR(n) and CHAR(n) in the SCHEMA definition must be an even and positive number.

LONG VARCHAR and LONG VARGRAPHIC are no longer supported as column types. LONG VARCHAR now corresponds to VARCHAR(64000). See “Using LONG VARCHAR with Unicode Character Sets” on page 393” for a discussion on how Teradata PT will handle this column type.

To view log files in UTF16 format, see “Viewing Logs in UTF16 Format” on page 281.

**Options for Specifying UTF8 and UTF16 in a Job Script**

Options for Specifying UTF8 and UTF16 in a Job Script

In most cases, all the elements of a job must use a common character set:

- Data
- Session character set, as specified in the job script
- Character set used to encode the job script

However, because of the unique properties of the Unicode character sets supported by Teradata PT on network-attached platforms (UTF8 and UTF16), it is possible for the session
character set and the job script encoding to use either the same or different Unicode character sets.

Specify the correct Unicode character set in a job script according to the following scenarios:

**Note:** Do not use the "-" after "UTF" in a job script.

Table 36: Specifying Unicode character sets in a Teradata PT Job Script

<table>
<thead>
<tr>
<th>Job Script Encoding</th>
<th>Session Character Set</th>
<th>Character Set Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTF8</td>
<td>UTF8</td>
<td>Specify the character set in the job script as follows: USING CHARACTER SET UTF8 DEFINE JOB...</td>
</tr>
<tr>
<td>UTF8</td>
<td>UTF16</td>
<td>Specify the character set in the job script as follows: USING CHARACTER SET UTF16 DEFINE JOB...</td>
</tr>
</tbody>
</table>
| UTF16               | UTF8                  | Do both of the following:  
  • Specify the character set in the job script as follows: USING CHARACTER SET UTF8 DEFINE JOB...  
  • Specify the character in the command line as follows: tbuild -e UTF16 |
| UTF16               | UTF16                 | Do both of the following:  
  • Specify the character set in the job script as follows: USING CHARACTER SET UTF16 DEFINE JOB...  
  • Specify the character in the command line as follows: tbuild -e UTF16 |

**Note:** Only network-attached clients running Teradata PT support the use of UTF8 and UTF16 character sets in job scripts. Channel-attached clients running Teradata PT do not support these character sets.

**Example 1**

The following script uses the Teradata PT Load operator to load a Teradata Database table with data extracted from a flat file, and uses the DataConnector operator as a file reader.

```sql
USING CHARACTER SET Kanjisjis_0s DEFINE JOB LOAD_F2_T_OP DESCRIPTION 'フラットファイルからTeradataテーブルへのロード（オペレーター経由）'
(
  DEFINE OPERATOR LoadOperator_TESTTABLE
  DESCRIPTION 'LOAD OPERATOR FOR Teradata PT'
  TYPE LOAD
  SCHEMA *
  ATTRIBUTES
  (  
```
Chapter 28: Extended Character Sets
Using Extended Character Sets

```sql
VARCHAR AccountID,
INTEGER BufferSize,
INTEGER ErrorLimit,
VARCHAR ErrorTable1 = 'testtable_err1',
VARCHAR ErrorTable2 = 'testtable_err2',
VARCHAR LogTable" = 'logtable',
INTEGER MaxSessions,
INTEGER MinSessions,
VARCHAR PauseAcq = 'No',
VARCHAR PrivateLogName,
VARCHAR TargetTable = 'テストテーブル',
VARCHAR TdpId = 'test',
INTEGER TenacityHours,
INTEGER TenacitySleep,
VARCHAR TraceLevel = 'None',
VARCHAR UserName = 'testuser',
VARCHAR UserPassword = 'testpassword',
VARCHAR WorkingDatabase
);

DEFINE SCHEMA SCHEMA_testtable
DESCRIPTION 'test/testuser.testtable 用'
(
    col1 INTEGER,
    col2 CHARACTER(10)
);

DEFINE OPERATOR DataconnectorOperator_TESTTABLE
DESCRIPTION 'DATACONNECTOR OPERATOR FOR Teradata PT'
TYPE DATACONNECTOR PRODUCER
SCHEMA SCHEMA_testtable
ATTRIBUTES
(
    VARCHAR AccessModuleInitStr,
    VARCHAR AccessModuleName,
    INTEGER BlockSize,
    VARCHAR DirectoryPath,
    VARCHAR ExpirationDate,
    VARCHAR FileName,
    VARCHAR Format,
    VARCHAR IndicatorMode = 'N',
    VARCHAR OpenMode,
    VARCHAR PrivateLogName,
    VARCHAR RecordFormat,
    VARCHAR RecordLength,
    INTEGER RetentionPeriod,
    INTEGER RowPerInstance,
    VARCHAR TraceLevel = 'None',
    VARCHAR TextDelimiter,
    INTEGER UnitCount,
    VARCHAR UnitType,
    VARCHAR VolSerNumber,
    INTEGER VolumeCount
);

APPLY
(
    'Insert Into テストテーブル (Col1, Col2) Values (:Col1,
    :Col2):'
)

```

Teradata Parallel Transporter User Guide
Example 2

The following example script extracts rows from a Teradata Database table using the Teradata PT Export operator, and stores the rows as records in a flat file using the DataConnector operator.

```sql
USING CHARACTER SET Kanjisjis_0s DEFINE JOB LOAD_F2_T_OP
DESCRIPTION 'テーブルからフラットファイルへのエクスポート (オペレーテー経 由)'
DEFINE LOG VIEW DEFAULT_LOGVIEW

DEFINE LOGGER DEFAULT_LOGGER
DESCRIPTION 'すべてのメッセージ'
DEFAULT VIEW DEFAULT_LOGVIEW
KEEP FILE
INFO
TRACE
DEBUG
ENABLE OTHER;

DEFINE OPERATOR DataconnectorConsumerTemplate
DESCRIPTION 'DATACONNECTOR OPERATOR FOR TERADATA PARALLEL TRANSPORTER'
DATACONNECTOR CONSUMER
SCHEMA *
ATTRIBUTES
```
Chapter 28: Extended Character Sets

Using Extended Character Sets

```sql
DEFINE SCHEMA SCHEMA_testtable
DESCRIPTION 'test/testuser.testtable
{
    col1 INTEGER,
    col2 CHARACTER(10)
}

DEFINE OPERATOR ExportOperator _ TESTTABLE
DESCRIPTION 'test/testuser.testtable 用エクスポートオペレーター'
TYPE EXPORT
ATTRIBUTES
{
    VARCHAR AccountId,
    INTEGER BlockSize,
    VARCHAR DateForm = 'IntegerDate',
    INTEGER MaxSessions,
    INTEGER MinSessions,
    VARCHAR PrivateLogName,
    VARCHAR SelectStmt = 'Select * from テストテーブル',
    VARCHAR TdpId + 'test',
    INTEGER TenacityHours,
    INTEGER TenacitySleep,
    VARCHAR TraceLevel = 'None',
    VARCHAR UserName = 'testuser',
    VARCHAR UserPassword = 'testpassword',
    VARCHAR WorkingDatabase,
}
LOAD INTO OPERATOR
{
    DataconnectorConsumerTemplate
ATTRIBUTES
{
    FileName = 'file002.txt',
    Format = 'Formatted'
}
```
Using LONG VARCHAR with Unicode Character Sets

Teradata supports a column type of LONG VARCHAR. When dealing with single-byte character sets (both client session character set and server storage character set) a LONG VARCHAR is interpreted as VARCHAR(64000).

When processing the script and coming across a column of type LONG VARCHAR, Teradata PT interprets this column type as VARCHAR(64000). Since the column type is passed on to the operators, some jobs may not run properly.

Problems may arise when the server side storage character set is Unicode or when the LONG VARCHAR column type is used in a schema definition. This is because the combination of the client side session character set and the server storage character set can cause the LONG VARCHAR specification in a DML USING clause to mean something other than VARCHAR(64000).

In summary:

- The use of LONG VARCHAR in a schema object definition is not recommended.
- Do not use LONG VARCHAR with Unicode character sets. Instead, specify VARCHAR(64000).
Chapter 28: Extended Character Sets
Using LONG VARCHAR with Unicode Character Sets
Operational metadata are data that describe the operational aspects of job execution. From Teradata PT point of view, operational metadata specifically refers to data that describe all aspects of operations, activities, timing and events, performance, and statistics that are associated with Teradata PT jobs executed in the data warehousing environment.

Topics include:
- Metadata Types
- Viewing Metadata
- Example Metadata Log Output
- Exporting and Loading Metadata
- Analyzing Job Metadata

By default, Teradata PT collects the basic types of metadata such as performance and statistical data for each instance of each operator at the beginning and end of each processing phase, which includes operator initialization, data acquisition, data application to target tables, and operator termination. There are three types of operational metadata collected and stored in the Teradata PT job log.

**Metadata Types**

Teradata PT is capable of providing three types of metadata.
- TWB_STATUS private log captures job performance metadata
- TWB_SRCTGT private log captures source and target metadata
- TWB_EVENTS private log captures operation event metadata

**TWB_STATUS Performance Metadata**

TWB_STATUS private log captures job performance data at different stages of the job. These stages, also known as operator processing methods, include the following:
- initialization of operators (INITIATE method)
- data acquisition performed by the operators (EXECUTE method)
- checkpoint processing (CHECKPOINT method)
- restart processing (RESTART method)
- termination of operators (TERMINATE method)
By default, Teradata PT collects performance data for each instance of the operator at the beginning and end of each method. Teradata PT also provides a `tbuild` command option for specifying the interval (in seconds) for collecting performance data. For details about all `tbuild` options, see Teradata Parallel Transporter Reference.

The performance data can be viewed as a relational table, which contains the following fields:

- The name of the job step
- The name of the operator
- Instance number
- Processing method (INITIATE, EXECUTE, CHECKPOINT, RESTART, TERMINATE)
- Start time of a method
- End time of a method
- CPU utilization (in seconds) for a method
- Number of buffers transferred since the beginning of data acquisition
- Number of rows sent (or received) by the instance since the beginning of data acquisition

This information is useful for evaluating the performance of a job in terms of throughput and the cost of exporting and loading of data by each operator. It is also useful for capacity planning by collecting the performance data for a period of time, summarizing the CPU utilization and elapsed time for each job, and then determining the trend of performance for the overall loading and exporting processes for a specific system configuration.

**TWB_SRCTGT Operator Source and Target Metadata**

Job operator source and target metadata are stored in the Teradata PT private log called TWB_SRCTGT. This metadata provides detailed information on the data accessed by Teradata PT operators, such as external data files processed, access module types, as well as actual Teradata PT tables populated while the job runs.

**TWB_EVENTS Operation Event Metadata**

Job event metadata are stored in the Teradata PT private log called TWB_EVENTS. These metadata provide timely and granular operational information, which includes event detection and notification at different levels and stages of Teradata PT jobs. Event data can be used to perform event analysis, enabling you to streamline and automate as many of the operational procedures as possible. Some examples of event metadata include:

- Rows processed since job start
- Rows processed since the last checkpoint
- Rows checkpointed since job start
- Rows applied to the target tables
- CPU time used by each operator since job start
- Elapsed time since job start (how long it has been running)
- Start and End of data acquisition and data application phases
- Start and end of a job
Example Metadata Log Output

The following sections show the schemas for the operational metadata logs.

Example: TWB_STATUS Performance and Statistical Metadata

The data schema for the TWB_STATUS' private log can be mapped to the following CREATE TABLE DDL statement:

```sql
CREATE TABLE Job_Status_Tbl
(
    Step_Name           varchar(21),
    Task_Name           varchar(21),
    Status_Message      varchar(21),
    Operator_Name       varchar(21),
    Instance_Count      varchar(5),
    Instance_Number     varchar(5),
    Status              varchar(21),
    Start_Time          varchar(9),
    Elapsed_Time        varchar(11),
    CPU_Time            varchar(11),
    Block_Size          varchar(11),
    Buffer_Count        varchar(11),
    Input_Rows          varchar(17),
    Output_Rows         varchar(17),
    Checkpoint_Interval varchar(6),
    Latency_Interval    varchar(6),
    End_of_Data         varchar(2),
    Multi_Phase         varchar(1)
);
```

Example: TWB_SRCTGT Job Operator Source and Target Metadata

The data schema for the TWB_SRCTGT private log can be mapped to the following CREATE TABLE DDL statement:

```sql
CREATE TABLE Job_SrcTgt_Tbl
(
    Step_Name       varchar(21),
    Task_Name       varchar(21),
    Operator_Name   varchar(21),
    SrcTgt_Type     varchar(21),
    SrcTgt_System   varchar(21),
    SrcTgt_Path     varchar(41),
    SrcTgt_Name     varchar(80)
);
```
Example: TWB_EVENTS Event Metadata

The data schema for the TWB_EVENTS private log can be mapped to the following CREATE TABLE DDL statement:

```sql
create table Job_Status_Tbl
{
    Job_ID        varchar(128),
    Event_Code    varchar(10),
    Event_String  varchar(128),
    Job_Name      varchar(128),
    Job_Step      varchar(128),
    Operator_Name varchar(128),
    Instance_Number varchar(4),
    Time_Stamp   varchar(24),
    Event_Data   varchar(64000)
};
```

Viewing Metadata

Use the `tlogview` command to retrieve job performance and statistical metadata will be collected, as follows.

To access the TWB_STATUS log, enter the following:

`tlogview -l <user log file name> -f TWB_STATUS > <output file name>`

To access the TWB_SRCTGT log, enter the following:

`tlogview -l <user log file name> -f TWB_SRCTGT > <output file name>`

To access the TWB_EVENTS log, enter the following:

`tlogview -l <user log file name> -f TWB_EVENTS > <output file name>`

where:

- `<user log file name>` is the Teradata PT log file name, typically ending with an `.out` extension.
- `<output file name>` is the user-supplied name of the file to receive the output from the command

After the performance data has been collected, you can load it into a set of relational tables so that queries against the data can be done with SQL. For example, about how to load the TWB_STATUS log into a relational table, see “Exporting and Loading Metadata” on page 399.
Exporting and Loading Metadata

Operational metadata are stored in the following TPT predefined private logs for each job:

- TWB_STATUS log
- TWB_SRCTGT log
- TWB_EVENTS log

Using the data schema described in "Obtaining Metadata" on page 385, operational metadata from these logs can be loaded into Teradata tables for SQL access. Use the scripts supplied in the Samples directory that is installed with Teradata PT. The script samples include instructions.

- To export performance and statistical metadata, use the script named `twb_status.txt`.
- To load operator source and target metadata, use the script named `twb_targets.txt`.
- To load job event metadata, use the script named `twb_events.txt`.

SQL examples for extracting operational metadata from Teradata tables are also stored in the Teradata PT Samples directory as follows:

- `sql1.txt` demonstrates how to extract job performance and statistical metadata.
- `sql2.txt` demonstrates how to extract job operator source and target metadata.

Each of the SQL files also provides examples for using an SQL join to extract combined metadata results from the operational metadata tables.

Analyzing Job Metadata

The following tips can be used to evaluate performance metadata and tune the job script:

- Determine the difference in CPU utilization between the producer and consumer operators. For example, if the CPU utilization of the producer operator is 2 times greater than that of the consumer operator, increasing the number of producer instances by a factor of 2 might improve the throughput of the job.

- Determine the difference between the CPU utilization and the elapsed time for performing the exporting and loading of data (i.e. the EXECUTE method). If the elapsed time is much higher than the CPU time, this could mean that some bottlenecks might have occurred either on the network, I/O system, or the Teradata Database server.

- Find out how many rows were sent by the producer operator (or received by the consumer operator) with the above CPU utilization. Dividing the numbers of rows by the CPU seconds spent on processing these rows would give you the number of rows per CPU second.
• The difference between the "start time" of two successive methods would indicate how long the job spent on a method.

• Find out how much time being spent on each checkpoint. Note checkpoint takes time and resources to process. Tuning the number of checkpoints to be taken by changing the checkpoint interval is necessary.
This appendix provides job script examples for a variety of ETL tasks, which are based on actual field applications of Teradata PT.

The examples are keyed to the common job scenarios shown in Chapters 5 through 7. They also include job setup and cleanup tasks discussed in Chapter 4. The scripts can be copied into Notepad or a UNIX editor and then customized to meet job-specific requirements, as shown in “Accessing and Editing Sample Scripts” on page 40.

This Appendix includes the following sections:

- How to Use Job Script Examples
- Job Script Example Library

**How to Use Job Script Examples**

The script examples shown in this appendix are meant to be used as templates. To begin with, this will probably entail copying the entire script and editing it to customize the function for your specific needs. Later, you may find it more useful to simply use the examples to remind you of key concepts, but draw the operator definitions and other script components from your own collection of customized scripts.

Using script examples to accomplish an ETL task involves the following steps:

1. Copy the script from either this appendix, or from the sample script file embedded in the Teradata PT software, into Notepad or a UNIX editor, as shown in “Accessing and Editing Sample Scripts” on page 40. Notice the differences in editing scripts for z/OS.
2. Determine the script components that need to be customized for your usage.
3. Set up global job variables
4. Give the script a unique job name
5. File the script under the job name or other unique name
6. Use the script file name in a `tbuild -f` statement to launch the customized job. For further information, see Chapter 21: “Launching a Job.”

Before editing an example script for your own use, you should read the following:

- For information on job components, see Chapter 2: “Teradata PT Job Components.”
- For information on a specific example script, see the Job Example of the corresponding number in Chapters 5 through 7.
Determining What to Edit in a Sample Script

Sample scripts contain all the script components necessary to accomplish the task defined for the script. However, many of these components will require editing before you can use the script. Typical items that require editing are as follows:

- Site-specific operator attributes such as Tdpid, UserName and UserPassword.
- Job-specific items such as:
  - Schema definition
  - DDL/DML in the APPLY statement
- Key operator attributes
- You may also decide to specify attributes not shown in the example operator definitions.
  For information on operator attributes, see *Teradata Parallel Transporter Reference*.
- Job variables

Setting Up Job Variables

The script examples provided by Teradata PT make extensive use of job variables. Using job variables allows for standardization of common attribute values. Such standardization simplifies assignment of attribute values and ensures consistency. Job variables also provide security, in that administrative username/password strings are not visible in the job script.

Example Job Variables File

The examples in this appendix use the following set of job variables:

```
jobvar_tdpid          = 'my_common_tdpid',
jobvar_username       = 'my_common_username',
jobvar_password       = 'my_common_password',
jobvar_datafiles_path = 'my_common_datafiles_path',
jobvar_tgt_dbname     = 'my_common_tgt_dbname',
jobvar_wrk_dbname     = 'my_common_wrk_dbname',
jobvar_archive_dir    = 'my_common_archive_directory',
jobvar_DB2Id          = 'my_common_DB2Id',
jobvar_DB2UserName    = 'my_common_DB2UserName',
jobvar_DB2Password    = 'my_common_DB2Password',
jobvar_outmod_name    = 'my_common_outmod_name',
jobvar_lobpath        = 'my_common_lobpath',
jobvar_lobfilename    = 'my_common_lobfilename',
jobvar_lobextension   = 'my_common_lobextension'
```

Use of job variables requires that you set up job variables either in the global job variables file or the local job variables file. For detailed information on the setup and use of job variables, see Chapter 3: "Job Setup Tasks."

Explanation of Job Variables Example

The job variables shown in the previous example are used throughout the script examples found in this appendix. When you use a script example, update the variable values with values for your system. Those values will be usable for all script examples that specify the variables.
Job Script Example Library

The following job scripts comprise the Script Example Library.

List of Script Examples

- Moving Data from an External Source into Teradata Database
  - Script Example 1A: High Speed Bulk Loading from Flat Files into an Empty Teradata Database Table
  - Script Example 1B: High Speed Bulk Loading from a Named Pipe into an Empty Teradata Database Table
  - Script Example 1C: High Speed Bulk Loading from Two Flat Files into an Empty Teradata Database Table Using UNION ALL
  - Script Example 2A: Read Data Direct from Source Files and Perform INSERT, UPDATE, and DELETE on Multiple Teradata Database Tables
  - Script Example 2B: Read Data from a Named Pipe and Perform INSERT, UPDATE, and DELETE on Multiple Teradata Database Tables
  - Script Example 3: Loading BLOB and CLOB Data into Teradata Database
  - Script Example 4A: Pre-process Data with an INMOD Routine Before Loading into an Empty Teradata Database Table (FastLoad Protocol)
  - Script Example 4B: Pre-process Data with an INMOD Routine Before Loading into Teradata Database Tables (Multiload Protocol)
  - Script Example 5A: Continuous Loading of Transactional Data from MQ
  - Script Example 5B: Continuous Loading of Transactional Data from JMS
  - Script example 5C: Intermediate File Logging Using Multiple APPLY Clauses with Continuous Loading of Transactional Data
  - Script Example 6: Loading Data from Other Relational Databases into an Empty Teradata Database Table
  - Script Example 7: Mini-Batch Loading into Teradata Database Tables
  - Script Example 8: Batch Directory Scan
  - Script Example 9: Active Directory Scan
- Moving Data from Teradata Database into an External Target
  - Script Example 10: Extracting Rows and Writing Them in Delimited Format
  - Script Example 11: Export Rows and Write Them as Binary or Indicator Mode Data
  - Script Example 12: Export Data and Process It with an OUTMOD Routine
  - Script Example 13: Export Data and Process It with an Access Module
  - Script Example 14: Extract BLOB/CLOB Data and Write It to an External Target
- Moving Data within the Teradata Database Environment
  - Script Example 15: Extract Data from a Teradata Database Staging Table and Load It into a Production Table
• Script Example 16: Export Data and then Use It to Perform Conditional Updates Against Production Tables
• Script Example 17: Bulk Delete of Data from a Teradata Database
• Script Example 18: Exporting BLOB/CLOB Data from One Teradata Database Table and Loading It into Another

**Script Example 1A: High Speed Bulk Loading from Flat Files into an Empty Teradata Database Table**

Use the following script as a template for jobs that read data directly from external flat files and load it into an empty Teradata Database table.

For a discussion of usage considerations, see *Job Example 1: High Speed Bulk Loading into an Empty Table*.

```sql
/*******************************************************************************
/*                                                                            */
/* Teradata Parallel Transporter                                            */
/* User Guide - Job Example 01A                                             */
/*                                                                            */
/*******************************************************************************
/* Description:                                                              */
/*                                                                            */
/* This example script uses two job steps.                                   */
/*                                                                            */
/* The first job step, called "Setup_Tables", uses the DDL                  */
/* Operator to setup the target table.                                       */
/*                                                                            */
/* The second job step, called "Load_Trans_Table", uses multiple instances of */
/* the DataConnector Operator to read rows from a file and uses multiple    */
/* instances of the Load Operator to write the rows into an empty           */
/* Teradata target table.                                                    */
/*******************************************************************************/

DEFINE JOB FILE_LOAD
DESCRIPTION 'Load a Teradata table from a file'
(

DEFINE SCHEMA Trans_n_Accts_Schema
(
    Account_Number VARCHAR(50),
    Trans_Number VARCHAR(50),
    Trans_Date VARCHAR(50),
    Trans_ID VARCHAR(50),
    Trans_Amount VARCHAR(50)
);

DEFINE OPERATOR DDL_OPERATOR
TYPE DDL
ATTRIBUTES
(
    VARCHAR PrivateLogName = 'ddl_log',
    VARCHAR TdpId = @jobvar_tdpid,
    VARCHAR UserName = @jobvar_username,
);```
DECLARE
 backgroundImage VARCHAR(50)
 = 'accounts.png',
 backgroundImageSize VARCHAR(50)
 = '2000x1000',
 userName VARCHAR(50)
 = 'John Doe',
 password VARCHAR(50)
 = 'password123',
 database VARCHAR(50)
 = 'exampledb',
 table VARCHAR(50)
 = 'example_table',
 query VARCHAR(50)
 = 'SELECT * FROM example_table';

CREATE TABLE example_table (
    id INT,
    name VARCHAR(50),
    age INT
);

SELECT * FROM example_table;

ALTER TABLE example_table
ADD COLUMN gender VARCHAR(10);

INSERT INTO example_table
VALUES (1, 'Alice', 25, 'Female'),
       (2, 'Bob', 30, 'Male'),
       (3, 'Carol', 28, 'Female');

UPDATE example_table
SET age = age + 1;

DELETE FROM example_table
WHERE age > 30;

CREATE VIEW example_view AS
SELECT * FROM example_table;

GRANT SELECT ON example_view TO public;

DROP VIEW example_view;

DROP TABLE example_table;
Script Example 1B: High Speed Bulk Loading from a Named Pipe into an Empty Teradata Database Table

Use the following script as a template for jobs that read data from Named Pipes and load it into an empty Teradata Database table.

For a discussion of usage considerations, see Job Example 1: High Speed Bulk Loading into an Empty Table.

```
/*************************************************************/
/*                                                            */
/* Teradata Parallel Transporter                              */
/* User Guide - Job Example 01B                              */
/*                                                            */
/*************************************************************/
/*                                                            */
/* Description:                                               */
/*                                                            */
/* This example script uses two job steps.                    */
/*                                                            */
/* The first job step, called "Setup_Tables", uses the DDL    */
/* Operator to setup the target table.                       */
/*                                                            */
/* The second job step, called "Load_Trans_Table", uses       */
/* multiple instances of the DataConnector Operator to read   */
/* rows from a named-pipe access module and uses multiple     */
/* instances of the Load Operator to write the rows into an   */
/* empty Teradata target table.                              */
/*                                                            */
/*************************************************************/

DEFINE JOB ACCESS_MODULE_LOAD
DESCRIPTION 'Load a Teradata table from an Access Module'
{
  DEFINE SCHEMA Trans_n_Accts_Schema
  (
      Account_Number VARCHAR(50),
      Trans_Number    VARCHAR(50),
      Trans_Date      VARCHAR(50),
      Trans_ID        VARCHAR(50),
      Trans_Amount    VARCHAR(50)
  );

  DEFINE OPERATOR DDL_OPERATOR
  TYPE DDL
  ATTRIBUTES
  {
      VARCHAR PrivateLogName = 'ddl_log',
      VARCHAR TdpId          = @jobvar_tdpid,
      VARCHAR UserName       = @jobvar_username,
      VARCHAR UserPassword   = @jobvar_password,
  }

  TO OPERATOR (LOAD_OPERATOR[2])
    SELECT * FROM OPERATOR(FILE_READER[2]);
};
```
VARCHAR ErrorList = '3807'
);

DEFINE OPERATOR ACCESS_MODULE_READER
TYPE DATACONNECTOR PRODUCER
SCHEMA Trans_n_Accts_Schema
ATTRIBUTES
{
  VARCHAR PrivateLogName = 'dataconnector_log',
  VARCHAR AccessModuleName = 'np_axsmod.so',
  VARCHAR AccessModuleInitStr,
  VARCHAR FileName = 'datapipe',
  VARCHAR Format = 'Formatted',
  VARCHAR OpenMode = 'Read'
);

DEFINE OPERATOR LOAD_OPERATOR
TYPE LOAD
SCHEMA *
ATTRIBUTES
{
  VARCHAR PrivateLogName = 'load_log',
  VARCHAR TdpId = @jobvar_tdpid,
  VARCHAR UserName = @jobvar_username,
  VARCHAR UserPassword = @jobvar_password,
  VARCHAR TargetTable = @jobvar_tgt_dbname || '.Trans',
  VARCHAR LogTable = @jobvar_wrk_dbname || '.LG_Trans',
  VARCHAR ErrorTable1 = @jobvar_wrk_dbname || '.ET_Trans',
  VARCHAR ErrorTable2 = @jobvar_wrk_dbname || '.UV_Trans'
};

STEP Setup_Tables
{
  APPLY
    ('DROP TABLE ' || @jobvar_wrk_dbname || '.ET_Trans;'),
    ('DROP TABLE ' || @jobvar_wrk_dbname || '.UV_Trans;'),
    ('DROP TABLE ' || @jobvar_tgt_dbname || '.Trans;'),
    ('CREATE TABLE ' || @jobvar_tgt_dbname
      || '.Trans (Account_Number VARCHAR(50),
      Trans_Number VARCHAR(50),
      Trans_Date VARCHAR(50),
      Trans_ID VARCHAR(50),
      Trans_Amount VARCHAR(50));')
  TO OPERATOR (DDL_OPERATOR);
};

STEP Load_Trans_Table
{
  APPLY
    ('INSERT INTO ' || @jobvar_tgt_dbname || '.Trans(Account_Number,
      Trans_Number, Trans_Date, Trans_ID, Trans_Amount)
    VALUES (:Account_Number, :Trans_Number, :Trans_Date, :Trans_ID, :Trans_Amount);')
TO OPERATOR (LOAD_OPERATOR[2])

    SELECT * FROM OPERATOR(ACCESS_MODULE_READER[2]);

_SCRIPT EXAMPLE 1C: High Speed Bulk Loading from Two Flat Files into an Empty Teradata Database Table Using UNION ALL_

Use the following script as a template for jobs that read data from two external flat files and use UNION ALL to combine the data into a single data stream before loading it into an empty Teradata Database table.

For a discussion of usage considerations, see “Job Example 1: High Speed Bulk Loading into an Empty Table” on page 107.

/**************************************************************/
/*                                                            */
/* Teradata Parallel Transporter                               */
/* User Guide - Job Example 01C                                */
/*                                                            */
/**************************************************************/
/*                                                            */
/* Description:                                               */
/*                                                            */
/* This example script uses two job steps.                     */
/*                                                            */
/* The first job step, called "Setup_Tables", uses the DDL     */
/* Operator to setup the target table.                        */
/*                                                            */
/* The second job step, called "Load_Data_From_Two_Files",    */
/* uses one copy of the DataConnector Operator to read data   */
/* from one file and another copy of the DataConnector        */
/* Operator to read data from a different file.               */
/*                                                            */
/* The UNION ALL operation combines the data into a single     */
/* data stream that is then applied to the Load Operator to    */
/* write that data to an empty Teradata table.                */
/*                                                            */
/***************************************************************/

DEFINE JOB FILE_LOAD_UNION_ALL
DESCRIPTION 'Load a Teradata table from two files with UNION ALL'
{
    DEFINE SCHEMA Trans_n_Accts_Schema
    (
        Account_Number VARCHAR(50),
        Trans_Number VARCHAR(50),
        Trans_Date VARCHAR(50),
        Trans_ID VARCHAR(50),
        Trans_Amount VARCHAR(50)
    );

    DEFINE OPERATOR DDL_OPERATOR
    TYPE DDL
    ATTRIBUTES
    (
        VARCHAR PrivateLogName = 'ddl_log',
    )

    /* Additional script code */
VARCHAR TdpId = @jobvar_tdpid,
VARCHAR UserName = @jobvar_username,
VARCHAR UserPassword = @jobvar_password,
VARCHAR ErrorList = '3807';

DEFINE OPERATOR FILE_READER
TYPE DATACONNECTOR PRODUCER
SCHEMA Trans_n_Accts_Schema
ATTRIBUTES
{
    VARCHAR PrivateLogName,
    VARCHAR DirectoryPath,
    VARCHAR FileName,
    VARCHAR Format,
    VARCHAR OpenMode,
    VARCHAR TextDelimiter
};

DEFINE OPERATOR LOAD_OPERATOR
TYPE LOAD
SCHEMA *
ATTRIBUTES
{
    VARCHAR PrivateLogName = 'load_log',
    VARCHAR TdpId = @jobvar_tdpid,
    VARCHAR UserName = @jobvar_username,
    VARCHAR UserPassword = @jobvar_password,
    VARCHAR TargetTable = @jobvar_tgt_dbname || '.Trans',
    VARCHAR LogTable = @jobvar_wrk_dbname || '.LG_Trans',
    VARCHAR ErrorTable1 = @jobvar_wrk_dbname || '.ET_Trans',
    VARCHAR ErrorTable2 = @jobvar_wrk_dbname || '.UV_Trans'
};

STEP Setup_Tables
{
    APPLY
    ('DROP TABLE   ' || @jobvar_wrk_dbname || '.ET_Trans;'),
    ('DROP TABLE   ' || @jobvar_wrk_dbname || '.UV_Trans;'),
    ('DROP TABLE   ' || @jobvar_tgt_dbname || '.Trans;'),
    ('CREATE TABLE ' || @jobvar_tgt_dbname
        || '.Trans (Account_Number VARCHAR(50),
            Trans_Number   VARCHAR(50),
            Trans_Date     VARCHAR(50),
            Trans_ID       VARCHAR(50),
            Trans_Amount   VARCHAR(50));')
    TO OPERATOR (DDL_OPERATOR);
};

STEP Load_Data_From_Two_Files
{
    APPLY
    ('INSERT INTO ' || @jobvar_tgt_dbname || '.Trans(Account_Number,
        Trans_Number, Trans_Date, Trans_ID, Trans_Amount)
        VALUES(:Account_Number,
        :Trans_Number,
        :Trans_Date,
        :Trans_ID,
        :Trans_Amount)');
}
### Script Example 2A: Read Data Direct from Source Files and Perform INSERT, UPDATE, and DELETE on Multiple Teradata Database Tables

Use the following script as a template for jobs that read data directly from external source files and then perform INSERT, UPDATE, and DELETE operations in multiple Teradata Database tables. This job is equivalent to a common use of the standalone MultiLoad utility.

For a discussion of usage considerations, see “Job Example 2: Perform INSERT, UPDATE, and DELETE in Multiple Tables” on page 110.

```sql
/* ************************************************** */
/* Teradata Parallel Transporter */
/* User Guide - Job Example 02A */
/* ************************************************** */
/* Description: */
/* */
/* This job example uses multiple instances of the */
/* DataConnector Operator to extract rows from a file and */
/* uses multiple instances of the Update Operator to perform */
/* upserts against two tables. */
/* */
/* ************************************************** */

DEFINE JOB FILE_UPDATE DESCRIPTION 'Load 2 TD tables from a file'
{
  DEFINE SCHEMA Trans_n_Accts_Schema (
    Account_Number VARCHAR(50),
    Number VARCHAR(50),
  )

  :Trans_Date, :Trans_ID, :Trans_Amount);

  TO OPERATOR (LOAD_OPERATOR[2])
  SELECT * FROM OPERATOR(FILE_READER
    ATTR (PrivateLogName = 'dataconnector1_log',
          DirectoryPath = @jobvar_datafiles_path,
          FileName = 'accounts1.txt',
          Format = 'Delimited',
          OpenMode = 'Read',
          TextDelimiter = '|'))

  UNION ALL

  SELECT * FROM OPERATOR(FILE_READER
    ATTR (PrivateLogName = 'dataconnector2_log',
          DirectoryPath = @jobvar_datafiles_path,
          FileName = 'accounts2.txt',
          Format = 'Delimited',
          OpenMode = 'Read',
          TextDelimiter = '|'));
```
Appendix A: Job Script Examples

Job Script Example Library

Teradata Parallel Transporter User Guide

Street VARCHAR(50),
City VARCHAR(50),
State VARCHAR(50),
Zip_Code VARCHAR(50),
Balance_Forward VARCHAR(50),
Balance_Current VARCHAR(50),
Trans_Number VARCHAR(50),
Trans_Date VARCHAR(50),
Trans_ID VARCHAR(50),
Trans_Amount VARCHAR(50)
);

DEFINE OPERATOR FILE_READER
TYPE DATACONNECTOR PRODUCER
SCHEMA Trans_n_Accts_Schema
ATTRIBUTES
{
  VARCHAR PrivateLogName = 'dataconnector_log',
  VARCHAR DirectoryPath  = @jobvar_datafiles_path,
  VARCHAR FileName       = 'accounts.txt',
  VARCHAR Format         = 'Delimited',
  VARCHAR OpenMode       = 'Read',
  VARCHAR TextDelimiter  = '|'  
}
);

DEFINE OPERATOR UPDATE_OPERATOR
TYPE UPDATE
SCHEMA *
ATTRIBUTES
{
  VARCHAR PrivateLogName    = 'update_log',
  VARCHAR TdpId             = @jobvar_tdpid,
  VARCHAR UserName          = @jobvar_username,
  VARCHAR UserPassword      = @jobvar_password,
  VARCHAR LogTable          = @jobvar_wrk_dbname || '.LG_Trans_n_Accts',
  VARCHAR ARRAY TargetTable = [@jobvar_wrk_dbname || '.Trans',
                              @jobvar_wrk_dbname || '.Accounts'],
  VARCHAR ARRAY ErrorTable1 = [@jobvar_wrk_dbname || '.ET_Trans',
                              @jobvar_wrk_dbname || '.ET_Accts'],
  VARCHAR ARRAY ErrorTable2 = [@jobvar_wrk_dbname || '.UV_Trans',
                              @jobvar_wrk_dbname || '.UV_Accts']
}
);

APPLY
('INSERT INTO ' || @jobvar_tgt_dbname
 || '.Trans(Trans_Number,
          Trans_Date,
          Account_Number,
          Trans_ID,
          Trans_Amount)
VALUES(:Trans_Number,
       :Trans_Date,
       :Account_Number,
       :Trans_ID,
       :Trans_Amount);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.Accounts SET Number          = :Number,
          Account_Name = :Account_Name,
          Account_Type = :Account_Type
VALUES(:Account_Name,
       :Account_Type,
       :Number,
       :Account_Name);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.ET_Trans_SET Number          = :Number,
          Account_Name = :Account_Name,
          Account_Type = :Account_Type
VALUES(:Account_Name,
       :Account_Type,
       :Number,
       :Account_Name);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.ET_Accts_SET Number          = :Number,
          Account_Name = :Account_Name,
          Account_Type = :Account_Type
VALUES(:Account_Name,
       :Account_Type,
       :Number,
       :Account_Name);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.UV_Trans_SET Number          = :Number,
          Account_Name = :Account_Name,
          Account_Type = :Account_Type
VALUES(:Account_Name,
       :Account_Type,
       :Number,
       :Account_Name);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.UV_Accts_SET Number          = :Number,
          Account_Name = :Account_Name,
          Account_Type = :Account_Type
VALUES(:Account_Name,
       :Account_Type,
       :Number,
       :Account_Name);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.LG_Trans_n_Accts_SET Number = :Number,
          Trans_Number = :Trans_Number,
          Trans_Date  = :Trans_Date,
          Account_Number = :Account_Number,
          Trans_ID     = :Trans_ID,
          Trans_Amount = :Trans_Amount
VALUES(:Trans_Number,
       :Trans_Date,
       :Account_Number,
       :Trans_ID,
       :Trans_Amount);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.LG_Accts_SET Number          = :Number,
          Account_Name = :Account_Name,
          Account_Type = :Account_Type
VALUES(:Account_Name,
       :Account_Type,
       :Number,
       :Account_Name);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.ET_Trans_SET Number          = :Number,
          Account_Name = :Account_Name,
          Account_Type = :Account_Type
VALUES(:Account_Name,
       :Account_Type,
       :Number,
       :Account_Name);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.ET_Accts_SET Number          = :Number,
          Account_Name = :Account_Name,
          Account_Type = :Account_Type
VALUES(:Account_Name,
       :Account_Type,
       :Number,
       :Account_Name);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.UV_Trans_SET Number          = :Number,
          Account_Name = :Account_Name,
          Account_Type = :Account_Type
VALUES(:Account_Name,
       :Account_Type,
       :Number,
       :Account_Name);'),

('UPDATE ' || @jobvar_tgt_dbname
 || '.UV_Accts_SET Number          = :Number,
          Account_Name = :Account_Name,
          Account_Type = :Account_Type
VALUES(:Account_Name,
       :Account_Type,
       :Number,
       :Account_Name);')

Teradata Parallel Transporter User Guide

411
Script Example 2B: Read Data from a Named Pipe and Perform INSERT, UPDATE, and DELETE on Multiple Teradata Database Tables

Use the following script as a template for jobs that read data directly from Named Pipes and then perform INSERT, UPDATE, and DELETE operations in multiple Teradata Database tables. This job is equivalent to a common use of the standalone MultiLoad utility.

For a discussion of usage considerations, see “Job Example 2: Perform INSERT, UPDATE, and DELETE in Multiple Tables” on page 110.
/* */
/*******/

DEFINE JOB ACCESS_MODULE_UPDATE
DESCRIPTION 'Load 2 TD tables from an access module'
(
  DEFINE SCHEMA Trans_n_Accts_Schema
  (  
    Account_Number VARCHAR(50),
    Number VARCHAR(50),
    Street VARCHAR(50),
    City VARCHAR(50),
    State VARCHAR(50),
    Zip_Code VARCHAR(50),
    Balance_Foward VARCHAR(50),
    Balance_Current VARCHAR(50),
    Trans_Number VARCHAR(50),
    Trans_Date VARCHAR(50),
    Trans_ID VARCHAR(50),
    Trans_Amount VARCHAR(50)
  );

  DEFINE OPERATOR ACCESS_MODULE_READER
  TYPE DATACONNECTOR PRODUCER
  SCHEMA Trans_n_Accts_Schema
  ATTRIBUTES
  (  
    VARCHAR PrivateLogName  = 'dataconnector_log',
    VARCHAR AccessModuleName = 'np_axsmod.so',
    VARCHAR AccessModuleInitStr,
    VARCHAR FileName         = 'datapipe',
    VARCHAR Format           = 'Formatted',
    VARCHAR OpenMode         = 'Read'
  );

  DEFINE OPERATOR UPDATE_OPERATOR
  TYPE UPDATE
  SCHEMA *
  ATTRIBUTES
  (  
    VARCHAR PrivateLogName  = 'update_log',
    VARCHAR TdpId           = @jobvar_tdpid,
    VARCHAR UserName        = @jobvar_username,
    VARCHAR UserPassword    = @jobvar_password,
    VARCHAR LogTable        = @jobvar_wrk_dbname || '.LG_Trans_n_Accts',
    VARCHAR ARRAY TargetTable = [@jobvar_wrk_dbname || '.Trans',  
                               @jobvar_wrk_dbname || '.Accounts'],
    VARCHAR ARRAY ErrorTable1 = [@jobvar_wrk_dbname || '.ET_Trans',  
                               @jobvar_wrk_dbname || '.ET_Accts'],
    VARCHAR ARRAY ErrorTable2 = [@jobvar_wrk_dbname || '.UV_Trans',  
                               @jobvar_wrk_dbname || '.UV_Accts']
  );

  APPLY
  ('INSERT INTO ' || @jobvar_tgt_dbname
   || '.Trans(Trans_Number,
   Trans_Date,
   Account_Number,
   Number,
   Street,
   City,
   State,
   Zip_Code,
   Balance_Foward,
   Balance_Current,
   Trans_Number,
   Trans_Date,
   Trans_ID,
   Trans_Amount)
   SELECT ',
   Account_Number,
   Number,
   Street,
   City,
   State,
   Zip_Code,
   Balance_Foward,
   Balance_Current,
   Trans_Number,
   Trans_Date,
   Trans_ID,
   Trans_Amount
   FROM ACCESS_MODULE_READER')
  WITH OPTIONS ('MAXROWS' => 10000000)
);
Script Example 3: Loading BLOB and CLOB Data into Teradata Database

Use this script example as a template for jobs that read inline BLOB and CLOB data from a non-Teradata source and write it to one or more Teradata Database tables.

For a discussion of usage considerations, see “Job Example 3: Loading BLOB and CLOB Data” on page 112.
Appendix A: Job Script Examples

Job Script Example Library

Teradata Parallel Transporter User Guide 415

/*==================================================================
/* Description: */
/* */
/* This example script uses three job steps. */
/* */
/* The first job step, called "Move_Data_Files", uses the OS */
/* Command Operator to copy all *.dat files from the current */
/* directory to another directory. */
/* */
/* The first job step, called "Setup_Table", uses the DDL */
/* Operator to create the target table. */
/* */
/* The second job step, called "Loading_LOB_Data", uses */
/* multiple instances of the DataConnector Operator to */
/* read large object (LOB) data from a file and uses the */
/* SQL Inserter Operator to load them into a Teradata table. */
/* */
/*==================================================================

DEFINE JOB LOB_FILE_LOAD
DESCRIPTION 'Load a large objects from a file to a table'
(
  DEFINE SCHEMA LOB_TABLE_SCHEMA
  DESCRIPTION 'PRODUCT INFORMATION SCHEMA'
  (
    COL1  INTEGER,
    COL2  INTEGER,
    COL3  INTDATE,
    COL4  CHAR(1),
    COL5  VARCHAR(30),
    COL6  VARCHAR(30),
    COL7  CLOB(10000) AS DEFERRED BY NAME,
    COL8  CLOB(40000),
    COL9  BLOB(10000) AS DEFERRED BY NAME,
    COL10 BLOB(20000)
  );

  DEFINE OPERATOR OS_COMMAND_OPERATOR
  TYPE OS COMMAND
  ATTRIBUTES
  (
    VARCHAR PrivateLogName = 'oscommand_log',
    VARCHAR OsCmd = 'cp .//*.dat ./temp/',
    VARCHAR IgnoreError = 'YES'
  );

  DEFINE OPERATOR DDL_OPERATOR
  TYPE DDL
  ATTRIBUTES
  (
    VARCHAR PrivateLogName = 'ddl_log',
    VARCHAR TdpId            = @jobvar_tdpid,
    VARCHAR UserName         = @jobvar_username,
    VARCHAR UserPassword     = @jobvar_password,
    VARCHAR ErrorList        = '3807'
  );

  DEFINE OPERATOR FILE_READER
  TYPE DATACONNECTOR PRODUCER
  
  DEFINE SCHEMA LOB_TABLE_SCHEMA
  DESCRIPTION 'PRODUCT INFORMATION SCHEMA'
  
/*==================================================================
SCHEMA LOB_TABLE_SCHEMA
ATTRIBUTES
    ( VARCHAR PrivateLogName = 'dataconnector_log',
     VARCHAR DirectoryPath = @jobvar_datafiles_path,
     VARCHAR FileName = 'lob_tst10.dat',
     VARCHAR Format = 'Formatted',
     VARCHAR OpenMode = 'Read');

DEFINE OPERATOR SQL_INSERTER
TYPE INSERTER
SCHEMA *
ATTRIBUTES
    ( VARCHAR PrivateLogName = 'inserter_log',
     VARCHAR TdpId = @jobvar_tdpid,
     VARCHAR UserName = @jobvar_username,
     VARCHAR UserPassword = @jobvar_password);

STEP Move_Data_Files
    ( APPLY TO OPERATOR (OS_COMMAND_OPERATOR);
    );

STEP Setup_Table
    ( APPLY
        ('DROP TABLE lob_tst10_tbl;
        'CREATE TABLE lob_tst10_tbl, FALLBACK (COL1_int INTEGER,
        COL2_int INTEGER,
        COL3_date DATE,
        COL4_char CHAR(1),
        COL5_var VARCHAR(30),
        COL6_var VARCHAR(30),
        COL7_clob CLOB(100) AS DEFERRED BY NAME,
        COL8_clob CLOB(50000),
        COL9_blob BLOB(100) AS DEFERRED BY NAME,
        COL10_blob BLOB(20000));'
        )
        TO OPERATOR (DDL_OPERATOR);
    );

STEP Loading_LOB_Data
    ( APPLY
        ('INSERT INTO lob_tst10_tbl values ( :COL1, :COL2, :COL3, :COL4, :COL5, 
        :COL6, :COL7, :COL8, :COL9, :COL10);'
        )
        TO OPERATOR (SQL_INSERTER)
        SELECT * FROM OPERATOR (FILE_READER[2]);
    );
Script Example 4A: Pre-process Data with an INMOD Routine Before Loading into an Empty Teradata Database Table (FastLoad Protocol)

Use this script example as a template for jobs that read data from a non-Teradata source and preprocess it with an INMOD routine before writing the data to an empty Teradata Database table. This job is equivalent to a common use of the standalone FastLoad utility.

For a discussion of usage considerations, see “Job Example 4: Pre-processing Data with an INMOD Routine Before Loading” on page 114.

```sql
/* Teradata Parallel Transporter User Guide - Job Example 04A */
/**************************************************************/
/* Description: */
/* This example script uses two job steps. */
/* The first job step, called "Setup_Tables", uses the DDL */
/* Operator to setup the target table. */
/* The second job step, called "Load_Table" uses the FastLoad */
/* INMOD Adapter Operator to read data from an INMOD and uses */
/* multiple instances of the Load Operator to write the data */
/* to a Teradata table. */
/**************************************************************/

DEFINE JOB INMOD_LOAD
  DESCRIPTION 'Load a table from an INMOD'
  {
    DEFINE SCHEMA PRODUCT_SOURCE_SCHEMA
      DESCRIPTION 'PRODUCT INFORMATION SCHEMA'
      {
        f1  INT,
        f2  CHAR(4),
        f3  CHAR(10),
        f4  CHAR(10),
      };

    DEFINE OPERATOR DDL_OPERATOR
      TYPE DDL
      ATTRIBUTES
      {
        VARCHAR PrivateLogName = 'ddl_log',
        VARCHAR TdpId = @jobvar_tdpid,
        VARCHAR UserName = @jobvar_username,
        VARCHAR UserPassword = @jobvar_password,
        VARCHAR ErrorList = '3807'
      };

    DEFINE OPERATOR INMOD_READER
      TYPE FASTLOAD INMOD
      SCHEMA PRODUCT_SOURCE_SCHEMA
      ATTRIBUTES
```
(     VARCHAR PrivateLogName = 'fastload_inmod_log',
     VARCHAR InmodName = './flin_013_3m.s',
     VARCHAR FileName = 'flin_013.txt',
     VARCHAR Format = 'Formatted',
     VARCHAR OpenMode = 'Read'
 );

DEFINE OPERATOR LOAD_OPERATOR
   TYPE LOAD
   SCHEMA *
   ATTRIBUTES
(     VARCHAR PrivateLogName = 'load_log',
     VARCHAR TdpId = @jobvar_tdpid,
     VARCHAR UserName = @jobvar_username,
     VARCHAR UserPassword = @jobvar_password,
     VARCHAR TargetTable = 'flin_013',
     VARCHAR LogTable = 'flin_013_log',
     VARCHAR ErrorTable1 = 'flin_013_e1',
     VARCHAR ErrorTable2 = 'flin_013_e2'
 );

STEP Setup_Tables
(   APPLY ('DROP   TABLE flin_013_e1;'),
    ('DROP   TABLE flin_013_e2;'),
    ('DROP   TABLE flin_013;'),
    ('CREATE TABLE flin_013 (Account_Number VARCHAR(50),
                              Trans_Number   VARCHAR(50),
                              Trans_Date     VARCHAR(50),
                              Trans_ID       VARCHAR(50),
                              Trans_Amount   VARCHAR(50));')
    TO OPERATOR (DDL_OPERATOR);
 );

STEP Load_Table
(   APPLY ('insert into flin_013 values (:f1, :f2, :f3, :f4);')
    TO OPERATOR (LOAD_OPERATOR[2])

    SELECT * FROM OPERATOR (INMOD_READER);
 );
Script Example 4B: Pre-process Data with an INMOD Routine Before Loading into Teradata Database Tables (Multiload Protocol)

Use this script example as a template for jobs that read data from a non-Teradata source and preprocess it with an INMOD routine before writing the data into a populated Teradata Database table. This job is equivalent to a common use of the standalone MultiLoad utility.

For a discussion of usage considerations, see “Job Example 4: Pre-processing Data with an INMOD Routine Before Loading” on page 114.

```
USES Teradata Parallel Transporter
USES User Guide - Job Example 04B
USES TERADATA
USES INMOD
USES MultiLoad
USES INMOD Adapter Operator
USES Update Operator
USES Product Source Schema
USES Table
USES Column
USES Column Type
USES Data Type
USES Private Log Name
USES Inmod Name
USES File Name
USES Format
USES Open Mode
USES Tdp Id
USES User Name
USES User Password
USES Define Job
USES Define Schema
USES Define Operator
USES Type
USES Attributes
USES Description
USES Load a table from an INMOD

DEFINE JOB INMOD_LOAD
DESCRIPTION 'Load a table from an INMOD'
{
  DEFINE SCHEMA PRODUCT_SOURCE_SCHEMA
  DESCRIPTION 'PRODUCT INFORMATION SCHEMA'
  {
    f1  INT,
    f2  CHAR(4),
    f3  CHAR(10),
    f4  CHAR(10)
  };

  DEFINE OPERATOR INMOD_READER
  TYPE MULTILOAD INMOD
  SCHEMA PRODUCT_SOURCE_SCHEMA
  ATTRIBUTES
  {
    VARCHAR PrivateLogName = 'multiload_inmod_log',
    VARCHAR InmodName = './mlinmod.so',
    VARCHAR FileName = 'mlin_013.txt',
    VARCHAR Format = 'Formatted',
    VARCHAR OpenMode = 'Read'
  };

  DEFINE OPERATOR UPDATE_OPERATOR
  TYPE UPDATE
  SCHEMA *
  ATTRIBUTES
  {
    VARCHAR PrivateLogName = 'update_log',
    VARCHAR TdpId = @jobvar_tdpid,
    VARCHAR UserName = @jobvar_username,
    VARCHAR UserPassword = @jobvar_password,
  }
```
VARCHARG TargetTable = 'mlin_013',
VARCHARG LogTable = 'mlin_013_log',
VARCHARG WorkTable = 'mlin_013_wt',
VARCHARG ErrorTable1 = 'mlin_013_e1',
VARCHARG ErrorTable2 = 'mlin_013_e2'
);

APPLY ('insert into mlin_013 values (:f1, :f2, :f3, :f4);')
TO OPERATOR (UPDATE_OPERATOR[2])

SELECT * FROM OPERATOR (INMOD_READER);
);

**Script Example 5A: Continuous Loading of Transactional Data from MQ**

Use this script example as a template for jobs that read data from MQ and perform
continuous INSERT, UPDATE, and DELETE operations in a Teradata Database table.

For a discussion of usage considerations, see "Job Example 5: Continuous Loading of
Transactional Data from JMS or MQ" on page 116.

/***************************************************************************/
/*                                                                      */
/* Teradata Parallel Transporter                                        */
/* User Guide - Job Example 05A                                         */
/*                                                                      */
/***************************************************************************/
/* Description:                                                         */
/*                                                                      */
/* This job example uses multiple instances of the                      */
/* DataConnector Operator to read data from MQ via the MQ               */
/* Access Module and performs an insert operation against a             */
/* Teradata table on a continuous basis using multiple                  */
/* instances of the Stream operator.                                    */
/*                                                                      */
/***************************************************************************/

DEFINE JOB MQ_LOAD
DESCRIPTION 'Load a Teradata table using MQSeries'
(
DEFINE SCHEMA MQ_SCHEMA
(
Associate_Name CHAR(20),
DOJ INTEGER,
Designation VARCHAR(25),
Loan_Amount DECIMAL(5,2),
Martial_Status CHAR(1),
No_Of_Dependents BYTEINT
);
DEFINE OPERATOR MQ_READER
TYPE DATACONNECTOR PRODUCER
SCHEMA MQ_SCHEMA
ATTRIBUTES
(
VARCHARG PrivateLogName = 'dataconnector_log',
VARCHARG FileName = 'DD:DATA',

APPLY ('insert into mlin_013 values (:f1, :f2, :f3, :f4);')
TO OPERATOR (UPDATE_OPERATOR[2])

SELECT * FROM OPERATOR (INMOD_READER);
);
Script Example 5B: Continuous Loading of Transactional Data from JMS

Use this script example as a template for jobs that read data from JMS and perform continuous INSERT, UPDATE, and DELETE operations in a Teradata Database table.

For a discussion of usage considerations, see “Job Example 5: Continuous Loading of Transactional Data from JMS or MQ” on page 116.
DEFINE JOB JMS_LOAD
DESCRIPTION 'Load a Teradata table using JMS Access Module'
(
  DEFINE SCHEMA JMS_SCHEMA
  (  
     Associate_Name    CHAR(20),
     DOJ               INTEGER,
     Designation       VARCHAR(25),
     Loan_Amount       DECIMAL(5,2),
     Martial_Status    CHAR(1),
     No_Of_Dependents  BYTEINT
  );

  DEFINE OPERATOR JMS_READER
  TYPE DATACONNECTOR PRODUCER
  SCHEMA JMS_SCHEMA
  ATTRIBUTES
  (  
     VARCHAR PrivateLogName = 'dataconnector_log',
     VARCHAR FileName = 'TLB1',
     VARCHAR Format = 'Formatted',
     VARCHAR OpenMode = 'Read',
     VARCHAR AccessModuleName = '/opt/teradata/client/lib/libjmsam.so',
     VARCHAR AccessModuleInitStr = '-url file:///tmp -icf com.sun.jndi.fscontext.RefFSContextFactory
-qcf ivtQCF -qnm ivtQ -ckfile myChkpt.log
-flush yes -tmd no -transx 5 -latency 5
-rwait 1 -logfill yes -log mylog.log -tlf
yes -TRCL 4 mytrace.log -debug yes'
  );

  DEFINE OPERATOR STREAM_OPERATOR
  TYPE STREAM
  SCHEMA *
  ATTRIBUTES
  (  
     VARCHAR PrivateLogName = 'stream_log',
     VARCHAR TdpId = @jobvar_tdpid,
     VARCHAR UserName = @jobvar_username,
     VARCHAR UserPassword = @jobvar_password,
     VARCHAR LogTable = 'sanity_test_JMS_log',
     VARCHAR ErrorTable1 = 'sanity_test_JMS_result'
  );

  APPLY
  ('INSERT INTO sanity_test_JMS_result VALUES (:Associate_Name,
     :DOJ,
     :Designation,
     :Loan_Amount,
     :Martial_Status,
     :No_Of_Dependents);')
  TO OPERATOR (STREAM_OPERATOR[2])

  SELECT * FROM OPERATOR (JMS_READER[2]);
);
Script example 5C: Intermediate File Logging Using Multiple APPLY Clauses with Continuous Loading of Transactional Data

Use this script example as a template for jobs that read data from JMS and perform continuous INSERT, UPDATE, and DELETE operations in a Teradata Database table, while simultaneously loading the same data into an external flat file.

For a discussion of usage considerations, see “Job Example 5: Continuous Loading of Transactional Data from JMS or MQ” on page 116.

```sql
/******************************************************************************
/*                                                            */
/* Teradata Parallel Transporter                                  */
/* User Guide - Job Example 05C                                   */
/*                                                            */
/******************************************************************************
/* Description:                                                   */
/*                                                            */
/* This example script demonstrates reading data from a          */
/* single source and simultaneously writing to two different     */
/* targets through the use of multiple APPLY statements.         */
/*                                                            */
/* This example script uses multiple instances of the            */
/* DataConnector Operator to read data from a JMS Access Module.  */
/* The job uses two APPLY statements to write data:              */
/*                                                            */
/* The first APPLY statement performs intermediate file          */
/* logging using the DataConnector Operator as a file writer.    */
/*                                                            */
/* The second APPLY statement performs an insert operation        */
/* against a Teradata table on a continuous basis using          */
/* multiple instances of the Stream operator.                   */
/*                                                            */
/******************************************************************************

DEFINE JOB JMS_LOAD_INTERMEDIATE_FILE_LOGGING
DESCRIPTION 'JMS Access Module loading to a table with intermediate file
logging'
{
  DEFINE_SCHEMA JMS_SCHEMA
  {
    Associate_Name    CHAR(20),
    DOJ               INTEGER,
    Designation       VARCHAR(25),
    Loan_Amount       DECIMAL(5,2),
    Marital_Status    CHAR(1),
    No_Of_Dependents  BYTEINT
  };

  DEFINE_OPERATOR JMS_READER
  TYPE DATACONNECTOR PRODUCER
  SCHEMA JMS_SCHEMA
  ATTRIBUTES
  {
    VARCHAR PrivateLogName = 'dataconnector_producer_log',
    VARCHAR FileName = 'TLB1',
    VARCHAR Format = 'Formatted',
  }

  /* Teradata Parallel Transporter */
  /* User Guide - Job Example 05C */
  /**/
  /* Description: */
  /**/
  /* This example script demonstrates reading data from a */
  /* single source and simultaneously writing to two different */
  /* targets through the use of multiple APPLY statements. */
  /* This example script uses multiple instances of the */
  /* DataConnector Operator to read data from a JMS Access */
  /* Module. The job uses two APPLY statements to write data: */
  /* The first APPLY statement performs intermediate file */
  /* logging using the DataConnector Operator as a file writer. */
  /* The second APPLY statement performs an insert operation */
  /* against a Teradata table on a continuous basis using */
  /* multiple instances of the Stream operator. */
  /**/
  /*/*******************************************************************************/
```
VARCHAR OpenMode = 'Read',
VARCHAR AccessModuleName = '/opt/teradata/client/lib/libjmsam.so',
VARCHAR AccessModuleInitStr = '-url file:///tmp -icf

com.sun.jndi.fscontext.RefFSContextFactory
-qcf ivtQCF -qnm ivtQ-cfile myChkpt.log
-flush yes -tdm no -transx 5 -latency 5
-rwait 1 -logfill yes -log mylog.log -tlf
-TRCL 4 mytrace.log -debug yes'
);

DEFINE OPERATOR FILE_WRITER
TYPE DATACONNECTOR CONSUMER
SCHEMA *
ATTRIBUTES
(
    VARCHAR PrivateLogName = 'dataconnector_consumer_log',
    VARCHAR DirectoryPath = @jobvar_datafiles_path,
    VARCHAR FileName = 'intermediate_file_log.dat',
    VARCHAR Format = 'Delimited',
    VARCHAR OpenMode = 'Write',
    VARCHAR TextDelimiter = '~'
);

DEFINE OPERATOR STREAM_OPERATOR
TYPE STREAM
SCHEMA *
ATTRIBUTES
(
    VARCHAR PrivateLogName = 'stream_log',
    VARCHAR TdpId = @jobvar_tdpid,
    VARCHAR UserName = @jobvar_username,
    VARCHAR UserPassword = @jobvar_password,
    VARCHAR LogTable = 'sanity_test_JMS_log',
    VARCHAR ErrorTable1 = 'sanity_test_JMS_error'
);

APPLY TO OPERATOR (FILE_WRITER),

APPLY ('INSERT INTO JMS_result VALUES (:Associate_Name,
    :DOJ,
    :Designation,
    :Loan_Amount,
    :Martial_Status,
    :No_Of_Dependents);')

TO OPERATOR (STREAM_OPERATOR[2])

SELECT * FROM OPERATOR (JMS_READER[2]);
);
Script Example 6: Loading Data from Other Relational Databases into an Empty Teradata Database Table

Use this script example as a template for jobs that read data from a non-Teradata relational database, such as Oracle, SQL Server, and DB2, and load it into an empty Teradata Database table.

For a discussion of usage considerations, see “Job Example 6: Loading Data from Other Relational Databases” on page 119.

```
/* Teradata Parallel Transporter User Guide - Job Example 06 */
/* Description: */
/* This example script uses two job steps. */
/* The first job step, called "Setup_Tables", uses the DDL Operator to setup the target table. */
/* The second job step, called "Insert_Into_Table", uses the ODBC Operator to read data from an external relational DBMS DB2 and uses the Load Operator to write the data to an empty Teradata table. */

DEFINE JOB ODBC_LOAD
DESCRIPTION 'ODBC LOAD PRODUCT DEFINITION TABLE'
{
  DEFINE SCHEMA Sanity_Test_ODBC_Schema
  (
    Associate_Id     INTEGER,
    Associate_Name   CHAR(25),
    Salary           FLOAT,
    DOJ              CHAR(10),
    Designation      VARCHAR(25),
    Loan_Amount      DECIMAL(5,2),
    Marital_Status   CHAR(1),
    No_Of_Dependents SMALLINT
  );

  DEFINE OPERATOR DDL_Operator
  TYPE DDL
  ATTRIBUTES
  {
    VARCHAR PrivateLogName = 'ddl_log',
    VARCHAR TdpId           = @jobvar_tdpid,
    VARCHAR UserName        = @jobvar_username,
    VARCHAR UserPassword    = @jobvar_password,
    VARCHAR ARRAY ErrorList = ['3807','3803']
  }

  DEFINE OPERATOR ODBC_Operator
```

Teradata Parallel Transporter User Guide
DESCRIPTION 'Teradata Parallel Transporter ODBC Operator'
TYPE ODBC
SCHEMA Sanity_Test_ODBC_Schema
ATTRIBUTES
(
  VARCHAR PrivateLogName  = 'odbc_log',
  VARCHAR DSNName         = @jobvar_DB2Id,
  VARCHAR UserName        = @jobvar_DB2UserName,
  VARCHAR UserPassword    = @jobvar_DB2Password,
  VARCHAR SelectStmt     = 'Select Associate_Id, ||
                            Associate_Name, ||
                            Salary, ||
                            DOJ, ||
                            Designation, ||
                            Loan_Amount, ||
                            Marital_Status, ||
                            No_Of_Dependents ||
                            FROM odbc_test_source;'
);

DEFINE OPERATOR Load_Operator
TYPE LOAD
SCHEMA *
ATTRIBUTES
(
  VARCHAR PrivateLogName  = 'load_log',
  VARCHAR TdpId           = @jobvar_tdpid,
  VARCHAR UserName        = @jobvar_username,
  VARCHAR UserPassword    = @jobvar_password,
  VARCHAR TargetTable     = 'odbc_test_result',
  VARCHAR LogTable        = 'odbc_test_log',
  VARCHAR ErrorTable1     = 'odbc_test_error1',
  VARCHAR ErrorTable2     = 'odbc_test_error2'
);

Step Setup_Into_Tables
(
  APPLY
  ('drop table odbc_test_result_e1;' ),
  ('drop table odbc_test_result_e2;' ),
  ('drop table odbc_test_result;' ),
  ('create table odbc_test_result ( Associate_Id integer,
                                 Associate_Name char(25),
                                 Salary float,
                                 DOJ date,
                                 Designation varchar(25),
                                 Loan_Amount decimal(5,2),
                                 Marital_Status char(1),
                                 No_Of_Dependents smallint );')
);

Step Insert_Into_Tables
(
  APPLY
  CASE
    WHEN (Associate_Id > 0 and Associate_Id < 5)
THEN 'INSERT INTO odbc_test_result (  
  :Associate_Id,  
  :Associate_Name,  
  :Salary,  
  :DOJ,  
  :Designation,  
  :Loan_Amount,  
  :Marital_Status,  
  :No_Of_Dependents );'
END

TO OPERATOR (Load_Operator)

SELECT
  Associate_Id,
  CASE
    WHEN Associate_Name = 'INDISPENSABLE_WORKER_NO_1' THEN 'INDISPENSABLE_WORKER_xxx5'
    WHEN Associate_Name = 'INDISPENSABLE_WORKER_NO_2' THEN 'INDISPENSABLE_WORKER_xxx5'
    WHEN Associate_Name = 'INDISPENSABLE_WORKER_NO_3' THEN 'INDISPENSABLE_WORKER_xxx5'
    WHEN Associate_Name = 'INDISPENSABLE_WORKER_NO_4' THEN 'INDISPENSABLE_WORKER_xxx5'
    WHEN Associate_Name = 'INDISPENSABLE_WORKER_NO_5' THEN 'INDISPENSABLE_WORKER_xxx5'
  END AS Associate_Name,
  Salary,
  DOJ,
  Designation,
  Loan_Amount,
  Marital_Status,
  No_Of_Dependents

FROM OPERATOR (ODBC_Operator)
WHERE Associate_Id > 0 and Associate_Id < 10;
);

Script Example 7: Mini-Batch Loading into Teradata Database Tables

Use this script example as a template for jobs that read data from external flat files and load them into a Teradata Database staging table before inserting them into a Teradata Database production table.

**Note:** This approach is necessary where join indexes or other factors restrict the Load operator from direct, high speed loading of the production table. Therefore it loads the staging table and then uses DDL operator (which has no such restrictions) to transfer them to the production table.

For details on mini-batch setup, see “Mini-Batch Loading” on page 375.

For a discussion of usage considerations, see “Job Example 7: Mini-Batch Loading” on page 120.
/* Teradata Parallel Transporter */
/* User Guide - Job Example 07 */

/* Description: */
/* This job example uses three job steps. */
/* The first job step, called "Step1", uses the DDL Operator to setup the target tables. */
/* The second job step, called "Step2", uses multiple instances of the DataConnector Operator to read data from a flat file and uses multiple instances of the Load to write data to an empty Teradata table. */
/* The third job step, called "Step3", uses the DDL Operator to issue Mini-Batch SQL Statements. */

DEFINE JOB MINI_BATCH
DESCRIPTION 'Mini-Batch Loading'
{
  DEFINE SCHEMA PRODUCT_SOURCE_SCHEMA
  {
    Associate_Id integer,
    Associate_Name char(25),
    Salary float,
    DOJ ansidate,
    Designation varchar(25),
    Loan_Amount decimal(5,2),
    Martial_Status char(1),
    No_Of_Dependents byteint
  };

  DEFINE OPERATOR DDL_OPERATOR
  TYPE DDL
  ATTRIBUTES
  {
    VARCHAR PrivateLogName = 'ddl_log',
    VARCHAR TdpId = @jobvar_tdpid,
    VARCHAR UserName = @jobvar_username,
    VARCHAR UserPassword = @jobvar_password,
    VARCHAR ARRAY ErrorList = ['3807','3803','5980']
  };

  DEFINE OPERATOR FILE_READER
  TYPE DATACONNECTOR PRODUCER
  SCHEMA PRODUCT_SOURCE_SCHEMA
  ATTRIBUTES
  {
    VARCHAR PrivateLogName = 'dataconnector_log',
    VARCHAR DirectoryPath = @jobvar_datafiles_path,
    VARCHAR FileName = 'accounts.txt',
    VARCHAR Format = 'Delimited',
  };
}
VARCHAR OpenMode       = 'Read',
VARCHAR TextDelimiter  = '|' ;

DEFINE OPERATOR LOAD_OPERATOR
TYPE LOAD
SCHEMA *
ATTRIBUTES
(
VARCHAR PrivateLogName = 'load_log',
VARCHAR TdpId          = @jobvar_tdpid,
VARCHAR UserName       = @jobvar_username,
VARCHAR UserPassword   = @jobvar_password,
VARCHAR TargetTable    = 'DDL046_stg',
VARCHAR LogTable       = 'DDL046_stg_log',
VARCHAR ErrorTable1    = 'DDL046_stg_e1',
VARCHAR ErrorTable2    = 'DDL046_stg_e2'
);

STEP STEP1
(
APPLY
('DROP TABLE DDL046_src;'),
('CREATE TABLE DDL046_src (Associate_Id     integer,
Associate_Name   char(25),
Salary           float,
DOJ              date,
Designation      varchar(25),
Loan_Amount      decimal(5,2),
Martial_Status   char(1),
No_Of_Dependents byteint);'),
('ins DDL046_src( 1,''Hard_Work_always_Pays_001'',10.12,''99-03-20'',''
Dedication_Plus_Sincer001'',110.12,''a'',1);',
ins DDL046_src( 2,''Hard_Work_always_Pays_002'',11.12,''99-03-21'',''
Dedication_Plus_Sincer002'',111.12,''b'',2);',
ins DDL046_src( 3,''Hard_Work_always_Pays_003'',12.12,''99-03-22'',''
Dedication_Plus_Sincer003'',112.12,''c'',3);',
ins DDL046_src( 4,''Hard_Work_always_Pays_004'',13.12,''99-03-23'',''
Dedication_Plus_Sincer004'',113.12,''d'',4);',
ins DDL046_src( 5,''Hard_Work_always_Pays_005'',14.12,''99-03-24'',''
Dedication_Plus_Sincer005'',114.12,''e'',5);',
ins DDL046_src( 6,''Hard_Work_always_Pays_006'',15.12,''99-03-25'',''
Dedication_Plus_Sincer006'',115.12,''f'',6);',
ins DDL046_src( 7,''Hard_Work_always_Pays_007'',16.12,''99-03-26'',''
Dedication_Plus_Sincer007'',116.12,''g'',7);',
ins DDL046_src( 8,''Hard_Work_always_Pays_008'',17.12,''99-03-27'',''
Dedication_Plus_Sincer008'',117.12,''h'',8);',
ins DDL046_src( 9,''Hard_Work_always_Pays_009'',18.12,''99-03-28'',''
Dedication_Plus_Sincer009'',118.12,''i'',9);
Appendix A: Job Script Examples

Job Script Example Library

430 Teradata Parallel Transporter User Guide

ins DDL046_src(10,'"Hard_Work_always_Pays_010"',19.12,'"99-03-29"'),
  ('Dedication_Plus_Sincer010',119.12,'"j"',10));
('DROP TABLE DDL046_stg;'),
('CREATE TABLE DDL046_stg (Associate_Id integer,
  Associate_Name char(25),
  Salary float,
  DOJ date,
  Designation varchar(25),
  Loan_Amount decimal(5,2),
  Martial_Status char(1),
  No_Of_Dependents byteint);')

TO OPERATOR (DDL_OPERATOR);

STEP STEP2
{
  APPLY ('INS INTO DDL046_stg (:Associate_Id,
    :Associate_Name,
    :Salary,
    :DOJ,
    :Designation,
    :Loan_Amount,
    :Martial_Status,
    :No_Of_Dependents);')
  TO OPERATOR (LOAD_OPERATOR[2])
  SELECT * FROM OPERATOR (FILE_READER[2]);
};

STEP STEP3
{
  APPLY
    ('DROP ERROR TABLE FOR DDL046_tar;'),
    ('DROP TABLE DDL046_tar;'),
    ('CREATE TABLE DDL046_tar (Assoc_Id integer,
      Assoc_Name char(25),
      Sal float,
      DOJ date,
      Desig varchar(25),
      Loan_Amt decimal(5,2),
      Martial_Stat char(1),
      No_Of_Deps byteint);'),
    ('CREATE INDEX index1 (Assoc_Id,Assoc_Name) on DDL046_tar;'),
    ('CREATE ERROR TABLE DDL046_etar for DDL046_tar;'),
    ('INSERT INTO DDL046_tar select * from DDL046_stg;'),
    ('MERGE INTO DDL046_tar USING DDL046_stg on Assoc_Id=Associate_Id
    WHEN MATCHED THEN UPDATE SET Sal=Salary+500 WHEN NOT MATCHED
    THEN INSERT (Associate_Id,Associate_Name,Salary,DOJ,
      Designation,Loan_Amount,Martial_Status,No_Of_Dependents);')
  TO OPERATOR (DDL_OPERATOR);
};
Script Example 8: Batch Directory Scan

Use the following script as a template for jobs that scan an external directory to read flat files before loading the data into an empty Teradata Database table.

Note: For loading into a populated Teradata Database table, substitute the Update operator for the Load operator in the following example and perform upsert operations.

For details on mini-batch setup, see “Batch Directory Scan” on page 375.

For a discussion of usage considerations, see “Job Example 8: Batch Directory Scan” on page 122.

```sql
/* Teradata Parallel Transporter User Guide - Job Example 08 */
/* Description: */
/* This job example uses multiple instances of the DataConnector Operator to read all files from a directory (Batch Directory Scan) and uses multiple instances of the Update Operator to write the data to a Teradata table. */

DEFINE JOB BATCH_DIRECTORY_SCAN
DESCRIPTION 'Batch directory scanning'
{
  DEFINE SCHEMA PRODUCT_SOURCE_SCHEMA
  {
    IN_RECV_DT   VARCHAR(10),
    IN_RTL_CO_NUM VARCHAR( 6),
    IN_STR_NUM   VARCHAR(11),
    IN_RECV_TMSTMP VARCHAR(23),
    IN_RECV_TYPE  VARCHAR( 4),
    IN_RECV_DOC   VARCHAR(10),
    IN_USER_ID    VARCHAR(11),
    IN_ITEM_UPC_CD VARCHAR(15),
    IN_RECV_QTY   VARCHAR(11)
  };

  DEFINE OPERATOR FILE_READER
  TYPE DATACONNECTOR PRODUCER
  SCHEMA PRODUCT_SOURCE_SCHEMA
  ATTRIBUTES
  {
    VARCHAR PrivateLogName    = 'dataconnector_log',
    VARCHAR DirectoryPath     = 'temp',
    VARCHAR FileName          = '*',
    VARCHAR Format            = 'Delimited',
    VARCHAR OpenMode          = 'Read',
    VARCHAR ArchiveDirectoryPath = 'archive'
  };

  DEFINE OPERATOR UPDATE_OPERATOR
```

Teradata Parallel Transporter User Guide
Script Example 9: Active Directory Scan

Use the following script as a template for jobs that scan transactional data being continuously loaded into two external flat files, reading only the new data from each file and then writing it to a Teradata Database table.

For details on Active Directory Scan setup, see “Active Directory Scan: Continuous Loading of Transactional Data” on page 376.

For a discussion of usage considerations, see “Job Example 9: Active Directory Scan” on page 124.

/**
 * Teradata Parallel Transporter
 * User Guide - Job Example 09
 */
/**
 * This example script uses two job steps.
 */
/**
 * The first job step, called "setup_tables", uses the DDL
 */
/* Operator to setup the target table.                        */
/*                                                            */
/* The second job step, called "load_records", uses one copy  */
/* of the DataConnector Operator to continuously read a set   */
/* of files from a directory (Active Directory Scan) and      */
/* uses another copy of the DataConnector Operator to        */
/* continuously read a different set of files from a         */
/* directory (Active Directory Scan).                        */
/*                                                            */
/* The UNION ALL operation combines the data into a single   */
/* data stream that is then applied to the Stream Operator to */
/* write that data to a Teradata table.                      */
/*                                                            */
/**************************************************************/

DEFINE JOB ACTIVEDIRECTORY_SCAN
DESCRIPTION 'Active directory scanning'
{
  DEFINE SCHEMA T2_SOURCE_SCHEMA
  {
    A_IN_ID1 VARCHAR(4),
    A_IN_C1  VARCHAR(10),
    A_IN_C2  VARCHAR(10),
    A_IN_C3  VARCHAR(10)
  };

  DEFINE SCHEMA T3_SOURCE_SCHEMA
  {
    B_IN_ID1 VARCHAR(4),
    B_IN_C1  VARCHAR(10),
    B_IN_C2  VARCHAR(10),
    B_IN_C3  VARCHAR(10)
  };

  DEFINE SCHEMA INDICATOR
  {
    IND CHAR(1)
  };

  DEFINE OPERATOR DDL_OPERATOR
  TYPE DDL
  ATTRIBUTES
  {
    VARCHAR PrivateLogName = 'ddl_log',
    VARCHAR TdpId          = @jobvar_tdpid,
    VARCHAR UserName       = @jobvar_username,
    VARCHAR UserPassword   = @jobvar_password,
    VARCHAR ErrorList      = '3807'
  };

  DEFINE OPERATOR FILE_READER_A
  TYPE DATACONNECTOR PRODUCER
  SCHEMA T2_SOURCE_SCHEMA
  ATTRIBUTES
  {
    VARCHAR PrivateLogName = 'M1.dtaclog',
    VARCHAR FileName,
    VARCHAR OpenMode       = 'Read',
    VARCHAR DirectoryPath,
VARCHAR ArchiveDirectoryPath = @jobvar_archive_dir,
VARCHAR IndicatorMode,
VARCHAR TraceLevel       = 'MILESTONES',
VARCHAR Format           = 'Delimited',
INTEGER VigilWaitTime    = 5, /* seconds */
INTEGER VigilElapsedTime = 1 /* minutes */
);

DEFINE OPERATOR FILE_READER_B
TYPE DATACONNECTOR PRODUCER
SCHEMA T3_SOURCE_SCHEMA
ATTRIBUTES
  (  
    VARCHAR PrivateLogName       = 'M2.dtaclog',
    VARCHAR FileName,
    VARCHAR OpenMode             = 'Read',
    VARCHAR DirectoryPath,
    VARCHAR ArchiveDirectoryPath = @jobvar_archive_dir,
    VARCHAR IndicatorMode,
    VARCHAR TraceLevel           = 'MILESTONES',
    VARCHAR Format               = 'Delimited',
    INTEGER VigilWaitTime        = 10, /* seconds */
    INTEGER VigilElapsedTime     = 1 /* minutes */
  );

DEFINE OPERATOR STREAM_OPERATOR
TYPE STREAM
SCHEMA *
ALLOW LATENCY CHECK
ATTRIBUTES
  (  
    VARCHAR PrivateLogName         = 'M2.strlog',
    VARCHAR PauseAcq               = 'N',
    INTEGER ErrorLimit             = 100,
    INTEGER BufferSize             = 64,
    INTEGER TenacityHours          = 2,
    INTEGER TenacitySleep          = 1,
    INTEGER MaxSessions            = 20,
    INTEGER MinSessions            = 1,
    INTEGER Pack                   = 20,
    INTEGER PackMaximum,
    VARCHAR Robust,
    VARCHAR TargetTable,
    VARCHAR TraceLevel             = 'NONE',
    VARCHAR TdpId                   = @jobvar_tdpid,
    VARCHAR UserName               = @jobvar_username,
    VARCHAR UserPassword           = @jobvar_password,
    VARCHAR AccountID,
    VARCHAR ErrorTable1            = 'M2_e1',
    VARCHAR ErrorTable2            = 'M2_e2',
    VARCHAR WorkTable              = 'M2_wt',
    VARCHAR LogTable               = 'M2_log',
    VARCHAR WorkingDatabase        = @jobvar_wrk_dbname
  );

STEP setup_tables
(  
  APPLY ('drop table M2_e1;'),
  ('drop table M2_e2;'),
('drop table M2_log;'),
('drop table M2_wt;'),
('drop table T2;'),
('drop table T3;'),
('CREATE SET TABLE T2 (c1 INTEGER NOT NULL,
c2 CHAR(10) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
c3 INTEGER NOT NULL)
PRIMARY INDEX (c1);'),
('CREATE SET TABLE T3 (c1 INTEGER NOT NULL,
c2 CHAR(10) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,
c3 INTEGER NOT NULL)
PRIMARY INDEX (c1);')
TO OPERATOR (DDL_OPERATOR());
);

STEP Load_Records
{
    APPLY CASE
    WHEN (IND = 'A')
    THEN ('INSERT INTO T2 (:A_IN_C1,:A_IN_C2,:A_IN_C3);')
    WHEN (IND = 'B')
    THEN ('INSERT INTO T3 (:B_IN_C1,:B_IN_C2,:B_IN_C3);')
    END
TO OPERATOR (STREAM_OPERATOR() [1])

SELECT 'A' as IND, A_IN_ID1, A_IN_C1, A_IN_C2, A_IN_C3,
CAST(NULL AS VARCHAR(4)) AS B_IN_ID1,
CAST(NULL AS VARCHAR(10)) AS B_IN_C1,
CAST(NULL AS VARCHAR(10)) AS B_IN_C2,
CAST(NULL AS VARCHAR(10)) AS B_IN_C3
FROM OPERATOR (FILE_READER_A[1]
    ATTR(FileName = 't2*,' DirectoryPath = @jobvar_datafiles_path))

UNION ALL

SELECT 'B' as IND,
    CAST(NULL AS VARCHAR(4)) AS A_IN_ID1,
    CAST(NULL AS VARCHAR(10)) AS A_IN_C1,
    CAST(NULL AS VARCHAR(10)) AS A_IN_C2,
    CAST(NULL AS VARCHAR(10)) AS A_IN_C3,
    B_IN_ID1, B_IN_C1, B_IN_C2, B_IN_C3
FROM OPERATOR (FILE_READER_B[1]
    ATTR(FileName = 't3*,' DirectoryPath = @jobvar_datafiles_path));
} ;
Script Example 10: Extracting Rows and Writing Them in Delimited Format

Use this script example as a template for jobs that read rows from a Teradata Database table and write them to a non-Teradata target file as delimited data.

For a discussion of usage considerations, see “Job Example 10: Extracting Rows and Sending Them in Delimited Format” on page 132.

```sql
/* Teradata Parallel Transporter User Guide - Job Example 10 */
/* Description: */
/* This job example uses the SQL Selector Operator to read */
/* data from a Teradata table and the DataConnector Operator */
/* to write to a flat file as delimited data. */

DEFINE JOB EXPORT_DELIMITED_FILE
DESCRIPTION 'Export rows from a Teradata table to a delimited file'
(
  DEFINE SCHEMA PRODUCT_SOURCE_SCHEMA
  (COL1 VARCHAR(8),
   COL2 VARCHAR(10))
);

DEFINE OPERATOR SQL_SELECTOR
  TYPE SELECTOR
  SCHEMA PRODUCT_SOURCE_SCHEMA
  ATTRIBUTES
  (VARCHAR PrivateLogName = 'selector_log',
   VARCHAR TdpId = @jobvar_tdpid,
   VARCHAR UserName = @jobvar_username,
   VARCHAR UserPassword = @jobvar_password,
   VARCHAR SelectStmt = 'select * from KORMLCH_TBL3',
   VARCHAR ReportModeOn)
);

DEFINE OPERATOR FILE_WRITER
  TYPE DATACONNECTOR CONSUMER
  SCHEMA *
  ATTRIBUTES
  (VARCHAR PrivateLogName = 'dataconnector_log',
   VARCHAR DirectoryPath = @jobvar_datafiles_path,
   VARCHAR FileName = 'test_file.dat',
   VARCHAR Format = 'Delimited',
   VARCHAR OpenMode = 'Write',
   VARCHAR TextDelimiter = '~')
);

APPLY TO OPERATOR (FILE_WRITER)
```
Script Example 11: Export Rows and Write Them as Binary or Indicator Mode Data

Use this script example as a template for jobs that read rows from a Teradata Database table and write them to a non-Teradata target file as binary or indicator-mode data.

For a discussion of usage considerations, see “Job Example 11: Extracting Rows and Sending Them in Indicator-mode Format” on page 133.

```
/**************************************************************/
/*                                                            */
/* Teradata Parallel Transporter                             */
/* User Guide - Job Example 11                               */
/*                                                            */
/**************************************************************/
/* Description:                                               */
/*                                                            */
/* This job example uses multiple instances of the Export     */
/* Operator to read data from a Teradata table and uses the   */
/* DataConnector Operator to write the data to a file in      */
/* indicator format.                                          */
/*                                                            */
/**************************************************************/

DEFINE JOB EXPORT_TO_FILE_INDICATOR_FORMAT
DESCRIPTION 'Export rows from a Teradata table to a file in indicator mode'
(
DEFINE SCHEMA PRODUCT_SOURCE_SCHEMA
(
Associate_Id     INTEGER,
Associate_Name   CHAR(25),
Salary           FLOAT,
DOJ              INTDATE,
Designation      VARCHAR(25),
Loan_Amount      DECIMAL(5,2),
Martial_Status   CHAR(1),
No_Of_Dependents BYTEINT
);
DEFINE OPERATOR EXPORT_OPERATOR
TYPE EXPORT
SCHEMA PRODUCT_SOURCE_SCHEMA
ATTRIBUTES
(
VARCHAR PrivateLogName = 'export_log',
VARCHAR TdpId          = @jobvar_tdpid,
VARCHAR UserName       = @jobvar_username,
VARCHAR UserPassword   = @jobvar_password,
VARCHAR SelectStmt     = 'SELECT * FROM TEMP18 ORDER BY 1';
);
DEFINE OPERATOR FILE_WRITER
TYPE DATACONNECTOR CONSUMER
);
### Script Example 12: Export Data and Process It with an OUTMOD Routine

Use this script example as a template for jobs that read rows from a Teradata Database table and send them to an OUTMOD for processing before writing the data to an external target.

See “Job Example 12: Export Data and Process It with an OUTMOD Routine” on page 135.

```sql
/* Teradata Parallel Transporter */
/* User Guide - Job Example 12 */
/* Description: */
/* This job example uses multiple instances of the Export Operator to read data from a Teradata table and uses the FastExport OUTMOD Adapter Operator to write the data to an OUTMOD. */

DEFINE JOB EXPORT_OUTMOD
DESCRIPTION 'Export rows from a Teradata table to an OUTMOD'
{
  DEFINE SCHEMA PRODUCT_SOURCE_SCHEMA
  {
    COL1 int,
    COL2 int
  };

  DEFINE OPERATOR EXPORT_OPERATOR
  TYPE EXPORT
  SCHEMA PRODUCT_SOURCE_SCHEMA
  ATTRIBUTES
  {
    VARCHAR PrivateLogName = 'export_log',
    VARCHAR TdpId = @jobvar_tdpid,
    VARCHAR UserName = @jobvar_username,
    VARCHAR UserPassword = @jobvar_password,
    VARCHAR SelectStmt = 'Select * From SAN012_src ORDER BY 1;'
  };

  APPLY TO OPERATOR (FILE_WRITER)
  SELECT * FROM OPERATOR (EXPORT_OPERATOR[2]);
}"
```
DEFINE OPERATOR FASTEXPORT_OUTMOD_ADAPTER
TYPE FASTEXPORT OUTMOD
SCHEMA *
ATTRIBUTES
{
    VARCHAR PrivateLogName = 'fastexport_outmod_log',
    VARCHAR FileName = 'SAN012.dat',
    VARCHAR Format = 'Formatted',
    VARCHAR OpenMode = 'Write',
    VARCHAR OutmodName = @jobvar_outmod_name
};

APPLY TO OPERATOR (FASTEXPORT_OUTMOD_ADAPTER)
SELECT * FROM OPERATOR (EXPORT_OPERATOR[2]);
);

**Script Example 13: Export Data and Process It with an Access Module**

Use this script example as a template for jobs that read rows from a Teradata Database table and send them to an access module for processing before writing them to an external target.

For a discussion of usage considerations, see “Job Example 13: Export Data and Process It with an Access Module” on page 136.

/**************************************************************/
/*                                                            */
/* Teradata Parallel Transporter                               */
/* User Guide - Job Example 13                                */
/*                                                            */
/**************************************************************/
/*                                                            */
/* Description:                                               */
/*                                                            */
/* This job example uses multiple instances of the Export      */
/* Operator to read data from a Teradata table and uses the    */
/* DataConnector Operator to write the data to a JMS access     */
/* module.                                                    */
/**************************************************************/

DEFINE JOB EXPORT_ACCESS_MODULE
DESCRIPTION 'Export rows from a Teradata table to an access module'
(
    DEFINE SCHEMA PRODUCT_SOURCE_SCHEMA
    {
        COL1 int,
        COL2 int
    };

    DEFINE OPERATOR EXPORT_OPERATOR
    TYPE EXPORT
    SCHEMA PRODUCT_SOURCE_SCHEMA
    ATTRIBUTES
    {
        VARCHAR PrivateLogName = 'export_log',
        VARCHAR TdpId = @jobvar_tdpid,
        VARCHAR UserName = @jobvar_username,
        VARCHAR UserName = @jobvar_username,
    };

    APPLY TO OPERATOR (EXPORT_OPERATOR)
VARCHAR UserPassword  = @jobvar_password,
VARCHAR SelectStmt     = 'Select * From SAN012_src ORDER BY 1;
)

DEFINE OPERATOR ACCESS_MODULE_WRITER
TYPE DATACONNECTOR CONSUMER
SCHEMA *
ATTRIBUTES
(  VARCHAR PrivateLogName      = 'dataconnector_log',
  VARCHAR FileName            = 'SAN012',
  VARCHAR Format              = 'Formatted',
  VARCHAR OpenMode            = 'Write',
  VARCHAR AccessModuleName    = '/opt/teradata/client/lib/libjmsam.so',
  VARCHAR AccessModuleInitStr = '-url file:///tmp -icf com.sun.jndi.fscontext.RefFSContextFactory
  -qcf ivtQCF -qnm ivtQ -ckfile myChkpt.log
  -flush yes -tdm no -transx 5 -latency 5
  -rwait 1 -logfill yes -log mylog.log -tlf
  yes -TRCL 4 mytrace.log -debug yes'
);

APPLY TO OPERATOR (ACCESS_MODULE_WRITER)
SELECT * FROM OPERATOR (EXPORT_OPERATOR[2]);

Script Example 14: Extract BLOB/CLOB Data and Write It to an External Target

Use this script example as a template for jobs that read BLOB/CLOB data from a Teradata Database table and write it to an external target file.

For a discussion of usage considerations, see "Job Example 14: Extract BLOB/CLOB Data and Write It to an External File" on page 137.

/*******************************************************************************/
/*                                                                       */
/* Teradata Parallel Transporter                                       */
/* User Guide - Job Example 14                                          */
/******* DESCRIPTION:                                                   */
/* This job example uses the SQL Selector Operator to read              */
/* large object (LOB) data from a Teradata table and uses               */
/* the DataConnector Operator to write the LOB data to an               */
/* external file.                                                      */
/*******/
/*******************************************************************************/

DEFINE JOB LOB_EXPORT_FILE
DESCRIPTION 'Export LOB data from a Teradata table to a file'
{
  DEFINE SCHEMA TEST_SCHEMA
  {
    COL1_INT  INTEGER,
}
Teradata Parallel Transporter User Guide

Appendix A: Job Script Examples

Job Script Example Library

Script Example 15: Extract Data from a Teradata Database Staging Table and Load It into a Production Table

Use this script example as a template for jobs that read rows from a Teradata Database staging table and then load them into an empty Teradata Database production table.

For a discussion of usage considerations, see “Job Example 15: Export Data from a Table and Load It into an Empty Table” on page 141.

/* *********************************************** */
/* * Teradata Parallel Transporter            */
/* * User Guide - Job Example 15            */
/* * *********************************************** */
/* * Description:                           */
/* * This job example uses two job steps.    */
/* * The first job step, called "Setup_Tables", uses the DDL */
/* Operator to setup the empty target table. */
/* */
/* The second job step, called "Load_Trans_Table", uses */
/* multiple instances of the Export Operator to read rows */
/* from a Teradata table and uses multiple instances of the */
/* Load Operator to write them into an empty Teradata table. */
/* */
/**************************************************************/

DEFINE JOB EXTRACT_TABLE_LOAD
DESCRIPTION 'Export rows from a Teradata table to another Teradata table'
{

DEFINE SCHEMA PRODUCT_SOURCE_SCHEMA
DESCRIPTION 'PRODUCT INFORMATION SCHEMA'
(
    Associate_Id     INTEGER,
    Associate_Name   CHAR(25),
    Salary           FLOAT,
    DOJ              INTDATE,
    Designation      VARCHAR(25),
    Loan_Amount      DECIMAL(5,2),
    Martial_Status   CHAR(1),
    No_Of_Dependents BYTEINT
);

DEFINE OPERATOR DDL_OPERATOR
TYPE DDL
ATTRIBUTES
(
    VARCHAR PrivateLogName = 'ddl_log',
    VARCHAR TdpId = @jobvar_tdpid,
    VARCHAR UserName = @jobvar_username,
    VARCHAR UserPassword = @jobvar_password,
    VARCHAR ErrorList = '3807'
);

DEFINE OPERATOR EXPORT_OPERATOR
TYPE EXPORT
SCHEMA PRODUCT_SOURCE_SCHEMA
ATTRIBUTES
(
    VARCHAR PrivateLogName = 'export_log',
    VARCHAR TdpId = @jobvar_tdpid,
    VARCHAR UserName = @jobvar_username,
    VARCHAR UserPassword = @jobvar_password,
    VARCHAR SelectStmt = 'SELECT * FROM TEMP18 ORDER BY 1;'
);

DEFINE OPERATOR LOAD_OPERATOR
TYPE LOAD
SCHEMA *
ATTRIBUTES
(
    VARCHAR PrivateLogName = 'load_log',
    VARCHAR TdpId = @jobvar_tdpid,
    VARCHAR UserName = @jobvar_username,
    VARCHAR UserPassword = @jobvar_password,
    VARCHAR TargetTable = @jobvar_tgt_dbname || '.Trans',
    VARCHAR LogTable = @jobvar_wrk_dbname || '.LG_Trans',
VARCHARG ErrorTable1 = @jobvar_wrk_dbname || '.ET_Trans',
VARCHARG ErrorTable2 = @jobvar_wrk_dbname || '.UV_Trans'
);

STEP Setup_Tables
{
  APPLY
  ('DROP TABLE   ' || @jobvar_wrk_dbname || '.ET_Trans;'),
  ('DROP TABLE   ' || @jobvar_wrk_dbname || '.UV_Trans;'),
  ('DROP TABLE   ' || @jobvar_tgt_dbname || '.Trans;'),
  ('CREATE TABLE ' || @jobvar_tgt_dbname
   || '.Trans (Trans_Number   VARCHAR(50),
          Trans_Date     VARCHAR(50),
          Account_Number VARCHAR(50),
          Trans_ID       VARCHAR(50),
          Trans_Amount   VARCHAR(50));')
  TO OPERATOR (DDL_OPERATOR);
};

STEP LOAD_Trans_Table
{
  APPLY TO OPERATOR (LOAD_OPERATOR[2])
  SELECT * FROM OPERATOR (EXPORT_OPERATOR[2]);
};

Script Example 16: Export Data and then Use It to Perform Conditional Updates Against Production Tables

Use this script example as a template for jobs that read rows from a Teradata Database table and then perform conditional updates against production tables.

For a discussion of usage considerations, see “Job Example 16: Export Data and then Use It to Perform Conditional Updates Against Production Tables” on page 143.

/**********************************************/
/*                                         */
/* Teradata Parallel Transporter          */
/* User Guide - Job Example 16             */
/*                                         */
/**********************************************/
/* Description:                           */
/*                                         */
/* This job example uses two job steps.    */
/*                                         */
/* The first job step, called "Load_to_Staging" uses the */
/* DataConnector Operator to read data from a flat file and */
/* writes that data to a staging table using multiple */
/* instances of the Update Operator.       */
/*                                         */
/* The second job step, called "Load_to_Target", uses */
/* multiple instances of the Export Operator to read data */
/* from the staging table and performs conditional updates */
/* against the target production table using multiple */
/* instances of the Update Operator.       */
/*                                         */
/***********************************************/
DEFINE JOB EXTRACT_TRANSFORM_LOAD
DESCRIPTION 'Load flat file to a staging table, then insert into 2 tables'
()
DEFINE SCHEMA In_Trans_n_Accts_Schema
(
    Account_Number VARCHAR(50),
    Number VARCHAR(50),
    Street VARCHAR(50),
    City VARCHAR(50),
    State VARCHAR(50),
    Zip_Code VARCHAR(50),
    Balance_Foward VARCHAR(50),
    Balance_Current VARCHAR(50),
    Trans_Number VARCHAR(50),
    Trans_Date VARCHAR(50),
    Trans_ID VARCHAR(50),
    Trans_Amount VARCHAR(50)
);

DEFINE SCHEMA Trans_n_Accts_Schema
(
    Trans_Number INTEGER,
    Trans_Date ANSI_DATE,
    Acct_Num_No_Xref INTEGER,
    Account_Number INTEGER,
    Trans_ID CHAR(4),
    Trans_Amount DECIMAL(10,2)
);

DEFINE OPERATOR FILE_READER
TYPE DATACONNECTOR PRODUCER
SCHEMA In_Trans_n_Accts_Schema
ATTRIBUTES
(
    VARCHAR PrivateLogName = 'dataconnector_log',
    VARCHAR DirectoryPath = @jobvar_datafiles_path,
    VARCHAR FileName = 'trans_and_accounts.txt',
    VARCHAR Format = 'Delimited',
    VARCHAR OpenMode = 'Read',
    VARCHAR TextDelimiter = '|' 
);

DEFINE OPERATOR EXPORT_OPERATOR
TYPE EXPORT
SCHEMA Trans_n_Accts_Schema
ATTRIBUTES
(
    VARCHAR PrivateLogName = 'export_log',
    VARCHAR TdpId = @jobvar_tdpid,
    VARCHAR UserName = @jobvar_username,
    VARCHAR UserPassword = @jobvar_password,
    VARCHAR SelectStmt = 'SELECT stg.Trans_Number ,stg.Trans_Date ,CASE WHEN acct.Account_Number IS NULL THEN 0 ELSE acct.Account_Number END (INTEGER) AS Acct_Num_No_Xref'
DEFINE OPERATOR UPDATE_OPERATOR
TYPE UPDATE
SCHEMA *
ATTRIBUTES {
  VARCHAR PrivateLogName = 'update_log',
  VARCHAR TdpId = @jobvar_tdpid,
  VARCHAR UserName = @jobvar_username,
  VARCHAR UserPassword = @jobvar_password,
  VARCHAR LogTable = @jobvar_wrk_dbname ||
    '.LG_Trans_n_Accts',
  VARCHAR ARRAY TargetTable = [@jobvar_wrk_dbname || '.Trans_n_Accts',
    @jobvar_wrk_dbname || '.Trans',
    @jobvar_wrk_dbname || '.Src_Exceptions'],
  VARCHAR ARRAY ErrorTable1 = [@jobvar_wrk_dbname ||
    '.ET_Trans_n_Accts',
    @jobvar_wrk_dbname || '.ET_Trans',
    @jobvar_wrk_dbname || '.ET_Src_Exceptions'],
  VARCHAR ARRAY ErrorTable2 = [@jobvar_wrk_dbname ||
    '.UV_Trans_n_Accts',
    @jobvar_wrk_dbname || '.UV_Trans',
    @jobvar_wrk_dbname || '.UV_Src_Exceptions']
};

STEP Load_to_Staging {
  APPLY ('INSERT INTO ' || @jobvar_tgt_dbname || '.Trans_n_Accts ( Account_Number
    ,Number
    ,Street
    ,City
    ,State
    ,Zip_Code
    ,Balance_Foward
    ,Balance_Current
    ,Trans_Number
    ,Trans_Date
    ,Trans_ID
    ,Trans_Amount
    ,stg.Account_Number
    ,stg.Trans_ID
    ,stg.Trans_Amount
  ) VALUES (
    :Account_Number
    ,:Number
    ,:Street
    ,:City
    ,:State
    ,:Zip_Code
    ,:Balance_Foward
    ,:Balance_Current
    ,:Trans_Number
    ,:Trans_Date
    ,:Trans_ID
    ,:Trans_Amount
    ,:m Account_Number;
  )
  )
}
TO OPERATOR (UPDATE_OPERATOR[2])

SELECT * FROM OPERATOR (FILE_READER);
)

STEP Load_to_Target
{
  APPLY
  CASE WHEN Acct_Num_No_Xref = 0
    THEN ('INSERT INTO ' || @jobvar_tgt_dbname || '.Src_Exceptions
      (
        Src_Sys
      ,Src_Obj
      ,Src_Name
      ,Src_Key_Name
      ,Src_Key_Value
      ,Tgt_Db_Name
      ,Tgt_Tbl_Name
      )
      VALUES
      (''SAP Transactions''
      ,''Flat File''
      ,''trans_and_accounts.txt''
      ,''Account_Number''
      ,:Account_Number
      ,''TPT''
      ,''Accounts''
      );')
    IGNORE DUPLICATE INSERT ROWS
  ELSE ('INSERT INTO ' || @jobvar_tgt_dbname || '.Trans
    (Trans_Number
    ,Trans_Date
    ,Account_Number
    ,Trans_ID
    ,Trans_Amount
    )
    VALUES
    (':Trans_Number
    ,':Trans_Date
    ,':Account_Number
    ,':Trans_ID
    ,':Trans_Amount
    );')
  END
}
Script Example 17: Bulk Delete of Data from a Teradata Database

Use this example script to delete a large quantity of data from a Teradata Database table.

For a discussion of usage considerations, see “Job Example 17: Bulk Delete of Data from a Teradata Database” on page 145.

```
/* Teradata Parallel Transporter User Guide - Job Example 17 */
/* Description: */
/* This job example uses the Update Operator as a standalone Operator to perform a bulk delete of data from a Teradata table based on a where condition. */
DEFINe JOB DELETE_DATA DESCRIPTION 'Bulk delete data from a table'
(  DEFINE OPERATOR UPDATE_OPERATOR
    TYPE UPDATE STANDALONE
    ATTRIBUTES
    (      VARCHAR PrivateLogName  = 'update_log',
            DeleteTask     = 'Y',
            TdpId          = @jobvar_tdpid,
            UserName       = @jobvar_username,
            UserPassword   = @jobvar_password,
            TargetTable    = 'Trans',
            LogTable       = 'LG_Trans',
            ErrorTable1    = 'ET_Trans',
            ErrorTable2    = 'UV_Trans'
    );
    APPLY ('DELETE FROM Trans WHERE Trans_Number < 100')
    TO OPERATOR (UPDATE_OPERATOR);
)
```

Script Example 18: Exporting BLOB/CLOB Data from One Teradata Database Table and Loading It into Another

Use the following script example as a template for jobs that move rows of data containing BLOB/CLOB data between two Teradata Database tables.
//**************************************************************/
/*                                                            */
/* Teradata Parallel Transporter                              */
/* User Guide - Job Example 18                                */
/*                                                            */
/********************** Teradata Parallel Transporter          */
/* User Guide - Job Example 18                               */
/*                                                            */
/****************************************************************/
/*                                                            */
/* Description:                                               */
/*                                                            */
/* This job example uses the SQL Selector Operator to read     */
/* large object (LOB) data from a Teradata table and uses      */
/* the SQL Inserter Operator to write the LOB data to another  */
/* Teradata table.                                            */
/*                                                            *//****************************************************************/

DEFINE JOB LOB_EXPORT_IMPORT
DESCRIPTION 'Export LOB from a table and Load into another table'
{
  DEFINE SCHEMA TEST_SCHEMA
  (
    COL1_INT INTEGER,
    COL2_INT INTEGER,
    COL3_CLOB CLOB(25000) AS DEFERRED BY NAME,
    COL4_BLOB BLOB(500) AS DEFERRED BY NAME,
    COL5_BLOB BLOB(29000) AS DEFERRED BY NAME
  );

  DEFINE OPERATOR SQL_SELECTOR
  TYPE SELECTOR
  SCHEMA TEST_SCHEMA
  ATTRIBUTES
  (
    VARCHAR PrivateLogName    = 'selector_log',
    VARCHAR TdpId             = @jobvar_tdpid,
    VARCHAR UserName          = @jobvar_username,
    VARCHAR UserPassword      = @jobvar_password,
    VARCHAR SelectStmt        = 'select * from tst5_src_tbl;',
    VARCHAR LobDirectoryPath  = @jobvar_lobpath,
    VARCHAR LobFileBaseName   = @jobvar_lobfilename,
    VARCHAR LobFileExtension  = @jobvar_lobextension
  );

  DEFINE OPERATOR SQL_INSERTER
  TYPE INSERTER
  SCHEMA *
  ATTRIBUTES
  (
    VARCHAR PrivateLogName = 'inserter_log',
    VARCHAR TdpId          = @jobvar_tdpid,
    VARCHAR UserName       = @jobvar_username,
    VARCHAR UserPassword   = @jobvar_password
  );

  APPLY TO OPERATOR (SQL_INSERTER)
  SELECT * FROM OPERATOR (SQL_SELECTOR);
};

The Teradata PT is GUI-based Windows application that simplifies the process of defining, modifying, and running simple load and export Teradata PT jobs that move data from a single source to a single destination.

Topics include:
- Launching TPT Wizard
- Overview
- Wizard Limitations
- Main Window
- Create a New Script
- Stop, Restart, Delete, Edit Jobs
- View Job Output
- Menus and Toolbars

Launching TPT Wizard

There are two ways to launch Teradata PT Wizard:
- From Start > Programs > FolderName > Teradata Parallel Transporter Wizard <version>
  The default value for FolderName is Teradata Client <version>
- From the desktop, double click Teradata Parallel Transporter Wizard <version> shortcut.

Overview

The basic workflow of the Wizard automatically creates the elements of a simple Teradata PT script. Following is a typical workflow, although variations to this flow often occur:

1. Type a job name and description.
2. Choose the data source.
3. Choose the data destination.
4. Map the source to a destination.
5. Select the type of processing, such as a simple insert or upsert, instead of using the Load, Update, or Stream operators.
Note: Depending on the data source, the Teradata PT Wizard uses the DataConnector, Export, or ODBC operators to extract data.

6 Generate job components.
7 Run the job.

The resulting script can be rerun, edited, or copied into another script.

When scripts are run, the following output is produced and displayed in various tabs on the main window:

- Click the Job Output tab to view a short synopsis of the job run.
- Click the Job Details tab to see a detailed table listing job instances. This table also shows the status of the running job and is updated dynamically as the job progresses.
- Click the Job Script tab to see the entire script. Each line has a line number contained in a comment.

Wizard Limitations

The Wizard has the following limitations:

- Jobs created on the Wizard can contain only a single job step.
- The job can only perform SELECTs, INSERTs, and UPSERTs. Wizard scripts must be manually altered to perform other functions.
- The Update and Stream operators can operate only against a single table. They only perform INSERTs, and UPSERTs.
Appendix B: Teradata PT Wizard

Main Window

The window of Teradata PT Wizard consists of two panes.

- A maximum of 450 columns can be defined in the source table or source record.
- The Wizard only supports the OCI driver type 2 of the Oracle JDBC driver.

Main Window

The left pane displays a directory structure (or job tree) with a node root named **Job**. Click on the Job root to display a list of previous jobs, along with a description in the right pane. Click a job name in the left pane to display the job summary.

As the session progresses and jobs are run, a history of job instances is built in the job tree in the left pane.

The right pane displays the name, description, and status of all jobs (or job nodes) that have been run.

Use the main window to run, delete, and modify jobs as follows:

- Click a job object in the job tree to see a description of the job, including the source and destination for the job, in the right pane.
Right-click a job name in the job tree to open a menu with options to edit the job (such as changing the data source or destination), to delete the job completely, or to rerun (re-submit) the job.

Click the plus sign (+) to display each instance of a job. Each time a job is run, an instance is created with one of the following icons:

- Fatal error
- Failed job
- Successful job
- Job is currently running

Click a run instance to view the job output, details, and job script for a specific instance.
Create a New Script

To create a new job script with the Wizard, do the following:

- Step 1 - Name the Job
- Step 2 - Select a Source and Select Data
- Step 3 - Select a Destination
- Step 4 - Run the Job

Step 1 - Name the Job

To name a new job

1. In the left pane of the main window, click Job to activate the new job icon. Do one of the following to open the Job Name/Description dialog box:
   - Click Edit > New.
   - Click the New Job icon.
   - Right-click, and click New.
   - Press Ctrl+N.

2. In the Job Name/Description dialog box, type a name and description for the job using no more than 128 characters.
The job name must start with a letter (upper or lower case) followed by a zero or more letters, and may contain digits (0-9). An underscore is also valid. If the text turns red during typing, the name does not meet these requirements. The following message appears when the Next button is clicked:

**Note**: When a job name is changed, Teradata PT Wizard creates a new job script with the new job name. The script with the old job name still exists.

The job description can remain blank; having a job description is not required. But like the job name, it appears in three places:

- In the second column next to the job name when the **Job** root is clicked in the left pane
- As the second line in the job summary
- In the Description statement in the job script

The job name and description can be changed when the job is edited.

3  (Optional) Click **Character Set** to change the language.

Teradata PT allows all character sets as long as they are supported by the Teradata Database. The default is the character set of the active platform; however, scripts and log output are in English only.

The default character sets for all Teradata PT jobs are:

- ASCII for network-attached client systems
- EBCDIC for channel-attached client systems

For information on extended character sets, see *Teradata Parallel Transporter User Guide* and *International Character Set Support*.

4  Click **Next** to open the **Job Source** dialog box.
Continue with Step 2 - Select a Source and Select Data.

**Step 2 - Select a Source and Select Data**

Use one of the following procedures, depending on the data source for the job.

- **Teradata Table as a Data Source**
- **File as a Data Source**
- **Oracle Table as a Data Source**
- **ODBC-Compliant Database as a Data Source**

**Logging onto a Data Source**

When using a Teradata table, an Oracle table or an ODBC-compliant database as a data source, a Logon dialog box appears to prompt for name, User ID and Password.

The Logon dialog box appears when creating a new script or editing an existing script. Logon information can be included in the Wizard scripts.

After supplying this information, the Teradata PT Wizard attempts to log on. If the connection can not be made, a message appears.

When running existing scripts, if the logon information has not been included in a script that has been submitted to run, information can be entered in the JobAttributes panel in the Run dialog box, as shown under step 4 on page 472.

**Teradata Table as a Data Source**

Use the Teradata Table option from the Job Source dialog box to log onto your Teradata system.

Then select a specific table as a data source for a job.
The Teradata Logon dialog box appears, optionally allowing the User ID and Password to be included in the Wizard job.

To export data from a Teradata table

1. From the Source Type list in the Job Source dialog box, select Teradata Table.
2. In the Teradata Logon dialog box, type the host name, user ID, and password to log on to your Teradata system.
3. (Optional) Select the check boxes to include your user ID and password in the generated scripts. The default is to enter placeholders.
4. Click OK.

The Job Source dialog box displays the directory structure of the Teradata system you logged onto.
In the left pane, select a database and a table to be the data source for the job.

**Caution:** Do not select tables that contain character large object (CLOB) or binary large object (BLOB) data types.

In the right pane, select up to 450 columns to include in the source schema, or click **Select All** or **Select None**. (Press Ctrl+click to select multiple columns.)

If a column name from a source table is a Teradata PT reserved word, the Teradata PT Wizard appends the phrase “_#” (where # is a numeric) so that the name differs from the keyword and the submitted script does not receive a syntax error.

For example, if the keyword DESCRIPTION is used as a column name, the name is changed to DESCRIPTION_1. Teradata PT keeps an internal counter for generating the appended number.

For the complete list of Teradata PT reserved words, see *Teradata Parallel Transporter Reference*.

**Note:** The values under **TPT Type** are names of the data types associated with the Teradata PT columns. The values under **DBS Type** are the data types from the source database. When Teradata PT gets a column name from a source table, it looks at the definition schema of the table to determine an accurate data type. Sometimes these types can be recorded incorrectly or as a “?” when the Wizard cannot properly determine the data type. This often occurs when reading user-defined data types (UDTs).
To change or correct a Teradata PT data type, click **Edit Type** (or right-click), and select the correct data type from the shortcut menu. Enter the length, precision, and scale if applicable. The precision and scale data types are only available when **Decimal/Numeric** is selected.

7. Click **Next** to open the **Job Destination** dialog box.

![Define Job Wizard](image)

### Job Destination

Select the type of the destination of the job's data. Then fill in the information for the selected data destination type.

**Destination Specification**

**Destination Type:**

8. Continue with **Step 3 - Select a Destination**.

**File as a Data Source**

Use the **File** option from the **Job Source** dialog box to browse for a flat file to use as the data source for a job.

---

**To export data from a flat file**

1. From the **Source Type** list in the **Job Source** dialog box, select **File**.

2. Do one of the following:
   - In **Directory Name** and **File Name**, type the path and name of the file to be used as the data source for the job.
   - Click **Select** to browse for the source file.
3 In **Format**, select either **Binary**, **Delimited**, **Formatted**, **Text**, or **Unformatted** as the format associated with the file.

For more information, see “Input File Formats”.

If specifying Delimited format, type the delimiter used in the source file into the **Delimiter** box. The wizard accepts delimiters up to 100 bytes in length. If no delimiter is provided, the TextDelimiter attribute defaults to the pipe character ( | ).

**Note:** When using a delimited flat file for input, all of the data types in the DEFINE SCHEMA must be VARCHARs. Defining non-VARCHAR data types results in an error when a job script is submitted to run.

4 (Optional) Select **Indicator Mode** to include indicator bytes at the beginning of each record. (Unavailable for delimited data.)

**Note:** If the file name contains a wildcard character (*), two additional input boxes are available. Type the number of minutes for a job to wait for additional data in the **Vigil Elapsed Time** box. Type the number of seconds to wait before Teradata PT checks for new data in **Vigil Wait Time** box.

5 Click **Next** to open the **Define Columns** dialog box.
In the **Define Columns** dialog box, specify the following, as needed:

- **Name** - Type the names of the columns in the source file.
- **Type** - Type the data type of each column. (Choices change depending on the type of format selected in the previous dialog box.)
  
  **Note:** When working with data from a file of delimited data, all fields must be defined as type VARCHAR.

- **Size** - Type the number of characters associated with each CHAR, VARCHAR, GRAPHIC, and VARGRAPHIC data types; and type the number of bytes associated with each BYTE and VARBYTE types. (All others are unavailable.)

(Optional) In **Number of Instances**, type the number of producer operator instances to process at the same time.

The **Precision** and **Scale** columns are only available for Decimal data types. Under **Precision**, type the number of digits to the left of the decimal point; under **Scale**, type the number of digits to the right of the decimal position. Otherwise, go to the next step.

After defining all the columns, click **Next** to open the **Job Destination** dialog box.

Continue with **Step 3 - Select a Destination**.
Oracle Table as a Data Source

Use the Oracle Table option from the Job Source dialog box to log onto an Oracle server and select a specific table as a data source. The Oracle Logon dialog appears, optionally allowing the User ID and Password to be included in the Wizard job.

To export data from an Oracle table

1. From the Source Type list in the Job Source dialog box, click Oracle Table.
2. At the logon prompt, type the TSN name (a net service name that is defined in a TNSNAMES.ORA file or in the Oracle directory server, depending on how the Oracle net service is configured on the Oracle client and server), user ID, and the password needed to build the Oracle JDBC connection.

Caution: The value you enter into the TSN Service Name box at logon is the value that the Wizard uses for the DSNname attribute in all scripts; however, systems are often configured with different values for the TSN Service Name and DSN name. If this is the case, you must manually edit the value of the DSNname attribute in scripts to match the TSN Service Name before submitting a job script that involves an Oracle server.

3. (Optional) Select the check boxes to include your user ID and password in the generated scripts. The default is to enter placeholders.
4. Click OK.

The Job Source dialog box displays the directory structure of the active Oracle server.

5. From the directory tree in the left pane, select a database and table that are the source of data for the job.

Caution: Do not select tables that contain character large object (CLOB) or binary large object (BLOB) data types.

6. In the right pane, select up to 450 columns to be included in the source schema, or click Select All or Select None.

Note: The values under TPT Type are names of the data types associated with the Teradata PT columns; the values under DBS Type are the data types from the source database. When Teradata PT gets a column name from a source table, it looks at the definition schema of the table to determine an accurate data type. Sometimes these types can be
recorded incorrectly or as a “?” when the Wizard cannot properly determine the data type. This often occurs when reading user-defined data types (UDTs).

To change or correct a Teradata PT data type, click Edit Type (or right-click), and select the correct data type from the shortcut menu. You can also enter the length, precision, and scale if it is applicable, but the precision and scale data types only appear when Decimal/Numeric is selected.

7 Click Next to open the Job Destination dialog box.

8 Continue with Step 3 - Select a Destination.

**ODBC-Compliant Database as a Data Source**

Use the ODBC DSN option from the Job Source dialog box to log onto an ODBC-compliant database. Then select a specific table as a data source for a job.

The ODBC Logon dialog box appears, optionally allowing the User ID and Password to be included in the Wizard job.

**To export data from an ODBC-compliant source**

1 From the Source Type list in the Job Source dialog box, select ODBC DSN.
2 In the ODBC Logon dialog box, type the host name, user ID, and password to log on.
3 (Optional) Select the check boxes to include your user ID and password in the generated scripts. The default is to enter placeholders.
4 Click OK.

The Job Source dialog box displays the database and table hierarchy of the ODBC-compliant data source you logged onto.
5 In the left pane, select a database and a table as the data source for the job.

**Caution:** Do not select tables that contain character large object (CLOB) or binary large object (BLOB) data types.

6 In the right pane, select up to 450 columns to be included in the source schema, or click **Select All** or **Select None**. (Press Ctrl+click to select multiple columns.)

**Note:** The values under **TPT Type** are names of the data types associated with the Teradata PT columns; the values under **DBS Type** are the data types from the source database. When Teradata PT gets a column name from a source table, it looks at the definition schema of the table to determine an accurate data type. Sometimes these types can be recorded incorrectly or as a “?” when the Wizard cannot properly determine the data type. This often occurs when reading user-defined data types (UDTs).

To change or correct a Teradata PT data type, click **Edit Type** (or right-click), and select the correct data type from the shortcut menu. You can also enter the length, precision, and scale if it is applicable, but the precision and scale data types are only available when **Decimal/Numeric** is selected.

7 Click **Next** to open the **Job Destination** dialog box.
Appendix B: Teradata PT Wizard
Create a New Script

8 Continue with Step 3 - Select a Destination.

**Step 3 - Select a Destination**

Regardless of whether the source for a job is a Teradata Database, a flat file, an ODBC-compliant source, or an Oracle database, the Wizard limits the load option in the *Job Destination* dialog box to the following:

- File as a Target
- Teradata Table as a Target

**File as a Target**

Use the *File* option in the *Job Destination* dialog box to export data to a flat file by using the following procedure.

**To load data to a file**

1 In *Destination Type* of the *Job Destination* dialog box, select *File*. 

---

**Define Job Wizard**

**Job Destination**

Select the type of the destination of the job's data. Then fill in the information for the selected data destination type.

**Destination Specification**

**Destination Type:** File

**File Specification**

**Directory Name:** C:\COURSES\Teradata Parallel Transporter\Test Files

**File Name:** Export_1000

**Output Mode**

- Create File
- Append File

**Format:** FORMATTED

---

**Define Job Wizard**

**Job Destination**

Select the type of the destination of the job's data. Then fill in the information for the selected data destination type.

**Destination Specification**

**Destination Type:** File

**File Specification**

**Directory Name:** C:\COURSES\Teradata Parallel Transporter\Test Files

**File Name:** Export_1000

**Output Mode**

- Create File
- Append File

**Format:** FORMATTED

---

Teradata Parallel Transporter User Guide
2 Do one of the following:
   • In **Directory Name**, type the directory that contains the destination file, then, in **File Name**, type the name of the destination file.
   • Click **Select** to browse for the destination file. If the file does not exist, type in the file name and press **Enter**. When the job script runs, the file will be created or appended, based on the option button choice made in the **Job Destination** dialog box’s **Output Mode**.

3 In the **Output Mode** group box, do one of the following:
   • Click **Create File** to export to an empty flat file.
   • Click **Append File** to add exported data to a file that already contains data.

4 In **Format**, select either **Binary**, **Delimited**, **Formatted**, **Text**, or **Unformatted** as the format associated with the destination file.

   **Note:** If the destination file is delimited, type the delimiter to be used in the file, up to 100 bytes, into the **Delimiter** box.

   When exporting delimited data, only VARCHAR columns can be exported from the source tables. If non-VARCHAR columns are needed, these steps must be done:
   a Convert these columns to VARCHAR.
   b Edit the values under the Teradata PT Type setting to VARCHAR for these columns. Do this by clicking Edit Type which is detailed in step 6 of the **File as a Data Source** procedure.
   c If needed, manually modify the SELECT statement in the attribute “SelectStmt” to cast non-VARCHAR columns to VARCHAR after generating the Wizard script.

5 (Optional) Select **Indicator Mode** to include indicator bytes at the beginning of each record. (Unavailable for delimited data.)

6 Click **Next** to open the **Finish Job** dialog box.

7 Continue with **Step 4 - Run the Job**.

**Teradata Table as a Target**

Use the **Teradata Table** option from the **Job Destination** dialog box to log onto your Teradata system, and to select a specific table as the destination for a job by using the following procedure.
To load data to a Teradata table

1. In Destination Type of the Job Destination dialog box, select Teradata Table.
2. In the Teradata Logon dialog box, type the host name, user ID, and password to log onto the target Teradata system.

3. (Optional) Select the check boxes to include your user ID and password in the generated scripts. The default is to enter placeholders.

4. Click OK to close the log on prompt and return to the Job Destination dialog box. For more information, see “Logging onto a Data Source” on page 455.

The directory structure and columns of the Teradata system are displayed. (The values are not editable.)

5. (Optional) In Number of Instances, type a number to designate the number of consumer operator instances to process at the same time.

6. Click Next to open the Operator Type Selection dialog box.
Select one of the following options depending on what Teradata PT operator or operation is to be used for the job. For more information about operators, see *Teradata Parallel Transporter Reference*.

- **Load Operator** - Use this option only if the destination table is empty; the job fails if it is not empty. This option transfers data much faster than the Update or Stream operators.
- **Update Operator** - Use this option to update an existing table regardless of whether it contains data. Selecting this option requires an additional selection of an insert or upsert operation.
- **Stream Operator** - Use this option to update a destination table from a source that generates constant data. Selecting this option requires an additional selection of an insert or upsert operation.
- **Insert Operation** - Use this option to copy data from the source to the destination.
- **Upsert Operation** - Selecting this option opens the *Update Row Specification* dialog box.
Use this option to select the destination columns that will get updated with data values from the source. Only the data values in the destination table that match the data values in the source are updated. When data does not match, a new row is created.

**Note:** At least one column must be selected, and at least one column must remain cleared.

8 Click **Next** to open the **Map Source to Destination** dialog box.

9 Click the **Automatic Mapping** button to map the first source column to the first destination column, the second source column to the second destination column, and so on.

- If the number of columns in **Source** is not the same as the number of columns in **Destination**, Teradata PT warns that it cannot map source columns to destination columns automatically and prompts you to map source and destination columns manually.
To map a source to a destination column manually, click a row in **Source** to open its drop-down list. Then select a data value for that source column row to map to a destination column. Note that one source column value can be mapped to multiple destination columns. Moreover, source columns can be left as **Column Not Mapped** as long as at least one column in the table is mapped.

- If the data types of the mapped columns are not the same, Teradata PT indicates that it cannot map source and destination columns and asks you to correct the mismatched data types.

- If you click **YES**, Teradata PT returns to the **Map Source to Destination** screen so you can map source to destination columns manually.

  To map a source to a destination column manually, click a row in **Source** to open its drop-down list. Then select a data value for that source column row to map to a destination column. One source column value can be mapped to multiple destination columns. Moreover, source columns can be left as **Column Not Mapped** as long as at least one column in the table is mapped.

- If you click **NO**, Teradata PT ignores the mismatched data types between columns.
and proceeds to map source and destination columns.

There are cases when the data types of the source and destination columns do not match, but the database can implicitly convert source column data types to destination column data types. For example, when the source column data type is smallint and the destination column data type is integer, the database can convert the source column data type to the destination column data type.

**Note:** To clear automatic mapping, click the **Clear Mapping** button. When you do, all automatic mapping is cleared and the **Clear Mapping** button is disabled. (That button is only enabled after the **Automatic Mapping** button is clicked.)

---

Once you have cleared automatic mapping, you can re-map source and destination columns automatically (by clicking the **Automatic Mapping** button) or manually (by clicking a row in **Source** to open its drop-down list and selecting a data value for that source column row to map to a destination column).

10 Click **Next** to open the **Finish Job** dialog box.
11 Continue with Step 4 - Run the Job.

**Step 4 - Run the Job**

The Finish Job dialog box displays a summary of the job.

---

**To run a job**

1 Decide to do one of the following:
   1. To run a new job, skip to step 4.
   2. To run a previously created job, continue with step 2.
   3. To save the job without running it (so you can run the script later), or to store the script (so you can copy it into another script), continue with step 2.

2 Review the job summary for accuracy, and do one of the following:
   1. To correct mistakes, click Back to return to the appropriate dialog box and make corrections.
   2. To store the job to be run or edited later, clear the Run Job Now option, and click Finish.
   3. To run the job now, select Run Job Now, then click Finish.

3 If you opted to run the job in Step 2, the Run Job dialog box opened. Otherwise, open the Run Job dialog box for a stored job now by right-clicking the job name in the job tree of the main Wizard window, and click Submit.
Appendix B: Teradata PT Wizard
Create a New Script

In **Job Name**, type the name of the job.

(Optional) In **Checkpoint Interval**, type the number of seconds between checkpoint intervals.

(Optional) In **Retry Limit**, type a positive number; the default value is 100. This option specifies how many times Teradata PT will automatically retry a job step after a restart. The **Retry Limit** option corresponds to the `tbuild` command's `-R` option.

(Optional) In **Latency Interval**, type the number of seconds until the Wizard flushes stale buffers.

**Note:** Currently, the Latency Interval option is available only for the Stream operator. For more information, see *Teradata Parallel Transporter Reference*.

(Optional) Select **Enable Trace** to enable the trace mechanisms.

If **Job Attributes** is available, type the name and password for the source table and the destination table. (This pane is available only if you did not select the two options to include the user ID and password in the generated script during log on. For information about these options, see “Teradata Table as a Target” on page 465 and “Oracle Table as a Data Source” on page 461.)

(Optional) View and edit the script before running it. Note that any changes made to the script will not be saved by the Wizard for the next use of the script. The changes will only apply for the current run when the OK button is clicked.

When you are ready to run the job, click **OK**.

While the job is running, the running icon 🔄 is displayed. When the job is complete, status can be viewed in the **Job Status** dialog box. For more information, see “View Job Output” on page 475.
Stop, Restart, Delete, Edit Jobs

Use the follow procedures to manage active jobs and jobs that have already been created in Teradata PT.

To stop a running job

1. At any point during the running of a job, select the run instance in the main window.
2. Do one of the following:
   - Click Jobs > Kill Job.
   - Click , which only available during job processing.
   - Press Ctrl+K.

To restart a job

1. From the main window, select a run instance in the job tree.
2. Do one of the following:
   - Click Jobs > Restart Job.
   - Click .
   - Right-click the job instance, then click Restart Job.
   - Press Ctrl+R.

The job begins from the point at which the job was stopped. Also see “To stop a running job” on page 473.

To delete a job

This procedure completely removes a job and all its instances from the Wizard.

1. From the main window, select a job name in the job tree.
2. Do one of the following:
   - Click Edit > Delete.
   - Click .
   - Right-click the job instance, then click Delete.
   - Press Ctrl+Shift+D.
3 A confirmation dialog box appears:

The left pane adjusts after clicking Yes.

---

**To delete a job instance**

This procedure completely removes a job instance from the Wizard.

1 From the main window, select a job instance in the job tree.

2 Do one of the following:
   - Click **Edit > Delete**.
   - Click .
   - Right-click the job instance, then click **Delete**.
   - Press Ctrl+Shift+D.

---

**To edit a previously created job**

1 In the job tree of the main window, do one of the following to open the **Job Name/Destination** dialog box, which allows editing:
   - Double-click the job name .
   - Right-click a job name, and click **Edit**.
   - Click **Edit > Edit**.
   - Click the Edit icon .
   - Press Ctrl+E.

2 Click **Next** to log on, and save your changes.

At this point, either close the script after modification, or continue to process the job to run it. To continue processing, start with “Step 2 - Select a Source and Select Data” on page 455.

---

**To run a previously created job**

1 In the job tree of the main window, do one of the following to open the **Run Job** window:
   - Double-click the job name .
   - Right-click the job name, and click **Submit**.
   - Click the Submit icon .
   - Press Ctrl+B.
View Job Output

Job output can be viewed in the following ways.

Job Status

Information about jobs is captured in the right pane of the main window as soon as a job starts. The three tabs in the right pane provide information about the status, output, and errors of each run instance of a job.

Job status information is also displayed when any run instance is clicked in the job tree.

Three tabs display the current state of a job:

- **Job Output** - Shows the name of the job and the job status. The Output box shows the results of the job run. The Errors box contains the location of the log file which includes errors that occurred during the run.

View the Teradata PT log at %SystemDrive%\Program Files\Teradata\Client\<version>\Teradata Parallel Transporter, or with the Wizard log viewer. See "Log Viewer" on page 476 for more information.
Appendix B: Teradata PT Wizard
View Job Output

- **Job Details** - Shows a table of job-related details. Columns include *Step Name, Task Name, Task Status,* and *Operator Name.* Use this tab to view a job as it runs. Each step is listed with its progress.
- **Job Script** - Shows the actual Teradata PT job script created by the Wizard for the specific job instance. The script can be copied into other scripts.

**Log Viewer**

Teradata PT keeps an extensive log of each job it runs. These logs are available in the *Log View Options* dialog box, which allows the selection of specific run instances.

**To view job logs**

1. In the job tree, do one of the following:
   - Select a run instance, then click *Job > View Log* on the menu bar.
   - Right-click a run instance, and click *View Log.*
2. Move field names to the right pane to include them in the job view; move field names to the left pane to remove them from the job view. To move field name, double-click them, or highlight a field and click *Add* or *Remove.*
3. (Optional) In the *Selection Criteria* box, add an SQL WHERE clause to narrow the amount of information that will be in the log.
4. Click *OK* to open the *View Log* dialog box with the information as requested.
Menus and Toolbars

The Teradata PT Wizard uses the following menu items. Many of these functions are also available by right-clicking icons in the main window and the job tree.

Table 37: Menu Items

<table>
<thead>
<tr>
<th>Menu</th>
<th>Menu choice</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Exit</td>
<td>Closes the wizard.</td>
</tr>
<tr>
<td>Edit</td>
<td>New</td>
<td>Creates a new job. See “Step 1 - Name the Job” on page 453.</td>
</tr>
<tr>
<td></td>
<td>Edit</td>
<td>Allows editing of the attributes of an existing job. See “To edit a previously created job” on page 474.</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>Deletes a job from the job tree, or deletes run instances from a specific job icon.</td>
</tr>
<tr>
<td></td>
<td>Refresh</td>
<td>Refreshes the wizard screen.</td>
</tr>
<tr>
<td>Jobs</td>
<td>Submit</td>
<td>Submits a selected job.</td>
</tr>
<tr>
<td></td>
<td>Kill Job</td>
<td>Stops the currently running job.</td>
</tr>
<tr>
<td></td>
<td>Restart Job</td>
<td>Restarts a run instance.</td>
</tr>
<tr>
<td></td>
<td>View Log</td>
<td>Opens the View Log Options dialog box.</td>
</tr>
<tr>
<td>Help</td>
<td>Teradata Parallel Transporter Help</td>
<td>Opens the online help.</td>
</tr>
<tr>
<td></td>
<td>About</td>
<td>Displays the active version of the Teradata PT Wizard.</td>
</tr>
</tbody>
</table>

Many of the following toolbar functions are also available by right-clicking a job instance in the job tree.

Table 38: Toolbar

<table>
<thead>
<tr>
<th>Buttons</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>New Job</td>
<td>Creates a new job. See “Create a New Script” on page 453.</td>
</tr>
<tr>
<td></td>
<td>Edit Item</td>
<td>Edits an existing job. See “To edit a previously created job” on page 474.</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>Deletes jobs or run instances from the Wizard. See “To delete a job” on page 473.</td>
</tr>
<tr>
<td></td>
<td>Kill Job</td>
<td>Stops an active job. See “To stop a running job” on page 473.</td>
</tr>
</tbody>
</table>
### Table 38: Toolbar (continued)

<table>
<thead>
<tr>
<th>Buttons</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Submit Job icon" /></td>
<td>Submit Job</td>
<td>Submits a job to be run. See “Step 4 - Run the Job” on page 471.</td>
</tr>
<tr>
<td><img src="image2" alt="Restart Job icon" /></td>
<td>Restart Job</td>
<td>Restarts a job. See “To restart a job” on page 473.</td>
</tr>
<tr>
<td><img src="image3" alt="View Log icon" /></td>
<td>View Log</td>
<td>Opens the View Log dialog box. See “Log Viewer” on page 476.</td>
</tr>
</tbody>
</table>
This appendix provides a brief description of the sample files distributed with Teradata PT for the z/OS platform.

These sample files can be found on the Teradata Tools and Utilities software release distribution tape and are loaded during the installation process into a SAMPLIB.

Please refer to the *Teradata Tools and Utilities Installation Guide for z/OS* and your software installer for the data set name.

The sample files include:

- Job Script Examples
- JCL Sample
- Job Attribute File
- Teradata PT Catalogued Procedure
- Teradata PTLV Catalogued Procedure

## Job Script Examples

The job script examples demonstrate some of the basic Teradata PT functions. Each script is fully documented and in particular describes the Teradata PT features used.

<table>
<thead>
<tr>
<th>Job Script</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GT11</td>
<td>Exports rows from one Teradata Database table and loads into another, empty Teradata Database table.</td>
</tr>
<tr>
<td>GT44</td>
<td>Exports rows from one Teradata Database table to a z/OS data set.</td>
</tr>
<tr>
<td>GT62</td>
<td>Loads rows into an empty Teradata Database table from a z/OS data set.</td>
</tr>
<tr>
<td>GT93</td>
<td>Loads rows into an empty Teradata Database table from 2 z/OS data sets.</td>
</tr>
<tr>
<td>GT94</td>
<td>Loads rows into 2 empty Teradata Database tables from a z/OS data set.</td>
</tr>
<tr>
<td>GT95</td>
<td>Updates a Teradata Database table from a z/OS data set.</td>
</tr>
</tbody>
</table>
A sample JCL has been included to execute the job script examples. It is member SAMPJCL. It contains the Teradata PT cataloged procedure inline and has a jobstep for each script example.

The z/OS data sets are unconditionally deleted and reallocated for this sample job. This allows repeated execution without any intervention; this would not be desirable for most production scenarios.

The use of the mainframe specific -S parameter lists the public and private logs.

There are several changes necessary for proper execution:

- The JOB statement must be modified to conform to your installation’s requirements.
- The SAMPLIB SET statement must be changed to the SAMPLIB data set name.
- The JOBLIB DD statement must also be updated with the appropriate TPT and CLI program library names.

**Job Attribute File**

The above jobstream references JOBVARS, the job variable file that contains the symbolic variables used in the script examples.

Symbolic variables are used for the information that would most likely change for different users. Thus the changes are localized to the job variable file, and the script examples themselves are static.

There are several changes necessary for proper execution:

- A Teradata TDP must be specified.
- A valid user name for the above TDP.
- The password associated with the above user name.
- Though not necessary, the FileName attribute format used to specify the z/OS data set for each script can be changed.
Teradata PT Catalogued Procedure

A sample TPT catalogued procedure has been included. There are 2 jobsteps.

- The first jobstep ALLOC allocates the checkpoint data sets if necessary. If they exist, the Teradata PT job uses the contained information in restart mode. ALLOC always deletes the log data set, if it exists, and allocates a new log for the current job.
- The second jobstep executes Teradata PT. Use symbolic parameters to specify the script and job attribute files, enter any TPT parameters, and create a unique high-level qualifier for the checkpoint and log data sets.
  Refer to SAMPJCL for an example of how symbolic parameters are set.

Teradata PTLV Catalogued Procedure

A sample TPTLV catalogued procedure is included, which will execute the tlogview program.

Use symbolic parameters to specify which logs and information are to be displayed.

The HLQUAL symbolic parameter is also used to designate the specific job log in a similar manner to the TPT cataloged procedure.
This appendix contains the following job log examples:

- Console Log for a Successful Job
- Public Log for a Successful Job
- Private Log for a Successful Job
- Public Log for an Unsuccessful Job
- Private Log for an Unsuccessful Job

Example Logs for a Successful Job

The following sections presents examples of the public and private logs for a successful job. The log examples that follow are based upon.

Console Log for a Successful Job

The following example shows a console log for a successful job. Note how the content differs from the public log example.

Teradata Parallel Transporter Version 13.10.00.00
Execution Plan generation started.
Execution Plan generation successfully completed.
Job id is root-1500, running on twesun4
Teradata Parallel Transporter SQL DDL Operator Version 13.10.00.00
DDL_OPERATOR: private log specified: ddllog
DDL_OPERATOR: connecting sessions
DDL_OPERATOR: sending SQL requests
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_e1' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_e2' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_log' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_wt' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
DDL_OPERATOR: disconnecting sessions
Job step setup_tables completed successfully
Teradata Parallel Transporter DataConnector Version 13.10.00.00
FILE_READER_A Instance 1 directing private log report to 'M1.dtaclog-1'.
FILE_READER_A DataConnector Producer operator Instances: 1
Teradata Parallel Transporter Stream Operator Version 13.10.00.00
STREAM_OPERATOR: private log specified: M2.strlog
Teradata Parallel Transporter DataConnector Version 13.10.00.01
FILE_READER_B Instance 1 directing private log report to 'M2.dtaclog-1'.
FILE_READER_B DataConnector Producer operator Instances: 1
STREAM_OPERATOR: Start-up Rate: UNLIMITED statements per Minute
STREAM_OPERATOR: Operator Command ID for External Command Interface: STREAM_OPERATOR15036
STREAM_OPERATOR: connecting sessions
FILE_READER_B Operator instance 1 processing file 't3_x2'.
FILE_READER_A Operator instance 1 processing file 't2_x2'.
STREAM_OPERATOR: entering Load Phase
FILE_READER_A Directory scan completed. 1 files processed.
FILE_READER_B Directory scan completed. 1 files processed.
FILE_READER_B Operator instance 1 processing file 't3_x2'.
FILE_READER_A Operator instance 1 processing file 't2_x2'.
FILE_READER_A Directory scan completed. 1 files processed.
FILE_READER_B Directory scan completed. 1 files processed.
FILE_READER_A The time value of attribute 'VigilStopTime' has expired (at 10:44:35).
FILE_READER_B The time value of attribute 'VigilStopTime' has expired (at 10:44:35).
STREAM_OPERATOR: entering Cleanup Phase
FILE_READER_B Total files processed: 2.
FILE_READER_A Total files processed: 2.
STREAM_OPERATOR: disconnecting sessions
Job step Load_Records completed successfully
Job root completed successfully

Public Log for a Successful Job

The following example shows a public log for a successful job.

Public log:

Teradata Parallel Transporter Coordinator Version 13.10.00.00
Teradata Parallel Transporter Executor Version 13.10.00.00
Teradata Parallel Transporter SQL DDL Operator Version 13.10.00.00
DDL_OPERATOR: private log specified: ddllog
DDL_OPERATOR: connecting sessions
DDL_OPERATOR: sending SQL requests
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_e1' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_e2' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_log' does not exist.
Appendix D: Example Logs

Example Logs for a Successful Job

DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_wt' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
FXTB_Terminate: Message Buffers Sent/Received = 0, Total Rows Received = 0, Total Rows Sent = 0
DDL_OPERATOR: disconnecting sessions
Job step setup_tables completed successfully
Teradata Parallel Transporter Coordinator Version 13.10.00.00
Teradata Parallel Transporter Executor Version 13.10.00.00
Teradata Parallel Transporter Executor Version 13.10.00.00
Teradata Parallel Transporter Stream Operator Version 13.10.00.00
STREAM_OPERATOR: private log specified: M2.strlog
Teradata Parallel Transporter DataConnector Version 13.10.00.01
FILE_READER_A Attribute 'TraceLevel' value reset to 'MILESTONES'.
FILE_READER_A Instance 1 directing private log report to 'M1.dtaclog-1'.
FILE_READER_A DataConnector Producer operator Instances: 1
FILE_READER_A Active Directory Vigil feature requested. VigilStartTime: 10:40:18, VigilStopTime: 10:41:18, VigilWaitTime: 30 seconds,
VigilNoticeFileName: .
FILE_READER_A Required attribute 'TextDelimiter' not found. Defaulting to '|'.
Teradata Parallel Transporter Executor Version 13.10.00.00
STREAM_OPERATOR: Start-up Rate: UNLIMITED statements per Minute
STREAM_OPERATOR: Operator Command ID for External Command Interface:
STREAM_OPERATOR14905
STREAM_OPERATOR: connecting sessions
Teradata Parallel Transporter DataConnector Version 13.10.00.01
FILE_READER_B Attribute 'TraceLevel' value reset to 'MILESTONES'.
FILE_READER_B Instance 1 directing private log report to 'M2.dtaclog-1'.
FILE_READER_B DataConnector Producer operator Instances: 1
VigilNoticeFileName: .
FILE_READER_B Required attribute 'TextDelimiter' not found. Defaulting to '|'.
FILE_READER_B Operator instance 1 processing file 't3_x2'.
FILE_READER_A Operator instance 1 processing file 't2_x2'.
Task(SELECT_3[0001]): checkpoint completed, status = Success
Task(APPLY_1[0001]): checkpoint completed, status = Success
Task(SELECT_2[0001]): checkpoint completed, status = Success
Task(SELECT_2[0001]) ready to checkpoint
STREAM_OPERATOR: entering Load Phase
Task(SELECT_3[0001]) ready to take internal checkpoint
Task(SELECT_2[0001]): checkpoint completed, status = Success
Task(SELECT_3[0001]): checkpoint completed, status = Success
Task(APPLY_1[0001]): checkpoint completed, status = Success
Task(SELECT_3[0001]) ready to checkpoint
Task(SELECT_2[0001]) ready to take internal checkpoint
Task(SELECT_2[0001]): checkpoint completed, status = Success
Task(SELECT_3[0001]): checkpoint completed, status = Success
Task(APPLY_1[0001]): checkpoint completed, status = Success
FILE_READER_A Directory scan completed. 1 files processed.
FILE_READER_B Directory scan completed. 1 files processed.
FILE_READER_A The time value of attribute 'VigilStopTime' has expired (at 10:41:18).
Task(SELECT_2[0001]) ready to checkpoint

FILE_READER_B The time value of attribute 'VigilStopTime' has expired (at 10:41:18).
Task(SELECT_3[0001]) ready to checkpoint

Task(SELECT_2[0001]): checkpoint completed, status = Success
Task(SELECT_3[0001]): checkpoint completed, status = Success
Task(APPLY_1[0001]): checkpoint completed, status = Success
Task(SELECT_2[0001]) ready to take the EOD checkpoint

Task(SELECT_3[0001]) ready to take the EOD checkpoint

Task(SELECT_2[0001]): checkpoint completed, status = Success
Task(SELECT_3[0001]): checkpoint completed, status = Success
Task(APPLY_1[0001]): checkpoint completed, status = Success
STREAM_OPERATOR: entering Cleanup Phase
PXTB_Terminate: Message Buffers Sent/Received = 3, Total Rows Received = 96, Total Rows Sent = 0
PXTB_Terminate: Message Buffers Sent/Received = 4, Total Rows Received = 0, Total Rows Sent = 48
PXTB_Terminate: Message Buffers Sent/Received = 4, Total Rows Received = 0, Total Rows Sent = 48
FILE_READER_B Total files processed: 1.
STREAM_OPERATOR: disconnecting sessions
FILE_READER_A Total files processed: 1.
Job step Load_Records completed successfully
Job root completed successfully

For information on evaluating the contents of a public log for a successful job, see “Chapter 23 Post-Job Considerations” on page 275.

Private Log for a Successful Job

The following is an example of a private log for a successful job.

Private log TWB_SRCTGT:

TPT_INFRA: TPT02241: Error: setup_tables     APPLY_1[0001]
DDL_OPERATOR   DBS     checks
TPT_INFRA: TPT02241: Error: Load_Records     APPLY_1[0001]
STREAM_OPERATOR DBS     checks
TPT_INFRA: TPT02241: Error: Load_Records     SELECT_2[0001]
FILE_READER_A FILE     temp  t2*
TPT_INFRA: TPT02241: Error: Load_Records     SELECT_3[0001]
FILE_READER_B FILE     temp  t3*
Private log PXCRM:

CheckPoint Resource Manager initialized.
Checking whether a valid CheckPoint exists for restart.
CheckPoint Resource Manager initialized.
Checking whether a valid CheckPoint exists for restart.
CheckPoint No. 1 started.
CheckPoint No. 1 finished successfully.
CheckPoint No. 2 started.
CheckPoint No. 2 finished successfully.
CheckPoint No. 3 started.
CheckPoint No. 3 finished successfully.
CheckPoint No. 4 started.
CheckPoint No. 4 finished successfully.
CheckPoint No. 5 started.
CheckPoint No. 5 finished successfully.
CheckPoint No. 6 started.
CheckPoint No. 6 finished successfully.

Private log TWB_STATUS:

TPT_INFRA: TPT02241: Error: setup_tables APPLY_1[0001]
Success DDL_OPERATOR 1 1 INITIATE-Started
10:40:06 0.0000 0.0000 65000 0 0
0 0 0 N Y
TPT_INFRA: TPT02241: Error: setup_tables APPLY_1[0001]
Success DDL_OPERATOR 1 1 INITIATE-Ended
10:40:07 1.0000 0.0800 65000 0 0
0 0 0 N Y
TPT_INFRA: TPT02241: Error: setup_tables APPLY_1[0001]
Success DDL_OPERATOR 1 1 EXECUTE-Started
10:40:07 0.0000 0.0000 65000 0 0
0 0 0 N Y
TPT_INFRA: TPT02241: Error: setup_tables APPLY_1[0001]
Success DDL_OPERATOR 1 1 EXECUTE-Ended
10:40:10 3.0000 0.0100 65000 0 0
0 0 0 Y Y
TPT_INFRA: TPT02241: Error: setup_tables APPLY_1[0001]
Success DDL_OPERATOR 1 1 TERMINATE-Started
10:40:10 0.0000 0.0000 65000 0 0
0 0 0 Y Y
TPT_INFRA: TPT02241: Error: setup_tables APPLY_1[0001]
Success DDL_OPERATOR 1 1 TERMINATE-Ended
10:40:10 0.0000 0.0000 65000 0 0
0 0 0 Y Y
TPT_INFRA: TPT02241: Error: Load_Records APPLY_1[0001]
Success STREAM_OPERATOR 1 1 INITIATE-Started
10:40:18 0.0000 0.0000 65000 0 0
0 0 0 N Y
TPT_INFRA: TPT02241: Error: Load_Records SELECT_2[0001]
Success FILE_READER_A 1 1 INITIATE-Started
10:40:18 0.0000 0.0000 65000 0 0
0 0 0 N Y
TPT_INFRA: TPT02241: Error: Load_Records SELECT_3[0001]
Success FILE_READER_B 1 1 INITIATE-Started
10:40:18 0.0000 0.0000 65000 0 0
0 0 0 N Y
TPT_INFRA: TPT02241: Error: Load_Records APPLY_1[0001]
Success STREAM_OPERATOR 1 1 INITIATE-Ended
Appendix D: Example Logs

Example Logs for a Successful Job

<table>
<thead>
<tr>
<th>Time</th>
<th>CPU Sec</th>
<th>DB Time</th>
<th>DB Size</th>
<th>DB MS</th>
<th>File Size</th>
<th>File YN</th>
<th>Exp</th>
<th>Time</th>
<th>CPU Sec</th>
<th>DB Time</th>
<th>DB Size</th>
<th>DB MS</th>
<th>File Size</th>
<th>File YN</th>
<th>Exp</th>
<th>Time</th>
<th>CPU Sec</th>
<th>DB Time</th>
<th>DB Size</th>
<th>DB MS</th>
<th>File Size</th>
<th>File YN</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:40:27</td>
<td>9.0000</td>
<td>0.9400</td>
<td>65000</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Y</td>
<td>10:40:27</td>
<td>9.0000</td>
<td>0.9400</td>
<td>65000</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Y</td>
<td>10:40:27</td>
<td>9.0000</td>
<td>0.9400</td>
<td>65000</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_3[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_B</td>
<td>1</td>
<td>1</td>
<td>INITIATE-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:40:48</td>
<td>30.0000</td>
<td>0.1000</td>
<td>65000</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Y</td>
<td>10:40:48</td>
<td>30.0000</td>
<td>0.1000</td>
<td>65000</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Y</td>
<td>10:40:48</td>
<td>30.0000</td>
<td>0.1000</td>
<td>65000</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_2[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_A</td>
<td>1</td>
<td>1</td>
<td>INITIATE-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10:40:48</td>
<td>0.0000</td>
<td>0.0000</td>
<td>65000</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Y</td>
<td>10:40:48</td>
<td>0.0000</td>
<td>0.0000</td>
<td>65000</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Y</td>
<td>10:40:48</td>
<td>0.0000</td>
<td>0.0000</td>
<td>65000</td>
<td>0</td>
<td>0</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_2[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_A</td>
<td>1</td>
<td>1</td>
<td>EXECUTE-Started</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_2[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_A</td>
<td>1</td>
<td>1</td>
<td>CHECKPOINT-Started</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_2[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_A</td>
<td>1</td>
<td>1</td>
<td>CHECKPOINT-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_3[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_B</td>
<td>1</td>
<td>1</td>
<td>EXECUTE-Started</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_3[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_B</td>
<td>1</td>
<td>1</td>
<td>CHECKPOINT-Started</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_3[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_B</td>
<td>1</td>
<td>1</td>
<td>CHECKPOINT-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_2[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_A</td>
<td>1</td>
<td>1</td>
<td>CHECKPOINT-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_2[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_A</td>
<td>1</td>
<td>1</td>
<td>CHECKPOINT-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_3[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_B</td>
<td>1</td>
<td>1</td>
<td>CHECKPOINT-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_3[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_B</td>
<td>1</td>
<td>1</td>
<td>CHECKPOINT-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_3[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_B</td>
<td>1</td>
<td>1</td>
<td>CHECKPOINT-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TPT_INFRA: TPT02241: Error: Load_Records SELECT_2[0001]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Success</td>
<td>FILE_READER_A</td>
<td>1</td>
<td>1</td>
<td>CHECKPOINT-Ended</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Example Logs for a Successful Job

<table>
<thead>
<tr>
<th>Time</th>
<th>Duration</th>
<th>CPU Time</th>
<th>Records</th>
<th>CPU Load</th>
<th>Disk Load</th>
</tr>
</thead>
<tbody>
<tr>
<td>10:40:50</td>
<td>2.0000</td>
<td>0.0100</td>
<td>65000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10:40:52</td>
<td>4.0000</td>
<td>0.0200</td>
<td>65000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10:40:52</td>
<td>4.0000</td>
<td>0.0200</td>
<td>65000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10:40:53</td>
<td>5.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10:40:53</td>
<td>5.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10:40:53</td>
<td>5.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10:40:54</td>
<td>6.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10:41:18</td>
<td>30.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10:41:18</td>
<td>30.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>10:41:19</td>
<td>31.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Time</td>
<td>Duration</td>
<td>Estimated</td>
<td>Actual</td>
<td>Status</td>
<td>Component</td>
</tr>
<tr>
<td>----------</td>
<td>-----------</td>
<td>-----------</td>
<td>---------</td>
<td>------------</td>
<td>-----------</td>
</tr>
<tr>
<td>10:41:20</td>
<td>32.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_A</td>
</tr>
<tr>
<td>10:41:20</td>
<td>32.0000</td>
<td>0.0400</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_B</td>
</tr>
<tr>
<td>10:41:20</td>
<td>32.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>Success</td>
<td>STREAM_OPERATOR</td>
</tr>
<tr>
<td>10:41:21</td>
<td>33.0000</td>
<td>0.0400</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_A</td>
</tr>
<tr>
<td>10:41:21</td>
<td>33.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_B</td>
</tr>
<tr>
<td>10:41:22</td>
<td>34.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_A</td>
</tr>
<tr>
<td>10:41:22</td>
<td>34.0000</td>
<td>0.0400</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_B</td>
</tr>
<tr>
<td>10:41:23</td>
<td>35.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_A</td>
</tr>
<tr>
<td>10:41:23</td>
<td>35.0000</td>
<td>0.0400</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_B</td>
</tr>
<tr>
<td>10:41:23</td>
<td>35.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>Success</td>
<td>STREAM_OPERATOR</td>
</tr>
<tr>
<td>10:41:23</td>
<td>35.0000</td>
<td>0.0400</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_B</td>
</tr>
<tr>
<td>10:41:23</td>
<td>35.0000</td>
<td>0.0300</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_A</td>
</tr>
<tr>
<td>10:41:23</td>
<td>35.0000</td>
<td>0.0400</td>
<td>65000</td>
<td>Success</td>
<td>FILE_READER_B</td>
</tr>
</tbody>
</table>
Private log ddllog:

===================================================================
= = = = = = TERADATA PARALLEL TRANSPORTER = = = = = =
= = = = SQL DDL OPERATOR VERSION 13.10.00.00 = =
= = OPERATOR SUPPORT LIBRARY VERSION 13.10.00.00 = =
= COPYRIGHT 2001-2008, TERADATA CORPORATION. ALL RIGHTS RESERVED. =
===================================================================

**** 10:40:07 Processing starting at: Thu May 22 10:40:07 2008

===================================================================
= = = = = = Module Identification = = = = = =
= = = = = = = = = = = = = = = = = =
===================================================================

SQL DDL Operator for SunOS release 5.10 on twesun4
DdlMain : 13.10.00.02
DdlCLI : 13.10.00.03
DdlUtil : 13.10.00.03
PcomCLI : 13.10.00.10
PcomMBCS : 13.10.00.02
PcomMsgs : 13.10.00.05
Example Logs for a Successful Job

**** 10:40:07 Connecting to RDBMS: 'checks'
**** 10:40:07 Connecting with UserId: 'px_user'

**** 10:40:07 Teradata Database Version: '06.01.01.95'
**** 10:40:07 Teradata Database Release: 'V2R.06.01.01.91'
**** 10:40:07 Maximum request size supported: 1MB
**** 10:40:07 Session character set: 'ASCII'
**** 10:40:07 Data Encryption: supported

**** 10:40:07 Starting to send DDL statements to the RDBMS.
**** 10:40:07 Submitting the following request for DDL Group 1:
drop table M2_e1;

**** 10:40:07 TPT10508: RDBMS error 3807: Object 'M2_e1' does not exist.
**** 10:40:07 TPT18046: Warning: error is ignored as requested in ErrorList

**** 10:40:07 Submitting the following request for DDL Group 2:
drop table M2_e2;

**** 10:40:07 TPT10508: RDBMS error 3807: Object 'M2_e2' does not exist.
**** 10:40:07 TPT18046: Warning: error is ignored as requested in ErrorList

**** 10:40:07 Submitting the following request for DDL Group 3:
drop table M2_log;

**** 10:40:07 TPT10508: RDBMS error 3807: Object 'M2_log' does not exist.
**** 10:40:07 TPT18046: Warning: error is ignored as requested in ErrorList

**** 10:40:07 Submitting the following request for DDL Group 4:
drop table M2_wt;

**** 10:40:07 TPT10508: RDBMS error 3807: Object 'M2_wt' does not exist.
**** 10:40:07 TPT18046: Warning: error is ignored as requested in ErrorList

**** 10:40:07 Submitting the following request for DDL Group 5:
   drop table T2;

**** 10:40:08 *** Command Successful.

**** 10:40:08 Submitting the following request for DDL Group 6:
   drop table T3;

**** 10:40:09 *** Command Successful.

**** 10:40:09 Submitting the following request for DDL Group 7:
   CREATE SET TABLE T2      (      c1 INTEGER NOT NULL,      c2
   CHAR(10) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,      c3 INTEGER
   NOT NULL)     PRIMARY INDEX ( c1 );

**** 10:40:09 *** Command Successful.

**** 10:40:09 Submitting the following request for DDL Group 8:
   CREATE SET TABLE T3      (      c1 INTEGER NOT NULL,      c2
   CHAR(10) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,      c3 INTEGER
   NOT NULL)     PRIMARY INDEX ( c1 );

**** 10:40:10 *** Command Successful.

**** 10:40:10 Finished sending DDL statements.

   Total Requests Sent:        8
   Total Requests Succeeded:   4
   Total Requests Failed:      4
   Total Requests Skipped:     0

===================================================================

=                                                                 =
=                        Logoff/Disconnect                        =
=                                                                 =
===================================================================

**** 10:40:10 Logging off all sessions

   Instance       Cpu Time
   =========       ===============
   1       0.09 Seconds

**** 10:40:10 Total processor time used = '0.09 Second(s)'
.   End   : Thu May 22 10:40:10 2008
.   Highest return code encountered = '0'.

**** 10:40:10 This job terminated

Private log M2.strlog:

===================================================================

= =
= = TERADATA PARALLEL TRANSPORTER =
= =
= = STREAM OPERATOR VERSION 13.10.00.00 =

Teradata Parallel Transporter User Guide
### Module Identification

Stream Operator for SunOS release 5.10 on twesun4
StreamMain : 13.10.00.00  
StreamCLI : 13.10.00.00  
StreamSess : 13.10.00.00  
StreamSort : 13.10.00.01  
StreamUtil : 13.10.00.00  
PcomCLI : 13.10.00.00  
PcomMBCS : 13.10.00.00  
PcomMsgs : 13.10.00.00  
PcomNtfy : 13.10.00.00  
PcomFx : 13.10.00.00  
PcomUtil : 13.10.00.00  
PXICU : 13.10.00.00  
TDICU : 13.10.00.00  
CLIv2 : 13.10.00.00  

### Attribute Definitions

**10:40:18** Options in effect for this job:
- **OperatorType**: Consumer  
- **Instances**: 1  
- **Character set**: Not specified; will use default  
- **Checkpoint**: No checkpoint in effect  
- **Notify**: Not enabled  
- **Error limit**: 24 rejected record(s)  
- **Tenacity**: 2 hour limit to successfully connect  
- **Sleep**: 1 minute(s) between connect retries  
- **Buffers**: 3  
- **Pack**: 20  
- **Robust**: In effect

### Column/Field Definition

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Offset</th>
<th>Length</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>-------------</td>
<td>--------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>-------------</td>
<td>--------</td>
<td>--------</td>
<td>------</td>
</tr>
</tbody>
</table>
Appendix D: Example Logs
Example Logs for a Successful Job

IND                                 0      1 CHAR
A_IN_ID1                            1      4 VARCHAR
A_IN_C1                             7     10 VARCHAR
A_IN_C2                            19     10 VARCHAR
A_IN_C3                            31     10 VARCHAR
B_IN_ID1                           43      4 VARCHAR
B_IN_C1                            49     10 VARCHAR
B_IN_C2                            61     10 VARCHAR
B_IN_C3                            73     10 VARCHAR

INDICATOR BYTES NEEDED: 2
EXPECTED RECORD LENGTH: 87

**** 10:40:18 Operator Command ID for External Command Interface:
STREAM_OPERATOR14905

===================================================================
=                                                                 =
=                   Control Session Connection                    =
=                                                                 =
===================================================================

**** 10:40:18 Connecting to RDBMS: 'checks'
**** 10:40:18 Connecting with UserId: 'px_user'

===================================================================
=                                                                 =
=                  Teradata Database Information                  =
=                                                                 =
===================================================================

**** 10:40:18 Teradata Database Version:      '06.01.01.95'
**** 10:40:18 Teradata Database Release:      'V2R.06.01.01.91'
**** 10:40:18 Maximum request size supported: 1MB
**** 10:40:18 Session character set:          'ASCII'
**** 10:40:18 RDBMS supports upsert SQL
**** 10:40:18 Data Encryption:                supported
**** 10:40:18 RDBMS supports Array Support
**** 10:40:20 Restart log table 'PX_USER.M2_log' has been created
**** 10:40:21 Error table 'PX_USER.M08143_38420_91483_ET' has been created

===================================================================
=                                                                 =
=                   Special Session Connection                    =
=                                                                 =
===================================================================

**** 10:40:26 Maximum number of special sessions requested: 20
 **** 10:40:26 Minimum number of special sessions required:  1

<table>
<thead>
<tr>
<th>Instance Assigned</th>
<th>Connected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>20</td>
<td>20   Successful</td>
</tr>
</tbody>
</table>

Total 20 20 Successful

**** 10:40:27 Current working DATABASE set: 'px_user'
Appendix D: Example Logs
Example Logs for a Successful Job

===================================================================
=                                                                 =
=                         DML Information                         =
=                                                                 =
===================================================================

**** 10:40:27 DML statement for DML Group: 1

INSERT INTO T2 (:A_IN_C1 ,:A_IN_C2 ,:A_IN_C3 );

DML Options in effect for this group:
Duplicate Inserts will be inserted into error table
Duplicate Updates will be inserted into error table
Missing Updates will be inserted into error table
Missing Deletes will be inserted into error table
Upsert feature is not specified
Extra Updates will be inserted into error table
Extra Deletes will be inserted into error table
Array Support is enabled

**** 10:40:27 DML statement for DML Group: 2

INSERT INTO T3 (:B_IN_C1 ,:B_IN_C2 ,:B_IN_C3 );

DML Options in effect for this group:
Duplicate Inserts will be inserted into error table
Duplicate Updates will be inserted into error table
Missing Updates will be inserted into error table
Missing Deletes will be inserted into error table
Upsert feature is not specified
Extra Updates will be inserted into error table
Extra Deletes will be inserted into error table
Array Support is enabled

===================================================================
=                                                                 =
=                           Load Phase                            =
=                                                                 =
===================================================================

**** 10:40:49 Starting to send data to the RDBMS
**** 10:40:51 Checkpoint complete. Rows sent: 69
**** 10:40:53 Checkpoint complete. Rows sent: 96
**** 10:41:19 Checkpoint complete. Rows sent: 96
**** 10:41:21 Checkpoint complete. Rows sent: 96
**** 10:41:22 Checkpoint complete. Rows sent: 96
**** 10:41:23 Finished sending rows to the RDBMS

<table>
<thead>
<tr>
<th>Instance</th>
<th>Rows Sent</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>Total</td>
<td>96</td>
</tr>
</tbody>
</table>

**** 10:41:23 Load Statistics for DML Group 1 :

<table>
<thead>
<tr>
<th>Type</th>
<th>Database</th>
<th>Table or Macro Name</th>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT</td>
<td>px_user</td>
<td>T2</td>
<td>48</td>
</tr>
</tbody>
</table>
**** 10:41:23 Load Statistics for DML Group 2 :
  Type          Database            Table or Macro Name      Activity
  =============== ======================== ========================= ============
  INSERT px_user                  T3                                  48
  = = = = =
  Operator Task Cleanup
  = = =
  = =

**** 10:41:25 Error table PX_USER.M08143_38420_91483_ET is EMPTY;
dropping table

**** 10:41:28 Restart log table 'PX_USER.M2_log' has been dropped

**** 10:41:28 Logging off all sessions

Instance      Cpu Time
========  ================
1        0.98 Seconds

**** 10:41:30 Total processor time used = '0.98 Second(s)'
  .    End   : Thu May 22 10:41:30 2008
  .    Highest return code encountered = '0'.

**** 10:41:30 This job terminated

Private log M1.dtaclog-1:

========================================================================
==
==
==                     TERADATA PARALLEL TRANSPORTER
=                        100.00.01
=                        
==
= DataConnector UTILITY LIBRARY VERSION 13.10.00.00 =
= COPYRIGHT 1997-2008, TERADATA CORPORATION. ALL RIGHTS RESERVED
= =

========================================================================
==
==
==
= Operator name: 'FILE_READER_A' instance 1 of 1 [Producer]

Appendix D: Example Logs
Example Logs for a Successful Job

========================================================================
==
=                    Operator module static specifics
=                      Compiled for platform: '32-bit Solaris'
=                       Operator module name:'dtacop', version:'13.10.00.00'
=                      pmddcomt_HeaderVersion: 'Common 13.10.00.00' - packing 'none'
=                      pmddamt_HeaderVersion: 'Common 13.10.00.00' - packing 'none'
==
========================================================================

Log will include stats only

Operator 'dtacop' main source version:'13.10.00.08'
**** 10:40:48 From file 't2_x2', starting to send rows.
**** 10:40:49 Finished sending rows for t2_x2 (index 1)
Rows sent: 48, CPU Time: 0.02 Seconds
Files read by this instance: 1
**** 10:41:28 Total processor time used = '0.02 Seconds(s)'
**** 10:41:28 Total files processed: 1

Private log M2.dtaclog-1:

========================================================================
==
=                     TERADATA PARALLEL TRANSPORTER
= DATACONNECTOR OPERATOR VERSION 13.10.00.00 =
= DataConnector UTILITY LIBRARY VERSION 13.10.00.00 =
= COPYRIGHT 1997-2008, TERADATA CORPORATION. ALL RIGHTS RESERVED
==
========================================================================

Operator name: 'FILE_READER_B' instance 1 of 1 [Producer]

========================================================================
Example Logs for an Unsuccessful Job

The following sections present examples of the public and private logs for an unsuccessful job. The job is similar to “Script Example 9: Active Directory Scan” on page 432.

Public Log for an Unsuccessful Job

The following is an example of a public log for an unsuccessful job

Public log:

Teradata Parallel Transporter Version 13.10.00.00
Execution Plan generation started.
Execution Plan generation successfully completed.
Job id is root-1413, running on twesun4
Teradata Parallel Transporter SQL DDL Operator Version 13.10.00.00
DDL_OPERATOR: private log specified: ddllog
DDL_OPERATOR: connecting sessions
DDL_OPERATOR: sending SQL requests
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_e1' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_e2' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
Appendix D: Example Logs
Example Logs for an Unsuccessful Job

Private Log for an Unsuccessful Job

The following is an example of a private log for an unsuccessful job.

Private log ddllog:

DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_log' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
DDL_OPERATOR: TPT10508: RDBMS error 3807: Object 'M2_wt' does not exist.
DDL_OPERATOR: TPT18046: Warning: error is ignored as requested in ErrorList
DDL_OPERATOR: disconnecting sessions
Job step setup_tables completed successfully
Teradata Parallel Transporter DataConnector Version 13.10.00.01
FILE_READER_B Instance 1 directing private log report to 'M2.dtaclog-1'.
FILE_READER_B DataConnector Producer operator Instances: 1
VigilNoticeFileName: .
Teradata Parallel Transporter Stream Operator Version 13.10.00.00
STREAM_OPERATOR: private log specified: M2.strlog
Teradata Parallel Transporter DataConnector Version 13.10.00.01
FILE_READER_A Instance 1 directing private log report to 'M1.dtaclog-1'.
FILE_READER_A DataConnector Producer operator Instances: 1
VigilNoticeFileName: .
STREAM_OPERATOR: Start-up Rate: UNLIMITED statements per Minute
STREAM_OPERATOR: Operator Command ID for External Command Interface:
STREAM_OPERATOR1763
STREAM_OPERATOR: connecting sessions
FILE_READER_A Operator instance 1 processing file 't2_x2'.
FILE_READER_B Operator instance 1 processing file 't3_x2'.
STREAM_OPERATOR: entering Load Phase
FILE_READER_A Directory scan completed. 1 files processed.
FILE_READER_B Directory scan completed. 1 files processed.
FILE_READER_A Operator instance 1 processing file 't2_x2'.
FILE_READER_B Operator instance 1 processing file 't3_x2'.
STREAM_OPERATOR: TPT10510: Error Limit has been exceeded
Of 145 row(s) sent to the RDBMS,
49 row(s) were recorded as errors.
STREAM_OPERATOR: disconnecting sessions
FILE_READER_B !WARNING! Archive operation NOT performed. Execute method was NOT completed
FILE_READER_B Total files processed: 2.
FILE_READER_A !WARNING! Archive operation NOT performed. Execute method was NOT completed
FILE_READER_A Total files processed: 2.
TPT_INFRA: TPT02772: Error: Job step Load_Records terminated (status 12)
Job root terminated (status 12)

For information on evaluating the contents of a public log for an unsuccessful job, see
“Chapter 24 Troubleshooting a Failed Job.”

SQL DDL Operator for SunOS release 5.10 on twesun4
DdlMain    : 13.10.00.00
DdlCLI     : 13.10.00.00
DdlUtil    : 13.10.00.00
PcomCLI    : 13.10.00.00
PcomMBCS   : 13.10.00.00
PcomMsgs   : 13.10.00.00
PcomNtfy   : 13.10.00.00
PcomPx     : 13.10.00.00
PcomUtil   : 13.10.00.00
PXICU      : 13.10.00.00
TDICU      : 13.10.00.00
CLTv2      : 13.10.00.00

**** 23:29:06 Connecting to RDBMS: 'checks'
**** 23:29:06 Connecting with UserId: 'px_user'

**** 23:29:06 Teradata Database Version: '06.01.01.95'
**** 23:29:06 Teradata Database Release: 'V2R.06.01.01.91'
**** 23:29:06 Maximum request size supported: 1MB
**** 23:29:06 Session character set: 'ASCII'
**** 23:29:06 Data Encryption: supported

SQL Requests
Example Logs for an Unsuccessful Job

**** 23:29:06 Starting to send DDL statements to the RDBMS.
**** 23:29:06 Submitting the following request for DDL Group 1:
  drop table M2_e1;

**** 23:29:06 TPT10508: RDBMS error 3807: Object 'M2_e1' does not exist.
**** 23:29:06 TPT18046: Warning: error is ignored as requested in ErrorList
**** 23:29:06 Submitting the following request for DDL Group 2:
  drop table M2_e2;

**** 23:29:06 TPT10508: RDBMS error 3807: Object 'M2_e2' does not exist.
**** 23:29:06 TPT18046: Warning: error is ignored as requested in ErrorList
**** 23:29:06 Submitting the following request for DDL Group 3:
  drop table M2_log;

**** 23:29:06 TPT10508: RDBMS error 3807: Object 'M2_log' does not exist.
**** 23:29:06 TPT18046: Warning: error is ignored as requested in ErrorList
**** 23:29:06 Submitting the following request for DDL Group 4:
  drop table M2_wt;

**** 23:29:06 TPT10508: RDBMS error 3807: Object 'M2_wt' does not exist.
**** 23:29:06 TPT18046: Warning: error is ignored as requested in ErrorList
**** 23:29:06 Submitting the following request for DDL Group 5:
  drop table T2;

**** 23:29:06 *** Command Successful.
**** 23:29:06 Submitting the following request for DDL Group 6:
  drop table T3;

**** 23:29:06 *** Command Successful.

**** 23:29:07 Submitting the following request for DDL Group 7:
  CREATE SET TABLE T2      (      c1 INTEGER NOT NULL,      c2
  CHAR(10) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,      c3 INTEGER
  NOT NULL)     PRIMARY INDEX ( c1 );

**** 23:29:08 *** Command Successful.

**** 23:29:08 Submitting the following request for DDL Group 8:
  CREATE SET TABLE T3      (      c1 INTEGER NOT NULL,      c2
  CHAR(10) CHARACTER SET LATIN NOT CASESPECIFIC NOT NULL,      c3 INTEGER
  NOT NULL)     PRIMARY INDEX ( c1 );

**** 23:29:08 *** Command Successful.

**** 23:29:08 Finished sending DDL statements.

Total Requests Sent: 8
Total Requests Succeeded: 4
Total Requests Failed: 4
Total Requests Skipped: 0
Example Logs for an Unsuccessful Job

**** 23:29:08 Logging off all sessions

<table>
<thead>
<tr>
<th>Instance</th>
<th>Cpu Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.08 Seconds</td>
</tr>
</tbody>
</table>

**** 23:29:08 Total processor time used = '0.08 Second(s)'

. Highest return code encountered = '0'.

**** 23:29:08 This job terminated

Private log M2.dtaclog-1:

Operator name: 'FILE_READER_B' instance 1 of 1 [Producer]


Operator module static specifics

Compiled for platform: '32-bit Solaris'

Operator module name:'dtacop', version:'13.10.00.00'

pmdcomt_HeaderVersion: 'Common 13.10.00.00' - packing 'none'

pmdamt_HeaderVersion: 'Common 13.10.00.00' - packing 'none'
Appendix D: Example Logs
Example Logs for an Unsuccessful Job

Log will include stats only

Operator 'dtacop' main source version: '13.10.00.08'

*** 23:29:15 From file 't3_x2', starting to send rows.
*** 23:29:25 Finished sending rows for t3_x2 (index 1)
Rows sent: 48, CPU Time: 0.02 Seconds

*** 23:29:50 From file 't3_x2', starting to send rows.
*** 23:29:50 Finished sending rows for t3_x2 (index 1)
Rows sent: 48, CPU Time: 0.02 Seconds
Files read by this instance: 2

*** 23:29:51 Total processor time used = '0.04 Seconds(s)'
*** 23:29:51 Total files processed: 2

Private log M2.strlog:
**** 23:29:15 Options in effect for this job:
OperatorType: Consumer
Instances: 1
Character set: Not specified; will use default
Checkpoint: No checkpoint in effect
Notify: Not enabled
Error limit: 24 rejected record(s)
Tenacity: 2 hour limit to successfully connect
Sleep: 1 minute(s) between connect retries
Buffers: 3
Pack: 20
Robust: In effect

===================================================================

=                                                                 =
=                     Column/Field Definition                     =
=                                                                 =
===================================================================

Column Name                    Offset Length Type
============================== ====== ======
========================
IND                                 0      1 CHAR
A_IN_ID1                            1      4 VARCHAR
A_IN_C1                             7     10 VARCHAR
A_IN_C2                            19     10 VARCHAR
A_IN_C3                            31     10 VARCHAR
B_IN_ID1                           43      4 VARCHAR
B_IN_C1                            49     10 VARCHAR
B_IN_C2                            61     10 VARCHAR
B_IN_C3                            73     10 VARCHAR

===================================================================

INDICATOR BYTES NEEDED: 2
EXPECTED RECORD LENGTH: 87

**** 23:29:15 Operator Command ID for External Command Interface:
STREAM_OPERATOR1763

===================================================================

=                                                                 =
=                   Control Session Connection                    =
=                                                                 =
===================================================================

**** 23:29:15 Connecting to RDBMS: 'checks'
**** 23:29:15 Connecting with UserId: 'px_user'

===================================================================

=                                                                 =
=           Teradata Database Information                       =
=                                                                 =
===================================================================

**** 23:29:15 Teradata Database Version: '06.01.01.95
**** 23:29:15 Teradata Database Release: 'V2R.06.01.01.91'
**** 23:29:15 Maximum request size supported: 1MB
**** 23:29:15 Session character set: 'ASCII'
**** 23:29:15 RDBMS supports upsert SQL
**** 23:29:15 Data Encryption: supported
**** 23:29:15 RDBMS supports Array Support
**** 23:29:17 Restart log table 'PX_USER.M2_log' has been created
**** 23:29:18 Error table 'PX_USER.M08134_84557_90329_ET' has been created

===================================================================

Special Session Connection

===================================================================

**** 23:29:23 Maximum number of special sessions requested: 20
**** 23:29:23 Minimum number of special sessions required: 1

Instance Assigned Connected Result
======== ======== ========= ======================
1        20       20    Successful
======== ======== ========= ======================
Total     20       20    Successful

**** 23:29:24 Current working DATABASE set: 'px_user'

===================================================================

DML Information

===================================================================

**** 23:29:24 DML statement for DML Group: 1

INSERT INTO T2 (:A_IN_C1 ,:A_IN_C2 ,:A_IN_C3 );

**** 23:29:24 DML statement for DML Group: 2

INSERT INTO T3 (:B_IN_C1 ,:B_IN_C2 ,:B_IN_C3 );

===================================================================

Load Phase

===================================================================

**** 23:29:25 Starting to send data to the RDBMS
**** 23:29:26 Checkpoint complete. Rows sent: 96
**** 23:29:49 Checkpoint complete. Rows sent: 144

**** 23:29:50 TPT10510: Error Limit has been exceeded
Of 145 row(s) sent to the RDBMS,
49 row(s) were recorded as errors.
Appendix D: Example Logs

Example Logs for an Unsuccessful Job

---

**** 23:29:50 Logging off all sessions

<table>
<thead>
<tr>
<th>Instance</th>
<th>Cpu Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.01 Seconds</td>
</tr>
</tbody>
</table>

**** 23:29:57 Total processor time used = '1.01 Second(s)'

. Highest return code encountered = '12'.

**** 23:29:57 This job terminated

Private log M1.dtaclog-1:

---

Operator name: 'FILE_READER_A' instance 1 of 1 [Producer]


---

Operator module static specifics

Compiled for platform: '32-bit Solaris'

Operator module name: 'dtacop', version: '13.10.00.00'

pmdcomt_HeaderVersion: 'Common 13.10.00.00' - packing 'none'

pmddart_HeaderVersion: 'Common 13.10.00.00' - packing 'none'

---
Log will include stats only

Operator 'dtacop' main source version:'13.10.00.08'
**** 23:29:15 From file 't2_x2', starting to send rows.
**** 23:29:25 Finished sending rows for t2_x2 (index 1)
   Rows sent: 48, CPU Time: 0.02 Seconds
**** 23:29:42 From file 't2_x2', starting to send rows.
**** 23:29:42 Finished sending rows for t2_x2 (index 1)
   Rows sent: 48, CPU Time: 0.02 Seconds
Files read by this instance: 2
**** 23:29:51 Total processor time used = '0.04 Seconds(s)'
**** 23:29:51 Total files processed: 2

For information on evaluating the contents of a private log for an unsuccessful job, see
“Chapter 24 Troubleshooting a Failed Job.”
A

**administrator** A special user responsible for allocating resources to a community of users.

C

**call-level interface (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.

**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.

D

**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 manipulation language (DML)** In Teradata SQL, the statements and facilities that manipulate or change the information content of the database. These statements include INSERT, UPDATE, and DELETE.

**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.

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 one is indicated as 0x00 0x01. In a little-endian computer, the number one is indicated as 0x01 0x00.

**export**  This refers to extracting or transferring system information from the tables and views of a given source and saving it so it can be manipulated or pulled into another system.

**field**  The basic unit of information stored in the Teradata Database. A field is either null, or has a single numeric or string value.

**JCL**  JCL (job control language) is a language for describing jobs (units of work) to the z/OS, and VSE operating systems, which run on IBM's 800/900 large server (mainframe) computers. These operating systems allocate their time and space resources among the total number of jobs that have been started in the computer. Jobs in turn break down into job steps. All the statements required to run a particular program constitute a job step. Jobs are background (sometimes called batch) units of work that run without requiring user interaction (for example, print jobs). In addition, the operating system manages interactive (foreground) user requests that initiate units of work. In general, foreground work is given priority over background work.

**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”.

**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 any value for a field.

**object**  In object-oriented programming, a unique instance of a data structure defined by 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**  This is 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.

**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.
operator  Is a term in Teradata PT used to describe a piece of software used to control loading and unloading data. There are different operators that perform different types of functions.

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 Teradata SQL statements granted to him against a table, database, user, macro, or view.

query  A Teradata SQL statement, such as a SELECT statement.

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.

SQL  See structured query language (SQL).

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 update the database objects used by a given query.

structured query language (SQL)  A standardized query language for requesting information from a database. SQL consists of a set of facilities for defining, manipulating, and controlling data in a relational database.

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 database.
**Teradata Parallel Transporter (Teradata PT)** Teradata PT is a load and unload utility that extracts, load, and updates data from one or more sources into one or more targets with parallel streams of data.

**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.

**Wizard** The Teradata PT Wizard. A GUI-based product that builds and runs simple load and unload job scripts.
Index

A
Active directory scan, DataConnector operator 154
AMP usage, Update operator 249

B
Batch directory scan, Data Connector operator 154
BTEQ, Teradata PT and 27

C
checking 221
Checkpoint
interval 265
interval, job performance and 266
restarts and 267
setting 264
types of 265
Checkpoint files
removing 323
Checkpointing
DDL operator 166
Export operator 173
Load operator 193
ODBC operator 205
SQL Inserter operator 214
SQL Selector operator 221
Stream operator 235
Update operator 248
Complication errors 296
Console log 277

D
Data acquisition errors 302
Data application errors 303
Data Connector (PIOM)
Teradata PT and 26
Data conversions 370
DataConnector operator
AccessModuleInitStr 159
AccessModuleName 159
as consumer
attributes, with access module 155
attributes, without access module 155
as consumer, definition 152
as producer
attributes with access module 153
attributes, active directory scan 154
attributes, batch directory scan 154
attributes, without access module 153
as producer, definition 152
as producer, flat files 151
capabilities 151
DirectoryPath 160
EnableScan 159
file size restrictions 160
FileName 155
Format 157
OpenMode 158
PrivateLogName 159
Vigil attributes 160
z/OS 160
DDL operator
attributes, security 163
capabilities 161
checkpointing 166
definition 162
ErrorList 164
PrivateLogName 164
restarting 166
specifications 163
SQL, supported 164
Decimal database feature 219
Delete task option, Update operator 253
DSNName, ODBC operator 202

E
Error capture, Update operator 246
Error handling, operator-specific 303
Error limit
Stream operator 228
Update operator 245
Error tables 311
ErrorList
DDL operator 164
Errors
Load operator 304
SQL Selector operator 316
Stream operator 307
Update operator 311
ErrorTable
Stream operator 228
Update operator 245
ErrorTable1
Load operator 189
Export operator
attributes, security 170
capabilities 167
checkpointing 173
definition 169
events 325
export jobs, limits on 172
FastExport utility and 168
instances 175
large decimal feature 173
multiple databases 175
performance 174
PrivateLogName 172
restarting 173
SelectStmt 171
sessions 175
SQL Selector operator and 215

FastExport OUTMOD Adapter operator
capabilities 177
definition 178
Format 178
OutmodName 178
PrivateLogName 179
specifications 178

FastLoad INMOD Adapter operator
capabilities 181
definition 182
Format 182
InmodName 182
PrivateLogName 182
specifications 182

FastLoad utility 186
Teradata PT and 26
FileName
DataConnector operator 155

IgnoreError, OS Command operator 208
Initialization errors 301
INMOD routine
support 52
InmodName
FastLoad INMOD Adapter operator 182
Multiload INMOD Adapter operator 197
Instances
Stream operator 235
Update operator 252

Job failure. See Failed Job
Job log
assessing 277
console 277
locations, operating systems and 280
private 279
public 277
using 277
Job script
comments 44
concepts 41
consumer operator, defining 53
creating 44
declarative section 41
executable section 41, 60
job variables 46
operators, defining 50
producer operator, defining 51
schema, defining 48
schema, multiple 49
standalone operator, defining 55
statement types 41
syntax 43
Job script compilation errors 296
Job. See Teradata PT job

Large decimal feature
Export operator 173
Large object. See LOB
Latency interval 234
Load operator
attributes, security 188
capabilities 185
checkpointing 193
definition 187
duplicate rows 191
ErrorLimit 191
errors 304
ErrorTable1 189
ErrorTable2
  ErrorTable2
    Load operator 190
FastLoad utility and 186
instances 193
load jobs, limits on 193
LogTable 189
  performance 192
  phases 186
    PrivateLogName 190
    restarting 193
  sessions 193
  space requirements 192
  specifications 188
  SQL Inserter operator and 212
  staged loading 192
TargetTable 189
Load operator and 186
LOB 366
  deferred method 366
  inline method 366
  loading attributes 219
  processing limitations 369
  schema, defining in 366
  user considerations 368
LogTable
  Load operator 189
  Stream operator 227
  Update operator 244

M
Multiload
  Teradata PT and 27
Multiload INMOD Adapter operator
  capabilities 195
    definition 196
    InmodName 196
    PrivateLogName 197
    specifications 196
Multiload INMOD Adapter operator
  Format 197
Multiload utility
  Update operator and 241
  MultiLoad utility and 241
  multiple statements 233

N
Non-robust mode
  Stream operator 231
Notify exit routines
  compiling 338
  events, monitoring 337
Export operator events 325
  linking 338
Load operator events
  Load operator
    events 328
Stream operator events 334
Update operator events 330

O
ODBC
  decimal data, handling 202
ODBC operator
  attributes, security 201
  capabilities 199
  checkpointing 205
  definition 200
  HP-UX and 205
  initialization files 203
  multiple parallel instances 202
  Oracle DATE fields 203
    PrivateLogName 202
    restarting 205
    specifications 201
  Teradata Database and 205
    TruncateData 203
ODBC operator, DSNName 202
OLE DB Access Module
  Teradata PT and 27
Online Archive 35
OpenMode
  DataConnector operator 158
Oracle DATE fields
  ODBC operator 203
OS Command operator
  capabilities 207
    definition 208
    IgnoreError 208
    OSCmd 208
    PrivateLogName 209
    specifications 208
    OSCmd
      OS Command operator 208
    OutmodName
      FastExport OUTMOD Adaptor operator 178
Index

P
Pre-processor errors 295
Private log 279
PrivateLogName
  DDL operator 164
  Export operator 172
  FastExport OUTMOD Adaptor operator 179
  FastLoad INMOD Adapter operator 182
  Load operator 190
  MultiLoad INMOD Adapter operator 197
  ODBC operator 202
  OS Command operator 209
  SQL Inserter operator 214
  SQL Selector operator 219
  Stream operator 229
  Update operator 248
product version numbers 3
Public log 277

Q
Query Banding 365

R
Restarting
  DDL operator 166
  Export operator 173
  Load operator 193
  ODBC operator 205
  SQL Inserter 214
  SQL Selector operator 221
  Stream operator 235
  Update operator 248
Restarting a job 318
Robust mode
  Stream operator 231

S
SelectStmt
  Export operator 171
  ODBC operator 202
  ODBC operator 202
  SQL Selector operator 218
SERIALIZE option 229
Session
  Stream operator 235
  Update operator 252
SQL errors 303
SQL Inserter operator
  attributes, security 213
  capabilities 211
  checkpointing 214
  data integrity 214
data loading 214
definition 212
Load operator and 212
PrivateLogName 214
restarting 214
specifications 213
SQL Selector operator 221
  advantages 216
  attributes, security 218
  capabilities 215
decimal database feature 219
definition, for LOB exporting 217
definition, not for LOB exporting 216
errors 316
Export operator and 215
LOB loading attributes 219
PrivateLogName 219
restarting 221
SelectStmt 218
specifications 217
Stream operator
  array support 234
  attributes, security 227
  capabilities 223
  checkpointing 235
definition, 226
dropped tables 236
error limit 228
ersors 307
ErrorTable 228
events 334
instances 235
latency interval 234
LogTable 227
macros 229
multiple statement requests 229, 233
non-robust mode 231
overhead 232
performance, data quality and 232
PrivateLogName 229
recovery logic 232
restarting 235
robust mode 231
SERIALIZE 229
sessions 235
space requirements 234
specifications 226
statement packing 233
TPump utility and 224
System resource errors 297

T
TargetTable
Load operator 189
Update operator 244
tbuild
checkpointing files and 262
errors 295
file name, specifying 261
job name, specifying 261
job variables, assigning 263
job, fatal error and 264
jobname syntax 262
setting options 261
unnamed jobs, running 262
Teradata PT
parallelism 29
platforms, supported 27
processing, basics of 28
Teradata PT job
exit codes
   Exit codes 276
launching 267
managing 269
post job checklist 275
restarting 318
Teradata PT Wizard. See Wizard
TextDelimiter
   DataConnector operator 158
TPump utility
   Stream operator and 224
   Teradata PT and 27
TruncateData
   ODBC operator 203
twbcmd
   job performance, monitoring 270
   job-level commands 270
   periodicity and 273
   rate and 273
twbkill, terminate job 274
twbstat
   current active jobs and 269
   TYPE DATACONNECTOR PRODUCER 35
   TYPE DDL 35
   TYPE EXPORT 35
   TYPE FASTEXPORT OUTMOD 35
   TYPE FASTLOAD INMOD 35
   TYPE INSERTER 35
   TYPE LOAD 35
   TYPE MULTILOAD INMOD 35
   TYPE MULTILOAD INMOD FILTER 35
   TYPE ODBC 35
   TYPE OS COMMAND 35
   TYPE SELECTOR 35
   TYPE STREAM 35
   TYPE UPDATE 35

U
Update operator 241
   AMP usage 249
   attributes, security 244
capabilities 239
checkpointing 248
definition 242
delete task option 253
dropped tables 252
Error capture 246
error limits 245
error tables 311
ers 311
ErrorTable 245
events 330
instances 252
LogTable 244
phases 240
PrivateLogName 248
restarting 248
sessions 252
space requirements 252
specifications 243
TargetTable 244
upserts 249
VARCHAR ARRAY tables 251
WorkTable 246
Update operators
   job limits 253

V
VARCHAR ARRAY table, Update operator 251
version numbers 3

W
WebSphere MQ Access Module
Teradata PT and 26
Wizard 450
   column limits 451
   limitations 450
   main dialog box 451
   Map Source to Destination dialog box 468
   menus 477
   toolbars 477
WorkTable, Update operator 246