# Teradata Parallel Transporter

## User Guide

Release 14.00
B035-2445-071A
June 2012

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

Teradata, Active Enterprise Intelligence, Applications-Within, Aprimo, Aprimo Marketing Studio, Aster, BYNET, Claraview, DecisionCast, Gridscale, MyCommerce, Raising Intelligence, Smarter. Faster. Wins., SQL-MapReduce, Teradata Decision Experts, "Teradata Labs" logo, "Teradata Raising Intelligence" logo, Teradata ServiceConnect, Teradata Source Experts, "Teradata The Best Decision Possible" logo, The Best Decision Possible, WebAnalyst, and Xkoto are trademarks or registered trademarks of Teradata Corporation or its affiliates in the United States and other countries.

Adaptec and SCSISelect are trademarks or registered trademarks of Adaptec, Inc.

AMD Opteron and Opteron are trademarks of Advanced Micro Devices, Inc.

Axeda is a registered trademark of Axeda Corporation. Axeda Agents, Axeda Applications, Axeda Policy Manager, Axeda Enterprise, Axeda Access, Axeda Software Management, Axeda Service, Axeda ServiceLink, and Firewall-Friendly are trademarks and Maximum Results and Maximum Support are servicemarks of Axeda Corporation.

Data Domain, EMC, PowerPath, SRDF, and Symmetrix are registered trademarks of EMC Corporation.

GoldenGate is a trademark of Oracle.

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 is a registered trademark 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.

NetVault is a trademark or registered trademark of Quest Software, Inc. in the United States and/or other countries.

Novell and SUSE are registered trademarks of Novell, Inc., in the United States and other countries.

Oracle, Java, and Solaris are registered trademarks of Oracle and/or its affiliates.

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.

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 registered trademark of Unicode, Inc. in the United States and other countries.

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 email: 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 © 2005-2012 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 Tools and Utilities is a group of client products designed to work with the Teradata Database.

Teradata PT provides high-performance data extraction, loading, and updating operations for 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 14.0
- Teradata Tools and Utilities 14.00
- Teradata Parallel Transporter 14.00

  **Note:** See "Verifying the Teradata PT Version" on page 36 to verify the Teradata Parallel Transporter version number.

To locate detailed supported-release information:

1. Go to http://www.info.teradata.com/.
2. Under **Online Publications**, click **General Search**.
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 6.

| Date and Release | Description |
|---|---|
| June 2012<br>14.00 | Moved the chapter on operational metadata from the Teradata PT Reference to this book.<br>Documented the use of the -r *checkpointDirectory* option of the **tbuild** command.<br>Documented how to specify the wait time for obtaining a file lock for a Teradata PT job.<br>Teradata PT supports loading DataTime data using the Schema-specific VARDATE column data type.<br>Revised “See” references for sample Teradata PT scripts.<br>Removed the following topic: “Setting up the Client System”.<br>Updated documentation of Teradata PT simplified syntax (called “Simplicity”).<br>Documented Teradata PT Best Practices for loading data. |
| November 2011<br>14.00 | Moved 14 chapters describing Teradata PT operators to the Teradata PT Reference.<br>Added appendix C (“Teradata PT Publications”).<br>Moved documentation of (1) the Notify Exit Routines, (2) Teradata Database considerations when running a Teradata PT job, (3) extended character sets, and (4) operational metadata to the Teradata PT Reference.<br>Moved documentation of Teradata PT Easy Loader to chapter 12.<br>Deleted “Example Logs” (formerly appendix C).<br>Teradata PT job supports Teradata Wallet for password security.<br>Documented the script that validates the installation of Teradata PT.<br>Documented the use of $JOBID and $$JOBID variables in Teradata PT scripts.<br>Documented how to specify ARRAY data types in a DEFINE SCHEMA statement.<br>Documentation of the tlogview command to access public and private logs revised. |

| Date and Release | Description |
|---|---|
| August 2011<br>14.00 | A new Schema Mapping operator enables users to verify that Teradata PT job schema definitions correctly describe input data.<br>Teradata PT job scripts support operator templates.<br>Teradata PT Easy Loader supports moving data from Teradata Database tables.<br>Stream operator supports the MacroCharSet option.<br>Procedures to set up the ODBC environment for Teradata PT on IBM z/OS have been documented.<br>Teradata PT samples and job variables files on MVS have been revised<br>Teradata PT notify exit samples on IBM z/OS have been revised.<br>Export operator supports VARCHAR and VARBYTE data with a length less than or equal to the defined length in the DEFINE SCHEMA definition.<br>Description of Update operator error limits has been revised.<br>Documentation of TextDelimiter and EscapeTextDelimiter attributes of the DataConnector operator have been revised.<br>SQL Selector operator supports a single SQL SELECT statement or multiple SELECT statements.<br>Stream operator supports statement independence. |

# Additional Information

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

| Type of Information | Description | Source |
|---|---|---|
| Release overview<br>Late information | Use the Release Definition for the following information:<br>• Overview of all the products in the release<br>• Information received too late to be included in the manuals<br>• Operating systems and Teradata Database versions that are certified to work with each product<br>• Version numbers of each product and the documentation for each product<br>• Information about available training and support center | 1 Go to http://www.info.teradata.com/.<br>2 Under **Online Publications**, click **General Search**<br>3 Type *2029* in the **Publication Product ID** box.<br>4 Click **Search.**<br>5 Select the appropriate Release Definition from the search results. |

<table>
<tr><th>Type of Information</th><th>Description</th><th>Source</th></tr>
<tr><td>Additional product information</td><td>Use the Teradata Information Products web site to view or download specific manuals that supply related or additional information to this manual.</td><td>1 Go to http://www.info.teradata.com/.<br>2 Under the Online Publications subcategory, Browse by Category, click Data Warehousing.<br>3 Do one of the following:<br>• For a list of Teradata Tools and Utilities documents, click Teradata Tools and Utilities, and then select an item under Releases or Products.<br>• Select a link to any of the data warehousing publications categories listed.<br>Specific books related to Teradata PT are as follows:<br>• Teradata Tools and Utilities Access Module Programmer Guide<br>B035-2424<br>• Teradata Tools and Utilities Access Module Reference<br>B035-2425<br>• Teradata Parallel Transporter Application Programming Interface Programmer Guide<br>B035-2516<br>• Teradata Parallel Transporter Operator Programmer Guide<br>B035-2435<br>• Teradata Parallel Transporter Quick Start Guide<br>B035-2501<br>• Teradata Parallel Transporter Reference<br>B035-2436<br>• Teradata Parallel Transporter User Guide<br>B035-2445<br>• Teradata Tools and Utilities for IBM z/OS Installation Guide<br>B035-3128<br>• Teradata Tools and Utilities Installation Guide for Microsoft Windows<br>B035-2407<br>• Teradata Tools and Utilities for Red Hat Enterprise Linux Installation Guide<br>B035-3121<br>• Teradata Tools and Utilities for SUSE Linux Installation Guide<br>B035-3122<br>• Teradata Tools and Utilities for s390x Linux Installation Guide<br>B035-3123<br>• Teradata Tools and Utilities for HP-UX Installation Guide<br>B025-3124</td></tr>
</table>

| Type of Information | Description | Source |
|---|---|---|
| | | • *Teradata Tools and Utilities for IBM AIX Installation Guide* B035-3125<br>• *Teradata Tools and Utilities for Oracle Solaris on AMD Opteron Systems Installation Guide* B035-3126<br>• *Teradata Tools and Utilities for Oracle Solaris on SPARC Systems Installation Guide* B035-3127 |
| 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/.<br>2 Under the **Online Publications** subcategory, **Browse by Category**, click **Data Warehousing**.<br>3 Click **CD-ROM Images**.<br>4 Follow the ordering instructions. |
| Ordering information for manuals | Use the Teradata Information Products web site to order printed versions of manuals. | 1 Go to http://www.info.teradata.com/.<br>2 Under **Print & CD Publications**, click **How to Order**.<br>3 Follow the ordering instructions. |
| General information about Teradata | The Teradata home page provides links to numerous sources of information about Teradata. Links include:<br>• Executive reports, case studies of customer experiences with Teradata, and thought leadership<br>• Technical information, solutions, and expert advice<br>• Press releases, mentions and media resources | • Go to Teradata.com/t/resources.<br>• Select a link. |

# Table of Contents

**Preface** . . . . . . . . . . 3

Purpose . . . . . . . . . . 3
Audience . . . . . . . . . . 3
Supported Releases . . . . . . . . . . 3
Prerequisites . . . . . . . . . . 4
Changes to This Book . . . . . . . . . . 4
Additional Information . . . . . . . . . . 6

## SECTION 1 Teradata PT Basics

**Chapter 1: Introduction to Teradata PT** . . . . . . . . . . 23

High-Level Description . . . . . . . . . . 23
Basic Processing . . . . . . . . . . 26
Teradata PT Parallel Environment . . . . . . . . . . 27
Operator Types . . . . . . . . . . 29
Access Modules . . . . . . . . . . 34
Data Streams . . . . . . . . . . 34
Validating Teradata PT after Installation . . . . . . . . . . 35
Verifying the Teradata PT Version . . . . . . . . . . 36
Switching Versions . . . . . . . . . . 36

**Chapter 2: Teradata PT Job Components** . . . . . . . . . . 37

Understanding Job Script Concepts . . . . . . . . . . 38
Creating a Job Script . . . . . . . . . . 41
Defining the Job Header and Job Name . . . . . . . . . . 41

Using Job Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
Defining a Schema . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
Defining Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
Coding the Executable Section . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
Defining Job Steps . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 62
Fast Track Job Scripting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 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 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70

### Chapter 4: Teradata Database Effects on Job Scripts . . . . . . . . . . . . . . . . . . . . . . . 73

Teradata Database Logon Security . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
Teradata Database Access Privileges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
Optimizing Job Performance with Sessions and Instances . . . . . . . . . . . . . . . . . . . . . . 80
Limits on Teradata PT Task Concurrency . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87

## 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

Common Jobs for Moving Data into a Teradata Database . . . . . . . . . . 97

## Chapter 6: Moving Data from Teradata Database to an External Target . . . . . . . . . . 109

Data Flow Description . . . . . . . . . . 109
Comparing Applicable Operators . . . . . . . . . . 110
Using Access Modules to Process Data Before Writing to External Targets . . . . . . . . . . 112
Common Data Movement Jobs . . . . . . . . . . 113

## Chapter 7: Moving Data within the Teradata Database Environment . . 119

Data Flow Description . . . . . . . . . . 119
Comparing Applicable Operators . . . . . . . . . . 120
Common Jobs to Move Data within a Teradata Database . . . . . . . . . . 121

# SECTION 4 Launching, Managing, and Troubleshooting a Job

## Chapter 8: Launching a Job . . . . . . . . . . 129

Setting tbuild Options . . . . . . . . . . 129
Setting Checkpoint Options . . . . . . . . . . 132
Launching a Teradata PT Job . . . . . . . . . . 136

## Chapter 9: Managing an Active Job . . . . . . . . . . 137

Managing an Active Job . . . . . . . . . . 137
Using twbstat to List Currently Active Jobs . . . . . . . . . . 137
Using the twbcmd Command to Monitor and Manage Job Performance . . . . . . . . . . 138
Using twbkill to Terminate a Job . . . . . . . . . . 142

## Chapter 10: Post-Job Considerations . . . . . 143

Post-Job Checklist . . . . . 143
Exit Codes . . . . . 144
Accessing and Using Job Logs . . . . . 145
Accessing and Using Error Tables . . . . . 150
Effects of Error Limits. . . . . . 154
Dropping Error Tables . . . . . 155
Restart Log Tables . . . . . 156
Strategies for Evaluating a Successful Job . . . . . 156

## Chapter 11: Troubleshooting a Failed Job . . . . . 161

Detecting and Correcting the Cause of Failure . . . . . 161
Common Job Failures and Remedies . . . . . 162
When the Job Fails to Begin Running . . . . . 162
When the Job Fails to Complete . . . . . 169
Operator-Specific Error Handling. . . . . . 171
Load Operator Errors . . . . . 172
Stream Operator Errors . . . . . 176
Update Operator Errors . . . . . 180
SQL Selector Operator Errors . . . . . 185
Additional Debugging Strategies for Complex Job Failures . . . . . 185
Restarting A Job . . . . . 186
Removing Checkpoint Files . . . . . 191
Specifying the Wait Time for a File Lock . . . . . 192

# SECTION 5 Advanced Topics

## Chapter 12: Teradata PT Easy Loader . . . . . 195

Using Teradata PT Easy Loader . . . . . 195

## Chapter 13: Advanced Scripting Strategies . . . . . 203

Data Acquisition and Loading Options . . . . . 203
Data Filtering and Conditioning Options . . . . . 208
Reusing Definitions with the INCLUDE Directive . . . . . 210
Simplifying Scripts with Operator Templates and Generated Schemas. . . . . 211
Using the Job Identifier in Your Job Script . . . . . 224
Using the Multiple APPLY Feature . . . . . 225
Using VARDATE Columns To Reformat DateTime Data . . . . . 226

## Chapter 14: Operational Metadata . . . . . 231

Metadata Types . . . . . 231
Example Metadata Log Output . . . . . 233
Viewing Metadata . . . . . 234
Exporting and Loading Metadata . . . . . 235
Analyzing Job Metadata . . . . . 235
Sending Operational Metadata to TMSM . . . . . 236

## Chapter 15: Best Practices . . . . . 239

Loading Data Using Teradata PT . . . . . 239

## Appendix A: IBM z/OS Samples Files . . . . . 253

Job Script Examples . . . . . 253
JCL Samples. . . . . 254
Job Attribute File. . . . . 254
Teradata PT Catalogued Procedure (PT#TPT) . . . . . 255
Teradata PTLV Catalogued Procedure (PT#TPTLV) . . . . . 255

## Appendix B: Teradata PT Wizard . . . . . 257

Launching TPT Wizard . . . . . 257
Overview . . . . . 257
Wizard Limitations . . . . . 258
Main Window . . . . . 259
Create a New Script . . . . . 261
Stop, Restart, Delete, Edit Jobs . . . . . 281
View Job Output . . . . . 283
Menus and Toolbars . . . . . 285

## Appendix C: Teradata PT Publications . . . . . 287

## Glossary . . . . . 289

## Index . . . . . 293

# List of Figures


Figure 1: Contrasting Traditional Utilities and Teradata PT . . . . . . . . . . . . . . . . . . . . . . . . . . 27

Figure 2: Teradata PT Pipeline Parallelism. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28

Figure 3: Teradata PT Data Parallelism. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

Figure 4: Job Flow Using a FastLoad INMOD Adapter Operator . . . . . . . . . . . . . . . . . . . . . . 32

Figure 5: Job Flow Using an INMOD Adapter Operator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

Figure 6: Job Flow Using an OUTMOD Adapter Operator . . . . . . . . . . . . . . . . . . . . . . . . . . 32

Figure 7: Data Streams . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35

Figure 8: Script Sections . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

Figure 9: Job Header and Job Name . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

Figure 10: Example Schema Definition. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

Figure 11: Defining producer operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50

Figure 12: Export Operator Definition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

Figure 13: Defining consumer operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

Figure 14: Load Operator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54

Figure 15: Example script for defining the DDL operator . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

Figure 16: Multiple Insert Statements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

Figure 17: SELECT Statement in an APPLY Statement. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

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. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71

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 . . . . . 98

Figure 22: Job Example 1B -- Reading Data from a Named Pipe for High Speed Loading . . 99

Figure 23: Job Example 1C -- Reading Data from Multiple Flat Files for High Speed Loading . 99

Figure 24: Job Example 2A -- Reading Data from a Flat File . . . . . . . . . . . . . . . . . . . . . . . . 100

Figure 25: Job Example 2B -- Reading Data from a Named Pipe. . . . . . . . . . . . . . . . . . . . . . 100

Figure 26: Job Example 3 -- Loading BLOB and CLOB Data . . . . . . . . . . . . . . . . . . . . . . . . 101

Figure 27: Job Example 4 -- Pre-processing Data with an INMOD Routine before Loading102

Figure 28: Job Example 5A -- Read Transactional Data from JMS and Load Using the Stream Operator. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

Figure 29: Job Example 5B -- Read Transactional Data from MQ and Load Using the Stream Operator. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

Figure 30: Job Example 5C -- Read Transactional Data from JMS or MQ and Simultaneously Load the Data to a Teradata Database and a Backup File . . . . . . . . . . . . . . . . . . . . . . . . . . . . .104
Figure 31: Job Example 6 -- Loading Data from Other Relational Databases. . . . . . . . . . . . .105
Figure 32: Job Example 7 -- Mini-Batch Loading . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .106
Figure 33: Job Example 8 -- Batch Directory Scan. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .107
Figure 34: Job Example 9 -- Active Directory Scan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .108
Figure 35: Moving Data from a Teradata Database into a Non-Teradata Target . . . . . . . . . .109
Figure 36: Job Example 10 -- Extracting Rows and Sending Them in Delimited Format . . .114
Figure 37: Job Example 11 -- Extracting Rows and Sending Them in Binary or Indicator-mode Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .115
Figure 38: Job Example 12 -- Export Data and Process It with an OUTMOD Routine. . . . .115
Figure 39: Job Example 13 -- Export Data and Process It with an Access Module. . . . . . . . .116
Figure 40: Job Example 14 -- Extract BLOB/CLOB Data and Write It to an External File . .117
Figure 41: Moving Data within the Teradata Database Environment . . . . . . . . . . . . . . . . . . .119
Figure 42: Job Example 15 -- Exporting Data and Loading It into Production Tables . . . . .122
Figure 43: Job Example 16 -- Export Data and Perform Conditional Updates Against Production Tables. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .123
Figure 44: Job Example 17: Delete Data from a Teradata Database Table . . . . . . . . . . . . . . .124
Figure 45: Job Example 18 -- Export BLOB/CLOB Data from One Teradata Database Table to Another . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .124
Figure 46: Parallel Reading and Loading of a File . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .243
Figure 47: Parallel Reading of MQ via UNION ALL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .243
Figure 48: Periodic Loading with Directory Scan . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .246
Figure 49: Switching Operator using a Job Variable . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .247
Figure 50: Extracting, Loading, and Transforming (ELT) . . . . . . . . . . . . . . . . . . . . . . . . . . . .248

# List of Tables

Table 1: Comparison of Teradata PT Operators and Teradata Utilities . . . . . . . . . . . . . . . . . 24
Table 2: Operator Summary. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
Table 3: Producer Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
Table 4: Consumer Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
Table 5: Standalone Operators. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
Table 6: Comparing Update and Stream Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
Table 7: Checkpoint Types and Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 133
Table 8: Load Errors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172
Table 9: Format of ErrorTable1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 173
Table 10: Error Table Columns . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 176
Table 11: Task Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 178
Table 12: Acquisition Error Table Format for the Update Operator . . . . . . . . . . . . . . . . . . . 182
Table 13: Application Error Table Format for the Update Operator. . . . . . . . . . . . . . . . . . . 182
Table 14: Task Example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 183
Table 15: Data Conditioning Syntax Using SELECT. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 208
Table 16: Teradata PT Operator Templates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 211
Table 17: Teradata PT Operator Templates . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 213
Table 18: TMSM Resource Types and Teradata PT Operators . . . . . . . . . . . . . . . . . . . . . . . 236
Table 19: Job Script Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 253
Table 20: Menu Items . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 285
Table 21: Toolbar . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 285

# SECTION 1 Teradata PT Basics

CHAPTER 1

# 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

## 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

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

Table 1: Comparison of Teradata PT Operators and Teradata Utilities (continued)

| Teradata PT Operator | Utility Equivalent | Purpose |
|---|---|---|
| MultiLoad INMOD Adapter operator | MultiLoad INMOD Routine | Reads and preprocesses data from a MultiLoad INMOD data source |
| ODBC operator | OLE DB Access Module | Exports data from any non-Teradata Database that has an ODBC driver |
| OS Command operator | Client host operating system | Executes host operating system commands |
| SQL Inserter operator | BTEQ | Inserts data into a Teradata table using SQL protocol |
| SQL Selector operator | BTEQ | Selects data from a Teradata table using SQL protocol |
| Stream operator | TPump | Continuously loads Teradata tables using SQL protocol |
| Update operator | MultiLoad | Updates, inserts, and deletes rows |

## Platforms

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

**Note:** The 14.00 Teradata PT products are compiled on the AIX 5.3 using 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.

## 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 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.

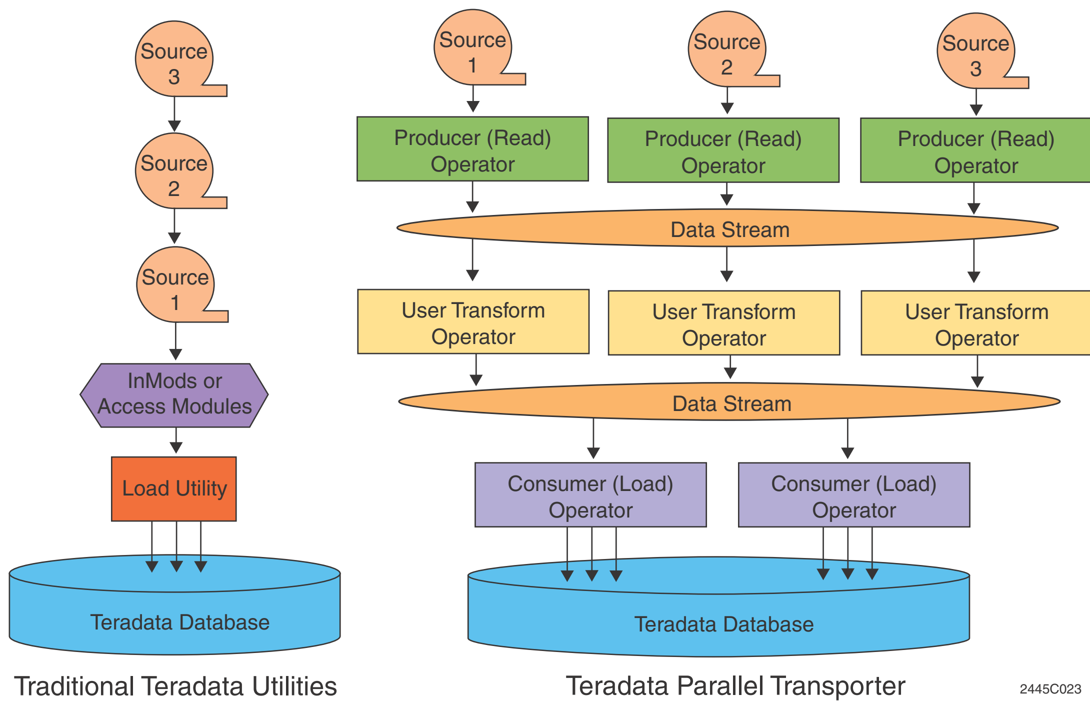
- Filter, condition, and cleanse data.

# 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.

Figure 1: Contrasting Traditional Utilities and 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.

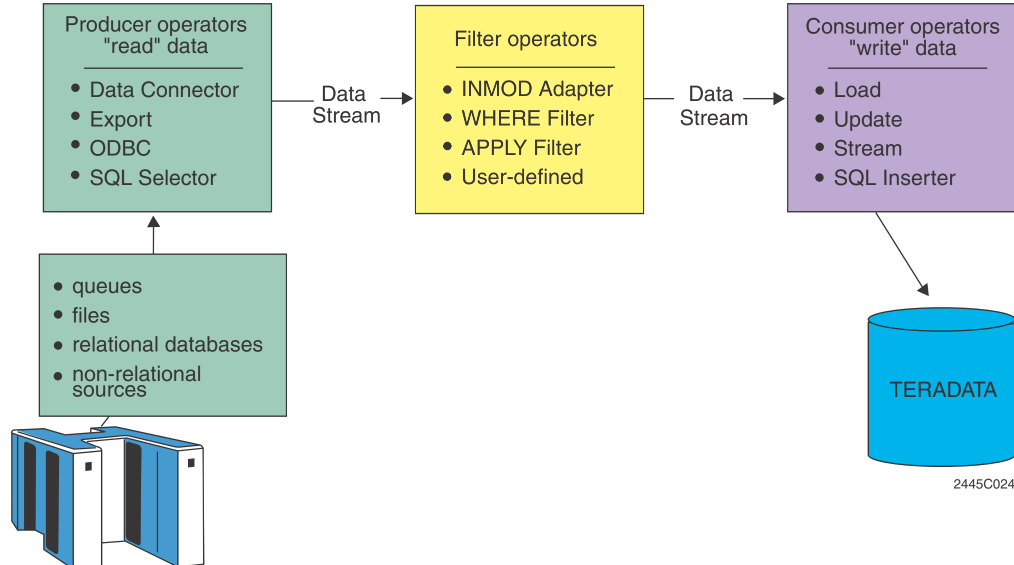
- A filter operator extracts data from the data stream, processes it, then writes it to another data stream.
- A load operator starts writing data to a target as soon as data is available from the data stream.

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

Figure 3: Teradata PT Data Parallelism



# 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

    - 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 32. 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 32. 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 32. 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 32. 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) adaptor 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

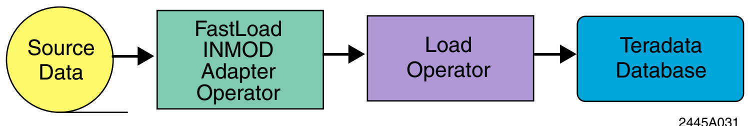
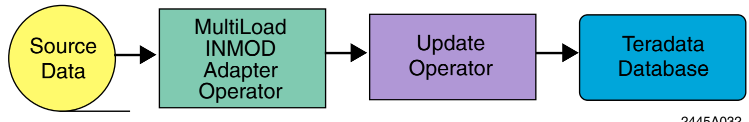



Figure 5 shows a sample job flow using the INMOD Adapter Operator.

Figure 5: Job Flow Using an INMOD Adapter Operator



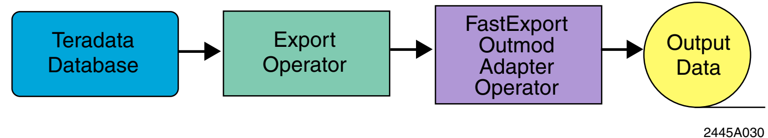

For detailed information, see "FastLoad INMOD Adapter Operator" or "MultiLoad INMOD Adapter Operator" in the *Teradata Parallel Transporter Reference*.

### 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

| Teradata PT Operator Needed | TYPE Definition | Action | Standalone Equivalent |
|---|---|---|---|
| DataConnector operator as a consumer | TYPE DATACONNECTOR CONSUMER | • Writes to flat files<br>• Interfaces with access modules | Data Connector |

Table 2: Operator Summary (continued)

| Teradata PT Operator Needed | TYPE Definition | Action | Standalone Equivalent |
|---|---|---|---|
| DataConnector operator as a producer | TYPE DATACONNECTOR PRODUCER | • Reads flat files<br>• Interfaces with access modules | Data Connector |
| DDL operator | TYPE DDL | Executes various DDL, DML, and DCL statements | DDL statements in utility scripts |
| Export operator | TYPE EXPORT | Reads bulk data from a Teradata Database | FastExport |
| FastLoad INMOD Adapter | TYPE FASTLOAD INMOD | Processes data prior to writing to a data stream | FastLoad INMOD |
| FastExport OUTMOD Adapter operator | TYPE FASTEXPORT OUTMOD | Processes data after an export | FastExport OUTMOD |
| Load operator | TYPE LOAD | Loads empty tables | FastLoad |
| MultiLoad INMOD Adapter operator | TYPE MULTILOAD INMOD | Processes data prior to updates | MultiLoad INMOD |
| MultiLoad INMOD Adapter operator | TYPE MULTILOAD INMOD FILTER | Filters and cleans input data | MultiLoad INMOD |
| ODBC operator | TYPE ODBC | Exports data from ODBC-compliant data sources | OLE DB Access Module |
| OS Command operator | TYPE OS COMMAND | Executes OS commands in a job | OS FastLoad command |
| SQL Inserter operator | TYPE INSERTER | Inserts data using SQL protocol | BTEQ |
| SQL Schema Mapping operator | TYPE SCHEMAMAPPER | Verifies that the schema definition correctly describes the input data and allows data to be displayed in various formats for debugging purposes | n/a |
| SQL Selector operator | TYPE SELECTOR | Exports data using SQL protocol | BTEQ |
| Stream operator | TYPE STREAM | Performs continuous updates, deletes, and inserts into multiple tables | TPump |
| Update operator | TYPE UPDATE | Performs bulk updates, deletes, and inserts | MultiLoad |
| Update operator as a standalone | TYPE UPDATE DeleteTask attribute | Deletes rows from a single table with a DELETE Task | MultiLoad DELETE |

**Note:** Avoid using the keywords TYPE CONSUMER, TYPE PRODUCER, TYPE FILTER, OR TYPE STANDALONE in any operator definition.

**Note:** Teradata PT now supports Online Archive. For more information, see *Database Administration*, B035-1093.

# 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 system 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.

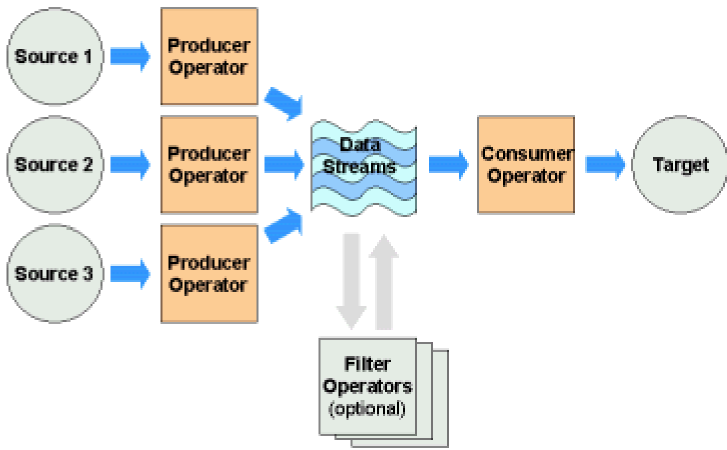
A Teradata PT job moves data from the sources to the targets through data streams. Data moves from producer operator(s) through the data streams to consumer operator(s):

- 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.

Figure 7: Data Streams




# Validating Teradata PT after Installation

To run the quick start validation scripts:

- Create a database user name and password.
- CD to the following directory:

  `$TWB_ROOT/sample/validate`

  - On Unix platforms, execute the following script:

    `./tptvalidate.ksh` *`Tdpid UserName UserPassword`*
  - On Windows platforms, execute the following script:

    `tptvalidate.bat` *`Tdpid UserName UserPassword`*

    where:

    the validation script executes the following scripts in the quickstart directory:

    - qsetup.txt
    - qstart1.txt
    - qsetup2.txt
    - qstart2.txt
    - qcleanup.txt.

    *TpdId* is a database ID.

    *UserName* is a database user name.

    *UserPassword* is a database user password.

# 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 Client installation guides listed in the Preface.

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:


- Understanding Job Script Concepts
- Creating a Job Script
- Fast Track Job Scripting

# Understanding Job Script Concepts

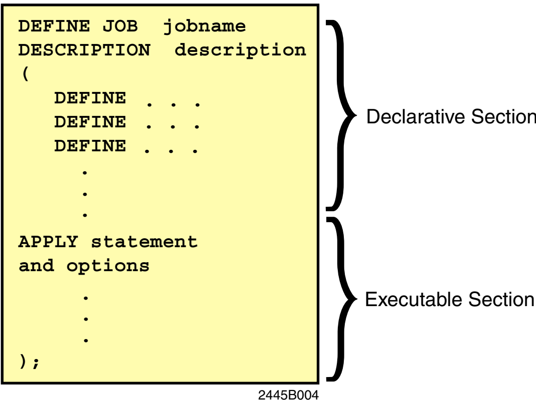
It is important to understand the following basic concepts about job script components and structure before attempting to create or edit a Teradata PT job script.

## 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:

- **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 208.

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.

## 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 data type 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
```

```
DETERMINISTIC
PARAMETER STYLE TD_GENERAL
NO SQL
RETURNS NULL ON NULL INPUT;')
```

- **Using Comments** - Teradata PT supports the use of C-style comments anywhere in a Teradata PT job script; for example:

```
/*<comment>*/
```

# 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.

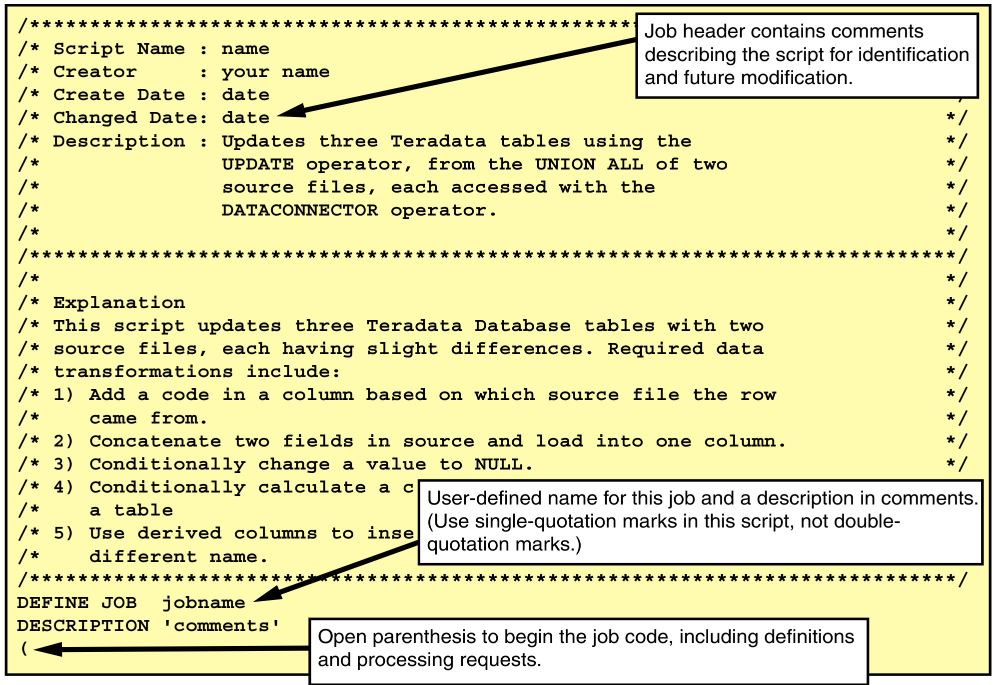
- Defining the Job Header and Job Name
- Defining a Schema
- Defining Operators
- Coding the Executable Section
- Defining Job Steps

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.

Figure 9: Job Header and Job Name




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 Updateddmmyy," 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 129.

# 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 in the Teradata PT configuration file by using the GlobalAttributeFile parameter.

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

- **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 can also be used in a quoted string for DML and DDL statements:

```
   'INSERT INTO TABLE ' || @TargTable || ' (:col1, :col2, . . ., :coln);'
```

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 131.

## 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.

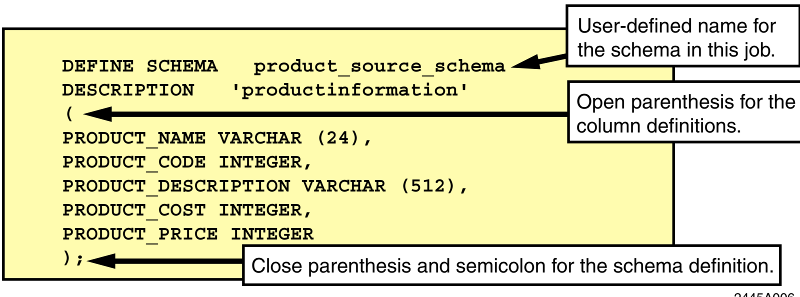
The value following the keyword SCHEMA in a DEFINE OPERATOR statement identifies the schema that the operator will use to process job data. Schemas specified in operator definitions must have been previously defined in the job script. To determine how many schemas you must define, observe the following 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 UTF16 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




## 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 203.

### Example: Multiple Schemas in a Job Script

```
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).

## Specifying ARRAY Data Types

A column that is defined as an ARRAY data type in a Teradata table must be specified as a VARCHAR data type in the DEFINE SCHEMA statement.

The external representation for an ARRAY data type is VARCHAR.

### Example 1

Here is a sample Teradata table definition that includes a one-dimensional ARRAY data type for the COL003 column:

```
CREATE SET TABLE SOURCE_TABLE ,NO FALLBACK ,
     NO BEFORE JOURNAL,
     NO AFTER JOURNAL,
     CHECKSUM = DEFAULT,
     DEFAULT MERGEBLOCKRATIO
     (
      EMP_ID INTEGER,
      EMP_NO BYTEINT,
      COL003 SYSUDTLIB.PHONENUMBERS_ARY,
      COL004 SYSUDTLIB.DECIMAL_ARY,
      COL005 SYSUDTLIB.INTEGER_ARY)
UNIQUE PRIMARY INDEX ( EMP_ID );
```

### Example 2

Here is a sample definition for the PHONENUMBERS_ARY data type:

```
CREATE TYPE PHONENUMBERS_ARY AS CHAR(10) CHARACTER SET LATIN ARRAY [2];
```

### Example 3

Here is a sample definition for the DECIMAL_ARY data type:

```
CREATE TYPE DECIMAL_ARY AS DECIMAL(5,2) ARRAY[2];
```

### Example 4

Here is a sample definition for the INTEGER_ARY data type:

```
CREATE TYPE INTEGER_ARY AS INTEGER ARRAY[2];
```

### Example 5

Here is a sample Teradata PT schema definition for the sample SOURCE_TABLE table:

```
   DEFINE SCHEMA EMPLOYEE_SCHEMA
   DESCRIPTION 'SAMPLE EMPLOYEE SCHEMA'
   (
      EMP_ID   INTEGER,
      EMP_NO   BYTEINT,
      COL003   VARCHAR(47),
      COL004   VARCHAR(17),
      COL005   VARCHAR(25)
   );
```

In the above example, the COL003 column is defined as VARCHAR(47), because it is the maximum representation for the COL003 column in the table.

Here is the calculation for the maximum representation for the COL003

column:

  1 byte for the left parenthesis

+  1 byte for the single quote

+ 10 to 20 bytes for the first element

+  1 byte for the single quote

+  1 byte for the comma

+  1 byte for the single quote

+ 10 to 20 bytes for the second element

+  1 byte for the single quote

+  1 byte for the right parenthesis

----

  47 bytes

Here are 2 sample data for the COL003 column:

```
Sample data 1: ('3105551234','3105551234')
Sample data 2: ('''''''''''''''''''''','''''''''''''''''''''')
```

Sample data 1 contains 2 elements of phone numbers.Sample data 2 contains 2 elements of all single quote characters.

In the above example, the COL004 column is defined as VARCHAR(17), because it is the maximum representation for the COL004 column in the table.

Here is the calculation for the maximum representation for the COL004 column:

  1 byte for the left parenthesis

+  1 to 7 bytes for the first element

+  1 byte for the comma

+  1 to 7 bytes for the second element

+  1 byte for the right parenthesis

----

  17 bytes

Here are 2 sample data for the COL004 column:

```
Sample data 1: (-123.45,888.10)
Sample data 2: (+123.45,-888.10)
```

In the above example, the COL005 column is defined as VARCHAR(25), because it is the maximum representation for the COL005 column in the table.

Here is the calculation for the maximum representation for the COL005

column:

  1 byte for the left parenthesis

+ 1 to 11 bytes for the first element

+ 1 byte for the comma

+ 1 to 11 bytes for the first element

+ 1 byte for the right parenthesis

----

 25 bytes

Here are 2 sample data for the COL005 column:

```
Sample data 1: (-2147483648,+2147483647)
Sample data 2: (0,0)
```

Use the Teradata SQL HELP TYPE statement to determine the maximum length for the ARRAY data type, as given by the value returned by MaxLength.

For more information about the external representations for the ARRAY data type, see Appendix B: "External Representation for UDTs" in *SQL Data Types and Literals*.

# Defining Operators

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 *Teradata Parallel Transporter Reference*.

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.

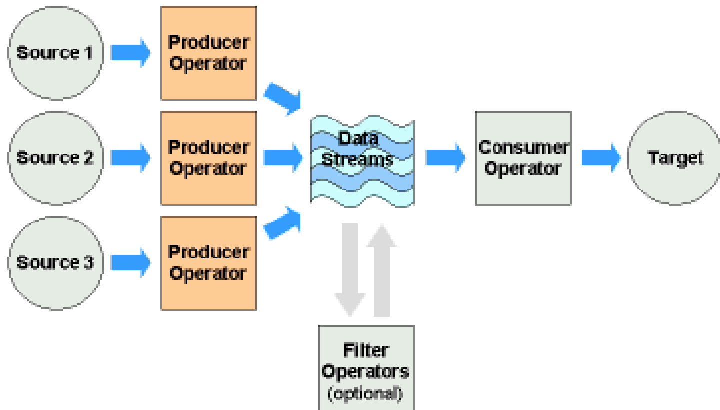
     For attribute definitions and syntax, see *Teradata Parallel Transporter Reference*.
   - The declaration of any optional attributes that are meaningful to the operator with the optional assignment of values.

     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

| Operator | Description |
|---|---|
| **Produces Data from** Teradata Database | |
| Export | 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. |
| SQL Selector | Selects data from Teradata tables using SQL sessions. The only producer operator that can handle LOB data. |
| **Produces Data from a Non-Teradata Data Sourc**e | |
| DataConnector (producer) | The DataConnector operator accesses files either directly or through an access module, and then writes it to the data stream. |
| ODBC | The ODBC operator extracts data from any ODBC provider, such as Oracle or SQL Server on Windows, UNIX, and z/OS platforms, and then writes it to the data stream. |
| **Produces and Processes Data from a Non-Teradata Data Sourc**e | |
| FastLoad INMOD Adapter | The FastLoad INMOD adapter uses FastLoad INMOD routines to read and preprocess input data from flat files, and then places it in the data stream. |
| MultiLoad INMOD Adapter | The MultiLoad INMOD adapter uses MultiLoad INMOD routines to read and preprocess input data from flat files, and then places it in the data stream. |

## Script Requirements

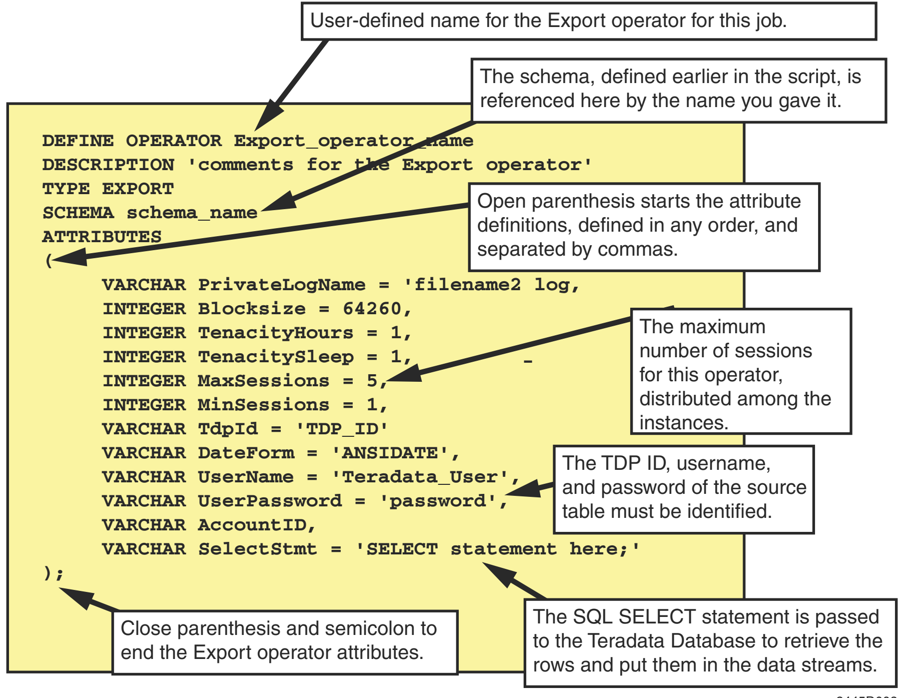
When you define a producer operator in a Teradata PT script, required specifications include:

- 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:

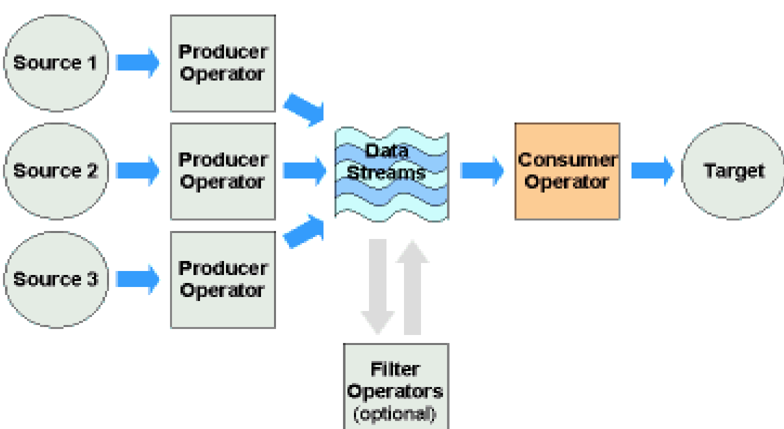
Figure 12: Export Operator Definition




## 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

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

## 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

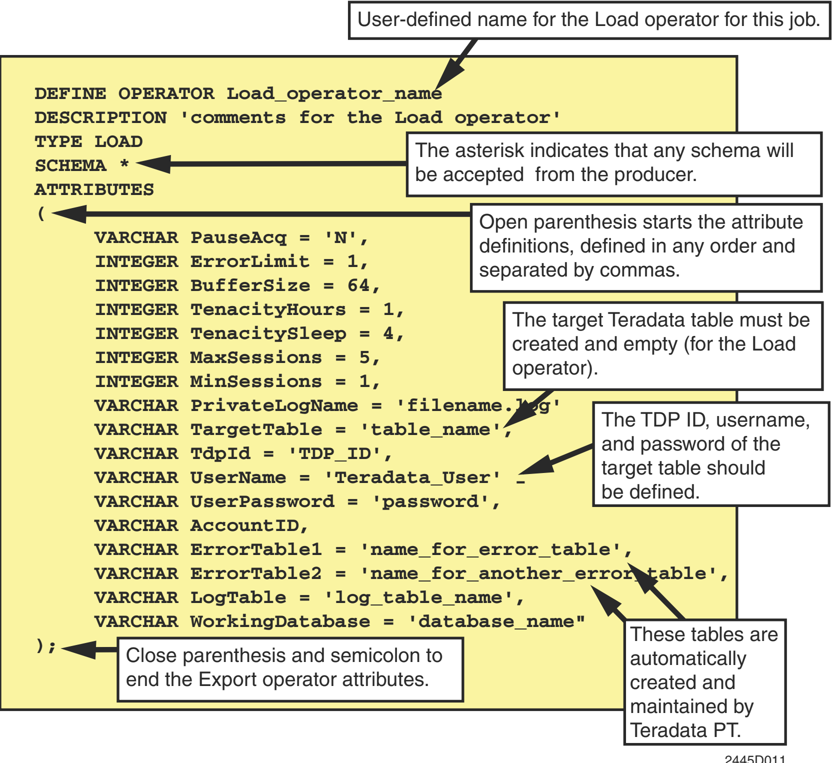
Teradata PT limits the number of tables consumers can load simultaneously, as follows:

| Teradata PT Operator | Maximum Target Tables |
|---|---|
| Load | 1 |
| Update | 5 |
| Stream | 127 |

| Teradata PT Operator | Maximum Target Tables |
|---|---|
| SQL Inserter | 1 |

### Example: Consumer Operator Definition

Figure 14: Load Operator




## Defining Standalone Operators

A standalone operator can be used for processing that does not involve sending data to or receiving data from the Teradata PT operator interface, and thus does not use data streams. If a standalone operator is used, it must be:

- the only operator used in the script, if job steps *are not* used
- the only operator used in the job step, if job steps *are* used

Table 5 contains a list of Teradata PT standalone operators.

Table 5: Standalone Operators

| Operator | Description |
|---|---|
| DDL | 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. |

Table 5: Standalone Operators (continued)

| Operator | Description |
|---|---|
| Update (standalone) | Use the Update operator as a standalone operator only in cases where it is performing the Delete Task and there is no data needed for the DELETE SQL request. |
| OS Command | Executes OS commands on the client host system as part of the Teradata PT job execution |

### 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'
);
```

2445E015

## 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 actions; *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,
```

```
);
```

**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 58.
  - 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 58.
- 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:

| Attribute | Operator |
|---|---|
| ErrorList | DDL |
| • TargetTable<br>• ErrorTable1<br>• ErrorTable2 | Load |
| • TargetTable<br>• WorkTable<br>• ErrorTable1<br>• ErrorTable2 | Update |
| TraceLevel | • DDL<br>• Export<br>• Load<br>• ODBC<br>• OS Command<br>• SQL Inserter<br>• SQL Selector<br>• Stream<br>• Update |

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 55.

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.

- 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 40.

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])
```

The APPLY statement is used with a consumer operator.

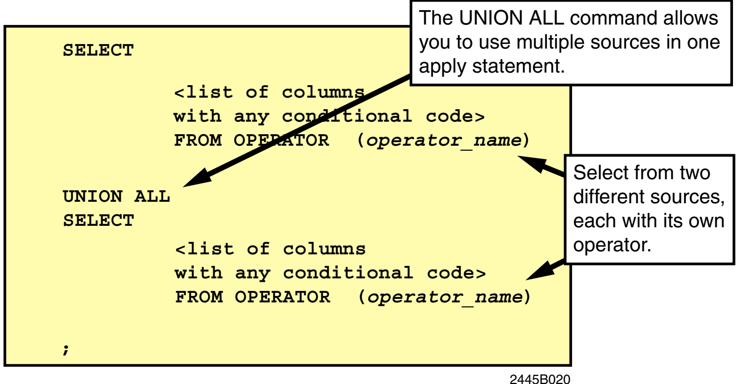
Conditional code using the CASE statement.

2445B019

2 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




## 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:

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

2445A017

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

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

2445A018

## 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 *Teradata Parallel Transporter Reference*.

# 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.

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 8: “Launching a Job.”

## Fast Track Job Scripting

Teradata PT provides some scripting aids that help make creating a Teradata PT job easier.

| Scripting Aid | Description |
| --- | --- |
| Example jobs and sample scripts | A collection of commonly used Teradata PT jobs is shown in:<br>• Chapter 5: “Moving External Data into Teradata Database.”<br>• Chapter 6: “Moving Data from Teradata Database to an External Target.”<br>• Chapter 7: “Moving Data within the Teradata Database Environment.”<br>Sample script for the common jobs shown in Chapters 5 through 7 are available in the following directory: <install-directory>/sample/userguide. |
| 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.<br>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.<br>For information, see Appendix B: “Teradata PT Wizard.” |

# SECTION 2 Pre-Job Setup

CHAPTER 3

# Job Setup Tasks

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.

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

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.

| Platform | Configuration File |
|---|---|
| Windows | `<installation directory>\twbcfg.ini` |
| UNIX | `<installation directory>/twbcfg.ini`<br>Configuration options can also be customized by placing the following configuration file in the home directory:<br>`$HOME/.twbcfg.ini` |
| z/OS | Specified through the DDNAME of *ATTRFILE*. |

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

For example, on a UNIX system:

```
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.

# 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 systems.

- **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:

- 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 43.

For information on specifying job variables on the command line at job launch, see "Assigning Job Variables on the Command Line" on page 131.

# 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 *Teradata Parallel Transporter Reference*.

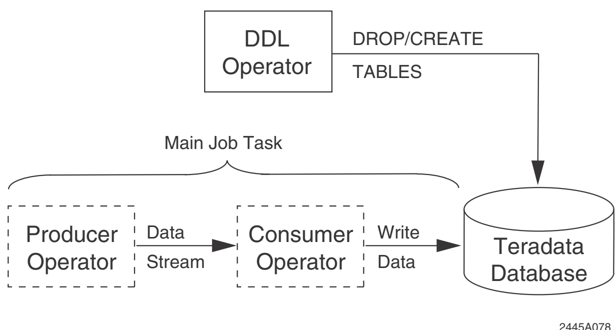
## Objective

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 18: Setup Tables Prior to Loading Data




### Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide01a.txt**: High Speed Bulk Loading from Flat Files into an Empty Teradata Database Table.

Other sample scripts 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.

## 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.

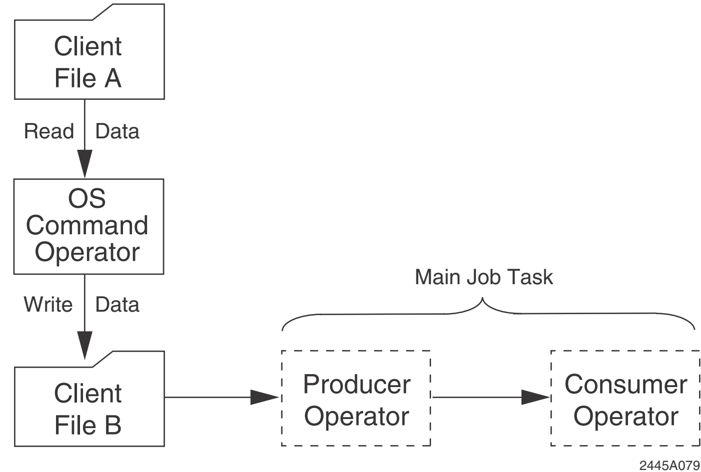
For information on the DDL operator, see *Teradata Parallel Transporter Reference*.

### 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 19: Copy Files from One Client Location to Another before Executing and Extract/Load Operation




## Sample Script

For the sample script that corresponds to this job, see the sample/userguide directory:

uguide03.txt: Loading BLOB and BLOB Data into Teradata Database.

Other scripts 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.

CHAPTER 4

# Teradata Database Effects on Job Scripts

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 *Teradata Parallel Transporter Reference*.

# 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
- Using Teradata Wallet in the Teradata PT Job
- 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*.

### *When Accessing a Teradata Database*

The following operators can directly access a Teradata Database and therefore require submission of more detailed logon information:

- DDL
- Export
- Load
- SQL Inserter
- SQL Selector
- Stream
- Update

## Teradata Database Authentication

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

| Security Attribute | Description | Strategy |
|---|---|---|
| UserName | The Teradata Database user name. | All users employing Teradata Database authentication must be defined in the Teradata Database.<br>For information about creating users, see *Database Administration*.<br>For information about assigning passwords, see *Security Administration*. |
| UserPassword | The Teradata Database password associated with the UserName | |
| 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:<br>• For channel-attached clients, specify the identity of the Teradata Director Program through which Teradata PT connects to the database. For example: `TDP6`<br>• 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 | User name, 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.

| Security Attribute | Description | Strategy |
|---|---|---|
| UserName | The name of the user being authenticated for access to Teradata Database | Required<br>Specify a Teradata Database user name and password. |
| UserPassword | The password associated with the UserName | |
| TdpId | Identifies the connection to the Teradata Database | Optional<br>If you don't specify a TdpId, the system will use the default TdpId, as defined in the Teradata Client clispb.dat.<br>To specify a TdpId, do the following:<br>• For channel-attached clients, this value is the identity of the Teradata Director Program through which Teradata PT connects to the database. For example:<br>`TDP6`<br>• For network-attached clients this value is the name of the interface to the Teradata Database system, or logical host group. For example:<br>`cs4400S3` |
| LogonMech | The security mechanism used to authenticate the user.<br>TD2 is required for all Teradata Database authentication. | Optional, depending on security setup.<br>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. |

## External Authentication

In some cases the user name 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 not use external authentication to log on with 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>
<tr><th>Security Attribute</th><th>Description</th><th>Strategy</th></tr>
<tr><td>UserName</td><td>The name used to log on to the network prior to launching the job script.</td><td rowspan="2">Optional:<br>• For single sign-on: The user name employed for the initial network logon must match a user name defined in the Teradata Database. No additional user name and password information is required.<br>• For other external authentication methods (for example, LDAP or Kerberos), specify the user name and password values in one of the following ways:<br>  • As values for the UserName and UserPassword attributes, except for logons that require use of LogonMechData (see below).<br>  • As the value for the LogMechData attribute.<br>Note: Do not declare the UserName or UserPassword attributes if you plan to enter user name and password data in LogonMechData.</td></tr>
<tr><td>UserPassword</td><td>The network password (not the Teradata Database password) associated with the UserName)</td></tr>
<tr><td>TdpId</td><td>Identifies the connection to the Teradata Database</td><td>Optional<br>If you don't specify a TdpId, the system will use the default Tdpid, as defined in the Teradata Client clispb.dat. Specify either:<br>• For channel-attached clients, specify the identity of the Teradata Director Program through which Teradata PT connects to the database. For example: <code>TDP6</code><br>• For network-attached clients, specify the name of the interface to the Teradata Database system, or logical host group. For example: <code>cs4400S3</code></td></tr>
<tr><td>LogonMech</td><td>The security mechanism that authenticate the user.<br>Similar to the <code>.logmech</code> statement in a Teradata Database logon string.</td><td>Required unless the external authentication mechanism is the default.<br>Choose among the following, depending on authentication method.<br>• Use LDAP for directory sign-on<br>• Use KRB5 or NTLM for single sign-on and sign-on as logons.</td></tr>
<tr><td>LogonMechData</td><td>Data required by external authentication mechanisms to complete the logon.<br>Similar to the <code>.logdata</code> statement in a Teradata Database logon string.</td><td>Optional<br>LogonMechData contains the user name, password, and in some cases, other information.<br>Entering user credential information in LogonMechData is required for all logons that specify profile=profilename or user=username, to differentiate among multiple applicable profiles or users.<br>Note: Do not declare the LogonMechData attribute if you plan to enter user name and password data in UserName and UserPassword.</td></tr>
</table>

## 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
- Load
- SQL Inserter
- SQL Selector
- Stream
- Update

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.

## Using Teradata Wallet in the Teradata PT Job

Users can store certain Teradata PT attributes using Teradata Wallet, which protects the attributes in encrypted form so that only the owning user can access them.

After an attribute is stored, you can use Teradata Wallet to retrieve the stored attribute using a string name.

For example, to specify the option for the Password attribute in a Teradata PT job using Teradata Wallet, use the following syntax:

```
VARCHAR Password = '"$tdwallet(password alias)"'
```

where *password alias* is the string name that contains the value for the stored password as shown in the following example:

```
VARCHAR Password = '"$tdwallet(MyPassword)"'
```

To specify the option for the TdpId attribute in a Teradata PT job script using Teradata Wallet, use the following syntax:

```
VARCHAR TdpId = '"$tdwallet(tdpid alias)"'
```

where *tdpid alias* is the string name that contains the value for the stored TdpId as shown in the following example:

```
VARCHAR TdpId = '"$tdwallet(MyTdpId)"'
```

You can only use the Teradata Wallet for the following attributes:

- Password
- Username
- TdpId
- AccountId

You cannot use Teradata Wallet to store any other attributes.

For information on Teradata Wallet, see *Security Administration*.

### 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 84 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 84 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* * *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.)

Formula for calculating shared memory allocated for a job:

```
[65000 x (n x m) x 2] bytes + [65000 x (n + m)] bytes
```

### Examples

Shared memory used by 2 producers and 2 consumers:

```
(65000 x 2 x 2 x 2) bytes + (65000 x (2 + 2)) bytes = 780000 bytes
```

Shared memory used by 4 producers and 4 consumers:

```
(65000 x 4 x 4 x 2) bytes + (65000 x (4 + 4)) bytes = 2600000 bytes
```

Shared memory used by 16 producers and 1 consumer:

```
(65000 x 16 x 1 x 2) bytes + (65000 x (16 + 1)) bytes = 3185000 bytes
```

**Note:** The default shared memory size for a job is 10M. You can use the **-h** option of the **tbuild** command to extend the shared memory size for a job. For more information about the **-h** option, please refer to the **tbuild** section of the *Teradata Parallel Transporter Reference*.

## 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 (A-K)*.

## 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

CHAPTER 5

# Moving External Data into Teradata Database

This chapter describes several 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
- 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

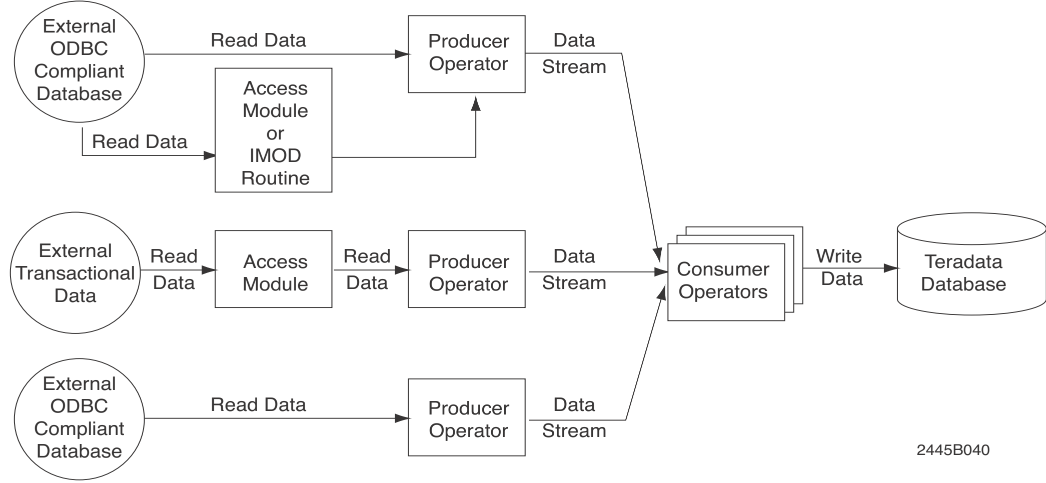



Figure 20 shows a composite of the possible paths for moving data from an external source to a Teradata Database. Note that Job Example 5C (Figure 30) allows for writing data to an external data target in parallel with writing data to a Teradata Database.

# 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.

  or

- 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.

  and

- 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 59 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:

| Operator | Description |
|---|---|
| **Operators that Read Data from External Sources** | |
| DataConnector Operator | Reads flat files from an external data source. Functions similarly to the standalone Teradata DataConnector utility.<br>**Features:**<br>• Can read a specific file or can be used to scan a directory.<br>• Interfaces with all Teradata PT supported access modules.<br>**Limitations:**<br>• Cannot read data from ODBC-compliant data sources.<br>For details, see *Teradata Parallel Transporter Reference*. |
| ODBC Operator | Reads data from most ODBC-compliant data sources.<br>**Limitations**:<br>• Cannot interface with access modules.<br>For details, see *Teradata Parallel Transporter Reference*. |

| Operator | Description |
|---|---|
| **Operators that Read and Modify Data from External Sources** | |
| FastExport INMOD Adapter Operator | Uses FastExport INMOD routines to read data from external files and then process it before sending it to the data stream.<br>For details, see *Teradata Parallel Transporter Reference*. |
| FastLoad INMOD Adapter Operator | Uses FastLoad INMOD routines to read data from external files and then process it before sending it to the data stream.<br>For details, see *Teradata Parallel Transporter Reference*. |
| MultiLoad INMOD Adapter Operator | Uses MultiLoad INMOD routines to read data from external files and then process it before sending it to the data stream.<br>For details, see *Teradata Parallel Transporter Reference*. |

## 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 59 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:

| Operator | Description |
|---|---|
| Load Operator | Inserts data at high speed into a single empty Teradata Database table. Function is similar to the standalone FastLoad utility.<br>**Features:**<br>• Best used for the initial data loads into Teradata Database tables.<br>**Limitations:**<br>• Does not support UPDATE, SELECT, or DELETE operations.<br>• The target table must be empty, with no defined secondary indexes.<br>• Multiple parallel instances of the Load operator can be used in a job, but they must all insert data into the same table.<br>For details, see *Teradata Parallel Transporter Reference*. |

| Operator | Description |
|---|---|
| SQL Inserter Operator | Uses a single SQL session to insert data into a Teradata Database in either an empty or populated table.<br>**Features:**<br>• Protects data integrity. If an error is encountered during the INSERT operation, SQL Operator backs out all rows it has inserted for the job since the last checkpoint.<br>• Use of multiple instances of SQL Inserter allows parallel loading of LOBs.<br>**Limitations:**<br>• Is slower than other operators capable of writing to Teradata Database.<br>• Will terminate the job if an attempted insert would duplicate an existing row in the target table.<br>For details, see *Teradata Parallel Transporter Reference*. |
| Stream Operator | Performs high-speed, low-steady volume SQL DML transactions, INSERT, UPDATE, DELETE, or UPSERT on Teradata Database tables. Function is similar to the standalone Teradata Tpump utility.<br>Stream operator or Update operator can be used for many similar tasks. For comparison of function to help you choose the best operator for the job, see "Comparing Update and Stream Operators" on page 95.<br>For details, see *Teradata Parallel Transporter Reference*. |
| Update Operator | Performs high-speed, high-volume SQL DML transactions, INSERT, UPDATE, DELETE, or UPSERT on Teradata Database tables. Function is similar to the standalone Teradata MultiLoad utility.<br>Update operator or Stream operator can be used for many similar tasks. For comparison of function to help you choose the best operator for the job, see "Comparing Update and Stream Operators" on page 95.<br>For details, see *Teradata Parallel Transporter Reference*. |

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

| Teradata PT Operator | Maximum Target Tables |
|---|---|
| Load | 1 |
| Update | 5 |
| Stream | 127 |
| SQL Inserter | 1 |

## 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

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

Table 6: Comparing Update and Stream Operators (continued)

| Parameter | Update Operator | Stream Operator |
|---|---|---|
| DML Statements | Uses actual DML statements. | Uses macros to modify tables rather than actual DML commands. |
| Work Tables | Requires one work table per target table. | Work tables not required. |

# 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.

| Access Module | Description |
|---|---|
| Java Message Service (JMS) | 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.<br>Teradata JMS Access Module caches the message throughput data stream in a fallback data file that supports checkpoint and restart functions. |
| Named Pipes | Provides access to data from a named pipe and sends it to a DataConnector producer operator.<br>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.<br>This access module is not available on z/OS platforms. |
| OLE DB | 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. |
| WebSphere MQ | Provides access to transactional data from IBM WebSphereMQ.<br>The Teradata WebSphere MQ Access Module also caches the output message stream in a fallback data file that supports checkpoint and restart functions. |

### 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 Teradata PT jobstream:

```
//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.

## Common Jobs for Moving Data into a Teradata Database

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 lists the most common job examples.

- 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

## 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

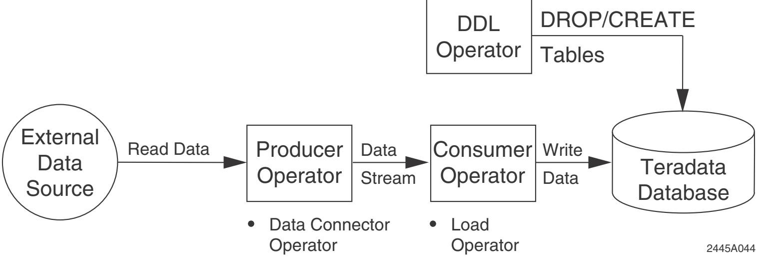


Figure 22: Job Example 1B -- Reading Data from a Named Pipe for High Speed Loading

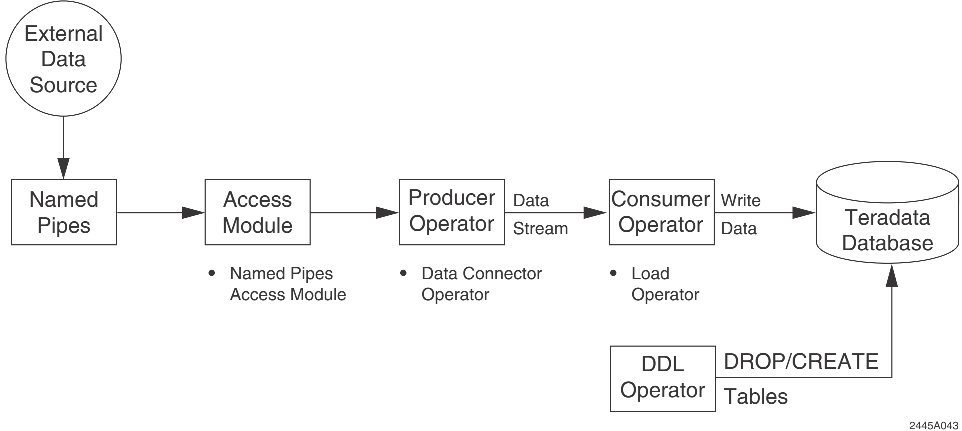
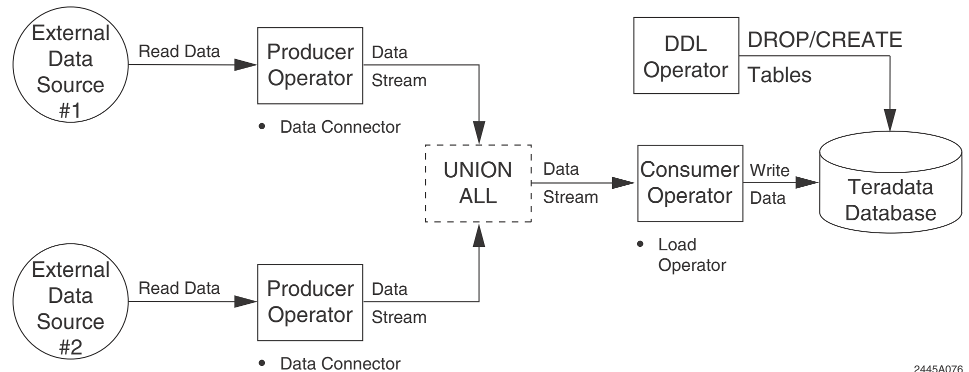



Figure 23: Job Example 1C -- Reading Data from Multiple Flat Files for High Speed Loading




## Samples Scripts

For the sample scripts that correspond to the three variations of this job, see in the sample/ userguide directory:

- **uguide01a.txt**: High Speed Bulk Loading from Flat Files into an Empty Teradata Database Table.
- **usguide01b.txt**: High Speed Bulk Loading from a Named Pipe into an Empty Teradata Database Table.
- **uguide01c.txt**: High Speed Loading from Two Flat Files into an Empty Teradata Database Table.

## 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.

## 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

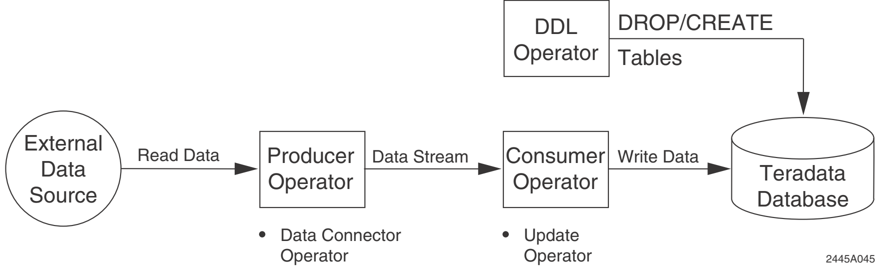
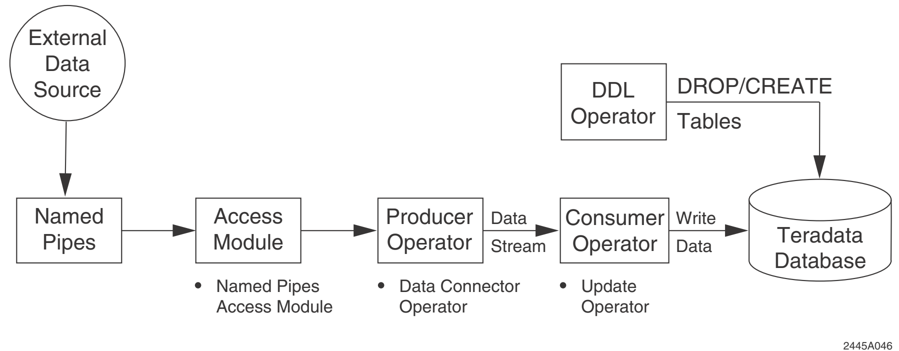



Figure 25: Job Example 2B -- Reading Data from a Named Pipe




### Sample Scripts

For the sample scripts that correspond to the two variations of this job, see in the sample/userguide directory:

- **uguide02a.txt:** Reading Data Direct from Source Files and Performing INSERT, UPDATE, and DELETE on Multiple Teradata Database Tables.

- **uguide02b.txt**: Reading Data from a Named Pipe and Performing INSERT, UPDATE, and DELETE on Multiple Teradata Database Tables.

### 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.

## Job Example 3: Loading BLOB and CLOB Data

### Job Objective

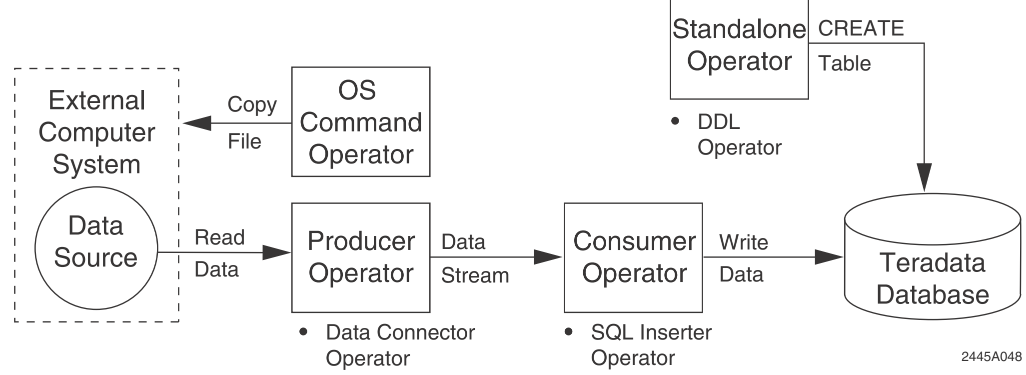
Extract inline BLOB/CLOB data from files and load it into one or more Teradata Database tables.

For detailed information on inline LOB processing, see *Teradata Parallel Transporter Reference*.

### 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




### Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide03.txt**: Loading BLOB and BLOB Data into Teradata Database.

### Rationale

This job uses:

- 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 the only operator that can load BLOB/CLOB data into Teradata Database tables.

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

### Job Objective

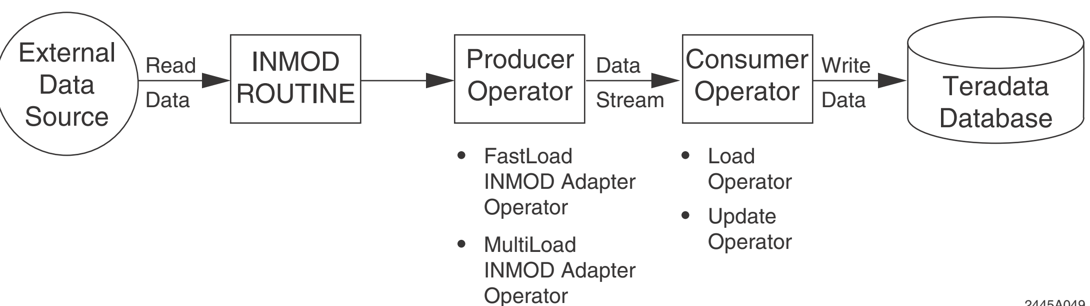
Read data from external source files and pre-process it with an INMOD routine before loading it into Teradata Database tables. There are two variations of this job:

- 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.

Figure 27: Job Example 4 -- Pre-processing Data with an INMOD Routine before Loading




### Sample Scripts

For the sample scripts that correspond to the two variations of this job, see in the sample/userguide directory:

- **uguide04a.txt**: Pre-processing Data with an INMOD Routine Before Loading It into an Empty Teradata Database Table (FastLoad Protocol).
- **uguide04b.txt**: Pre-processing Data with an INMOD Routine Before Loading It into Teradata Database Tables (MultiLoad Protocol).

### 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).

## 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

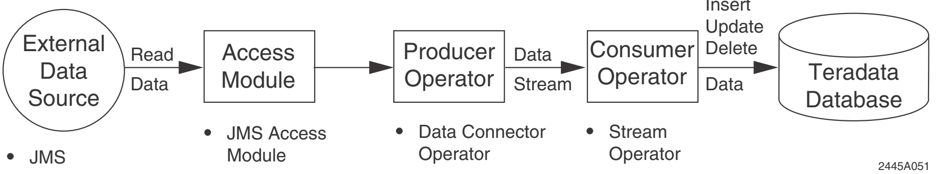



Figure 29: Job Example 5B -- Read Transactional Data from MQ and Load Using the Stream Operator

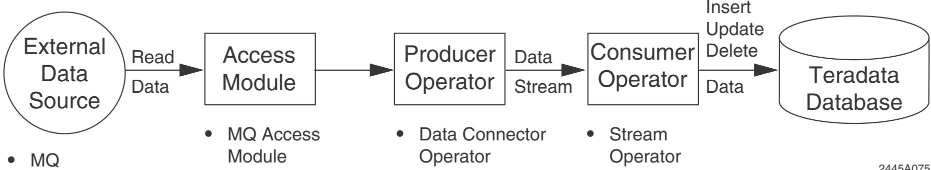
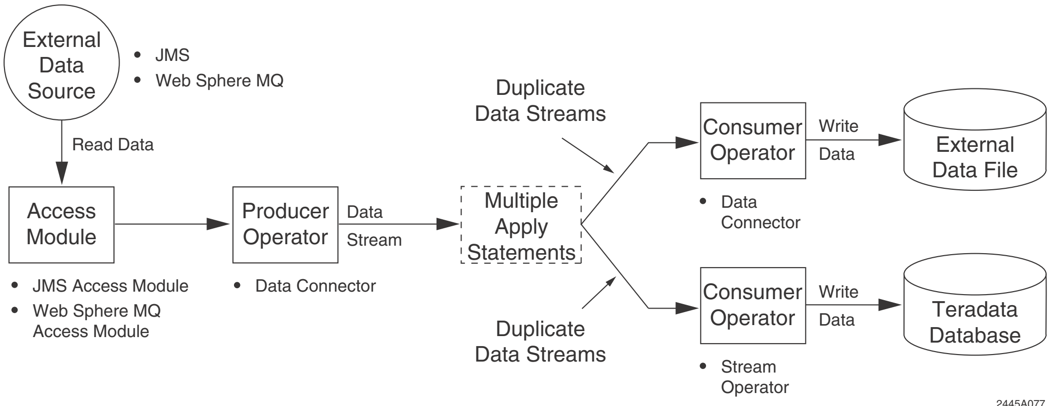


Figure 30: Job Example 5C -- Read Transactional Data from JMS or MQ and Simultaneously Load the Data to a Teradata Database and a Backup File




### Sample Scripts

For the sample scripts that correspond to the three variations of this job, see in the sample/userguide directory:

- **uguide05a.txt**: Continuous Loading of Transactional Data from MQ.
- **uguide05b.txt**: Continuous Loading of Transactional Data from JMS.
- **uguide05c.txt**: Intermediate File Logging Using Multiple APPLY Clauses with Continuous Loading of Transactional Data.

### 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.

## Job Example 6: Loading Data from Other Relational Databases

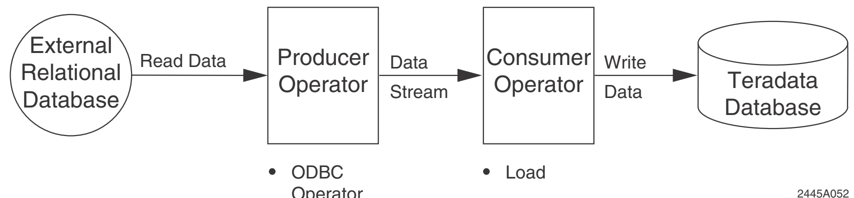
### Job Objective:

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

### Data Flow Diagram

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

Figure 31: Job Example 6 -- Loading Data from Other Relational Databases




### Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide06.txt**: Loading Data from Other Relational Databases into an Empty Teradata Database Table.

### 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.

## Job Example 7: Mini-Batch Loading

### Job Objective:

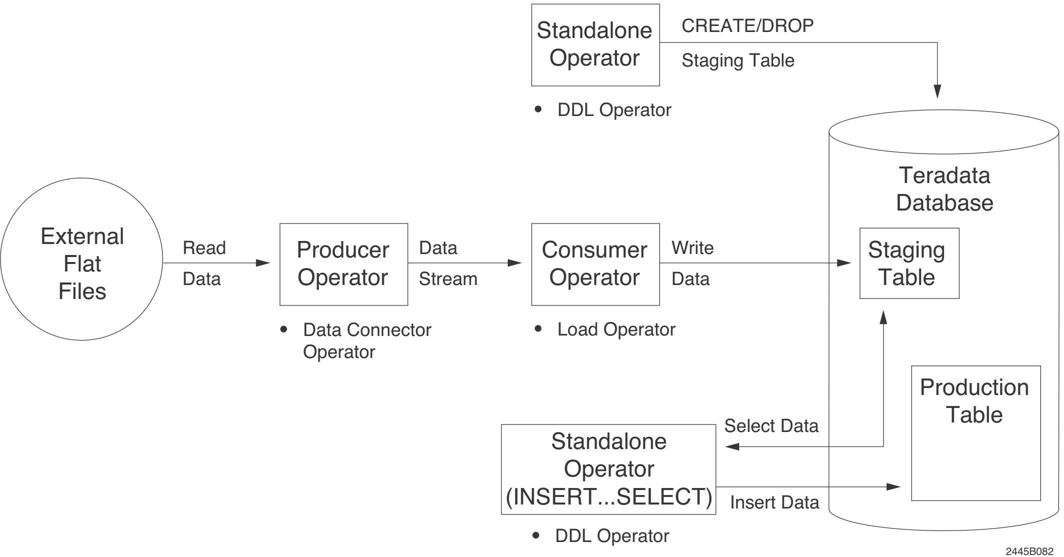
Read data directly from one or more external flat files and write it to a Teradata Database table.

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.

Figure 32: Job Example 7 -- Mini-Batch Loading




### Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide07.txt**: Mini-Batch Loading into Teradata Database Tables.

### 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.

## 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 to a Teradata Database table.

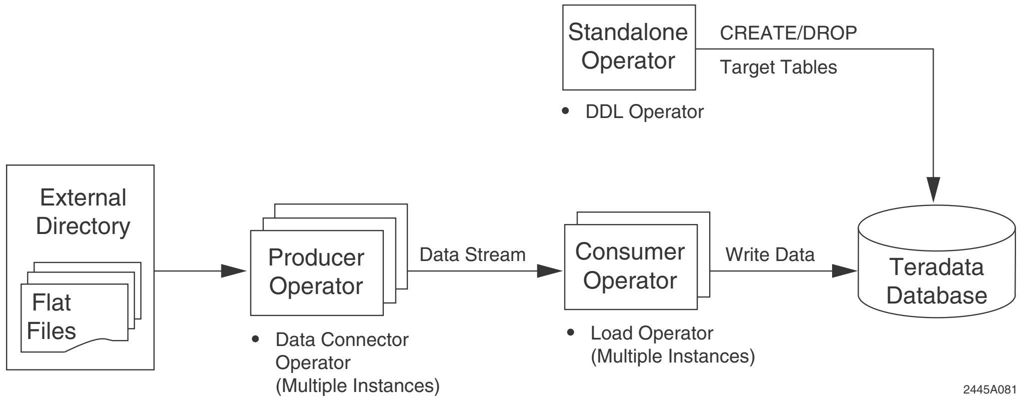
Note: If the Teradata Database table is populated, use the Update operator for the load operation.

For strategies on using this method, see "Batch Directory Scan" on page 205 and the DataConnector Operator chapter in *Teradata Parallel Transporter Reference*.

### Data Flow Diagrams

Figure 33 shows a flow diagram of the elements in Job Example 8.

Figure 33: Job Example 8 -- Batch Directory Scan




### Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide08.txt**: Batch Directory Scan.

### 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.

## 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 206 and the DataConnector Operator chapter in *Teradata Parallel Transporter Reference*.

### Data Flow Diagrams

Figure 34 shows a flow diagram of the elements in Job Example 9.

Figure 34: Job Example 9 -- Active Directory Scan

## Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide09.txt**: Active Directory Scan.

## 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.

CHAPTER 6

# Moving Data from Teradata Database to an External Target

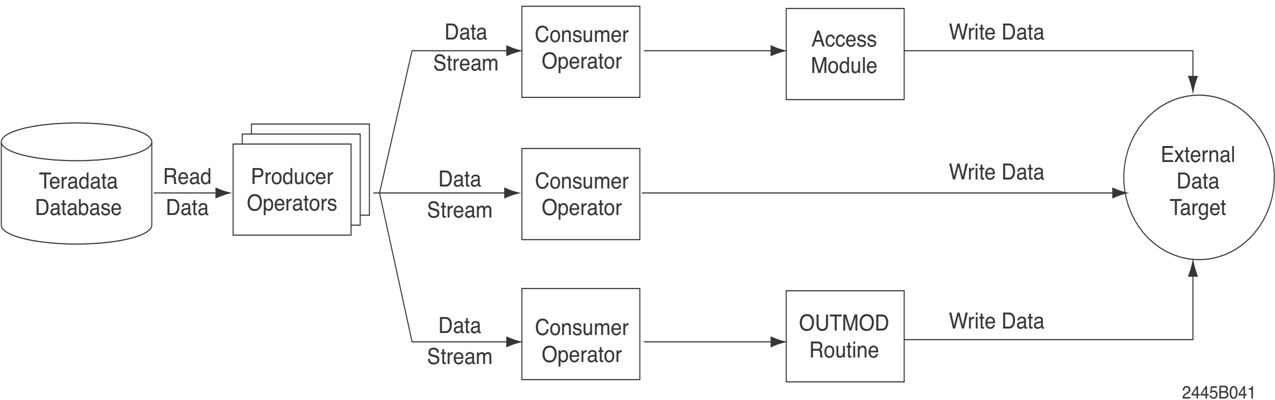
This chapter describes several methods for using Teradata PT to move data from a a Teradata Database into a non-Teradata target. It includes the following topics:


- 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.

# 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.

  and
- A consumer operator that takes data from the data stream and writes it to the data target.

  or
- 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 59 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:

| Operator | Description |
|---|---|
| Export Operator | Extracts large volumes of data from a Teradata Database at high speed. Function is similar to the standalone Teradata FastExport utility.<br>**Features:**<br>• Allows use of multiple parallel instances.<br>• 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.<br>**Limitations:**<br>• 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.<br>• 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.<br>For details, see *Teradata Parallel Transporter Reference*. |

| Operator | Description |
|---|---|
| SQL Selector Operator | Submits a single SQL SELECT statement to the Teradata Database to retrieve data from a table.<br>**Features:**<br>• Use to retrieve data in TEXT mode and write it to target files in the TEXT or VARTEXT (delimited) format.<br>• Can retrieve LOB data from the Teradata Database.<br>**Limitations:**<br>• Much slower than Export operator.<br>For details, see *Teradata Parallel Transporter Reference*. |

## 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 59 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:

| Operator | Description |
|---|---|
| **Operators that Write Data to a non-Teradata Target** | |
| DataConnector Operator | Writes data to flat files and functions similarly to the DataConnector standalone utility.<br>**Features:**<br>• Can write directly to an external file or through an access module.<br>For details, see *Teradata Parallel Transporter Reference*. |
| **Operators that Pre-process Data before Writing to a non-Teradata Target** | |
| FastExport OUTMOD Adapter Operator | Uses a FastExport OUTMOD routine to pre-process data before writing it to the data target.<br>For details, see *Teradata Parallel Transporter Reference*. |

# 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.

| Access Module | Description |
|---|---|
| OLE DB | Provides write access to a flat file or a table in an OLE DB-compliant DBMS, such as SQL Server, Oracle or Connix. |

## 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

# Job Example 10: Extracting Rows and Sending Them in Delimited Format

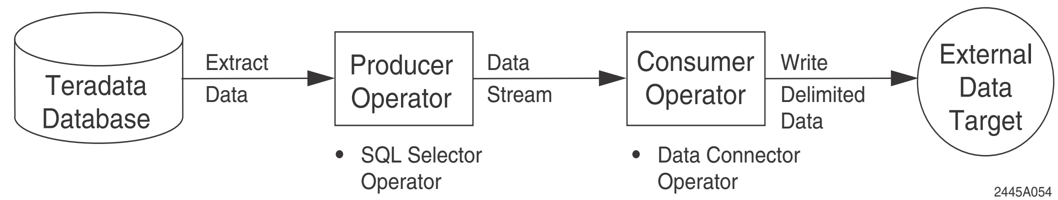
## Job Objective

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




## Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide10.txt**: Extracting Rows and Writing Them in Delimited Format.

## 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.

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

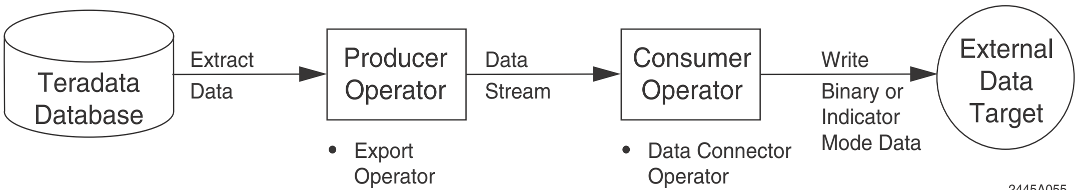
## Job Objective

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




### Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide11.txt**: Exporting Rows and Writing Them as Binary or Indicator Mode Data.

### 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.

## Job Example 12: Export Data and Process It with an OUTMOD Routine

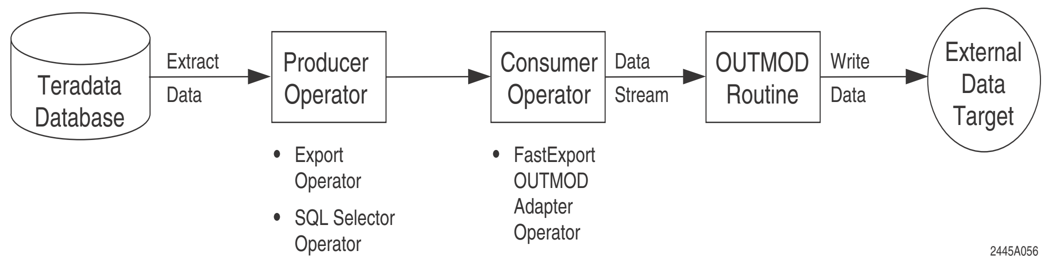
### Job Objective

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




### Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide12.txt**: Exporting Data and Processing It with an OUTMOD Routine.

### 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.

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

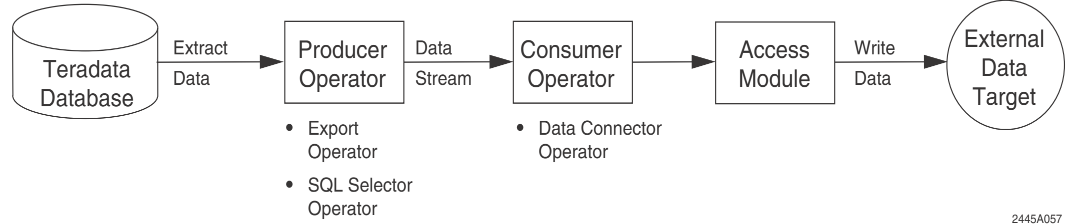
### Job Objective

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




### Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide13.txt:** Exporting Data and Processing It with an Access Module.

### 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.

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

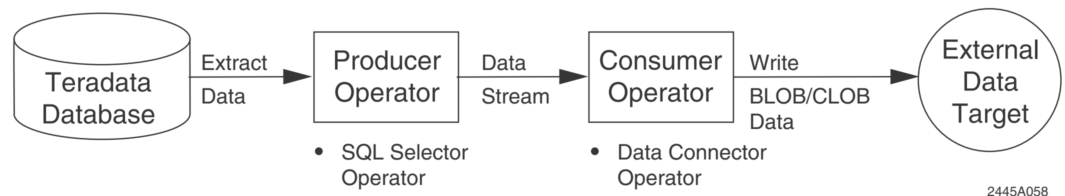
### Job Objective

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




### Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide 14.txt**: Extracting BLOB/CLOB Data and Writing It to an External Target.

### 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.

CHAPTER 7

# Moving Data within the Teradata Database Environment

This chapter describes the methods 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.

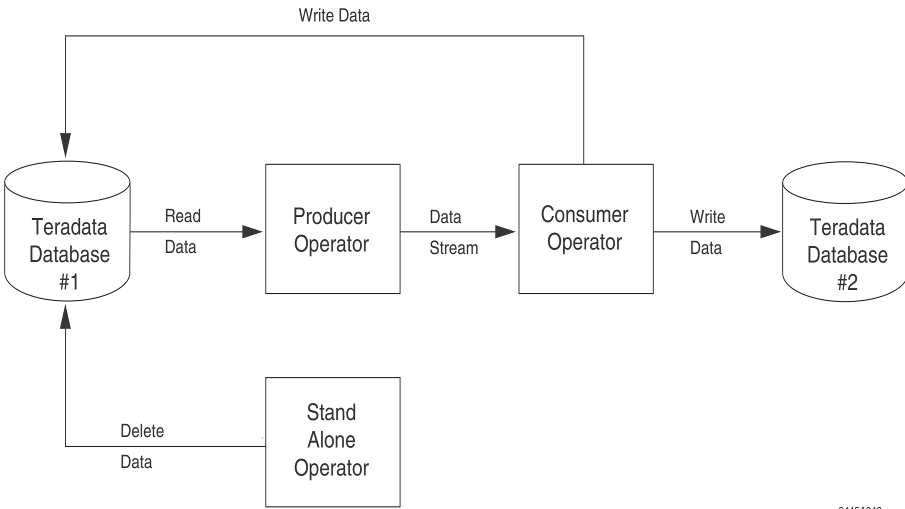
The chapter includes the following topics:

- 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.

## Using Teradata PT Easy Loader

Teradata PT Easy Loader is a command-line interface for loading data from Teradata Database table(s) without requiring you to write a Teradata PT script.

### Example 1

To load data from Teradata Database table(s), you can specify all job options that describe the data source in a job variables file, as follows:

```
SourceTdpId  = 'TdpId',
SourceUserName  = 'username',
SourceUserPassword = 'userpassword',
SelectStmt = 'Teradata SQL SELECT statement'
```

Note: Rather than specifying in a SELECT statement in the job variables file, you can specify the table name of the source table, as follows:

```
SourceTable = 'tablename'
```

Then execute the following command:

```
tdload -t tablename -u username -j jobvariablesfile
```

### Example 2

To load data from Teradata table(s), you can specify all job options that describe the data source as well as the target table in the job variable files, as follows:

```
TargetTable = 'tablename',
TargetTdpId  = 'TdpId',
TargetUserName  = 'username',
TargetUserPassword = 'userpassword',
SourceTdpId  = 'TdpId',
SourceUserName  = 'username',
SourceUserPassword = 'userpassword',
SelectStmt = 'Teradata SQL SELECT statement'
```

Then execute the following command:

```
tdload -j jobvariablesfile
```

For information on defining a job variables file and on executing the **tdload** command, see "Chapter 12 Teradata PT Easy Loader" on page 195.

# 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

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

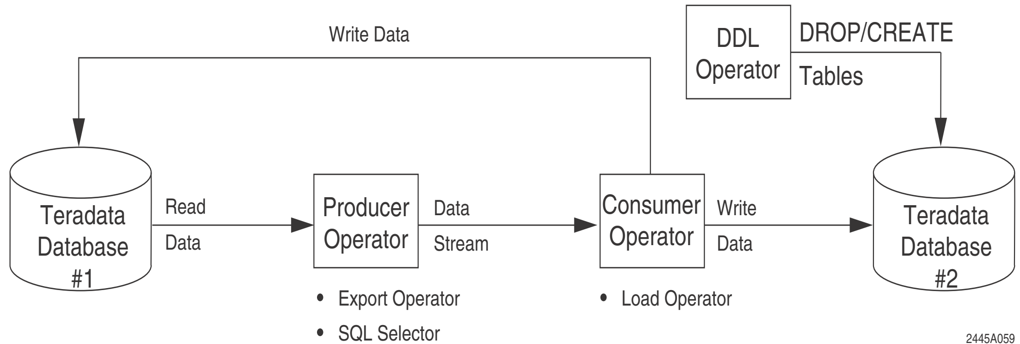
### Job Objective

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




### Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide15.txt**: Extracting Data from a Teradata Database Staging Table and Loading It into a Production Table.

### 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.

## 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.

Figure 43: Job Example 16 -- Export Data and Perform Conditional Updates Against Production Tables

## Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide 16.txt**: Exporting Data and then Use It to Performing Conditional Updates Against Production Tables.

## 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.

# Job Example 17: Bulk Delete of Data from a Teradata Database

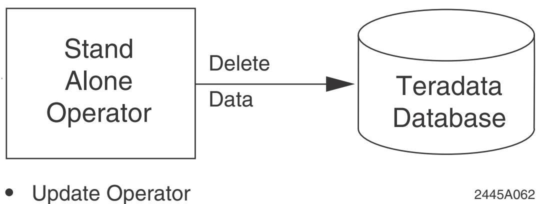
## Job Objective

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




## Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide17.txt**: Bulk Delete of Data from a Teradata Database.

## 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.

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

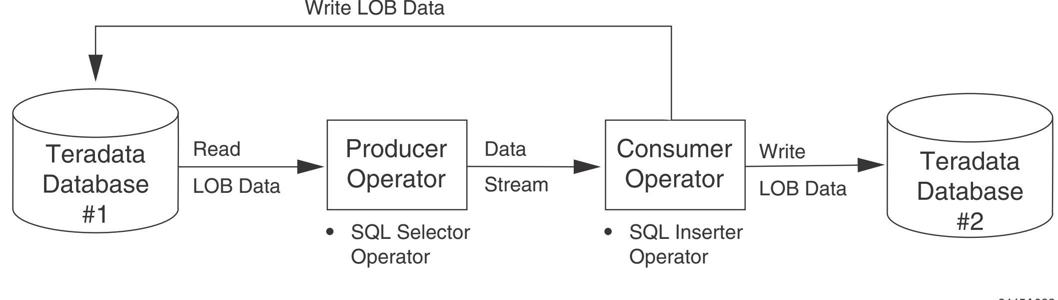
## Job Objective

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.

Figure 45: Job Example 18 -- Export BLOB/CLOB Data from One Teradata Database Table to Another



## Sample Script

For the sample script that corresponds to this job, see in the sample/userguide directory:

**uguide18.txt**: Exporting BLOB/CLOB Data from One Teradata Database Table and Loading It into Another.

## 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.

# SECTION 4 Launching, Managing, and Troubleshooting a Job

CHAPTER 8

# Launching a Job

This chapter explains how to set up Teradata PT job management options in the **tbuild** command before using that command to launch a job.

Topics include:

- Setting tbuild Options
- Setting Checkpoint Options
- Launching a Teradata PT Job

## Setting tbuild Options

The **tbuild** command, which is used to launch a Teradata PT job, enables you to specify job management options *before* you launch a job. Take time to become familiar with the available **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 **tbuild** options and how to employ them. For details about all **tbuild** options, see *Teradata Parallel Transporter Reference*.

### Specifying a File Name

The `-f <filename>` option is required in the **tbuild** command. The filename references the name of the file containing the job script you want to launch, as follows:

- If you run **tbuild** from the same directory that contains the job script, only the file name of the job script is required.
- If you run **tbuild** from a directory other than the directory that contains the job script file, the `-f <filename>` 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 Systems

Although it is not required, Teradata recommends that all jobs specify a job name on the **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 systems, 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 systems, 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 132.

### Jobname Syntax

The syntax to specify a job name is:

```
tbuild -f <filename> -j <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 can not be restarted.
- 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 7: Checkpoint Types and Functions

| Checkpoint Type | Function |
|---|---|
| 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:<br>• a Start-of-Data checkpoint<br>• an End-of-Data checkpoint<br>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.<br>• 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.<br>• 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.<br>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.<br>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.<br>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 <filename> -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 module checkpoint operation be 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 186.)

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.

## Setting the Checkpoint Directory

If a default checkpoint directory is defined in the global configuration file or the local configuration file, or both, submit a job by entering the following command:

```
tbuild -f <job script name> -z <checkpoint interval> <unique job name>
```

To overwrite the definition of a checkpoint directory in the global and local configuration files, specify the directory you want with the following using the following command:

```
tbuild -f <job script name> -r <checkpoint directory> -z <checkpoint
interval> <unique job name>
```

As a job runs, Teradata PT automatically searches for associated checkpoint files in the checkpoint directory. If a checkpoint file is found from a previous run, the job restarts where it left off.

To restart a job with the default checkpoint directory, enter the following command:

```
tbuild -f <job script name> -z <checkpoint interval> <unique job name>
```

To restart a job with the user-specified checkpoint directory, enter the following command:

```
tbuild -f <job script name> -r <checkpoint directory> -z <checkpoint
interval> <unique job name>
```

# 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'
```

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.

CHAPTER 9
# Managing an Active Job

This chapter describes how to manage an active Teradata PT job.

Topics include:


- 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


## 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
cl151001-112
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*.

| Command Option | Description |
|---|---|
| JOB CHECKPOINT | Takes an immediate checkpoint, then continues the job. |
| JOB PAUSE | Takes an immediate checkpoint, then suspends processing. |
| JOB RESUME | Resumes a paused job. |
| JOB TERMINATE | Takes an immediate checkpoint, then terminates the job. The job retains the checkpoint files, and is therefore restartable. |
| JOB STATUS | Writes a status record for each active operator instance to the TWB_STATUS log, and displays row processing statistics while continuing the job. |

## 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*.

| Syntax Element | Description |
|---|---|
| rate=statementRate | Option that specifies the maximum number of DML statements per minute the Stream operator can submit to the Teradata Database.<br>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.<br>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.<br>The specified Rate value must be either:<br>• a whole number greater than zero<br>or,<br>• unlimited<br>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.<br>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.<br>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.<br>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.<br>Valid values are between 1 and 600.<br>The default value is 4, that is, four 15 second intervals per minute.<br>If the statement rate is unlimited, then the *periodicity* value will be ignored.<br>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. |

### 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 **twbkil**l 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
```

CHAPTER 10

# Post-Job Considerations

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 144.

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 145.

   - If the job failed to complete, refer to the troubleshooting procedure in Chapter 11: "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 156.

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:

| Code | Description |
|---|---|
| Exit Code = 0 | The Teradata PT job or job step completed successfully with at most, minor warnings. |
| Exit Code = 4 | The Teradata PT job or job step completed successfully, but issued one or more warnings. |
| Exit Code = 8 | A user error, such as a syntax error in the script, terminated the job. |
| Exit Code = 12 | A fatal error terminated the job.<br>A fatal error is any error other than a user error, for example:<br>• Incompatible data types encountered during reading of data sources.<br>• Data errors exceeding the value specified in the ErrorLimit attribute.<br>• Insufficient system resources, such as shared memory or semaphores, to execute the job. |

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 156.
- 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 129.

# 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.

## 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.

### Accessing a Public Log by Job Name

To access the public log for a particular Teradata PT job, enter the following **tlogview** command:

```
tlogview -j <jobname>-<job sequence number>
```

where:

- `<jobname>` 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 (a UNIX command shown):

```
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 OS**

  `cd $TWB_ROOT/logs`
- **Windows**

  `chdir %TWB_ROOT%\logs`
- **Linux**

  `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:

| Log Section Heading | Description |
|---|---|
| Private log PXCRM | The checkpoint log for the job |
| Private log TWB_STATUS | The log of statistical performance metadata for operations carried out by the job. |
| Private log TWB_SRCTGT | The log of metadata for operations on the data source and data target carried out by the job. |
| Private log <PrivateLogName> | 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.<br>The individual operator logs contain such informations as:<br>• Operator name<br>• Operator version information<br>• Separate sections for each stage of operator task execution, including data/time stamps, SQL submitted, and errors encountered. |

### 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> -f "*" -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:

```
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 **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.
- *<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:

| Operating System | Location |
|---|---|
| UNIX<br>(including AIX, HP-UX, Solaris running of a SPARC system, and Linux) | The default directory is:<br>`/opt/teradata/client/<version_number>/tbuild/logs`<br>The log directory cannot be a relative directory path and the directory.<br>Log directory can be user-specified by modifying the "LogDirectory" entry in the local or global twbcfg.ini file |
| z/OS | Logs are user-specified in the Teradata PT job JCL<br>Note: The **tlogview** command should be run in the batch environment by the appropriate JCL. The **tlogview** 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. |
| Windows | Default directory is:<br>`%ProgramFiles%\Teradata\Client\<version_number>\Teradata Parallel Transporter\logs` |

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 UTF16 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:

- **ErrorTable1** (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:

| Operator | Description |
| --- | --- |
| Load | Generates acquisition and application error tables (ErrorTable1 and ErrorTable2). |
| Update | Error tables are named in one of the following two ways:<br>• The operator automatically names the table in terms of the target table, as follows:<br>  • For ErrorTable1: `TargetTableName_ET`<br>  • For Errortable2: `TargetTableName_UV`<br>• The ErrorTable1 and ErrorTable2 attributes name error tables. |
| Stream | 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.<br>The error table is named in one of the following two ways:<br>• Use the ErrorTable attribute to name the error table.<br>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:<br>`DatabaseName.ErrorTableName`<br>For information, see chapters on Load and Update operators in *Teradata Parallel Transporter Reference.*<br>• If no name is specified for the ErrorTable attribute the operator automatically names the error table, for instance: M<yy><doy>_<seconds>_<LSN>_ET<br>where:<br>  • M is the default prefix<br>  • <yy> is the last two digits of the year<br>  • <doy> is the day of the year<br>  • <seconds> is the seconds of the day<br>  • <LSN> is the logical session number<br>  • ET is the default suffix (meaning “error table”)<br>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 172.

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 update 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 152:

### ErrorTable1

```
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

```
SELECT dbcerrorcode, sourceseq, dmlseq FROM t3_e2;
```

If the operator definition does not specify a name for the ErrorTable2 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.

### 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.

```
SELECT errorcode, errorfield, sourceseq, dmlseq From t2_e1;

 *** Query completed. One row found. 4 columns returned.
 *** Total elapsed time was 1 second.

     ErrorCode  ErrorField                          SourceSeq  DMLSeq
```

```
--------------  ----------------------------  --------------  ------
          2679  A_IN_C1                                   49       1
```

The following explains the error table entry above:

| Error table Column | Value | Explanation |
|---|---|---|
| ErrorCode | 2679 | The Teradata Database error code associated with the error.<br>In this case, message 2679 indicates: The format or data contains a bad character.<br>**Note:** This error code also appears in the job logs. |
| ErrorField | A_IN_C1 | Indicates where the error was generated.<br>In this case: In column A_IN_C1, as defined in the INSERT INTO statement for the Stream operator. |
| SourceSeq | 49 | The sequence number of the data row that caused the error. |
| DMLSeq | 1 | The sequence number of the DML statement (within its DML Group) that caused the error. |

## 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.

```
select dbcerrorcode, sourceseq, dmlseq from t3_e2;

 *** Query completed. 2 rows found. 3 columns returned.
 *** Total elapsed time was 1 second.

  DBCErrorCode       SourceSeq  DMLSeq
--------------  --------------  ------
          2793              50       2
          2793              49       2
```

The following explains the error table entry above:

| Error table Column | Value | Explanation |
|---|---|---|
| ErrorCode | 2793 | The Teradata Database error code associated with the error.<br>In this case, message 2793 indicates: The format or data contains a bad character.<br>**Note:** This error code also appears in the job logs. |
| SourceSeq | 49, 50 | The sequence numbers of the data rows that caused the error. |
| DMLSeq | 2 | The sequence number of the DML statement (within its DML Group) that caused the error. |

## 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 172
- "Stream Operator Errors" on page 176
- "Update Operator Errors" on page 180

# 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 106. 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 107, 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:

- High-volume jobs with large error limits may overrun the space allocation for the error tables. If this happens you need to either increase the space allocation, or set the elapsed time to a shorter duration. It may be useful to periodically save the error tables to an alternate location to allow time for evaluation.
- Jobs containing operators with high values for the ErrorLimit attribute may accumulate a large number of bad rows. Make sure to set the DropErrorTable attribute to **No** so the error tables will be retained at the end of the VigilElapsedTime, to allow time for manual processing of the bad rows. Make sure to manually drop the error tables before the next run, or set the AppendErrorTable attribute to **Yes**.

# 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 186.

# 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 11: "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 the Teradata Tools and Utilities installation guide for your platform.

### 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.

CHAPTER 11

# Troubleshooting a Failed Job

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 145.

For information on the content of error tables and how to access them, see "Accessing and Using Error Tables" on page 150.

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 162.

- *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 169.

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 150.

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 186.

# 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.

## Environment Variable Errors

**For an IBM AIX system**: When the LANG and LC_FASTMSG environment variables are set to "C" and "true", respectively, the following messages will appear after running a Teradata PT job using the **tbuild** command:

**Message Catalog Error**: Message 4000 was not found

**Message Catalog Error**: Message 2007 was not found

**Cause**: Environment variable settings

**Corrective Action**: Use one of the following actions to solve the error messages:

- Change the value for the LANG environment variable to "en_US"
- Change the value for the LC_FASTMSG environment variable to "false"

Then re-run the Teradata PT job.

**Configuration Data**:

- TPT v12
- TPT v13

## 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 210.
- 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 43.

### 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 Common 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:

```
DEFINE SCHEMA DAILY_SALES
(
  Store_Number     INTEGER,
      :             :
  Sales_Date          ANSIDATE,
  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.

### 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 erno: 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
```

**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:**

Error 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
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'.
READ_DATA: Operator instance 1 processing file 'File00005'.
READ_DATA: Operator instance 1 processing file 'File00016'.
READ_DATA: Operator instance 1 processing file 'File00011'.
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 'File00002'.
READ_DATA: Operator instance 1 processing file 'File00010'.
READ_DATA: Operator instance 1 processing file 'File00004'.
STREAM_OPERATOR: connecting sessions
PXTB_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 83.

**Corrective Action:**

1. Do one of the following
   - Decrease the number of consumer or producer instances.

     or,
   - 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 169.

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 *value*K 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 *value*M 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 83.

### 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 150.

## 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 AceptExcessColumns o 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 8: Load Errors

| Error Condition | Cause of Rejection |
|---|---|
| Constraint violation | Records do not comply with the range constraints you defined when creating the table. |
| Unavailable AMP condition | Records are destined for a nonfallback table on an AMP that is down. |
| Data conversion errors | Refers to records from the input file that fail a specific data type conversion. |
| Unique primary index violation | Records contain a value for the unique primary index field that already exists, but is not a duplicate row. |
| Duplicate row | Records are exact duplicates of existing rows. |

## 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:

Table 9: Format of ErrorTable1

| Column | Contents |
|---|---|
| ErrorCode | Teradata Database return code for the error condition, as specified in the messages reference documentation for your operating system environment. |
| ErrorFieldName | Name of the data item that caused the error condition. |
| DataParcel | Entire data record, as provided by the source producer operator. DataParcel is used as the primary index for the first error table. The data record string can be up to 64,000 bytes, depending on which version of the DBS the job is run against. |

- The application error table is formatted to match the target table.

## 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 ErrorTable1attribute:

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 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:

```
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:

```
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:

```
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.

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 10: Error Table Columns

| Column. | Contents |
|---|---|
| DataSeq | Sequence number assigned to the input source in which the error occurred. |
| DMLSeq | Sequence number assigned to the DML group in which the error occurred. |
| ErrorCode | Code for the error. |

Table 10: Error Table Columns (continued)

| Column. | Contents |
|---|---|
| ErrorField | This field is zero for the Stream operator. |
| ErrorMsg | The Teradata Database error message for the error. |
| HostData | Client data being processed when the error occurred. |
| LoadStartTime | Queue Insertion TimeStamp (QITS) value indicates when the job started. On restart, it indicates when the job restarted. |
| RowInsertTime | Indicates when the row was inserted into the Stream operator error table. |
| SourceSeq | Sequence number assigned to the row from the input source (the DataSeq number) in which the error occurred. |
| STMTSeq | Sequence number of the DML statement within the DML group (as indicated by the previous column DMLSeq) being executed when the error occurred. |

## 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 154.

### 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:

```
APPLY
'INSERT INTO table1 VALUES (:FIELD1,:FIELD2 );
UPDATE table2 SET field3 = :FIELD3 WHERE field4 = :FIELD4;'
```

### 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 11: Task Example

| Sequence | | |
|---|---|---|
| Type | Number | Statement |
| 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.

| DataSeq | DMLSeq | SMTSeq | SourceSeq | ErrorCode | ErrorField |
|---|---|---|---|---|---|
| 002 | 001 | 002 | 20456 | 2679 | 000 |

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
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.

- ErrorTable1uses 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 150.

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 181 and "Application Error Table" on page 182.

## 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 12: Acquisition Error Table Format for the Update Operator

| Column | Contents |
|---|---|
| ApplySeq | Sequence number assigned to the DML group in which the error occurred (the same as DMLSeq). It can be ignored in error handling. |
| DMLSeq | Sequence number assigned to the DML statement within the DML group in which the error occurred. |
| ErrorCode | Code for the error. |
| ErrorField | Field name of the target table in which the error occurred.<br>**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. |
| HostData | Client data being processed when the error occurred. |
| ImportSeq | Sequence number assigned to the input source in which the error occurred. |
| SourceSeq | Sequence number assigned to the row from the input source (the ImportSeq number) in which the error occurred. |
| STMTSeq | Sequence number of the DML statement within the DML group (as indicated by the previous column DMLSeq) being executed when the error occurred. |

## 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 13: Application Error Table Format for the Update Operator

| Column | Contents |
|---|---|
| ApplySeq | Sequence number assigned to the DML group in which the error occurred (the same as DMLSeq). It can be ignored in error handling. |
| DBCErrorCode | Code for the error. |

Table 13: Application Error Table Format for the Update Operator (continued)

| Column | Contents |
|---|---|
| DBCErrorField | Field name of the target table in which the error occurred.<br>**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. |
| DMLSeq | Sequence number assigned to the DML statement within the DML group in which the error occurred. |
| ImportSeq | Sequence number assigned to the input source in which the error occurred. |
| SourceSeq | Sequence number assigned to the row from the input source (the ImportSeq number) in which the error occurred. |
| STMTSeq | Sequence number of the DML statement within the DML group (as indicated by the previous column DMLSeq) that is executed when the error occurred. |
| Uniqueness | Value that prevents duplicate row errors in the error table. It can be ignored in error handling. |

## 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 14: Task Example

| Sequence | | |
|---|---|---|
| Type | Number | Statement |
| DML | 001 | `'INSERT INTO table1 VALUES (:FIELD1,:FIELD2 );`<br>`UPDATE table2 SET field3 = :FIELD3 WHERE field4 =`<br>`:FIELD4;'` |
| STMT | 001 | `INSERT INTO table1 VALUES (:FIELD1, :FIELD2 );` |

Table 14: Task Example (continued)

| Sequence | | |
|---|---|---|
| Type | Number | Statement |
| 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:

| ImportSeq | DMLSeq | SMTSeq | ApplySeq | Source Seq | ErrorCode | ErrorField |
|---|---|---|---|---|---|---|
| 002 | 001 | 002 | 001 | 20456 | 2679 | field3 |

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
- 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 running on a SPARC system, 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 <filename>
  - Run the operators in your Teradata PT job in trace mode using the TraceLevel attribute.

    ```
    TraceLevel = 'all'
    ```

- Provide truss output (UNIX system 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>.

# Restarting A Job

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 129.

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 an error that allows a retry, such as a database restart or deadlock, occurs before, during, or after loading 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 <filename> -s <job step identifer>
```

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 system only

  On a UNIX system, 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 <filename> -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
- <high-level qualifier>.LVCP

where <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 *<username>.LVCP*.

Jobs executed under the same login name, therefore, use the same *<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 systems only, as follows:

If the job script specifies a job name:

```
twbrmcp  <job name>
```

If the job script 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 Systems

- To delete checkpoint files from a directory defined by Teradata PT, enter the following command:
  - On UNIX OS:
    ```
    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 OS:

```
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 dataset 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:

```
//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 Appendix C: "Teradata PT Publications."

# Specifying the Wait Time for a File Lock

Setting FileLockWaitLimit, a configuration directive added to the Teradata PT global configuration file, twbcfg.ini, to any positive integer specifies the number of seconds Teradata PT jobs wait to obtain a file lock.

Below is an example twbcfg.ini with the new directive:

```
CheckpointDirectory='/opt/teradata/client/14.0/tbuild/checkpoint'
LogDirectory='/opt/teradata/client/14.0/tbuild/logs'
FileLockWaitLimit='30'
```

If a job cannot acquire the file lock within the wait limit time, the job terminates.

If FileLockWaitLimit is not set, the default wait limit is five seconds.

See "Setting Up Configuration Files" on page 67.

# SECTION 5 Advanced Topics

CHAPTER 12

# Teradata PT Easy Loader

This chapter describes using Teradata PT Easy Loader, 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.

## Using Teradata PT Easy Loader

### Required Tasks

The tasks in this section are required when using Teradata PT Easy Loader 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. The layout of the data records in the flat file must match that of the target table.

Note: Unicode is not supported in the command line or in the job variables file.

### The tdload Command

The Teradata PT Easy Loader command, **tdload**, has the following syntax:

```
tdload jobOptions jobname
```

### Reference Information

| Information on... | Is available in... |
|---|---|
| the definition of **tdload** job options | *Teradata Parallel Transporter Reference* |
| the definition of **tdload** job name | *Teradata Parallel Transporter Reference* |
| delimited format files | *Teradata Parallel Transporter Reference* |

| Information on... | Is available in... |
|---|---|
| determining the optimum number of sessions and instances | “Optimizing Job Performance with Sessions and Instances” on page 80 |
| error limits | “Effects of Error Limits” on page 154. |

## Task 1: Define a Job Variables File

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

**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.

### 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 Easy Loader. Each job variable must be defined on a single line separated by commas, using the following format:

*option* = *value*

where:

| Syntax Element | Explanation |
|---|---|
| *option* | A single-letter or a multi-letter (long) option.<br>**Note:** The option name is case-sensitive. |
| *value* | An integer or a character string.<br>You must enclose character strings within single quotes. |

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 Easy Loader 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 Easy Loader Job

### Procedure

Follow these steps to launch a Teradata PT Easy Loader 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:

```
tdload -f filename -u username -p password -h tdpid -t 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 Easy Loader job in Example 1 above. Using the -j option eliminates the need to type the options when executing **tdload**.

```
tdload -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 Easy Loader 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 Easy Loader Job

You can monitor and manage a Teradata PT Easy Loader 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.

### Reference Information

| Information on... | Is available in... |
|---|---|
| the **twbcmd** command | • "Using the twbcmd Command to Monitor and Manage Job Performance" on page 138<br>• *Teradata Parallel Transporter Reference* |

### 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:

```
twbcmd job_id JOB STATUS
```

where *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 (if, for example, continuation of the job could either cause system failures or significantly impact overall system performance), you can use the **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:

```
twbkill job_id
```

where *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

| Information on... | Is available in... |
|---|---|
| the **twbkill** command | • "Using twbkill to Terminate a Job" on page 142<br>• *Teradata Parallel Transporter Reference* |

## Task 4: Evaluate a Completed Teradata PT Easy Loader Job

You can evaluate a completed Teradata PT Easy Loader 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.

| Exit Code | Description. |
|---|---|
| 0 | Completed successfully. |
| 4 | Completed successfully, but issued one or more warnings. |
| 8 | Terminated due to a user error, such as a syntax error. |
| 12 | Terminated due to a fatal error.<br>A fatal error is any error other than a user error. |

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.

| Types of Log | Explanation |
|---|---|
| Console | The console log displays messages in the Command window where the **tbuild** command was issued.<br>This log contains high-level information about Teradata PT operators and infrastructure. It updates continuously while the job runs. |
| Public | The public log contains general information about the job. Use the **tlogview** command to access this log. |
| Private | 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. |

### 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.

| Error Table | Name of Table | Explanation |
|---|---|---|
| 1 | Acquisition | Exports constraint violations, bad data, and data conversion errors |
| 2 | Application | Contains any rows that cause violations of the unique primary index, for instance, duplicate rows.<br>This error table is not used when the target table has a nonunique primary index. |

### Reference Information

| Information on... | Is available in... |
|---|---|
| exit codes | "Chapter 10 Post-Job Considerations" on page 143 |
| job steps | "Chapter 2 Teradata PT Job Components" on page 37 |
| specifying checkpoint intervals | "Chapter 8 Launching a Job" on page 129 |
| accessing public job logs and how to read them, including example logs | "Chapter 10 Post-Job Considerations" on page 143 |
| accessing private job logs and how to read them, including example logs | "Chapter 10 Post-Job Considerations" on page 143 |
| accessing the error tables and how to read them, including example tables | "Chapter 10 Post-Job Considerations" on page 143 |

| Information on... | Is available in... |
|---|---|
| the **tlogview** command | • "Accessing and Using Job Logs" on page 145<br>• *Teradata Parallel Transporter Reference* |
| strategies for evaluating a successful job | "Chapter 10 Post-Job Considerations" on page 143 |

## Task 5: Troubleshoot a Failed Teradata PT Easy Loader Job, If Necessary

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

### Reference Information

| Information on... | Is available in... |
|---|---|
| detecting and correcting the cause of failure | "Chapter 11 Troubleshooting a Failed Job" on page 161 |
| common job failures and remedies | "Chapter 11 Troubleshooting a Failed Job" on page 161 |
| operator-specific error handling | "Chapter 11 Troubleshooting a Failed Job" on page 161 |
| debugging strategies for complex job failures | "Chapter 11 Troubleshooting a Failed Job" on page 161 |
| restarting a job | "Chapter 11 Troubleshooting a Failed Job" on page 161 |
| removing checkpoint files | "Chapter 11 Troubleshooting a Failed Job" on page 161 |
| the notify exit routines | *Teradata Parallel Transporter Reference* |

CHAPTER 13
# 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
- Simplifying Scripts with Operator Templates and Generated Schemas
- Using the Job Identifier in Your Job Script
- 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 46).

  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 and the APPLY statement, even if all producer schemas are not UNION-compatible.

### Code Example

```
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 98.

For a sample Teradata PT script, see in the sample/userguide directory:

**uguide01c.txt**: High Speed Bulk Loading from Two Flat Files to an Empty Teradata Database Using UNION ALL.

## 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 *Teradata Parallel Transporter Reference*.

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 103.

For a sample Teradata PT script, see in the sample/userguide directory:

**uguide05c.txt**: Intermediate File Logging Using Multiple APPLY Clauses with Continuous Loading of Transactional Data.

## 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 sample Teradata PT script, see in the sample/userguide directory:

**uguide07.txt**: Mini-Batch Loading into Teradata Database Tables.

## 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 sample Teradata PT script, see in the sample/userguide directory:

**uguide08.txt**: Batch Directory Scan.

## Active Directory Scan: Continuous Loading of Transactional Data

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:

| Attribute | Setup Requirements |
|---|---|
| VigilStartTime | Required to specify the start time for the initial directory scan. |
| VigilStopTime | Specifies the time after which no more scans will begin. Any scan that begins before the stop time will run to completion.<br>This attribute is interchangeable with the VigilElapsedTime attribute. Using one of these two attributes is required. |
| VigilWaitTime | Specifies the time in seconds between the beginning of one scan and the beginning of the next scan. |
| VigilElapsedTime | 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. |

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 107.

For a sample Teradata PT script, see in the sample/userguide directory:

uguide09.txt: Active Directory Scan.

# 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 15 contains some examples of data conditioning using SELECT and the syntax for each.

Table 15: Data Conditioning Syntax Using SELECT

| Filtering Task | Code Example |
|---|---|
| Rename a column from "price" to "original price" | `SELECT price AS original_price` |
| Assign new values to a column based on the value of other columns, literals, and arbitrary expressions, such as calculating a discounted price of 20% off the base price. | `SELECT price*0.8 AS discounted_price` |
| Assign NULL values to a column. | `SELECT NULL(VARCHAR(n)) AS product_name`<br>where n is the size of the VARCHAR column as defined in the DEFINE SCHEMA statement. |
| Concatenate columns using the concatenation operator (\|\|), such as loading both an area code and the telephone number into the same column to create a customer number | `SELECT AREA_CODE\|\|PHONE_NUMBER AS CUSTOMER_NUMBER` |
| Load a value into a column that does not exist in the source, such as putting the value "123" in a column called "JOB_ID" | `SELECT '123' as JOB_ID` |
| Assign different values to a column in an output row based on conditions on columns in the corresponding input row, using a CASE value expression. | `APPLY 'INSERT INTO SALES_TABLE`<br>`(original_price, discounted_price,`<br>`product_name)' TO OPERATOR (UPDATE_OPERATOR`<br>`[4])`<br>`SELECT price AS original_price, price*0.8 AS`<br>`discounted_price,`<br>`CASE WHEN product_name = ' ' THEN`<br>`NULL(VARCHAR) ELSE product_name`<br>`END AS product_name`<br>`FROM OPERATOR (DATA_CONNECTOR [2])` |

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

```
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 Expressions

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 > 100000000 )
       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
```

```
(
      FirstName    VARCHAR(32),
      LastName     VARCHAR(32),
      ...
);
```

Then, to include this schema definition in the job script:

```
DEFINE JOB CustomerUpdate
(
      INCLUDE 'customer.schema';
      DEFINE OPERATOR...
);
```

You can also use job variables (`@<variableName>`) in files declared by the INCLUDE statement. Assignment of values to job variables takes place after the contents of the declared files have been incorporated into the job script file, from the highest priority job variable source. For more information on the various ways to set job variables and the processing priority for variable sources, see "Setting Up Job Variables" on page 43.

# Simplifying Scripts with Operator Templates and Generated Schemas

## Using Operator Templates

You can simplify your job script and reduce its size by using *operator templates*. An operator template is a stored DEFINE OPERATOR statement that is automatically imported into your job script when you reference in an APPLY statement a standard Teradata PT-supplied operator by its template name, as shown in the following example:

```
APPLY  'INSERT INTO TABLE_X ( :col1, ..., :coln );'
    TO OPERATOR( $LOAD() ) ... ;
```

In this example, $LOAD is the template name of the standard Teradata PT-supplied Load operator. The DEFINE OPERATOR statement for the $LOAD operator is stored in template file $LOAD.txt in the Teradata PT template directory. Template operator names are the standard operator types prefixed by the dollar sign ($), unless a shorter yet still descriptive name exists.

The following table lists the operator templates that Teradata PT supplies. All operator templates are located in the Teradata PT Install directory in the template directory and begin with the dollar sign ($).

Table 16: Teradata PT Operator Templates

| Template Operator Name | Standard Operator Type | Template File Name |
|---|---|---|
| $DATACONNECTOR_ CONSUMER | DATACONNECTOR CONSUMER | *$DATACONNECTOR_ CONSUMER.txt* |
| $DATACONNECTOR_ PRODUCER | DATACONNECTOR PRODUCER | *$DATACONNECTOR_ PRODUCEF* |

Table 16: Teradata PT Operator Templates (continued)

| Template Operator Name | Standard Operator Type | Template File Name |
|---|---|---|
| $DDL | DDL | *$DDL.txt* |
| $DELETER | UPDATE STANDALONE | *$DELETER.txt* |
| $EXPORT | EXPORT | *$EXPORT.txt* |
| $FE_OUTMOD | FASTEXPORT OUTMOD | *$FE_OUTMOD.txt* |
| $FILE_READER | DATACONNECTOR PRODUCER | *$FILE_READER.txt* |
| $FILE_WRITER | DATACONNECTOR CONSUMER | *$FILE_WRITER.txt* |
| $FL_INMOD | FASTLOAD INMOD | *$FL_INMOD.txt* |
| $INSERTER | INSERTER | *$INSERTER.txt* |
| $LOAD | LOAD | *$LOAD.txt* |
| $ML_INMOD | MULTILOAD INMOD | *$ML_INMOD.txt* |
| $ODBC | ODBC | *$ODBC.txt* |
| $OS_COMMAND | OS COMMAND | *$OS_COMMAND.txt* |
| $SCHEMAP | SCHEMAMAPPER | *$SCHEMAP.txt* |
| $SELECTOR | SELECTOR | *$SELECTOR.txt* |
| $STREAM | STREAM | *$STREAM.txt* |
| $UPDATE | UPDATE | *$UPDATE.txt* |

Using an operator template is like storing your own DEFINE OPERATOR statement in a separate file and then importing the file into your job script using a Teradata PT INCLUDE directive, except that when you use an operator template, you do not need to code any INCLUDE directive.

Since the DEFINE OPERATOR statement for a template is not in your script when you code it, you do not have normal access to its operator attribute declarations and cannot assign job-scope default values to these attributes. To make them accessible to you, all operator attributes that are defined for each standard operator are declared in its template definition and assigned conventionally-named job variables as their job-scope default values.

The following, for example, is the template definition for the DDL operator:

```
DEFINE OPERATOR $DDL
  DESCRIPTION 'Teradata Parallel Transporter DDL Operator'
  TYPE DDL
  ATTRIBUTES
  (
    VARCHAR UserName            = @TargetUserName,
    VARCHAR UserPassword        = @TargetUserPassword,
    VARCHAR TdpId               = @TargetTdpId,
```

```
    VARCHAR AccountId           = @TargetAccountId,
    VARCHAR WorkingDatabase     = @TargetWorkingDatabase,
    VARCHAR LogonMech           = @TargetLogonMech,
    VARCHAR LogonMechData       = @TargetLogonMechData,
    VARCHAR DataEncryption      = @DDLDataEncryption,
    VARCHAR ARRAY ErrorList     = @DDLErrorList,
    VARCHAR LogSQL              = @DDLTargetLogSQL,
    VARCHAR PrivateLogName      = @DDLPrivateLogName,
    VARCHAR QueryBandSessInfo   = @DDLQueryBandSessInfo,
    VARCHAR ReplicationOverride = @DDLReplicationOverride,
    VARCHAR TraceLevel          = @DDLTraceLevel
  );
```

Notice that all DDL Operator attributes can be assigned values via job variables for the execution of any job script that references the $DDL template. The names of these job variables are formed from the corresponding attribute names as follows:

- Logon attribute names, that is, attributes associated with logging on to Teradata Database, such as UserName and UserPassword, are prefixed with 'Source' in producer templates and 'Target' in consumer and standalone templates.
- All other template attributes names are prefixed with a unique name, either the corresponding operator type or a short name or mnemonic that is easily associated with the template.

This template job variable naming convention ensures that assigned job variables for any template cannot inadvertently affect attribute values of other referenced templates.

The following table lists the job variable name prefix for each of the standard operator template supplied by Teradata PT:

Table 17: Teradata PT Operator Templates

| Template Operator Name | Standard Operator Type | Job Variable Name Prefix |
|---|---|---|
| $DATACONNECTOR_ CONSUMER | DATACONNECTOR CONSUMMER | DCC |
| $DATACONNECTOR_ PRODUCER | DATACONNECTOR PRODUCER | DCP |
| $DDL | DDL | DDL |
| $DELETER | UPDATE STANDALONE | Deleter |
| $EXPORT | EXPORT | Export |
| $FE_OUTMOD | FASTEXPORT OUTMOD | FEOutmod |
| $FILE_READER | DATACONNECTOR PRODUCER | FileReader |
| $FILE_WRITER | DATACONNECTOR CONSUMER | FileWriter |
| $FL_INMOD | FASTLOAD INMOD | FLInmod |
| $INSERTER | INSERTER | Inserter |

Table 17: Teradata PT Operator Templates (continued)

| Template Operator Name | Standard Operator Type | Job Variable Name Prefix |
|---|---|---|
| $LOAD | LOAD | Load |
| $ML_INMOD | MULTILOAD INMOD | MLInmod |
| $ODBC | ODBC | ODBC |
| $OS_COMMAND | OS COMMAND | OSCommand |
| $SCHEMAP | SCHEMAMAPPER | Smap |
| $SELECTOR | SELECTOR | Selector |
| $STREAM | STREAM | Stream |
| $UPDATE | UPDATE | Update |

When no value has been assigned to the job variable of any particular template attribute, Teradata PT interprets the attribute as having been declared without a job-scope default value assignment. If the $DDL job variable *TargetLogonMech*, for example, is not assigned any value when a script using the $DDL template is executed, then its attribute declaration:

```
VARCHAR LogonMech = @TargetLogonMech,
```

will be interpreted by Teradata PT as if it had been declared in the $DDL template as

```
VARCHAR LogonMech,
```

Of course, like the attributes of script-defined operators, attributes of a template operator can always be assigned values at the place in the APPLY statement where the template is referenced. Such assignments definitively determine the attribute values for that specific invocation of the template operator, whether or not the corresponding job variables were assigned values, as shown below:

```
SELECT *
FROM OPERATOR
(
  $EXPORT()
  ATTR
  (
    PrivateLogName = 'weekly_sample.log',
    SelectStmt     = 'Select * from Weekly_Trans;'
  )
)
```

In the above script extract, the values assigned to attributes PrivateLogName and SelectStmt are those that this particular execution of the Export Operator will use, even if the corresponding job variables in the $EXPORT template definition have different values. Attribute value specification works the same way for template operators as it does for script-defined operators.

Assigning template attribute values for specific template references in an APPLY statement will even be necessary if a job script contains more than one reference to a given template and a single set of job variable assignments will not work for all template references.

## Array Type Template Attributes

A small number of operator attributes can be declared to be either of ARRAY (multi-valued) type or non-array (single-valued) type. For example, the Update operator has an attribute called QueryBandSessInfo that can be declared either as

```
VARCHAR ARRAY QueryBandSessInfo,
```

or

```
VARCHAR QueryBandSessInfo
```

Since only one form of this declaration can be in the $UPDATE template, the ARRAY form is the one declared in the template:

```
VARCHAR ARRAY QueryBandSessInfo = @UpdateQueryBandSessInfo
```

and if job variable *QueryBandSessInfo* has been assigned an array of values, there is agreement between type and number of values. If *UpdateQueryBandSessInfo* has been assigned a single value without array brackets, however, Teradata PT adds array brackets to the value, creating a one-element array to prevent a type mismatch between the attribute and its singular value.

All attributes that can have either an ARRAY or a scalar declaration are declared with the ARRAY form in their templates.

## Limitations to Template Use

With the availability of templates, including user-defined templates, there is no reason to define consumer or standalone operators in your job scripts. Use of producer operator templates, however, is potentially limited because the definition of a producer operator requires the specification of an explicit (script-defined) schema as part of the DEFINE OPERATOR syntax.

Standalone operators do not employ schemas and, therefore, do not require a schema specification. Consumer operators specify their schemas as follows:

```
SCHEMA *
```

This means that operators accept the schema of whatever data is sent to them in the data streams connecting them to the data sources. But producer operators must identify a script-defined schema that describes the data they place onto the data stream(s) with the syntax phrase

```
SCHEMA schemaName
```

where *schemaName* is the name of a schema defined by a DEFINE SCHEMA statement in your job script. If a producer template actually specified a schema, use of that template would be limited to scripts that included a definition of that particular schema and further limited to occurrences of the template where that schema is the appropriate one.

To overcome this potentially severe restriction on the use of producer operator templates, Teradata PT supports a feature allowing the schema to be specified explicitly or implicitly as part of a producer operator template reference in the APPLY statement, such shown in the following three examples:

```
... FROM OPERATOR( $FILE_READER( DAILY_SALES ) ) ...
```

```
... FROM OPERATOR( $EXPORT( 'Product_Shipments' ) ) ...
... FROM OPERATOR( $FILE_READER( DELIMITED 'Gross_Receipts_YTD' ) )
     ...
```

1. In the first example, the execution of the DataConnector producer operator, invoked by this particular reference to its template name $FILE_READER, will use 'DAILY_SALES' as its schema. Such an explicit schema reference can be made only to a schema that was defined earlier in the your job script through a DEFINE SCHEMA statement.
2. In the second example, the execution of the Export operator, invoked by this particular reference to its template name $EXPORT, will use a Teradata PT-generated schema based on the column descriptions of the Teradata Database table Product_Shipments. Such an implicit schema reference can be made only via the name of a Teradata Database table that already exists at the time your job is submitted.
3. In the third example, the execution of the DataConnector producer operator, will use the delimited-file-format version of a Teradata PT-generated schema based on the column descriptions of existing Teradata Database table Gross_Receipts_YTD.

## Generated Schemas

As examples two and three in the previous section show, Teradata PT will accept the name of a Teradata Database table as a stand-in for a Teradata PT schema that it will generate from the column descriptions of the table. Such generated schemas can be a major convenience when the number of schema columns is large, reducing keyboarding time and keystroke errors and ultimately enabling job scripts to be simpler to write.

To generate a Teradata PT DEFINE SCHEMA statement from a Teradata Database table, Teradata PT makes a HELP TABLE call to the Teradata Database to get the descriptions of the table's columns, and constructs a DEFINE SCHEMA statement, with a generated schema name, for all script invocations of the template operator that specify their schema via this table name. Teradata PT then uses that generated name in the copy of the producer template definition that it imports into the script, for all references to this particular template that share the same schema.

### Example 1

Suppose Teradata Database table Invoice_Counts has 4 columns of the four integer types. Its Teradata PT-constructed DEFINE SCHEMA statement would be as follows:

```
DEFINE SCHEMA $SCHEMA_GEN_TBL001
DESCRIPTION 'SOURCE INFORMATION SCHEMA'
(
  "IC1"    BYTEINT,
  "IC2"    SMALLINT,
  "IC4"    INTEGER,
  "IC8"    BIGINT
);
```

and the copy of any producer template imported into the script that specified its schema via the table name 'Invoice_Counts' would have

```
SCHEMA $SCHEMA_GEN_TBL001
```

as its schema specification. Notice that the generated schema name incorporates a sequence number ('001' in this example) that enumerates the number of schemas generated for any job script. Notice also that the generated schema column names are enclosed in double quotes to avoid any potential conflicts with Teradata PT reserved words.

If our example Teradata Database table Invoice_Counts, whose generated schema is shown above, were to be the target table for a delimited data file source, then the job script would require the delimited-file-format version of the schema; it can be requested by the DELIMITED keyword, as shown in the third producer template reference example above, and again here:

```
... FROM OPERATOR( $FILE_READER( DELIMITED 'Invoice_Counts' ) ) ...
```

In this case, all columns in the producer's generated schema will be type VARCHAR to match the character format of the data, regardless of how the target columns are defined in the Teradata Database target table. (The Teradata Database performs all necessary data conversions from VARCHAR to non-character data type representations at loading time.) With the DELIMITED keyword present, as shown the above example, Teradata PT would generate the following delimited-file-format version of the schema for table Invoice_Counts:

```
DEFINE SCHEMA $SCHEMA_GEN_D_TBL003
DESCRIPTION 'SOURCE INFORMATION SCHEMA'
(
  "IC1"    VARCHAR(4),
  "IC2"    VARCHAR(6),
  "IC4"    VARCHAR(11),
  "IC8"    VARCHAR(20)
);
```

Notice that the generated schema name contains an "_D" appended to it so as to distinguish it from the name of the non-delimited version of the schema, in case a job script might need both versions of the schema.

### Example 2

Teradata PT will generate a schema from a Teradata Database table in one additional script context as illustrated by the following two DEFINE SCHEMA statements:

```
DEFINE SCHEMA TODAYS_TRANSACTIONS FROM TABLE 'Daily_Trans';
DEFINE SCHEMA INVOICE_COUNTS FROM TABLE DELIMITED 'Invoice_Counts';
```

Teradata PT will generate a DEFINE SCHEMA statement based on the column descriptions of the identified Teradata Database table and substitute it in the script for the original, abbreviated DEFINE SCHEMA statement.

The only difference from schemas generated from producer template references is that the generated schema will have the name supplied in the original DEFINE SCHEMA statement rather than a generated name: TODAYS_TRANSACTIONS and INVOICE_COUNTS, respectively, in the above two examples. There is no functional difference between a schema defined via a Teradata Database table and a schema defined by a fully-coded DEFINE SCHEMA statement.

## Generated Schemas Based on SQL SELECT Statements

Teradata PT will also generate a DEFINE SCHEMA statement based on the result columns of an SQL SELECT statement:

### Example 3

```
DEFINE SCHEMA PROD_EXT FROM SELECT 'Select a,b,c,sum(d) from Products;';
DEFINE SCHEMA TRANS FROM SELECT OF OPERATOR EXPORT2;
```

The first of these DEFINE SCHEMA statements requests Teradata PT to pass the specified SQL SELECT statement to the Teradata Database and generate a fully-specified DEFINE SCHEMA statement from the definitions of the result columns of the SQL SELECT statement, as determined by the Teradata Database.

The second of these DEFINE SCHEMA statements requests Teradata PT to obtain the SQL SELECT statement that is the specified value of the *SelectStmt* attribute of the script-defined operator named EXPORT2, to pass it to the Teradata Database, and then to generate a fully-specified DEFINE SCHEMA statement from the result columns of that SQL SELECT statement.

The capability to generate schemas from SQL SELECT statements can be quite useful for complex data extractions, for example SELECT statements involving joins, and for tables with column-by-column character encodings, for which the Teradata Database can be relied on to calculate the correct column length values.

There is no functional difference between a schema defined via an SQL SELECT statement and a schema defined by a fully-coded DEFINE SCHEMA statement.

## Inferred Schemas

When a producer operator template reference does not include a schema specification, that is, when it does not include either a schema name or a Teradata Database table name, Teradata PT may be able to infer the appropriate schema and, if not a script-defined schema, generate a DEFINE SCHEMA statement for it.

Teradata PT analyzes all operator usage in the job step containing the template reference and attempts to determine what the schema of the template should be in order for the APPLY statement of the job set to make sense.

### Example 1

Consider this example.

```
STEP LOAD_2
(
  APPLY  <DML statement(s)>

  TO OPERATOR( $INSERTER() )

  SELECT * FROM OPERATOR( $EXPORT() )

  UNION ALL

  SELECT * FROM OPERATOR( EXPORT_OPER2() );
```

```
);
```

The script-defined producer operator EXPORT_OPER_2 must have been defined previously in the job script, with a schema specification of the name of a script-defined schema. Since the source data extracted by EXPORT_OPER_2 is merged into a single input data stream with the source data extracted by producer template operator $EXPORT, Teradata PT can infer that the schema for both producer operators must be the same and can then use the name of the schema for EXPORT_OPER_2 in the copy of the $EXPORT template that it imports into the job script.

### Example 2

In this example, an SQL SELECT statement is inferred as the basis for generating the schema for the $EXPORT producer template:

```
STEP INSERT_DAILY_TRANS
(
  APPLY  <DML statement(s)>

  TO OPERATOR( $INSERTER() )

  SELECT *

  FROM OPERATOR
  (
    $EXPORT()
    ATTR
    (
      PrivateLogName = 'daily_trans.log',
      SelectStmt     = 'Select * from Daily_Trans;'
    )
  );
);
```

In the above job step, the SelectStmt attribute of the $EXPORT template has been assigned the value Select * from Daily_Trans that, according to the definition of this attribute, must be an SQL SELECT statement. By querying the Teradata Database, Teradata PT can generate a schema based on the columns of the result table of the SELECT statement since that will accurately describe the source data produced by the Export operator invoked via template $EXPORT.

Basing the generated schema on the value of the SelectStmt attribute works for producer templates $EXPORT and $SELECTOR, since both of their underlying operators require this attribute.

### Example 3

In this example, Teradata PT determines the identity of the target table of a consumer operator in the job step and infers that a schema based on this table will be the basis for generating a schema for the producer template(s) in the job step:

```
STEP INSERT_MONTHLY_SHIPMENTS
(
  APPLY  <DML statement(s)>
```

```
    TO OPERATOR
    (
      $LOAD()
      ATTRIBUTES
      (
        PrivateLogName = 'monthly_ship.log',
        TargetTable    = 'Monthly_Shipments'
      )
    )

    SELECT * FROM OPERATOR( $FILE_READER() );
  );
```

In the above example, Teradata PT can determine, from the 'TargetTable' attribute of the $LOAD operator template, that the target for data to be loaded in this job step is Teradata Database table Monthly_Shipments.

In the absence of any script specification of the schema from the producer template, and the absence of any script indicators of a possible basis for generating this source schema, Teradata PT makes the assumption that source data will be loaded unchanged into the target table and thus that the target Teradata Database table can be inferred to be the basis for generating the source schema for the producer template, $FILE_READER.

Since this assumption may not be correct in all cases in which only the target table is known to Teradata PT, some inferred schemas may not accurately describe the source data from a producer template and the resulting schema mismatch will be detected by the Teradata PT and cause the job to fail. This is a limitation to the inferred schema feature: for the schema for the producer template to be correctly inferred from a Teradata Database target table, the source data must move unchanged through Teradata PT to the target. To be more specific: target columns may not be a projection of source columns nor include any derived columns. Only SELECT * can describe the rows from the producer template operator in order for Teradata PT to be able to infer the correct schema for that producer template.

Teradata PT will not attempt to infer a schema for a producer template from a target table when the job step contains multiple target tables, such as syntax that employs the multiple APPLY feature, or when the step consumer operator is the Update operator (or $UPDATE template) and it is going to update more than one target table.

Job steps with no basis for Teradata PT to infer a schema for a producer template, as well as job steps that result in schema mismatch errors when Teradata PT infers an incorrect schema for a template can always be remedied through the use of a set of special job variables, as described in the next section.

## Special Job Variables for Inferring and Generating a Schema

To generate a schema, either based on a Teradata Database table or based on the results table of an SQL SELECT statement, Teradata PT must first logon to the appropriate Teradata Database to get the column descriptions of the schema to be generated. If the template operator reference itself has logon attribute assignments in its ATTRIBUTES list, then those values will be used for Teradata Database logon. If not, then Teradata PT queries one of two

sets of special conventionally-named job variables, either of which, when assigned values, will control Teradata PT's logon to the desired Teradata Database:

| | |
|---|---|
| SourceTdpId | TargetTdpId |
| SourceUserName | TargetUserName |
| SourceUserPassword | TargetUserPassword |
| SourceAccountId | TargetAccountId |
| SourceWorkingDatabase | TargetWorkingDatabase |

When Teradata PT attempts to generate the schema for a producer template, it checks three conditions in order to determine which of the above sets of logon job variables will be queried to provide Teradata Database logon parameters:

1 If the job variable SelectStmt has a value, or if the template reference includes a SelectStmt attribute assignment, then that value, an SQL SELECT statement, from either of these sources will be the basis for the generated schema, and the Source set of logon job variables will be queried to obtain DSB logon parameters.
2 If the job variable SourceTable has a value, then that value, a Teradata Database table name, will be the basis for the generated schema, and the Source set of logon job variables will be queried to obtain DSB logon parameters.
3 If the job variable TargetTable has a value, or if the consumer operators in the job step of producer template collectively identifies one and only one target table, via a TargetTable attribute assignment, then that Teradata Database table will be the basis for the generated schema, and the Target set of logon job variables will be queried to obtain Teradata Database logon parameters.

If the set of logon job variables queried yields logon parameters, and Teradata PT gets the schema column descriptions from the requested Teradata Database, Teradata PT still must determine which schema format to generate: the delimited-file format or the regular column-mapped format. Teradata PT supports one more special job variable that provides this information:

```
SourceFormat
```

Before generating the DEFINE SCHEMA statement, Teradata PT queries the special job variable SourceFormat. If has the value Delimited, then the generated schema will be in delimited-file format; otherwise, it will be in the normal format in which the schema column definitions closely match the Teradata Database table's column descriptions.

## Step-Scope and Job-Scope Special Job Variables

Through the special job variables described above Teradata PT provides for complete control of the schema inference and generation processes, which essentially overcome the limitations to the use of producer operator templates. However, typical jobs will have multiple job steps and a single set of values for these controlling job variables would not necessarily lead to the correct results in every job step.

For that reason Teradata PT supports a step-specific variant of each of these job variables. For example, for a job step named LOAD3, if the job variable:

```
LOAD3_SourceTable
```

is assigned a value, that value identifies the Teradata Database table that would be the basis for generating the schema for any producer template in job step LOAD3 which did not already have its own schema specification. Similarly, all of the other special job variables have step-scope variants of the general form:

```
<step name>_<job variable name>
```

Support for step-scope variants of these special job variables opens up their "stepless" variants (job variable names not prefixed by a step name) to provide job-scope default values. For example, if the step-scope job variable LOAD3_TdpId has not been assigned a value, but the job-scope default job variable TdpId has been assigned a value, then Teradata PT will use the value of TdpId to logon to a Teradata Database, if job step LOAD3 requires any generated schemas.

To make it easier to recognize the job scope of these special stepless job variables Teradata PT also supports them with the job_ prefix. For example, if assigned a value, the job variable:

```
job_UserName
```

provides the job-scope default database logon user name.

In summary, the precise value assignments to these special job variables, implemented to support the generation of correct schemas for producer templates, combined with schema specifications in producer template references allow you to use producer templates in almost all APPLY statements.

## When Schema-generation Job Variables are Not Used

If logon parameters are not assigned to either step-scope or job-scope logon job variables, Teradata PT may still be able to determine which Teradata Database to log onto to get schema column descriptions, using logon parameters assigned to operator logon attributes in your job script:

- If the producer template reference itself has an ATTRIBUTES list that contains logon attribute assignments, then those attribute values will be used to logon to the specified Teradata Database.
- If all other producer references in the same job step are collectively assigned exactly one unique set of logon attribute values, then these logon attribute values will be used to logon to the specified Teradata Database. If more than one set of producer logon attributes exist in the job step, Teradata PT will terminate the job.
- If all other producer references in the entire job collectively are assigned exactly one unique set of logon attribute values, then these logon attribute values will be used to logon to the specified Teradata Database. If more than one set of unique logon attributes exist within the job, Teradata PT will terminate the job.
- If all consumer references in the same job step collectively are assigned exactly one unique set of logon attribute values, then these logon attribute values will be used to logon to the

specified Teradata Database. If more than one unique set of consumer logon attributes exist in the job step, Teradata PT will terminate the job.

- If all consumer references in the entire job collectively are assigned exactly one unique set of logon attribute values, then these logon attribute values will be used to logon to the specified Teradata Database. If more than one unique set of consumer logon attributes exist in the job, Teradata PT will terminate the job.

## Generated SQL Insert Statements

Teradata PT supports one additional feature to reduce script size and eliminate keystroke errors: it will generate any SQL INSERT statement, if the target table is specified or can be unambiguously determined from your script. For example, if an SQL INSERT statement is coded in your script as

```
$INSERT 'Invoice_Counts'
```

and Teradata Database table Invoice_Counts has the 4 columns I1, I2, I4 an I8, then Teradata PT will replace "$INSERT 'Invoice_Counts'" in your script with the following generated SQL INSERT statement:

```
'INSERT INTO Invoice_Counts VALUES (
          :I1,
          :I2,
          :I4,
          :I8);'
```

Using a job variable you name to identify the target table would work equally well. For example:

```
$INSERT  @Insert1
```

will cause Teradata PT to use the value of job variable 'Insert1' as the Teradata Database table name in the resulting generated INSERT statement. In general the more columns a Teradata Database table has, the more useful the $INSERT macro is, if the job script requires an SQL INSERT statement for that table.

The use of table-specifying $INSERT macros makes possible APPLY statements such as the following:

```
STEP LOAD_QRTRS
(
  APPLY

    CASE

      WHEN( TRANS_DATE <= '2011-03-31' )

        THEN  $INSERT 'Q1_Trans'

      WHEN( TRANS_DATE <= '2011-06-30' )

        THEN  $INSERT 'Q2_Trans'

      WHEN( TRANS_DATE <= '2011-09-30' )

        THEN  $INSERT 'Q3_Trans'
```

```
        ELSE  $INSERT 'Q4_Trans'
      END
    TO OPERATOR( $LOAD() )
    SELECT * FROM OPERATOR( $FILE_READER() );
  );
```

The use of $INSERT without a Teradata Database table specifier is still supported in any job step, if Teradata PT can identify the target table by the following procedure:

- If the step-scope job variable

  `<step name>_TargetTable`

  has been assigned a value, then that value will be used as the target table in the generated SQL INSERT statement.
- If the job-scope job variable

  `TargetTable`

  has been assigned a value, then that value will be used as the target table in the generated SQL INSERT statement.
- If the 'TargetTable' attribute of the consumer operator(s) referenced in the job step collectively identifies a single target table, then that table will be the target table in the generated SQL INSERT statement.

If the target table of an instance of $INSERT without a table specifier is ambiguous or cannot be identified at all, then Teradata PT will terminate the job with an explanatory error message.

# Using the Job Identifier in Your Job Script

Teradata PT constructs a unique identifier for each job submitted for execution. Even though it is not generated until your job executes, you can reference its unique job identifier in your job script via the new script keyword $JOBID and $$JOBID.

- Use $JOBID to get the job identifier as a quoted string.
- Use $$JOBID to get the job identifier as an unquoted token but with the underscore character replacing the hyphen between the job name and the job sequence number.

For example, $JOBID becomes 'ws150002-02335' and $$JOBID becomes ws150002_02335.

The $JOBID and $$JOBID feature allows you to use the unique Teradata PT-constructed job identifier in the names of script items, which can be especially useful for items that persist after job execution such as log files.

# 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 204.

## 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 203.

## 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);
```

# Using VARDATE Columns To Reformat DateTime Data

Often varying length date, time, timestamp, and interval data is loaded into the Teradata Database as VARCHAR data. The Teradata Database refers to this as DateTime data. For a full definition of DateTime data, see *SQL Data Types and Literals*.

In some cases, DateTime data is formatted in a way that does not match the format that the Teradata Database expects. To load this DateTime data, Teradata PT enables you reformat it by specifying in the DEFINE SCHEMA statement of your load job a column type of VARDATE along with input and output format strings. These format strings reformat incoming DateTime data, enabling disparate sources of DateTime data to be loaded into a single Teradata Database.

Columns defined with type VARDATE must be followed by a:

- Column size
- FORMATIN string
- FORMATOUT string.

The FORMATIN string specifies the format for the incoming DateTime data.

The FORMATOUT string specifies the desired output format and must match the format of the database column into which data is being loaded.

The size of the VARDATE column must always be equal to or greater than the length of the larger formatting string.

For the DEFINE SCHEMA statement syntax for VARDATE, see *Teradata Parallel Transporter Reference*.

## Example: Using the VARDATE Column Data Type

If the first column of a Teradata Database table is defined with type DATE during table creation, only input data in the form 'YY/MM/DD' is accepted. For information about the default DateTime formats that Teradata Database accepts, see *SQL Data Types and Literals*.

However, the data we want to load into our example column looks as follows:

```
JAN-21 1999 | …
FEB-03 1997 | …
AUG-24 2001 | …
```

To load this data into our table successfully, we must define a SCHEMA statement whose first column is of type VARDATE and whose format strings match the parameters described above:

```
DEFINE SCHEMA VARDATE_EXAMPLE(
       COL1 VARDATE(15) FORMATIN('MMM-DDBYYYY') FORMATOUT('YY/MM/DD');
       …
       …
           );
```

For complete VARDATE syntax, see *Teradata Parallel Transporter Reference*.

The following table provides a non-exhaustive set of VARDATE examples with their column definitions and their column type specifications for a DEFINE SCHEMA statement. Some column definitions include the FORMAT phrase.

For more information, see “Data Type Formats and Format Phrases” in *SQL Data Types and Literals*.

| Example Data | Column Definition | Example Schema |
|---|---|---|
| 12:03:25 | DATE | VARDATE(8)<br>FORMATIN('YY:MM:DD')<br>FORMATOUT('YY/MM/DD') |
| FEB 01 1999 | DATE FORMAT 'MMM-DD-YY | VARDATE(12)<br>FORMATIN('MMMBDDBYYYY')<br>FORMATOUT('MMM-DD-YY') |
| 10:12 AM | TIME | VARDATE(8)<br>FORMATIN('HH:MIBT')<br>FORMATOUT('HH:MI:SS') |
| 08:55:00+02:00 | TIME FORMAT 'HH.MI.SS | VARDATE(14)<br>FORMATIN('HH:MI:SSZ')<br>FORMATOUT('HH.MI.SS') |
| 01:15 PM AUG 18 2005 | TIMESTAMP | VARDATE(26)<br>FORMATIN('HH:MIBTBMMMBDDBYYYY')<br>FORMATOUT('YYYY-MM-DDBHH:MI:SST') |
| 2012-02-29 12:35+02:00 | TIMESTAMP FORMAT 'HH:MI MM-DD-YY' | VARDATE(22)<br>FORMATIN('YYYY-MM-DDBHH:MIZ')<br>FORMATOUT('HH:MI MM-DD-YY') |
| 2012-04-06 18:04:01 | INTERVAL YEAR(2) TO MONTH | VARDATE(20)<br>FORMATIN('YYYY-MM-DDBHH:MI:SS')<br>FORMATOUT('YY-MM') |
| 2012-04-06 18:04:01 | INTERVAL DAY(2) TO HOUR | VARDATE(20)<br>FORMATIN('YYYY-MM-DDBHH:MI:SS')<br>FORMATOUT('DD HH') |

## Supported Formatting Characters

Teradata PT provides a set of formatting characters so FORMATIN strings can represent all types of incoming DateTime data and FORMATOUT strings can represent all Teradata Database formats, both default and user defined.

The following table shows the formatting characters for the FORMATIN and FORMATOUT strings that Teradata PT supports.

| Formatting Character | Description | Examples |
|---|---|---|
| Separators | Any non-alphanumeric character that separates two alphanumeric formatting characters. | "/": Slash Separator<br>",": Comma Separator<br>"'": Apostrophe Separator<br>":": Colon Separator<br>".": Period Separator<br>"-": Dash Separator<br>"B": Space Separator |
| YY | A two-digit year that is valid for the calendar, numeric characters only. When converting a two digit year into a four digit year, the two digit year is appended with '19'. | 03<br>(stored as 1903 when converted to four digit year) |
| YYYY<br>Y4 | A four-digit year that is valid for the calendar, numeric characters only. | 2003 |
| MM | The month of the year as two numeric characters that are valid for the calendar. | 02 |
| MMM<br>M3 | A three-character month that matches one of the names specified by ShortMonths in the current Specification for Data Formatting (SDF) file. | Jan<br>Feb<br>May |
| MMMM<br>M4 | A full month name that matches one of the names specified by LongMonths in the current SDF. | January<br>February<br>May |
| EEEE<br>E3 | An abbreviated day of the week name that matches one of the names specified by ShortDays in the current SDF. | Fri |
| EEEE<br>E4 | A day of the week name that matches one of the names specified by LongDays in the current SDF. | Friday |
| DD | The day of the month as two numeric characters that are valid for the calendar. | 03<br>24 |
| HH | Represents the hour as two numeric digits. | 23 |

| Formatting Character | Description | Examples |
|---|---|---|
| MI | Represents the minute as two numeric digits. | 59 |
| SS | Represents the second as two numeric digits. | 01 |
| T | Represents time in 12-hour format instead of 24-hour format. The appropriate time of day, as specified by AMPM in the current SDF is copied to the output string where a T appears in the FORMAT phrase. | 01:20:01PM<br>('HH:MI:SST') |
| Z | Time zone.<br>The Z controls the placement of the time zone in the output of TIMESTAMP data, and can only appear at the beginning or end of the time formatting characters. | 98-01-01 +00:00<br>('YY-MM-DDBZ') |

## DataConnector Operator Support

Only the DataConnector producer operator supports VARDATE formatting. This means that DateTime data being loaded into the Teradata Database must come from a flat file and not from another database.

When the DataConnector producer operator is in delimited mode, all of the Teradata PT schema's columns that are not being reformatted must be of type VARCHAR. A VARDATE column in the Teradata PT DEFINE SCHEMA statement means that the data being read in from the flat file for the given column will be modified based on formatting strings before being sent to the Teradata database.

VARDATE formatting does not affect consumer operators.

Even though in some cases Teradata PT attempts to prepend 0's to double digit times and dates that are missing a first digit, you must provide formatting strings that match every character of the incoming data, as well as the format of target columns.

For every row, all columns of incoming data with type VARDATE must conform to the format specified by the FORMATIN string of that column. If the formatting of a given row differs significantly from the specified FORMATIN string, or if the desired conversion from FORMATIN to FORMATOUT is unsuccessful due to a lack of data, a conversion failure occurs. If it does and you have specified the RowErrFile Name attribute of the DataConnector operator, erroneous rows are sent to an error file; otherwise the job fails.

CHAPTER 14

# Operational Metadata

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
- Sending Operational Metadata to TMSM

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.

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

- Job step that has finished execution or encountered an error
- Storage fragmentation has reached a high-water mark
- Shared memory usage has reached a high-water mark
- Job termination request received from user

# 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:

```
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:

```
create 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:

```
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 235.

# Exporting and Loading Metadata

Operational metadata are stored in the following Teradata PT predefined private logs for each job:

- TWB_STATUS log
- TWB_SRCTGT log
- TWB_EVENTS log

Using the data schema described in "Viewing Metadata" on page 234, 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.

# Sending Operational Metadata to TMSM

Teradata Multi-System Manager (TMSM) is the monitoring and control facility for a variety of Dual Active Solutions. The users of this facility include Enterprise Data Warehouse (EDW) users or anyone who needs to monitor and control processes including, but not limited to, Teradata Load and Unload Utilities, Teradata SQL, ETL tools, and Teradata Database.

To integrate with TMSM, Teradata PT has been enhanced to collect operational metadata and event data from operators. To do this, Teradata PT obtains the Unit of Work ID (UOW ID) from TMSM for a job, and sends it to TMSM using the *send event* interface. By default, a Teradata PT job sends events to TMSM as long as TMSM is active. If TMSM is not active, the job runs without sending events to TMSM. For more information about the send event interface, see the TMSM Event System API Reference.

Teradata PT allows the following Teradata PT operators, which can be regarded as resource types from the TMSM point of view, to be monitored:

Table 18: TMSM Resource Types and Teradata PT Operators

| TMSM Predefined Resource Types | Teradata PT Operators |
| --- | --- |
| TMSM_RESOURCETYPE_TPT_EXPORT | Export Operator |
| TMSM_RESOURCETYPE_TPT_UPDATE | Update Operator |
| TMSM_RESOURCETYPE_TPT_LOAD | Load Operator |
| TMSM_RESOURCETYPE_TPT_STREAM | Stream Operator |
| TMSM_RESOURCETYPE_TPT_INSERT | Schema Mapping Operator |

Simple ETL process monitoring tracks a process from the start to the end. A process can include multiple steps, each of which represents an activity or event to be monitored. For example, a Teradata PT load job can be regarded as such a process.

TMSM requires the following Teradata PT flow:

1. Obtain a system-generated UOW ID from TMSM for a Teradata PT job.
2. Send a "start" event to TMSM along with the UOW ID.
3. (Option) Send one or more "step" events to TMSM along with the UOW ID.
4. Send an "end" event to TMSM along with the UOW ID.

## Example: Load Job

- Start Event - *Connecting sessions* message containing a Teradata PT job ID, a Teradata PT step name, and a TDPID Step Event message (*Acquisition begins*).
- Step Event - *Checkpoint completes* message.
- Step Event - *Acquisition completes* message.
- Step Event - *Application completes* message.
- Step Event - *Rows inserted* message containing the number of rows inserted into the target table.
- End Event - *Job terminating* message containing the total number of rows sent to the DBS.

## Example: Update Job

- Start Event - *Connecting sessions* message containing a Teradata PT job ID, a Teradata PT step name, and a TDPID.
- Step Event - *Checkpoint completes* message.
- Step Event - *Acquisition completes* message.
- Step Event - *Application completes* message.
- Step Event - *Rows inserted* message containing the number of rows inserted into the target table (1 message per table).
- Step Event - *Rows updated* message containing the number of rows that were updated against each of the target tables (1 message per table).
- Step Event - *Rows deleted* message containing the number of rows that were deleted from each of the target tables (1 message per table).
- End Event - *Job terminating* message containing the total number of rows sent to the DBS.

## Example: Stream Job

- Start Event - *Connecting sessions* message containing a Teradata PT job ID, a Teradata PT step name, and a TDPID.
- Step Event - *Loading begins* message.
- Step Event - *Loading completes* message.
- Step Event - *Rows inserted* message containing the number of rows inserted into the target table (1 message per table).
- Step Event - *Rows updated* message containing the number of rows that were updated against each of the target tables (1 message per table).
- Step Event - *Rows deleted* message containing the number of rows that were deleted from each of the target tables (1 message per table).
- End Event - *Job terminating* message containing the total number of rows sent to the DBS.

## Example: Export Job

- Start Event - *Connecting sessions* message containing a Teradata PT job ID, a Teradata PT step name, and a TDPID.

- Step Event - *Retrieving rows* message.
- End Event - *Job terminating* message containing the total number of rows exported.

CHAPTER 15
# Best Practices

This chapter describes Teradata PT best practices for loading data into the Teradata Database.

## Loading Data Using Teradata PT

Teradata PT provides a comprehensive set of practices for loading data into the Teradata Database, including:

- "Writing Job Scripts for Reusability and Manageability"
- "Writing Job Scripts for Scalable Performance"
- "Determining System Resource Usage at the Job Level"
- "Using Teradata PT Periodic Loading for Active Data Warehousing"
- "Using the ELT Approach for Loading"
- "Managing and Monitoring Teradata PT Jobs"
- "Writing Load Scripts for Restartability and Availability"

### Writing Job Scripts for Reusability and Manageability

Teradata PT employs a single scripting language for extracting and loading data Teradata also provides:

- A simplified version of its scripting language, called "Simplicity," that uses pre-defined reusable operator templates easy to maintain. Using pre-defined operator templates vastly reduces the number of lines of code you have to write. Job schemas are generated dynamically based on the source or target table. For information on simplified Teradata PT scripts, see "Simplifying Scripts with Operator Templates and Generated Schemas" on page 211.
- A utility called Easy Loader that allows you to run a Teradata PT load job using a command line interface. This makes it unnecessary to write a job script. For information on Easy Loader, see Chapter 12: "Teradata PT Easy Loader."

#### Using the Simplicity Job Script

Simplicity features include operator templates that are imported into the job script when operators are referenced in an APPLY/SELECT statement, as shown in the following code excerpt:

```
APPLY 'INSERT INTO TABLE_X (:col1, :col2);'
TO OPERATOR ($LOAD)
SELECT * FROM OPERATOR ($FILE_READER);
```

By referencing the $LOAD and $FILE_READER operator template nameS, Teradata PT knows to import the Load template into a job script. If reducing the amount of code for loading or exporting and simplifying job maintenance is important to you, use the simplified job script method.

The following two examples illustrate loading a delimited file into a Teradata table:

### Example 1: Job Script with Simplified Syntax

```
DEFINE JOB PLOAD_JOB
DESCRIPTION 'PLOAD JOB'
(
  /* Use the schema of the TargetTable for TPT_SCHEMA */
  DEFINE SCHEMA TPT_SCHEMA DELIMITED @LoadTargetTable;

  APPLY $INSERT @LoadTargetTable TO OPERATOR ($LOAD [@LoadInstances])
  SELECT * FROM OPERATOR ($FILE_READER(TPT_SCHEMA) [@ReaderInstances]);
);
```

### Example 2: Job Script without Simplified Syntax

```
DEFINE JOB PLOAD_JOB
DESCRIPTION 'PLOAD JOB'
(
  DEFINE SCHEMA TPT_SCHEMA
  DESCRIPTION 'TPT SCHEMA'
  (
    COL001    VARCHAR(100),
    COL002    VARCHAR(100),
    COL003    VARCHAR(100)
  );

  DEFINE OPERATOR LOAD_OPERATOR
  DESCRIPTION 'TPT Load Operator'
  TYPE LOAD
  SCHEMA *
  ATTRIBUTES
  (
    VARCHAR TdpId             = @LoadTdpId,
    VARCHAR UserName          = @LoadUserName,
    VARCHAR UserPassword      = @LoadUserPassword,
    VARCHAR TargetTable       = @LoadTargetTable,
    VARCHAR LogTable          = @LoadLogTable,
    VARCHAR ErrorTable1       = @LoadErrorTable1,
    VARCHAR ErrorTable2       = @LoadErrorTable2,
    VARCHAR PrivateLogName    = @LoadPrivateLogName
  );

  DEFINE OPERATOR FILE_READER_OPERATOR
  DESCRIPTION 'TPT DataConnector Producer Operator'
  TYPE DATACONNECTOR PRODUCER
  SCHEMA TPT_SCHEMA
  ATTRIBUTES
  (
    VARCHAR FileName          = @FileReaderFileName,
    VARCHAR Format            = @FileReaderFormat,
    VARCHAR OpenMode        = @FileReaderOpenMode,
    VARCHAR TextDelimiter   = @FileReaderTextDelimiter,
```

```
    VARCHAR MultipleReaders   = @FileReaderMultipleReaders,
    VARCHAR PrivateLogName    = @FileReaderPrivateLogName
  );

  APPLY
  (
    'INS ' || @LoadTargetTable || ' (
     :COL001,
     :COL002,
     :COL003
   );'
 TO OPERATOR (LOAD_OPERATOR [@LoadInstances])
 SELECT *
 FROM OPERATOR ( FILE_READER_OPERATOR [@ReaderInstances]);
);
```

As “Example 1: Job Script with Simplified Syntax” on page 240 shows, using templates eliminates the need to write DEFINE OPERATOR statements for operators in the job script itself. The reference to the job variable `@LoadTargetTable` in the DEFINE SCHEMA statement allows Teradata PT to generate the job schema at runtime based on the value provided for the job variable.

### Value in Using Job Variables File

Using job variables defined in a job variables file allows the job script compiler to assign values to job variables at script execution time rather than having to have you code values as constants in every script. Each job variable in a script should be prefixed with an `@` sign so that the script compiler replaces each variable with the corresponding value when the Teradata PT job executes.

Teradata PT allows unlimited variable substitution in a script, maximizing the reusability of scripts across systems. Moreover, using a job variables file allows multiple jobs to share a common set of variable values.

For information on a job variables file, see “Setting Up the Job Variables Files” on page 68.

### Advantages of a Simplicity Script

- Smaller and simpler job scripts
- Less job script maintenance with the use of operator templates
- Scripts that can be customized by using user-defined templates
- Generated schema objects
- Generated SQL insert statements
- Standardized job variable names corresponding to operator attributes

## Writing Job Scripts for Scalable Performance

### Using Multiple Operator Instances for Scalability

As a multi-process application that exploits the parallel and scalable framework, Teradata PT makes it possible to use additional CPU processing power, shorten the load process, and reduce overall job execution time. You can specify the number of operator instances in your

job script. This gives you the control over the scalability and performance of the data loading process.

In addition, Teradata PT allows data extraction and data loading to run completely asynchronously from each other. This supports broader parallelism, which further improves performance.

With traditional Teradata standalone utilities, such as FastLoad, MultiLoad, and Tpump, which rely on a single system process to perform data extraction and loading, a single process can reach a threshold beyond which CPU speed cannot increase, a critical limiting factor.

### Using Directory Scan for Loading Files in Parallel

Teradata PT provides a feature called Directory Scan that enables data files in a directory to be processed in a parallel and scalable manner as part of the loading process. In addition, if multiple directories are stored across multiple disks, a special feature in Teradata PT called UNION ALL can be used to process these directories of files in parallel, thus achieving more throughputs across disks. See “Combining Multiple Sources using UNION ALL” below.

Directory scans also provide an option that lets users select files for processing based on file names, which include wildcard specifications. The DataConnector operator provides scalable and parallel access to multiple files in a load-balancing manner. By *load balancing* we mean that the files are distributed as evenly as possible based on file sizes among operator instances.

Teradata standalone utilities, such as FastLoad, MultiLoad, and Tpump only allow one file to be processed at a time.

### Reading a Large File in Parallel

One of the most common loading scenarios is loading huge data file into the data warehouse. As opposed to the standalone utilities, which are limited by the CPU speed within a single system process, Teradata PT allows a file to be processed in parallel using multiple instances of the Data Connector operator, each of which is executed under a different system process, thus increasing the speed of reading a file.

As shown in Figure 46, users can specify multiple instances of the DataConnector operator to read a file by setting the MultipleReaders attribute to “Yes”. Once this attribute is enabled, each instance of the DataConnector operator processes an equal subset of records in the file.

For example, with 2 instances, instance 1 would return record sequence 1, 3, 5, and so on, and instance 2 would return record sequence 2, 4, 6, and so on. Although multiple instances are used to read the file, this does not necessarily result in double I/O because most operating systems support file caching and satisfy application read requests from the cache. As long as the instances stay synchronized and each instance reads the same amount of data, these reads are satisfied from the file cache by the operating system. This can proceed at the maximum sequential read rate of the disk.

Figure 46: Parallel Reading and Loading of a File

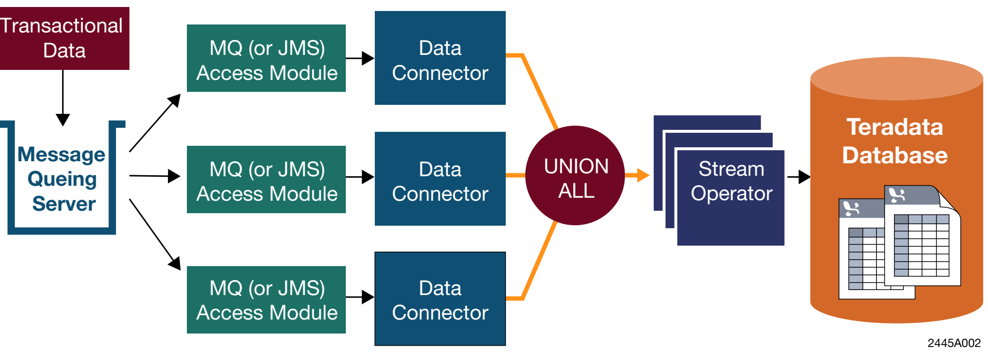
Rows
1, 3, 5, 7, 9,...
Data Connector
File
Data Connector
Rows
2, 4, 6, 8, 10,...
Load Operator
Teradata Database
Production Table
2445A001

### Combining Multiple Sources using UNION ALL

Similar to the UNION ALL operation which allows multiple UNION-compatible tables to be combined, the Teradata PT UNION ALL feature allows similar or dissimilar data sources to be combined into a single source that can be processed in a parallel and scalable manner. This operation also eliminates the need for manually merging multiple data sources as input for loading.

As shown in Figure 47, multiple copies of access modules can be launched by multiple instances of the DataConnector operator for reading transactional data from the same or different message queues. This parallel arrangement, which enables data parallelism, can significantly improve the performance of data extraction.

Figure 47: Parallel Reading of MQ via UNION ALL




## Determining System Resource Usage at the Job Level

### Determining the Use of the Number of Instances

Although most operators in Teradata PT can be scaled to use multiple instances for achieving maximum throughput, excessive use of instances can lead to over-parallelism, which can affect performance adversely. Each instance added to a job may introduce more data streams for data transfer, resulting in more shared memory, more semaphores, and additional system processes.

The following methods are recommended to manage the use of the number of instances:

- Do not create more instances than needed because this will consume system resources. Start with 2 instances and work your way up.

  You probably only need 1 to 4 instances of any given operator in most loading scenarios. However, scenarios like Directory Scan and LOB Loading may require more producer instances and consumer instances, respectively.
- Measure where a bottleneck may be occurring when data is being loaded. Teradata PT can be scaled to eliminate data I/O and load process CPU bottlenecks.
- Read the TWB_STATUS private log which displays statistics showing how much data was processed by each instance. Job performance is evaluated based on the number of CPU seconds and elapsed time in seconds in the TWB_STATUS log.
- Reduce the number of instances if you see underutilized instances of operators. Both operator private logs and the TWB_STATUS log provide detailed information at the instance level. See "Using the TWB_STATUS Private Log to Obtain Job Status" below.

## Determining the Use of Shared Memory

More instances of operators in a job require more allocated shared memory for data streams. By default, Teradata PT provides 10M of shared memory per job. If you want to employ more producer or consumer instances to boost parallelism and scalability, you need to allocate more shared memory. The **tbuild -h** option can be used to increase shared memory size. See the section on the **tbuild** command in the *Teradata Parallel Transporter Reference*.

The following formula for calculating the size of shared memory required for a job with multiple producer/consumer instances appears below:

Note: The formula used in Example 1 and Example 2 is for non-Buffer Mode loading.

```
[65000 x (Producer_count x Consumer_count) x 2] bytes + [65000 x
(Producer_count + Consumer_count)] bytes
```

**Example 1**

Shared memory used by 2 producers and 2 consumers:

```
(65000 x 2 x 2 x 2) bytes + (65000 x (2 + 2)) bytes = 780000 bytes
```

**Example 2**

Shared memory used by 4 producers and 4 consumers:

```
(65000 x 4 x 4 x 2) bytes + (65000 x (4 + 4)) bytes = 2600000 bytes
```

For Buffer Mode loading, see "Determining the Size of Shared Memory for Buffer Mode Loading" below. This section provides more information about shared memory allocation.

For a Directory Scan that reads a very large number of files from a directory, add the following amount of shared memory to account for the size of the checkpoint record the DataConnector operator creates:

```
1K + file_count * 580 bytes
```

## Determining Semaphore Usage per Job

Semaphores are used for synchronizing processes and access to resources in a parallel execution environment. For example, when a data stream is used to transfer data buffers from the producer instance (one process) to the consumer instance (another process), semaphores are used to synchronize the access to the data stream that the producer and consumer instances share. If more instances are used in a job, more semaphores are needed.

Use the following formula to calculate the required semaphores for a job with multiple producer and consumer instances is:

```
Nprocs = MAX( 25, Consumer_count + Producer_count + 2)
Semaphores = 2 * (Nprocs + 3) + 5
```

where:

*Nprocs* are the number of job processes, including the processes that the Teradata PT infrastructure uses.

## Determining the Size of Shared Memory for Buffer Mode Loading

Buffer Mode in Teradata PT is a loading mechanisms for transferring data buffers directly from the producer operator to the consumer operator without using the CPU-intensive row-by-row processing in the Teradata PT infrastructure and, in this way, increasing throughput performance.

For a producer or consumer job to be eligible for Buffer Mode, the job script cannot contain filtering criteria such as CASE/WHEN or WHERE clauses in the Teradata PT SELECT statement.

**Note:** Not all operators support Buffer Mode. Currently, the Export, Select, ODBC, and DataConnector producer operators and the Load and DataConnector consumer operators support Buffer Mode. LOB importing and exporting are not Buffer-Mode eligible.

The following are the typical operations that are Buffer-Mode eligible:

- Exporting rows from a Teradata Database table and:
  - Writing them to files
  - Loading them into another Teradata Database table
- Extracting rows:
  - From files and loading them into a Teradata Database table
  - From an ODBC source table and loading them into a Teradata Database table
  - Through INMOD and access modules and loading them into a Teradata Database table

Buffer Mode also allows blocking of multiple buffers into a single data stream message so as to minimize buffer transfers in data streams. The main challenge with blocked Buffer Mode is determining a *blocking factor,* that is, the number of buffers in a message. The blocking factor is determined based on the following formula:

```
Buffers/Block = (MemoryPercent * TotalSharedMemory) / ((ProducerCount +
(QueueDepth * ProducerCount + 1) * ConsumerCount) * BufferSize).
```

Where MemoryPercent is the percentage of shared memory to be dedicated to data stream messages and QueueDepth is the maximum number of messages that can be placed on a data stream. The consumer operator sets the BufferSize dynamically.

Teradata PT provides the default setting of the *blocking factor*, but it may not be optimal because it only takes the default values for the MemoryPercent (80), TotalSharedMemory (10M), and the QueueDepth (2) when deciding the blocking factor. If you want to use a larger blocking factor to minimize the number of buffers being transferred through data streams, you need to increase the shared memory at the job level using the tbuild -h option.

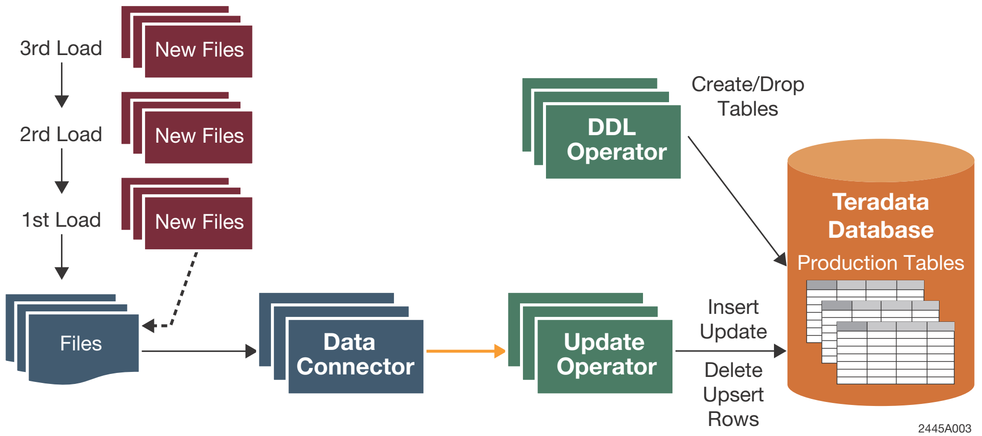
## Using Teradata PT Periodic Loading for Active Data Warehousing

Unlike the *batch file* processing, where the number of files is predefined and each file usually contains a very large number of rows, files that represent transactions in the Active Data Warehousing (ADW) environment are “dynamic” relatively small in size (a few hundred rows per file in average). *Dynamic* means that files can be created, processed, and removed from the directory while the loading job is running. Since dynamic files represent real-time transactional data flow, they are usually created in short-time duration, made available for updates once they are created, processed in or close to time-sequence order, and committed to the data warehouse in a timely manner as quickly as possible.

### Periodic File Collection and Loading

As shown in Figure 48, transactional files can be collected periodically and processed by the DataConnector operator before they are loaded into Teradata tables using the Update operator.

Figure 48: Periodic Loading with Directory Scan




Both active and batch directory scan can be used for periodic loading.

For active directory scan, there are multiple scans (based on the VigilWaitTime value) of the directory for new files; the job does not terminate until the VigilElapsedTime or VigilStopTime expires.

For batch directory scan (which does not require that the VigilWaitTime and VigilElapsedTIme attributes be set), there is only one scan of the directory for files; the job terminates once all the files collected by that scan are processed.

For more information on active and batch directory scan attributes, see Chapter 5: "Moving External Data into Teradata Database."

## Switching the Load Protocol for Periodic Loading

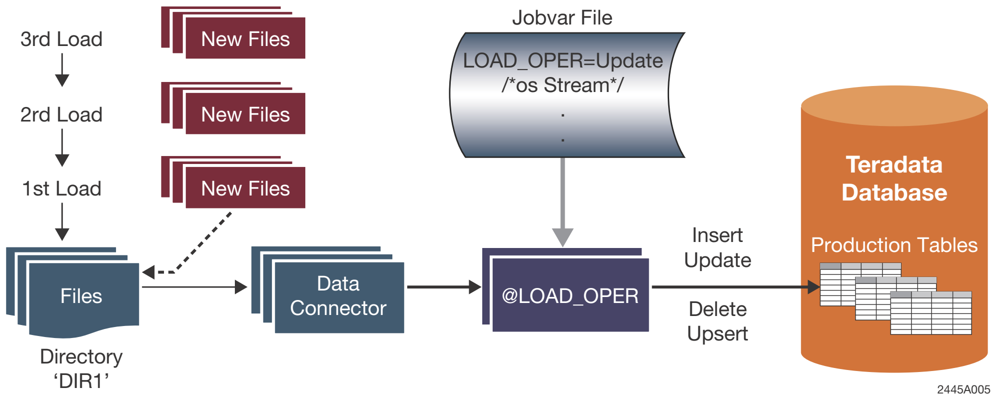
One of the most distinguishable advantages of using Teradata PT for active or periodic loading is that you can switch the load protocol and job parameters without modifying the job script itself. However, not all job scripts allow you to switch the load (or export) protocol because the features supported by one operator may not be applicable to the others. In other words, you need to take into consideration that the Teradata PT SELECT and APPLY operations being used in the job are applicable to the operators that are to be switched.

Switching the load protocol is desirable for the following reasons:

- The current load protocol cannot process the current volume of transactional files fast enough
- The number of concurrent load jobs that require "load slots" has reached a limit the Teradata Database imposes.
- The system cannot sustain the current usage of system resources with the current load strategy.
- "Catch up" is required after a system failure or a sudden increase in the volume of transactions.

As shown in Figure 49, if you want to change the load protocol from Stream to Update, or vice versa, you can define the operator type with a job variable name that starts with the @ sign (for example, @LOAD). For more details about how to use job variables, see "Setting Up the Job Variables Files" on page 68.

Figure 49: Switching Operator using a Job Variable



## Using the ELT Approach for Loading

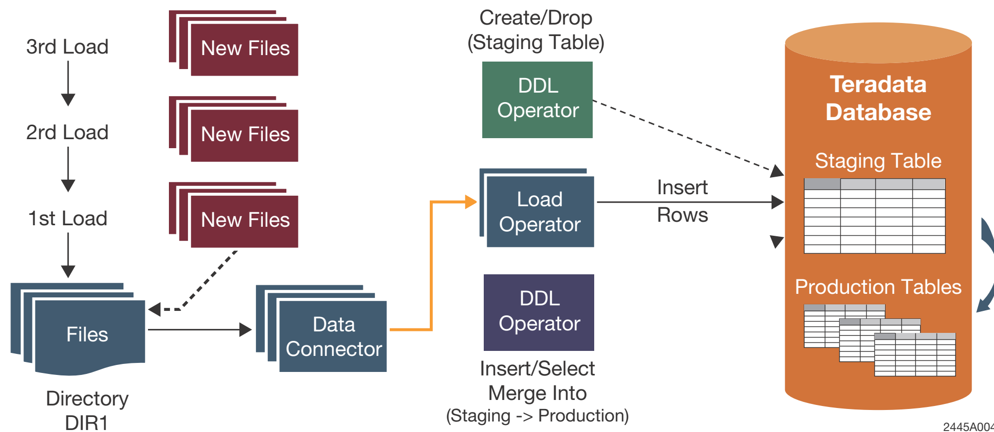
While most 3rd party data warehousing products usually provide a wide variety of data transformation tools for cleansing and filtering data before loading, Teradata PT's primary focus is fast data loading data into Teradata tables where data can be further processed using the power of SQL within Teradata. This feature of the Teradata Database gives rise to a new loading approach called Extract, Load, and Transform (ELT).

Aside from loading with scalable performance, there are additional advantages to using the ELT approach. For example, the Load and Update operators do not support the loading of target tables with USI (Unique Secondary Index), JI (Join Index), RI (Referential Integrity) or Triggers. With the ELT approach, however, you can avoid the above restrictions by first loading data into a staging table and then using such SQL statements as INSERT-SELECT or MERGE-INTO to move the data from the staging table to the target table.

To implement ELT, you can use the Teradata PT "job step" feature to encapsulate the loading step, the SQL INSERT-SELECT step, and the cleanup step into a single job, as shown in Figure 50.

Each step within a job is restartable, which means that whenever a step fails for any reason, the resubmission of the job would cause it to resume execution at the failed step and continue. The ELT approach also offers the flexibility of performing a number of different operations on the source rows, such as data cleansing, transformation, integrity checks, and so on, via different steps within the job, before inserting data into the target table.

Figure 50: Extracting, Loading, and Transforming (ELT)




## Managing and Monitoring Teradata PT Jobs

### Using the External Command Interface to Monitor Jobs

The Teradata PT Asynchronous Command Interface is a software component, through which you can issue commands to Teradata PT jobs at runtime. The term *asynchronous* implies two important things:

1. That you can issue commands to Teradata PT jobs from outside the Teradata PT address space; thus the term *external commands*.
2. That Teradata PT processes commands in an asynchronous manner, while it is in the middle of performing loading or exporting operations.

The purpose of the external commands you issue include, but are not limited to, the following:

- Suspending and resuming a job to allow for better management of load time and system resources
- Taking a checkpoint on a demand or timely basis
- Terminating a job in a graceful manner for later restart
- Obtaining the status of a job on a demand basis
- Committing transactions that are in-flight by taking an immediate checkpoint
- Defining external procedures or rules for driving external commands
- Synchronizing multiple data targets that can be loaded concurrently
- Collecting operational metadata such as performance statistics, source or target information, job events, and so on, which can be stored in relational tables

External commands can be targeted to different Teradata PT execution levels, that is, the job level and the operator level. The different level-of-command execution gives you more flexibility in defining monitoring and modification procedures for controlling job execution, regulating system resource usage, and refining job performance.

### Using the TWB_STATUS Private Log to Obtain Job Status

When a job runs, Teradata PT creates a special log called the TWB_STATUS log to capture job performance data at different stages of the job. These stages, known as "processing methods," include:

- 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. You can view the performance data in the TWB_STATUS log 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 job performance in terms of throughput and the cost of performing the exporting and loading of data by each operator. The information is also useful for capacity planning when performance data, collected for a period of time, summarizes the CPU utilization and elapsed time for each job, so you can determine performance trends for the overall loading and exporting processes.

## Writing Load Scripts for Restartability and Availability

One of the main challenges for data warehousing design is how to recover from a failure as quickly as possible. Recovery usually involves fixing the client or server systems, changing configuration parameters or system resources, restarting the interrupted jobs based on their last checkpoints, and bringing the system back to normal without resorting to rigorous manual efforts or writing piece-meal recovery procedures.

Most of the time, jobs may also be required to perform "catch up" so that transactions that were accumulated during the "failure window" can be applied to the target systems as quickly as possible.

To this end, Teradata PT provides some unique features that allow you to speed up the recovery process without resorting to changing job scripts after a job failure. These features include:

- Making all jobs checkpoint restartable by default.
- Archiving transactional data in a readily-loaded format concurrently with the loading of such transactions into target tables using the Duplicate APPLY feature, which allows the same data to go into different targets.
- Defining a single script language for all operators, which not only results in common approaches for defining operators, but also allows substantial reusability of metadata and operators.
- Supporting unlimited variable substitution using a job variables file so that changeable and common job parameters, called “attributes,” can be isolated in a single place for value assignments.
- Having complete independence between the producer operator (for data extraction) and the consumer operator (for data loading) in a job substantially simplifies the process of "switching export/load protocols". In other words, changing either the producer operator or the consumer operator in a job would not impact the other.

To take advantage of the above features for restartabilty, some best practices for designing and implementing job scripts are necessary. The best practices presented below speak to reusability and manageability of job scripts, the flexibility of building and enhancing them to deal with ever increasing data volumes and changes in execution environments, and restartability after job failures. These practices can also be regarded as standard guidelines in building data warehousing processes.

- Always use a job name to execute a job.
- Use job variable files to capture changeable and common parameters such as user ID, password, file names, source or archive directory names, the number of producer and consumer instances, and so on.
- Run with backup or archive using the Duplicate APPLY feature so that each APPLY statement can send the same data to a different target.
- Define checkpoint frequency to control load granularity in case of failure. The smaller the frequency, the less time to recover a job, but more time to take checkpoints.
- Switch the load protocol (for example, Stream to Update) for purposes of catch up after a system failure.
- Always execute a job with the job variables file so that parameters are defined in one place instead of being distributed across job scripts.

## Restarting a Job from a Job Failure

### Automatic Restart

An *automatic restart* means that a job can restart on its own without manual resubmission. With the default start-of-data and end-of-data checkpoints, a job can automatically restart itself when a retryable error occurs, such as a database restart or deadlock before, during, or after data loading. Consider the following when dealing with automatic restarts:

- Jobs can automatically restart as many times as is specified by the value of the RETRY option of the Teradata PT job launching command (the **-r** option). By default, a job can restart up to five times.
- If no checkpoint interval is specified for a job, and the job fails during processing, the job restarts either at the start-of-data checkpoint or the end-of-data checkpoint, depending on which one is the last recorded checkpoint in the checkpoint file.
- To avoid reloading data from the beginning, especially for a long running job, specify a checkpoint interval when launching a job so the restart can be done based on the most recent checkpoint taken.

### Manual Restart

If a job fails and terminates, you can manually restart it by resubmitting the same job with the original job-launching command. By default, all Teradata PT jobs are checkpoint-restartable using one of the two checkpoints taken before data loading and after data loading. When jobs have multiple steps, a checkpoint is created for each successful step, allowing a job to restart from the failed step.

**Restarting a Job "Catch Up"**

Here are the steps for switching the load protocol to perform "catch up":

- Terminate the current job with the TERMINATE command. This forces the job to take a checkpoint before it terminates.
- Switch the load protocol by either changing the operator in the job variables file or by using another job variables file that has the new operator. The latter method is highly recommended because it prevents users from modifying existing job variables files.
- Resubmit the same job with the same command options.

  Note: Do not cleanup the Teradata PT checkpoint files left from the previous run.

The above 3 steps can be easily automated because performing "catch up" is very similar to restarting a job. In most of the "catch-up" cases, you do not need to modify the original scripts. This is all due to the advantages of having a single script language, external job variables to isolate changes to one place, and a common protocol for checkpoint restart across operators.

APPENDIX A

# IBM z/OS Samples Files

This appendix provides a brief description of the sample files distributed with Teradata PT for the IBM 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 dataset. For the dataset name, see the *Teradata Tools and Utilities for IBM z/OS Installation Guide* and your software installer.

The sample files include:

- Job Script Examples
- JCL Samples
- Job Attribute File
- Teradata PT Catalogued Procedure (PT#TPT)
- Teradata PTLV Catalogued Procedure (PT#TPTLV)

For Teradata PT API samples, see *Teradata Parallel Transporter Application Programming Interface Programmer Guide*

## Job Script Examples

The job script examples demonstrate some of the basic Teradata PT functions. Each script is fully documented and describes the Teradata PT features used.

Table 19: Job Script Examples

| Job Script | Description |
|---|---|
| PTS00001 | Exports rows from one Teradata Database table and loads them into an empty Teradata Database table. |
| PTS00002 | Exports rows from one Teradata Database table to a z/OS dataset. |
| PTS00003 | Loads rows into an empty Teradata Database table from a z/OS dataset. |
| PTS00004 | Loads rows into an empty Teradata Database table from 2 z/OS datasets. |
| PTS00005 | Loads rows into 2 empty Teradata Database tables from a z/OS dataset. |
| PTS00006 | Updates a Teradata Database table from a z/OS dataset. |

Table 19: Job Script Examples (continued)

| Job Script | Description |
|---|---|
| PTS00007 | Loads rows into an empty Teradata Database table via the DDL operator. |
| PTS00008 | Exports rows from a Teradata Database table to a USS file and copies the file via the OS Command operator. |

# JCL Samples

A sample JCL is included with the SAMPLIB dataset to execute the job script examples. It is member PT$TPTAL. It contains the Teradata PT cataloged procedure inline and has a jobstep for each script example.

The z/OS datasets are unconditionally deleted and reallocated for this sample job. This allows repeated execution without any user intervention. User intervention 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 of the job.

- The JOB statement must be modified to conform to your installation's requirements.
- The TTUPREF SET statement must be changed to the installation dataset PREFIX.

Another sample JCL, member PT$TPT01 (identical to PT$TPTAL), executes one sample script.

# Job Attribute File

The above JCL references PT@JBVAR, the job variable file that contains the variables used in the script examples.

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 of the JCL.

- A Teradata TDP must be specified.
- A valid user name for the Teradata TDP must be specified.
- The password associated with the user name for the Teradata TDP must be specified.

**Note:** The FileName attribute format used to specify the z/OS dataset for each script can be changed.

# Teradata PT Catalogued Procedure (PT#TPT)

A sample Teradata PT catalogued procedure has been included. There are 2 jobsteps.

- The first jobstep, ALLOC, allocates the checkpoint datasets if necessary.

  If they exist, the Teradata PT job uses the contained information in restart mode. ALLOC always deletes the log dataset, if it exists, and allocates a new log for the current job.

- The second jobstep, TPT, executes Teradata PT.

  Use symbolic parameters to specify the script and job attribute files, enter any Teradata PT parameters, and create a unique high-level qualifier for the checkpoint and log datasets.

  See PT$TPTAL for an example of how symbolic parameters are set.

# Teradata PTLV Catalogued Procedure (PT#TPTLV)

A sample TPTLV catalogued procedure included in the SAMPLIB dataset executes **tlogview**. Use the JCL variables to specify which logs and information are to be displayed.

Note: The HLQUAL symbolic parameter is also used to designate the specific job log in a similar manner to the Teradata PT cataloged procedure.

APPENDIX B
# Teradata PT Wizard

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.

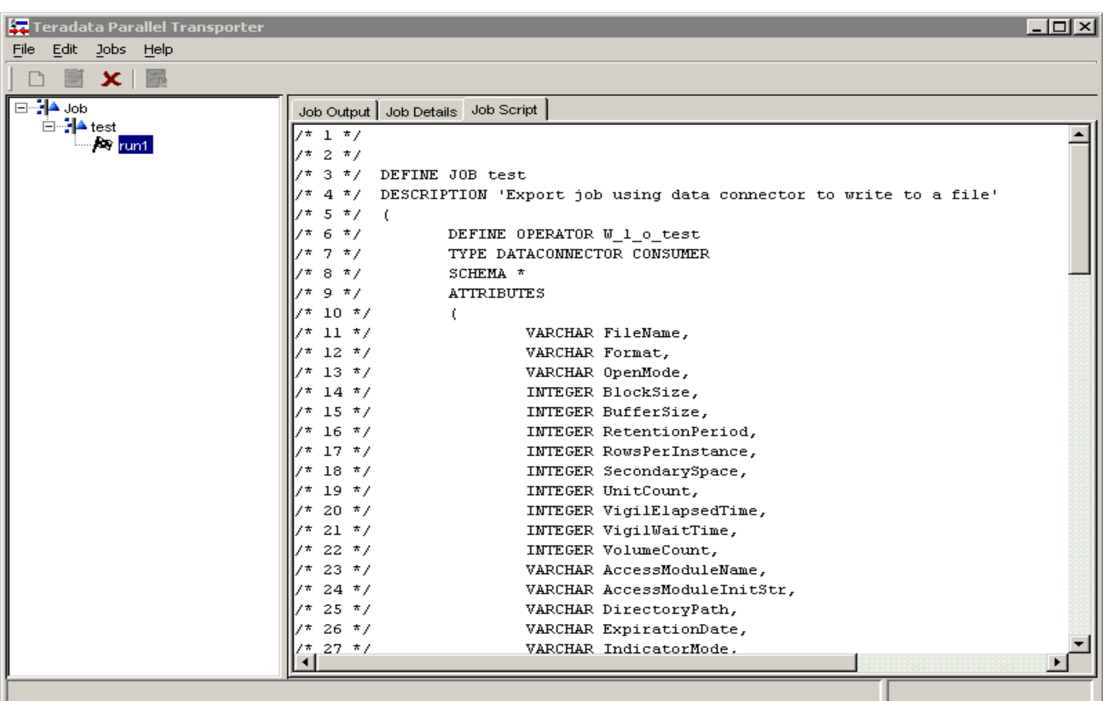
7 Run the job.

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

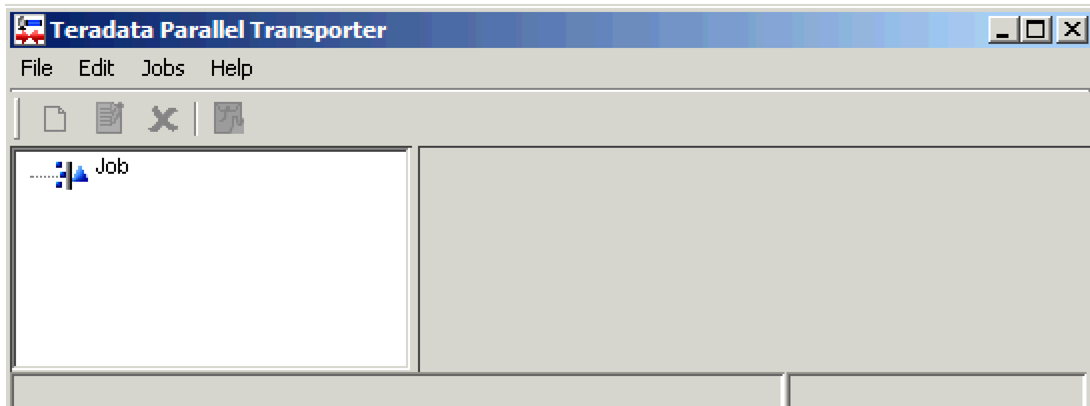
The Wizard has the following limitations:

- 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.

- 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.

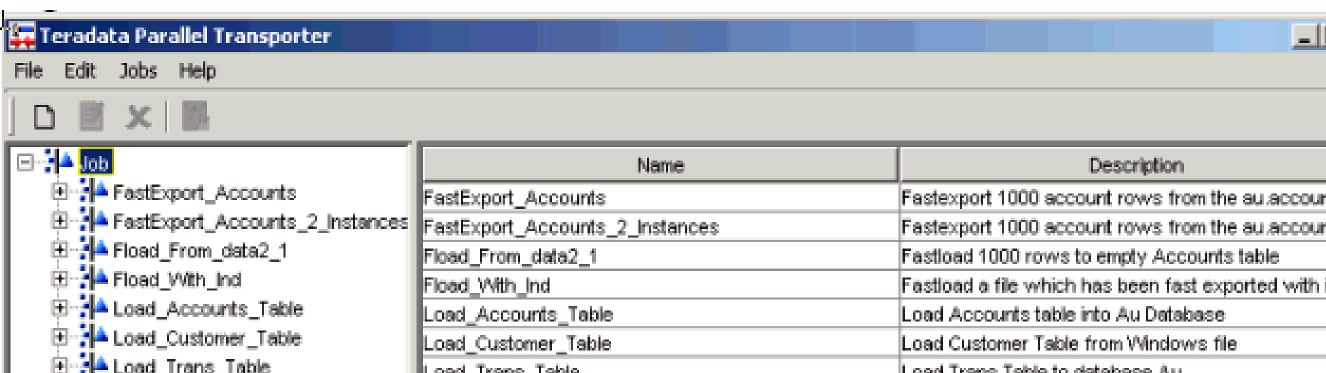
# Main Window

The window of Teradata PT Wizard consists of two panes.




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.

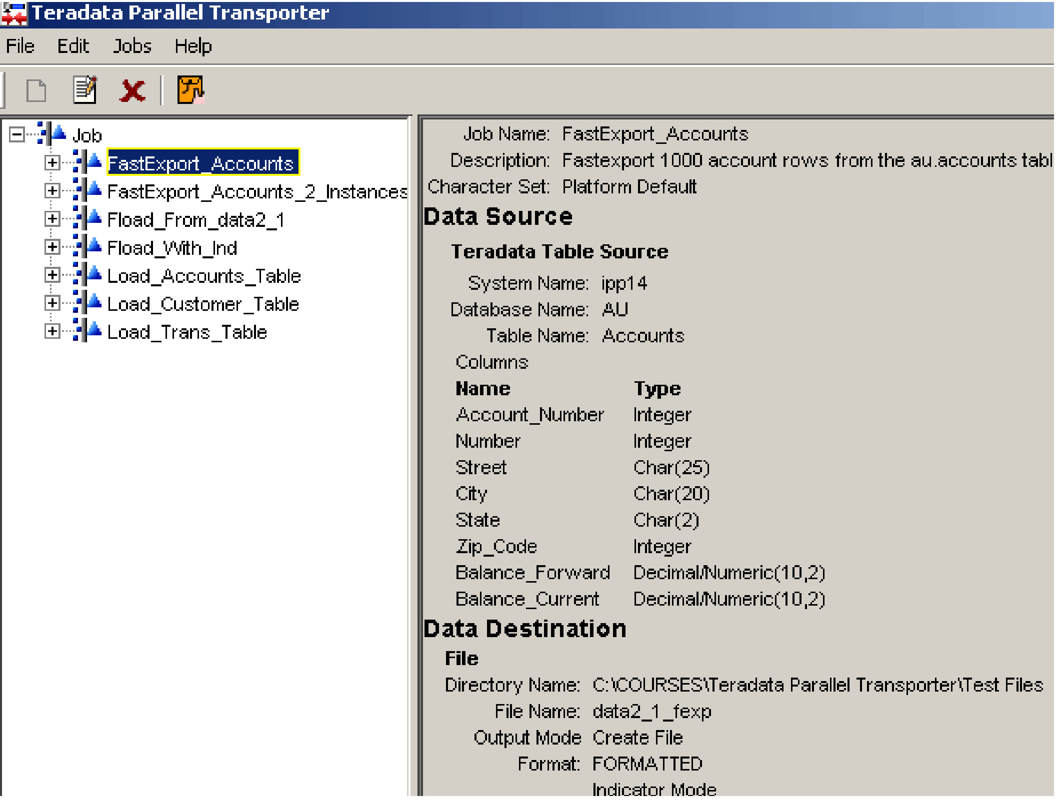
As the session progresses and jobs are run, a history of job instances is built in the job tree in the left pane.



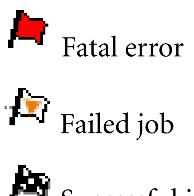

The right pane displays the name, description, and status of all jobs (or job nodes) that have been run.

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.

- Right-click a run instance to open a menu with options to restart the instance, delete the instance (not the job itself) or to view the job log.

# 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

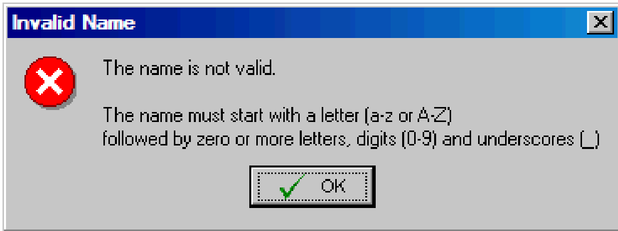
### To name a new job

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.

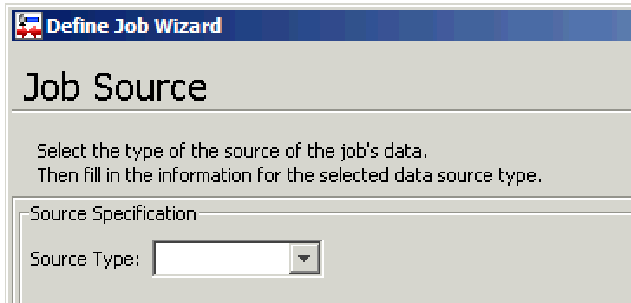
Teradata PT allows all character sets as long as they are supported by the Teradata Database. The default is the character set of the active platform; however, scripts and log output are in English only.

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 Reference* and *International Character Set Support*.

4 Click **Next** to open the **Job Source** dialog box.




5 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.

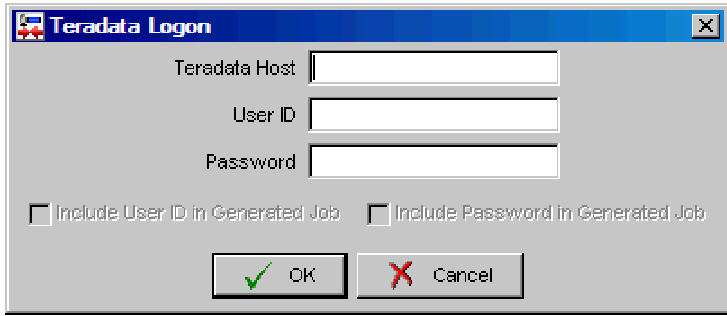
When running existing scripts, if the logon information has not been included in a script that has been submitted to run, information can be entered in the JobAttributes panel in the Run dialog box, as shown under step 4 on page 280.

### 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

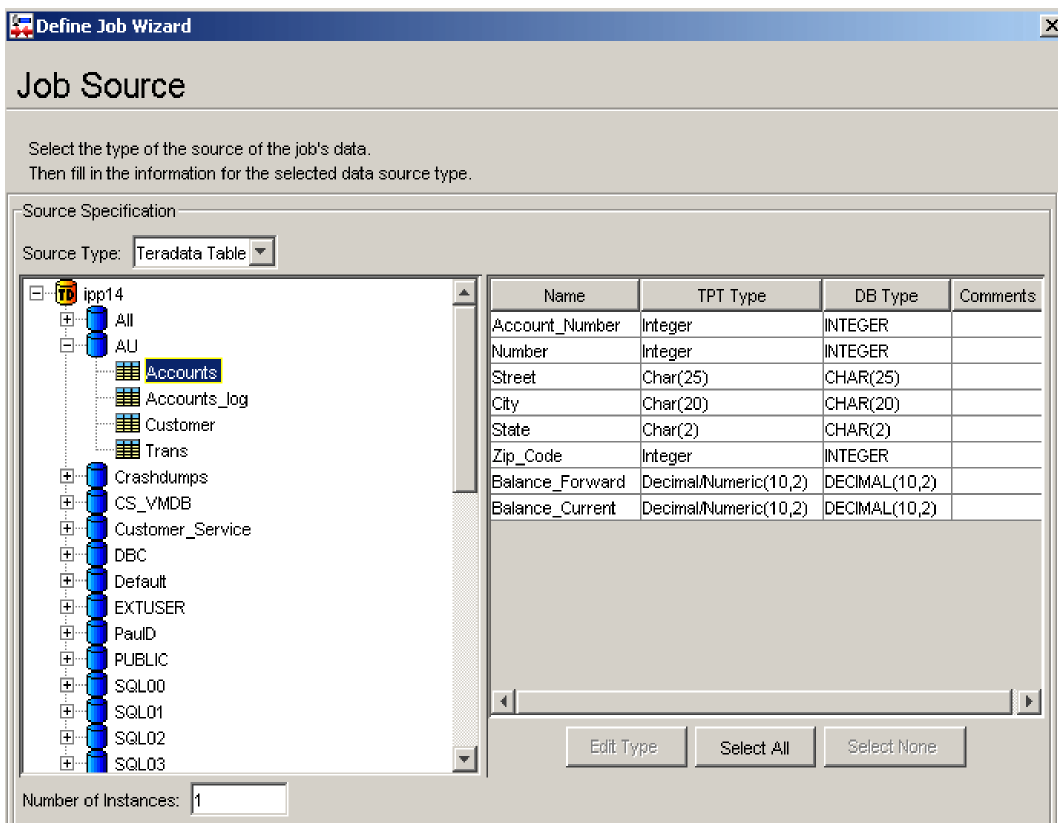
1 From the **Source Type** list in the **Job Source** dialog box, select **Teradata Table**.

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.




5 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.

6 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.

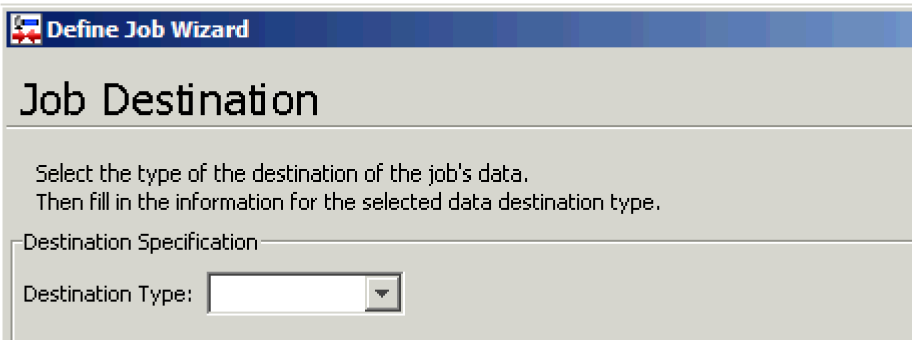
For example, if the keyword DESCRIPTION is used as a column name, the name is changed to DESCRIPTION_1. Teradata PT keeps an internal counter for generating the appended number.

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.




8 Continue with Step 3 - Select a Destination.

## File as a Data Source

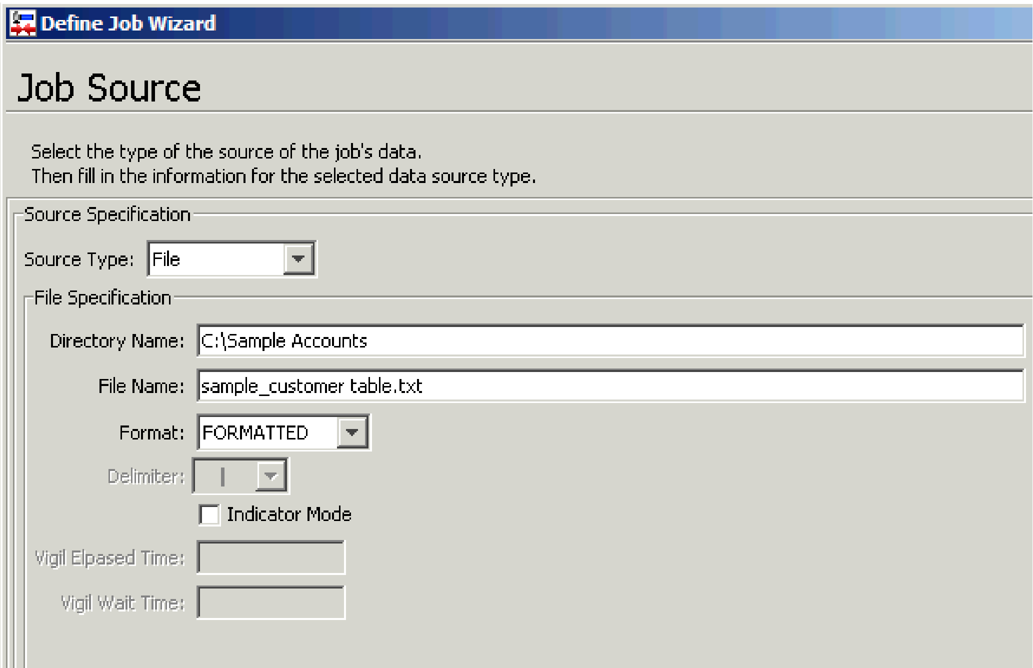
Use the **File** option from the **Job Source** dialog box to browse for a flat file to use as the data source for a job.

### 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”.

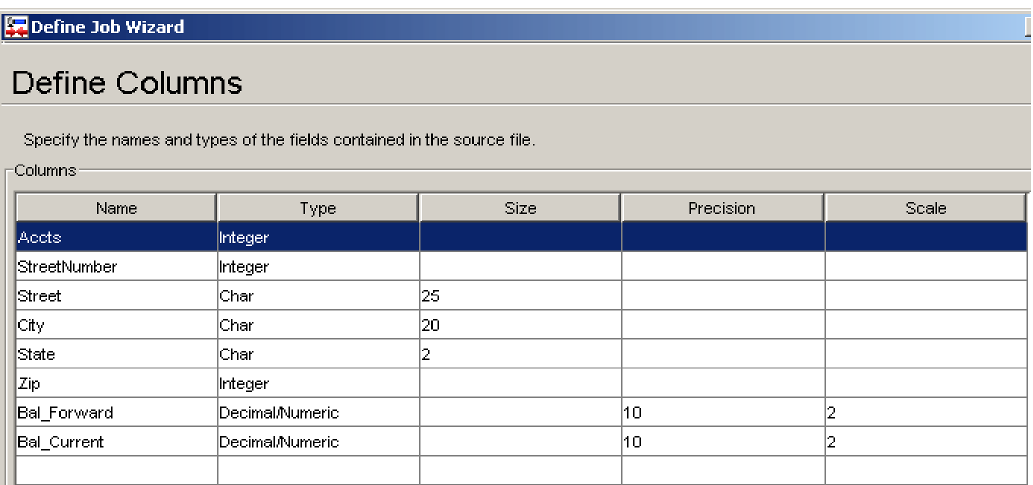
If specifying Delimited format, type the delimiter used in the source file into the **Delimiter** box. The wizard accepts delimiters up to 100 bytes in length. If no delimiter is provided, the TextDelimiter attribute defaults to the pipe character ( | ).

**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.




6 In the **Define Columns** dialog box, specify the following, as needed:

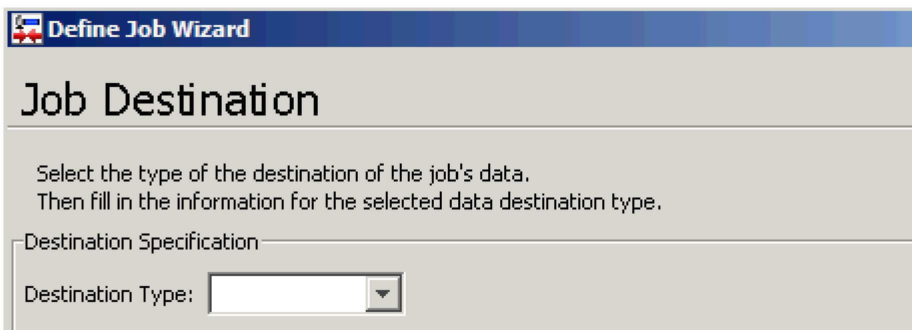
- **Name** - Type the names of the columns in the source file.
- **Type** - Type the data type of each column. (Choices change depending on the type of format selected in the previous dialog box.)

  **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.)

7 (Optional) In **Number of Instances**, type the number of producer operator instances to process at the same time.

8 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.

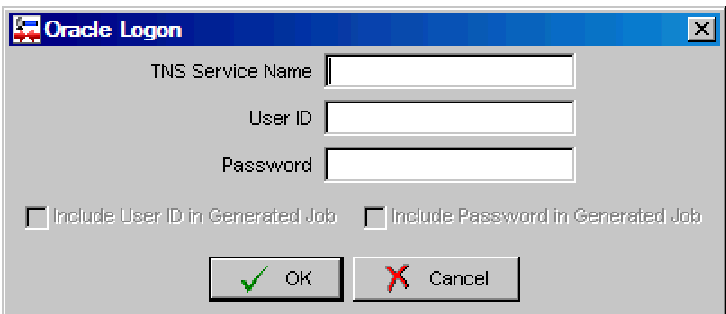
9 After defining all the columns, click **Next** to open the **Job Destination** dialog box.




10 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.

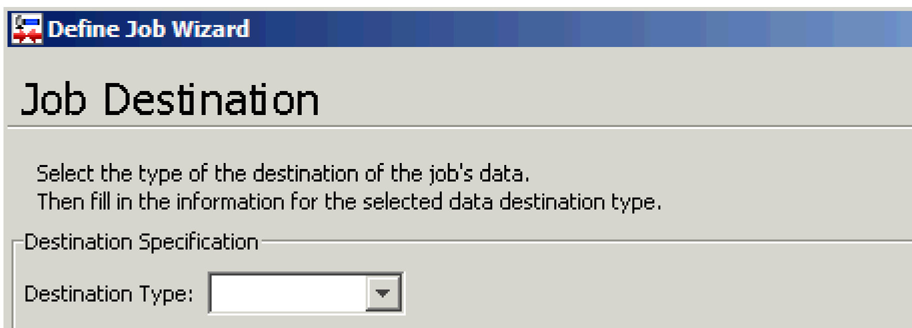
6 In the right pane, select up to 450 columns to be included in the source schema, or click **Select All** or **Select None.**

**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.

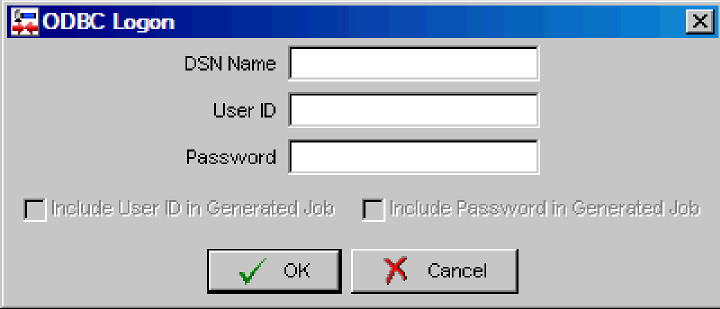



8 Continue with Step 3 - Select a Destination.

## 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

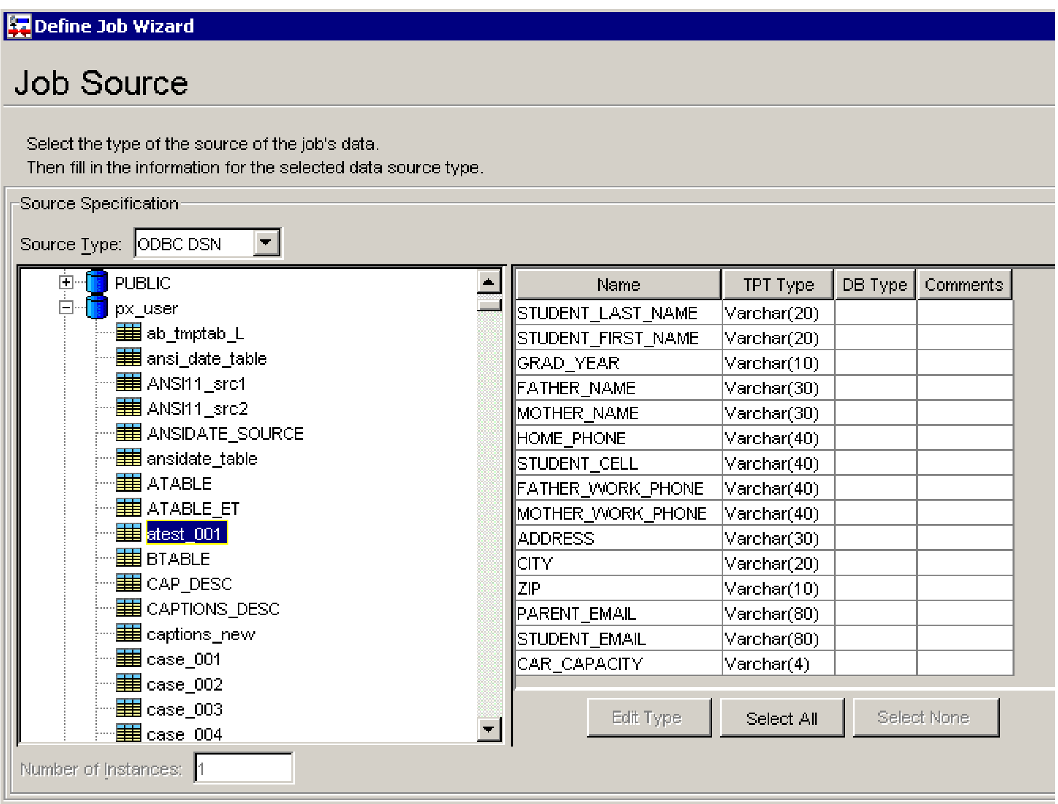
1 From the **Source Type** list in the **Job Source** dialog box, select **ODBC DSN.**

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.

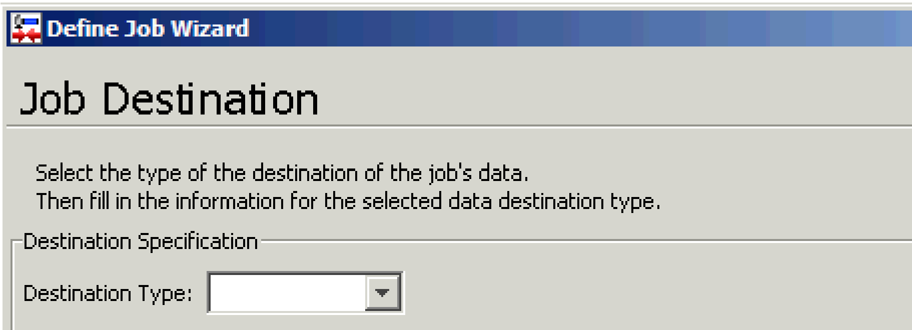
**Caution:** Do not select tables that contain character large object (CLOB) or binary large object (BLOB) data types.

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.




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:

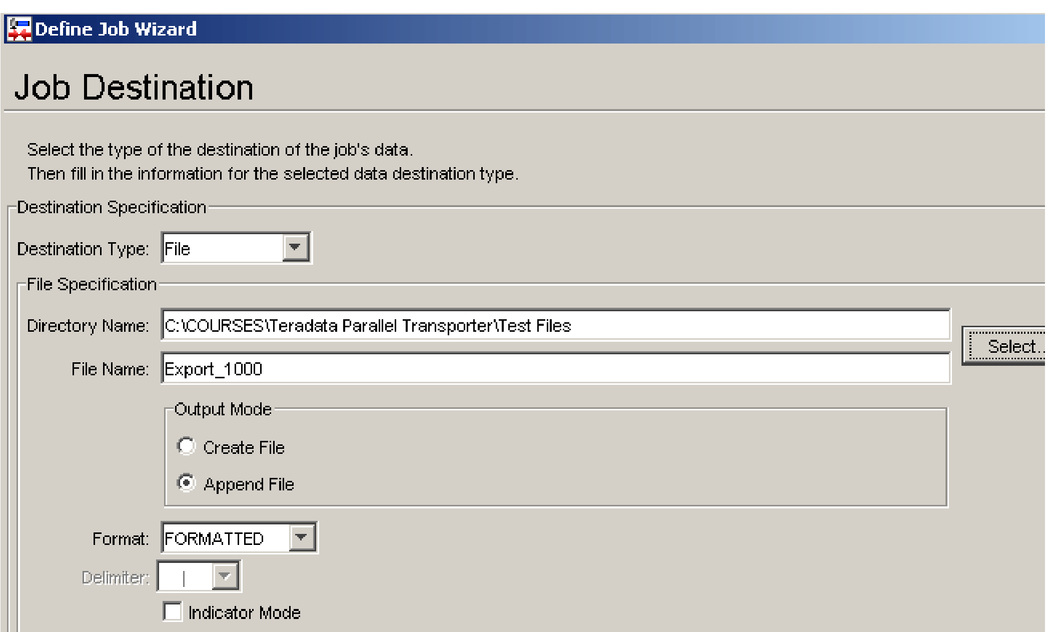
- File as a Target
- Teradata Table as a Target

### 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**.



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.

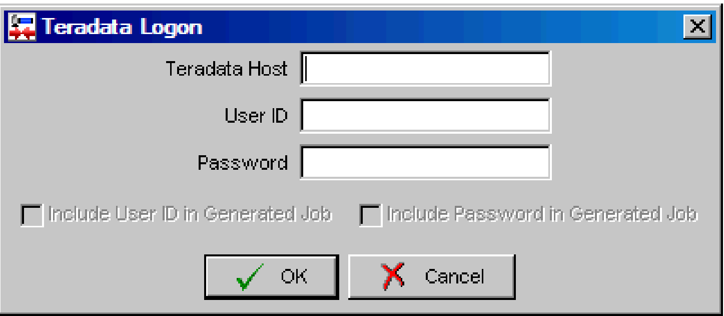
## Teradata Table as a Target

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**.

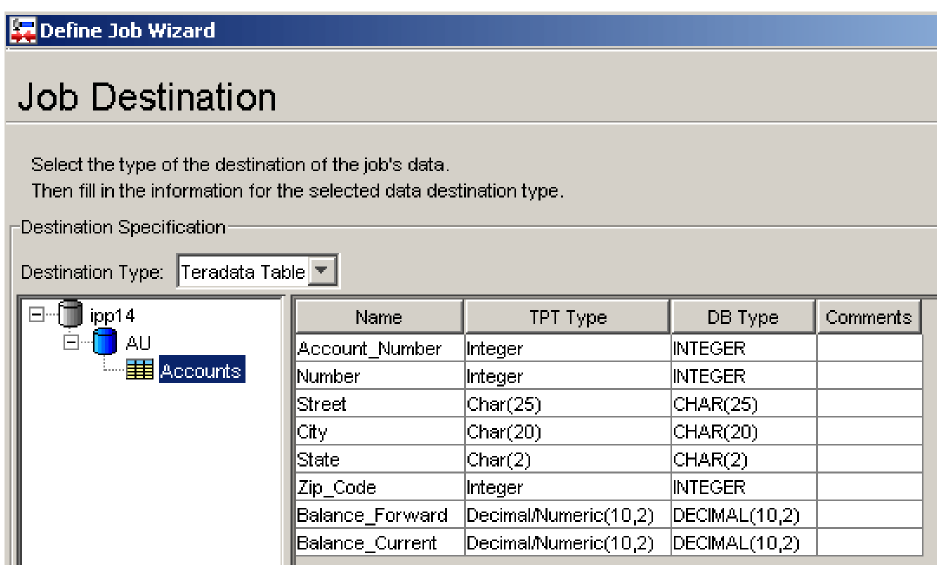
2 In the **Teradata Logon** dialog box, type the host name, user ID, and password to log onto the target Teradata system.




3 (Optional) Select the check boxes to include your user ID and password in the generated scripts. The default is to enter placeholders.

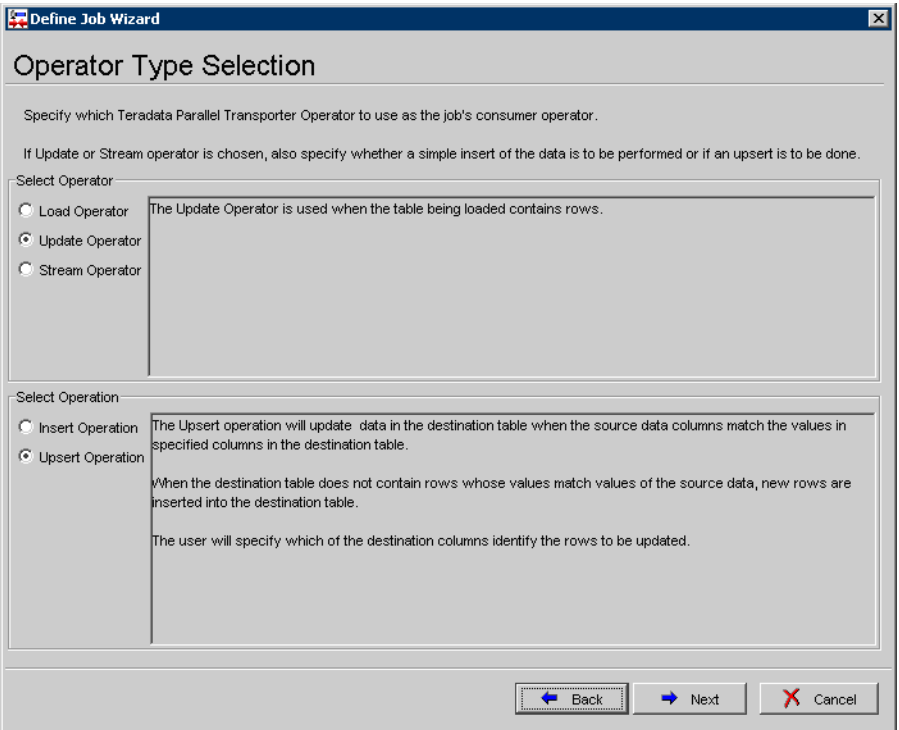
4 Click **OK** to close the log on prompt and return to the **Job Destination** dialog box. For more information, see "Logging onto a Data Source" on page 263.

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.



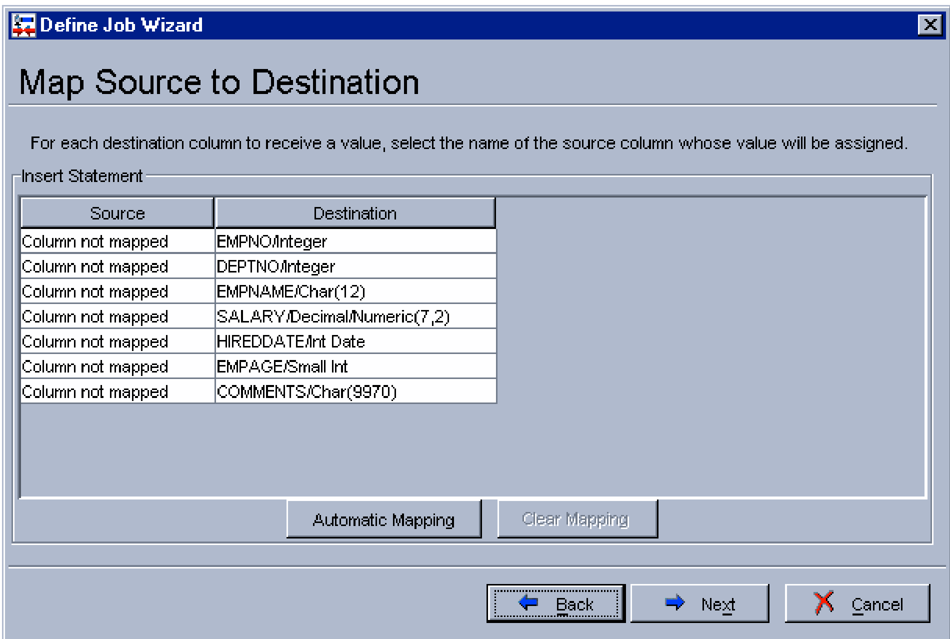

7 Select one of the following options depending on what Teradata PT operator or operation is to be used for the job. For more information about operators, see *Teradata Parallel Transporter Reference*.

- **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.

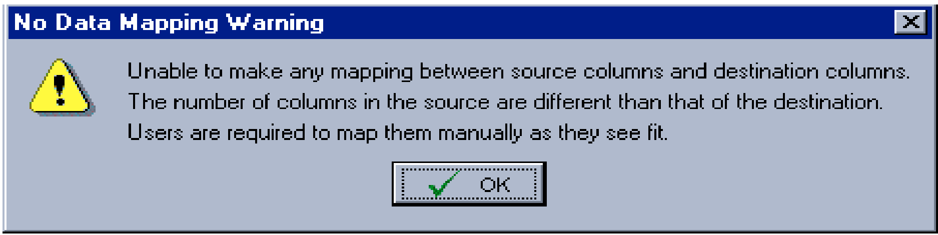
**Note:** At least one column must be selected, and at least one column must remain cleared.

8 Click **Next** to open the **Map Source to Destination** dialog box.



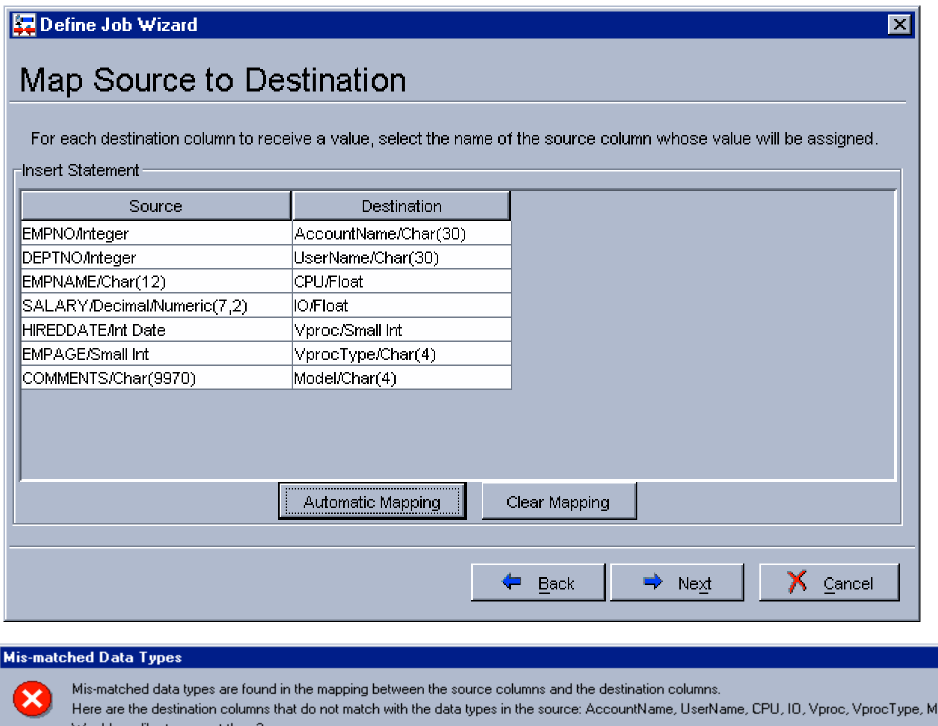

9 Click the **Automatic Mapping** button to map the first source column to the first destination column, the second source column to the second destination column, and so on.

- 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.




Mis-matched Data Types

Mis-matched data types are found in the mapping between the source columns and the destination columns.
Here are the destination columns that do not match with the data types in the source: AccountName, UserName, CPU, IO, Vproc, VprocType, Model
Would you like to correct them?

Yes No Help

- If you click **YES**, Teradata PT returns to the **Map Source to Destination** screen so you can map source to destination columns manually.

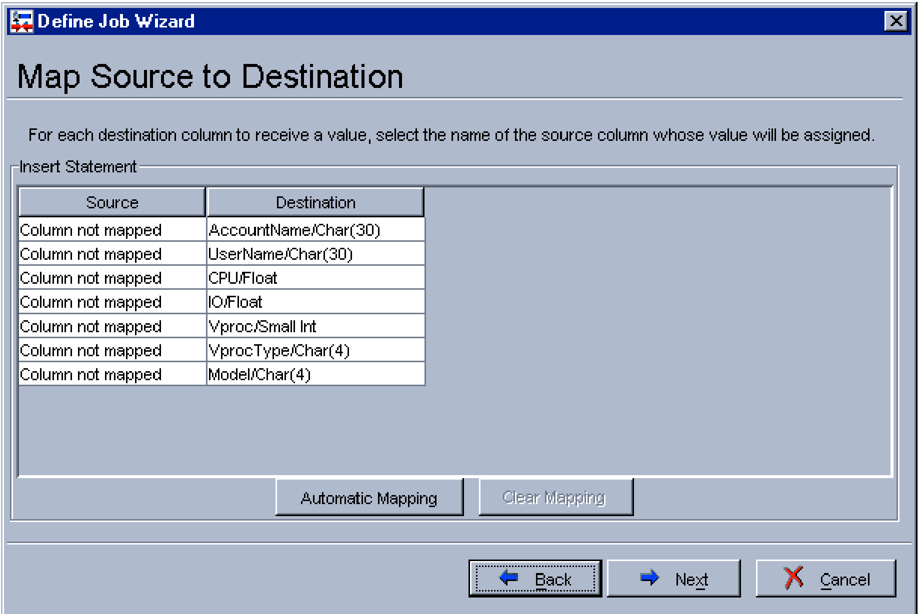
  To map a source to a destination column manually, click a row in **Source** to open its dropdown list. Then select a data value for that source column row to map to a destination column. One source column value can be mapped to multiple destination columns. Moreover, source columns can be left as **Column 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.

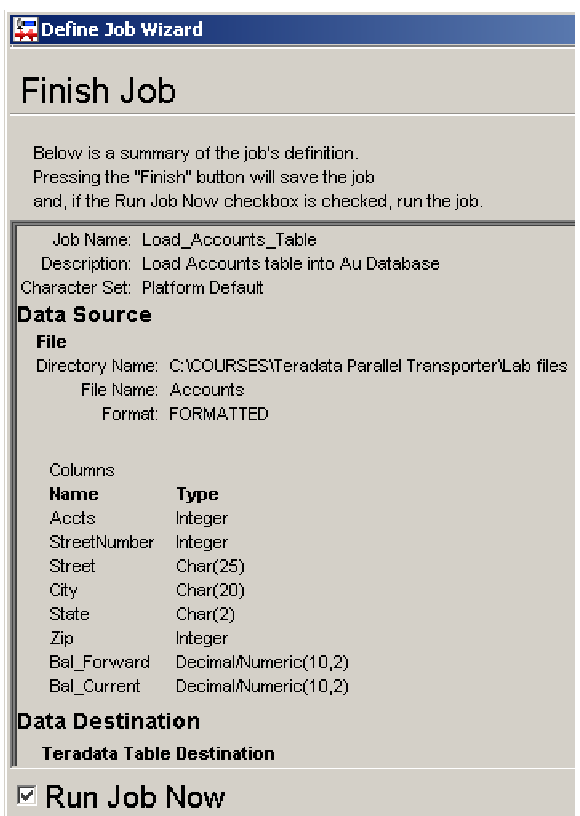
There are cases when the data types of the source and destination columns do not match, but the database can implicitly convert source column data types to destination column data types. For example, when the source column data type is smallint and the destination column data type is integer, the database 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 dropdown 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:

- To run a new job, skip to step 4.
- To run a previously created job, continue with step 2.
- 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:

- To correct mistakes, click **Back** to return to the appropriate dialog box and make corrections.
- To store the job to be run or edited later, clear the **Run Job Now** option, and click **Finish**.
- 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**.

Run Job

Job Name:

Checkpoint Interval:

Retry Limit:

Latency Interval:

Enable Trace

Teradata Parallel Transporter Script

```
DEFINE JOB db2_tstl
DESCRIPTION 'Using DataDirect ODBC driver to retrieve data from a DB2 table w
on mainframe.'
(
        DEFINE OPERATOR W_1_o_db2_tstl
        TYPE DATACONNECTOR CONSUMER
        SCHEMA *
        ATTRIBUTES
```

4 In **Job Name**, type the name of the job.

5 (Optional) In **Checkpoint Interval**, type the number of seconds between checkpoint intervals.

6 (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 -R** option.

7 (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*.

8 (Optional) Select **Enable Trace** to enable the trace mechanisms.

9 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 273 and "Oracle Table as a Data Source" on page 269.)

10 (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.

11 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 283.

# 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 `Crtl+R`.

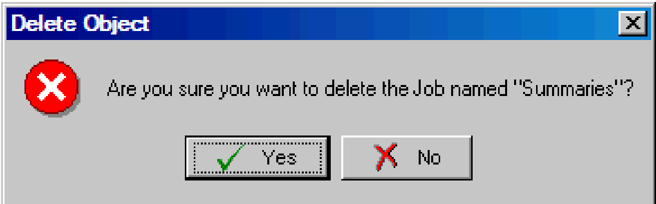
   The job begins from the point at which the job was stopped. Also see "To stop a running job" on page 281.

### 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 263.

## 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`.

2 Start with step 2 of “Step 4 - Run the Job” on page 279.

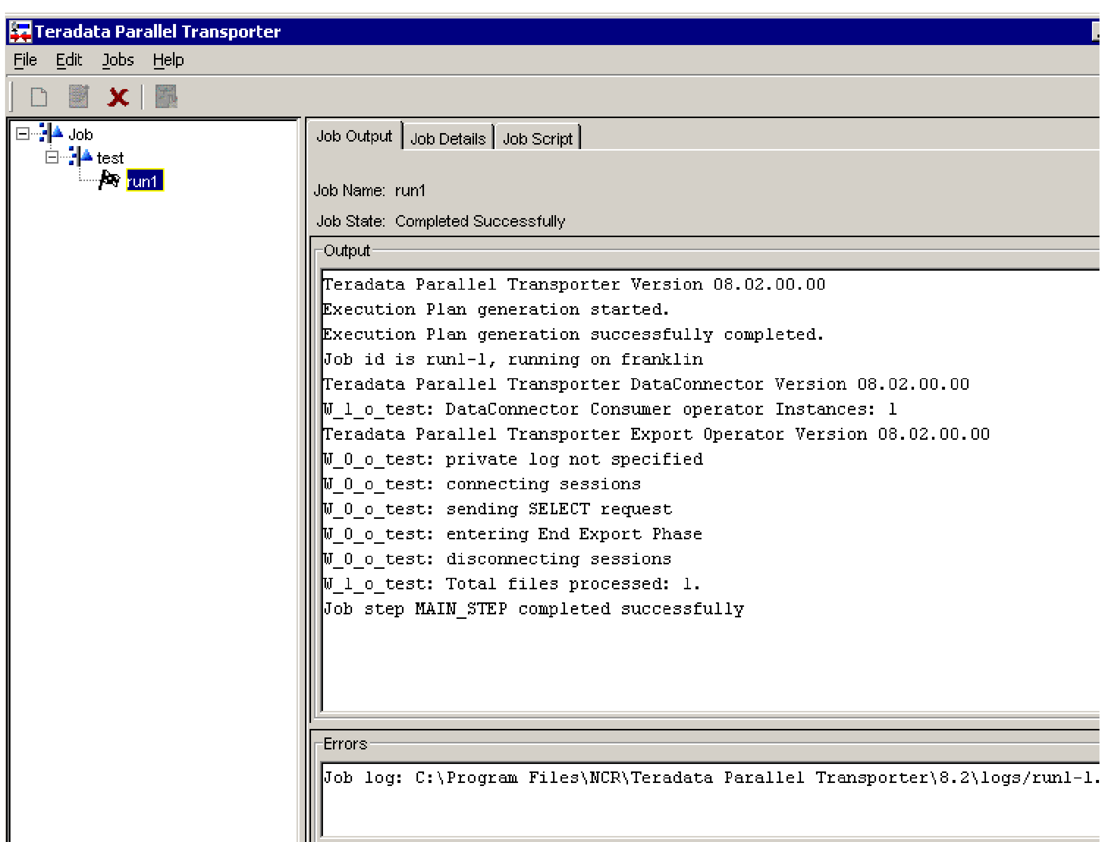
# View Job Output

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 284 for more information.

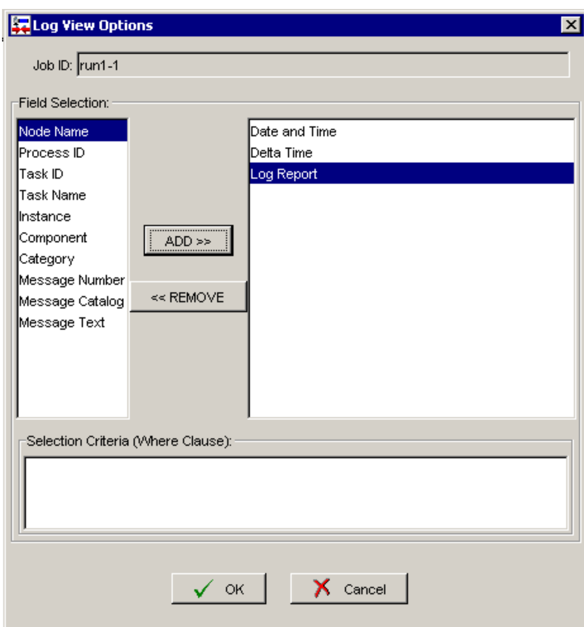
- **Job Details** - Shows a table of job-related details. Columns include **Step Name**, **Task Name**, **Task Status**, and **Operator Name.** Use this tab to view a job as it runs. Each step is listed with its progress.
- **Job Script** - Shows the actual Teradata PT job script created by the Wizard for the specific job instance. The script can be copied into other scripts.

## Log Viewer

Teradata PT keeps an extensive log of each jobs 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 20: Menu Items

| Menu | Menu choice | Description |
|---|---|---|
| File | Exit | Closes the wizard. |
| Edit | New | Creates a new job. See "Step 1 - Name the Job" on page 261. |
| | Edit | Allows editing of the attributes of an existing job. See "To edit a previously created job" on page 282. |
| | Delete | Deletes a job from the job tree, or deletes run instances from a specific job icon. |
| | Refresh | Refreshes the wizard screen. |
| Jobs | Submit | Submits a selected job. |
| | Kill Job | Stops the currently running job. |
| | Restart Job | Restarts a run instance. |
| | View Log | Opens the **View Log Options** dialog box. |
| Help | Teradata Parallel Transporter Help | Opens the online help. |
| | About | Displays the active version of the Teradata PT Wizard. |

Many of the following toolbar functions are also available by right-clicking a job instance in the job tree.

Table 21: Toolbar

| Buttons | Name | Function |
|---|---|---|
| | New Job | Creates a new job. See "Create a New Script" on page 261. |
| | Edit Item | Edits an existing job. See "To edit a previously created job" on page 282. |
| | Delete | Deletes jobs or run instances from the Wizard. See "To delete a job" on page 281. |
| | Kill Job | Stops an active job. See "To stop a running job" on page 281. |

Table 21: Toolbar (continued)

| Buttons | Name | Function |
|---|---|---|
| | Submit Job | Submits a job to be run. See "Step 4 - Run the Job" on page 279. |
| | Restart Job | Restarts a job. See "To restart a job" on page 281. |
| | View Log | Opens the View Log dialog box. See "Log Viewer" on page 284. |

APPENDIX C

# Teradata PT Publications

User documentation for Teradata PT is distributed among the following books.

## Teradata PT Publications

| Publication | Contents |
|---|---|
| *Teradata Parallel Transporter Quick Start Guide*<br>B035-2501 | Provides getting-started information for using Teradata PT. Includes Teradata PT job examples for:<br>• Reading data from a flat file and loading it into a Teradata Database target table.<br>• Exporting data from a Teradata Database source table and writing it to a flat file.<br>• Exporting data from a Teradata Database source table and loading it to a Teradata Database target table. |
| *Teradata Parallel Transporter User Guide*<br>B035-2445 | Detailed strategies for planning, implementing, and debugging Teradata PT.<br>The book includes chapters on:<br>• Writing Teradata PT template job scripts, the kind of job scripts illustrated in the *Teradata Parallel Transporter Quick Start Guide*<br>• Writing Teradata PT defined schema job scripts that:<br>  • Move data to and from data targets<br>  • Move data within the Teradata environment<br>• Describing individual Teradata PT operators and access modules<br>• Launching, managing, and troubleshooting a Teradata PT job |
| *Teradata Parallel Transporter Reference* (this book)<br>B035-2436 | A reference book that defines:<br>• Teradata PT command line utility commands.<br>• Object definition statements that make up the declarative section of a Teradata PT job script.<br>• The APPLY statement that makes up the executable section of a Teradata PT job script.<br>• Syntax for each Teradata PT operator. |

| Publication | Contents |
|---|---|
| *Teradata Parallel Transporter Application Programming Interface Programmer Guide*<br>B035-2435 | Provides information about:<br>• Setting up the interface.<br>• Coding.<br>• Error reporting.<br>• Checkpointing and restarting. |
| *Teradata Parallel Transporter Operator Programmer Guide*<br>B035-2435 | Provides information on developing custom operators, including all interface functions that allow communication between the Teradata PT operators and the Teradata PT infrastructure. |

# Glossary

## 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.

## 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.

## F

**field** The basic unit of information stored in the Teradata Database. A field is either null, or has a single numeric or string value.

## J

**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.

## L

**log** A record of events. A file that records events. Many programs produce log files. Often you will look at a log file to determine what is happening when problems occur. Log files have the extension *.log*.

## N

**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.

## O

**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.

## P

**parameter** A variable name in a macro for which an argument value is substituted when the macro is executed.

**privilege** A user's right to perform the Teradata SQL statements granted to him against a table, database, user, macro, or view.

## Q

**query** A Teradata SQL statement, such as a SELECT statement.

## R

**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.

## S

**session** Also called a Teradata Database session. A session begins when the user logs on to Teradata Database and ends when the user logs off Teradata Database. In client software, a logical connection between an application program on a host and the Teradata Database. The connection permits the application program to send one request at a time to and receive one response at a time from Teradata Database.

**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.

## T

**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.

## U

**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.

## V

**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.

## W

**Wizard** The Teradata PT Wizard. A GUI-based product that builds and runs simple load and unload job scripts.

# Index


## A

About Teradata, general information 9
Active directory scan 206
Array type template attributes 215

## B

Batch directory scan 205
Best practices, Teradata PT 239
BTEQ, Teradata PT and 25

## C

CASE DML expressions 209
CASE value expressions 210
CD-ROM images 9
Checkpoint
  files, removing 191
  interval 134
  restarts and 135
  setting 132
  types of 133
Checkpoint directory, setting 135
Complication errors 164
Console log 145

## D

Data acquisition 203
Data acquisition errors 170
Data application errors 171
Data conditioning 208
Data Connector (PIOM), Teradata PT and 24
Data filtering 208
Data loading 203
DDL operator, Teradata PT and 24
Dual Active Solutions, monitoring 236

## E

Easy Loader 195
Error handling, operator-specific 171
Error tables 180
Errors
  Load operator 172
  SQL Selector operator 185
  Stream operator 176
  Update operator 180
Exit codes 144

## F

Failed job
  common 162
  complex failures, debugging 185
  correcting 161
  data application errors 171
  detecting 161
  failure to complete 169
  initialization errors 170
  remedies 162
  SQL errors 171
FastExport, Teradata PT and 24
FastLoad utility, Teradata PT and 24

## G

Generated schema 216

## I

Immediate file logging 204
Inferred schema 211, 218
Information Products web site 9
Initialization errors 170
INMOD routine 51
Insert statements, generated 223

## J

Job failure. See Failed Job
Job log
  assessing 145
  console 145
  locations, operating systems and 148
  private 146
  public 145
  using 145
Job script
  comments 41
  compilation errors 164
  concepts 38
  consumer operator, defining 52
  creating 41
  declarative section 38
  executable section 38, 59

job variables 43
operators, defining 49
producer operator, defining 50
schema, defining 45
schema, multiple 46
standalone operator, defining 54
statement types 38
syntax 40
Job strategies
moving data from Teradata Database 109
moving data within the Teradata Database 119
moving external data into Teradata Database 91
Job variable identifier 224
Job variables 43
generating schema and 220
inferring schema and 220
job-scope 221
step-scope 221
Job. See Teradata PT job

## L

Load operator, errors 172

## M

Mini-batch loading 205
monitor 236
MulitLoad utility, Teradata PT and 25
Multiple APPLY 225

## O

OLE DB Access Module, Teradata PT and 25
Online Archive 33
Operator templates 211
OS Command operator, Teradata PT and 25

## P

Pre-processor errors 163
Private log 146
product version numbers 3
Public log 145
Publications, ordering 9

## R

Restarting, jobs 186
Reusing definitions, with INCLUDE directive 210

## S

Schema
generated 216
inferred 211, 218
Scripts, simplifying 211
SQL errors 171
SQL Selector operator, errors 185
SQL Selector operator, Teradata PT and 25
Stream operator, errors 176
System resource errors 166

## T

tbuild
checkpointing files and 130
errors 163
file name, specifying 129
job name, specifying 129
job variables, assigning 131
job, fatal error and 132
jobname syntax 130
setting options 129
unnamed jobs, running 130
tdload command. See Easy Loader
Template use, limitations 215
Template, operator 211
Teradata PT
description 23
parallelism 27
platforms, supported 25
processing, basics of 26
Teradata PT best practices 239
Teradata PT job
exit codes 144
launching 136
managing 137
post job checklist 143
restarting 186
TMSM Export Job example 237
TMSM Load Job example 237
TMSM Stream Job example 237
TMSM Update Job example 237
Teradata PT operators 236
Teradata PT Wizard. See Wizard
TMSM resource types 236
TPump utility, Teradata PT and 25
twbcmd
job performance, monitoring 138
job-level commands 138
periodicity and 141
rate and 141
twbkill, terminate job 142
twbstat
current active jobs and 137
TYPE DATACONNECTOR PRODUCER 33
TYPE DDL 33
TYPE EXPORT 33
TYPE FASTEXPORT OUTMOD 33

TYPE FASTLOAD INMOD 33
TYPE INSERTER 33
TYPE LOAD 33
TYPE MULTILOAD INMOD 33
TYPE MULTILOAD INMOD FILTER 33
TYPE ODBC 33
TYPE OS COMMAND 33
TYPE SCHEMAMAPPER 33
TYPE SELECTOR 33
TYPE STREAM 33
TYPE UPDATE 33

## U

UNION ALL, multiple sources and 203
Update operator
  error tables 180
  errors 180

## V

VARDATE columns
  DataConnector operator support for 230
  formatting characters for 228
  reformatting DataTime data and 226
version numbers 3

## W

WebSphere MQ Access Module
  Teradata PT and 24
Wizard 258
  column limits 259
  limitations 258
  main dialog box 259
  Map Source to Destination dialog box 276
  menus 285
  toolbars 285