

RED BRICK® WAREHOUSE
Version 5.1

**TABLE MANAGEMENT UTILITY
REFERENCE GUIDE**

5.1

The information in this document is subject to change without notice and does not represent a commitment by Red Brick Systems. The software and/or databases described in this document are furnished under a license agreement and can be used or copied only in accordance with the terms of the agreement. Except as permitted by such license, no part of this document and/or database may be reproduced or transmitted in any form or by any means, electronic or mechanical (including photocopying and recording), or transferred to information storage and retrieval systems without the written permission of Red Brick Systems.

Restricted Rights: Use, duplication, or disclosure by the United States Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software—Restricted Rights at 48 CFR 52.227-19, as applicable.

© Copyright 1991–1998 Red Brick Systems, Inc.
All rights reserved.
Printed in the USA.

RISQL, Red Brick, the Red Brick logo ( RED BRICK®), and The Data Warehouse Company are registered trademarks of Red Brick Systems, Inc. Red Brick Data Mine, Red Brick Data Mine Builder, Red Brick Vista, STARindex, STARjoin, TARGETindex, and TARGETjoin are trademarks of Red Brick Systems, Inc.

Solaris™ and SPARC™ are registered trademarks and Sun®, SunOS®, SunSoft®, and NFS® are trademarks of Sun Microsystems, Inc.

All other trademarks are the properties of their respective companies.

Revision number: 1
January, 1998
Part number: 409051

Red Brick Systems, Inc.
485 Alberto Way
Los Gatos, California 95032
USA
Telephone: +1 408 399 3200
+1 800 777 2585



Contents

About This Document

Purpose	ix
Audience	ix
Organization	x
Related Documentation	xii
Conventions	xiv
Syntax Notation	xiv
Syntax Diagrams	xv
Keywords and Punctuation	xvii
Identifiers and Names	xvii
Customer Support	xviii
Support Solutions Warehouse	xviii
General and Technical Questions	xviii
Troubleshooting Tips	xx
Documentation Questions and Comments	xx

1 Introduction to the Table Management Utility

TMU Operations and Functionality	1-2
TMU Control Files and Statements	1-3
Termination	1-3
Comments	1-3
Locales and Multibyte Characters	1-4
USER Statement	1-4
LOAD DATA and SYNCH Statements	1-4
UNLOAD Statements	1-5
GENERATE Statements	1-5
REORG Statements	1-6
BACKUP and RESTORE Statements	1-6
UPGRADE Statements	1-7
SET Statements	1-7

	TMU Options	1-8
	Parallel TMU	1-8
	Enterprise Copy Management	1-8
	Auto Aggregate	1-8
2	<i>Running the TMU and PTMU</i>	
	User Access and Required Permission	2-2
	Operating System Access	2-2
	Database Access	2-2
	Syntax for rb_tmu and rb_ptmu Programs	2-3
	Exit Status Codes	2-4
	Step-by-Step Procedure and Examples	2-5
	USER Statement for Username and Password	2-9
	SET Statements and Parameters to Control Behavior	2-11
	Lock Behavior	2-12
	Buffer Cache Size	2-13
	Temporary Space Management	2-14
	Parallel Processing Tasks (PTMU Only)	2-17
	Serial Mode Operation (PTMU Only)	2-19
	Format of Datetime Values in TMU Statements	2-20
	Load Information Limit	2-21
	Suggestions for Effective PTMU Operations	2-22
	Operations That Use Parallel Processing	2-22
	Discard Limits on Parallel Load Operations	2-22
	AUTOROWGEN with the PTMU	2-23
	Multiple Tape Drives with the PTMU	2-23
	3480/3490 Multiple-Tape Drive with the PTMU	2-24
3	<i>Loading Data into a Warehouse Database</i>	
	The LOAD DATA Operation	3-2
	Inputs, Outputs, and Options	3-2
	Data Processing During Load Operations	3-4
	Procedure for Loading Data	3-7
	Some Preliminary Decisions	3-9
	Determining Table Order	3-9
	Ordering Input Data	3-9
	Maintaining Referential Integrity with Automatic Row Generation	3-10
	Writing a LOAD DATA Statement	3-17
	LOAD DATA Syntax	3-18
	Input Clause	3-20
	Format Clause	3-24
	Locale Clause	3-29
	Discard Clause	3-33
	Syntax	3-34
	Usage Notes	3-42

Optimize Clause	3-45
Table Clause	3-49
Selective Column Updates with RETAIN and DEFAULT	3-53
Field Specifications (Simple, Concat, Constant, Sequence, Increment)	3-55
Table Clause: Simple Fields	3-57
Table Clause: Concatenated Fields	3-62
Table Clause: Constant Fields	3-65
Table Clause: Sequence Fields	3-68
Table Clause: Increment Fields	3-70
Segment Clause	3-72
Criteria Clause	3-75
Comment Clause	3-80
Fieldtypes	3-82
Character Fieldtype	3-84
Numeric External Fieldtypes	3-85
Packed and Zoned Decimal Fieldtypes	3-87
Integer Binary Fieldtypes	3-89
Floating-Point Binary Fieldtypes	3-90
Datetime Fieldtypes	3-91
Datetime Format Masks for Datetime Fields	3-93
Subfield Components	3-94
Example: Loading Datetime Data	3-99
Restricted Datetime Masks for Numeric Fields	3-101
Writing a SYNCH Statement	3-104
Format of Input Data	3-106
Disk Files	3-107
Tape Files on UNIX Systems	3-109
Fieldtype Conversions	3-111
LOAD DATA Syntax Summary	3-114
4 Unloading Data from a Table	
The UNLOAD Operation	4-2
Internal Format	4-3
External Format	4-3
Data Conversion to External Format	4-4
UNLOAD Syntax	4-5
Unloading or Loading Internal-Format Data	4-10
Unloading or Loading External-Format Data	4-12
Converting a Table to Multiple Segments	4-14
Moving a Database	4-14
Loading External-Format Data into Third-Party Tools	4-14
Unloading Selected Rows	4-15
Example: UNLOAD Statements and External Format Data	4-16

- 5 *Generating CREATE TABLE and LOAD DATA Statements***
 - Generating CREATE TABLE Statements 5-2
 - Generating LOAD DATA Statements 5-4
 - Example: GENERATE Statements and External-Format Data 5-7
- 6 *Reorganizing Tables and Indexes***
 - The REORG Operation 6-1
 - REORG Syntax 6-3
 - Usage Notes 6-4
- 7 *Performing Backup and Restore Operations***
 - Backup/Restore Policy 7-2
 - TMU BACKUP and RESTORE Operations 7-3
 - Backup Level 7-3
 - Backup Locking Operations 7-4
 - Database Locale 7-4
 - BACKUP Syntax 7-5
 - RESTORE Syntax 7-7
 - Database Backup and Restore Procedures 7-10
 - Segment Restore Procedures 7-12
- 8 *Moving Data with the Copy Management Option***
 - The rb_cm Utility 8-2
 - System Requirements 8-3
 - Database Security Requirements 8-4
 - rb_cm Syntax 8-5
 - TMU Control Files for Use with rb_cm 8-8
 - LOAD and UNLOAD Statements 8-8
 - SYNCH Statement 8-10
 - SET Statements 8-10
 - Examples of rb_cm Operations 8-11
 - Example: Copying Data Between Different Computers 8-11
 - Example: Copying Data Between Tables on the Same Computer 8-14
 - Verifying Results of rb_cm Operations 8-16
- A *Example: Using the TMU in AGGREGATE Mode***
 - Background A-1
 - Strategy A-2
 - The Dimension Tables A-4
 - The Sales Table A-6
 - Load Procedure: Refresh Loads A-8
 - Load Procedure: Daily Loads A-10
 - Results A-12

B Loading Data from Existing Systems

Oracle's ORACLE B-1

Oracle's Rdb B-2

Metaphor's DBS 2xx B-2

IBM's DB2 B-3

C Defined Locales***Index***



About This Document

Purpose

This guide provides information needed to use the Table Management Utility (both the TMU and its parallel version, the PTMU) to load and maintain the tables and indexes in Red Brick® Warehouse databases. It includes information necessary for the effective use of the TMU, as well as syntax definitions and procedural descriptions. It is intended to be used in conjunction with the *Warehouse Administrator's Guide* to develop and maintain an efficient data warehouse operation.

This book is intended for use on both UNIX-based and Windows NT systems supported for Red Brick Warehouse. Differences between the two types of systems and features and options that are not available or applicable to both systems are so indicated.

For operating-system or platform-specific information, refer to Red Brick Systems' *Installation and Configuration Guide* for the platform at your site or to the documentation that accompanied the hardware and operating system.

Audience

The intended users of this guide are the database administrators and operators who are responsible for loading and maintaining the tables and indexes in Red Brick Warehouse. In this guide, these users are collectively designated as the warehouse administrator.

Knowledge of the operating system, basic system administration procedures, and relational databases is assumed.

Organization

This guide is organized as follows:

Chapter 1, “Introduction to the Table Management Utility,” provides an overview of the Table Management Utility (TMU), the various types of control files that control its activities, and its optional features.

Chapter 2, “Running the TMU and PTMU,” describes how to invoke the TMU and PTMU and provides runtime and performance information.

Chapter 3, “Loading Data into a Warehouse Database,” describes how to use LOAD DATA statements to load data with the TMU. It also includes detailed information about the Automatic Row Generation feature, datatype formats, fieldtype conversions, input data formats, and the SYNCH statement that is used to synchronize newly loaded offline segments.

Chapter 4, “Unloading Data from a Table,” describes how to use UNLOAD statements to unload data with the TMU in order to move it to another platform or use it in an analysis tool.

Chapter 5, “Generating CREATE TABLE and LOAD DATA Statements,” describes the GENERATE command that automatically generates CREATE TABLE and LOAD DATA statements for use in reloading the unloaded data or in replicating databases.

Chapter 6, “Reorganizing Tables and Indexes,” describes how to use REORG statements to reorganize a table and its indexes to improve performance and maintain referential integrity.

Chapter 7, “Performing Backup and Restore Operations,” briefly describes the need for a backup/restore plan and the recommended solutions; it also provides a complete description of the discontinued TMU incremental backup and restore option, available only to users who purchased it prior to the release of Red Brick Warehouse Version 5.1.

Chapter 8, “Moving Data with the Copy Management Option,” describes how to use the configuration management facility included in the Enterprise Control and Coordination option to copy and move data among the databases of an enterprise.

Appendix A, “Example: Using the TMU in AGGREGATE Mode,” contains an example illustrating the use of the TMU Auto Aggregate option to generate quarterly and yearly sales aggregates.

Appendix B, “Loading Data from Existing Systems,” describes how to load existing databases from several common relational database management systems into a warehouse database.

Appendix C, “Defined Locales,” defines compatible locales that can be used to interpret input data when data is loaded into a warehouse database.

Related Documentation

The standard documentation set for Red Brick Warehouse includes the following documents:

<i>Installation and Configuration Guide</i>	Installation and configuration information, as well as platform-specific material, about Red Brick Warehouse and related products. Customized for either UNIX-based or Windows NT systems.
<i>Warehouse Administrator's Guide</i>	Description of warehouse architecture, supported schemas, and other concepts relevant to warehouse databases. Procedural information for designing and implementing a warehouse database, maintaining a database, and tuning a database for performance. Includes a description of the system tables and the configuration file (<i>rbw.config</i>). Customized for either UNIX-based or Windows NT systems.
<i>Table Management Utility Reference Guide</i>	Description of the Table Management Utility, including all activities related to loading and maintaining data. Also includes information about data replication and the <i>rb_cm</i> copy management utility.
<i>SQL Reference Guide</i>	Complete language reference for the Red Brick Systems SQL implementation and RISQL [®] extensions for warehouse databases.
<i>SQL Self-Study Guide</i>	Example-based review of SQL and introduction to the RISQL extensions, the macro facility, and Aroma, the sample database.
<i>RISQL Entry Tool and RISQL Reporter User's Guide</i>	Complete guide to the RISQL Entry Tool, a command-line tool used to enter SQL statements, and the RISQL Reporter, an enhanced version of the RISQL Entry Tool with report-formatting capabilities.
<i>Messages and Codes Reference Guide</i>	Complete listing of all informational, warning, and error messages generated by warehouse products, including probable causes and recommended responses. Also includes event log messages that are written to the log files.
<i>Release Notes</i>	Information pertinent to the current release that was unavailable when the documents were printed.

In addition to the standard documentation set, the following documents are included for specific sites:

<i>Red Brick Vista User's Guide</i>	Description of the Red Brick Vista™ aggregate navigation and advice system, including procedures for rewriting queries and getting advice on the best set of aggregate tables and views to create. Includes detailed examples of queries whose performance can be dramatically increased by using aggregate navigation.
<i>SQL-BackTrack for Red Brick Warehouse User's Guide</i>	Complete guide to SQL-BackTrack™ for Red Brick Warehouse, a command-line interface for backing up and recovering warehouse databases. Includes procedures for defining backup configuration files, performing online and checkpoint backups, and recovering the database to a consistent state.
<i>Client Connector Pack Installation Guide</i>	Procedures for installing and configuring the Red Brick ODBC Driver, the RISQL Entry Tool, and the RISQL Reporter on client systems. Included for those sites that purchase the Client Connector Pack.
<i>ODBC Connectivity Guide</i>	Information about ODBC conformance levels and instructions for compiling and linking an ODBC application using the Red Brick ODBClib SDK.
<i>Red Brick Data Mine User's Guide</i>	Description of the data mining process, and procedural information for using the Red Brick Data Mine™ SQL-based interface to find hidden or unpredictable relationships among the data in a data set. Included for those sites that purchase the Red Brick Data Mine option.
<i>Red Brick Data Mine Builder™ User's Guide</i>	Description of the data mining process, and procedural information for performing data mining using Red Brick's GUI-based product in a Microsoft Windows environment.

Additional references you might find helpful include:

- An introductory-level book on SQL
- An introductory-level book on relational databases
- Documentation for your hardware platform and operating system

Online Documentation

The Red Brick Warehouse documentation is also available in Adobe Acrobat format on the Red Brick Warehouse CD-ROM.

Conventions

Throughout Red Brick Systems technical publications, the following notation and syntax conventions are used:

- Computer input and output, including commands, code, and examples, appear in *Courier*.
- Information that you enter or that is being emphasized in an example appears in **Courier bold** to help you distinguish it from other text.
- Filenames, system-level commands, and variables appear in *Palatino italic* or *Courier italic*, depending on the context.
- Document titles always appear in *Palatino italic*.
- Names of database tables and columns are capitalized (Sales table, Dollars column). Names of system tables and columns are in all uppercase (RBW_INDEXES table, TNAME column).

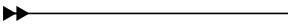





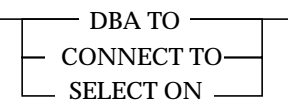
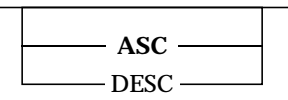
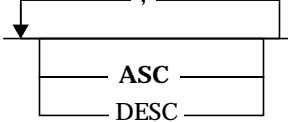
Syntax Notation

This guide uses the following conventions to describe the syntax of operating-system commands:

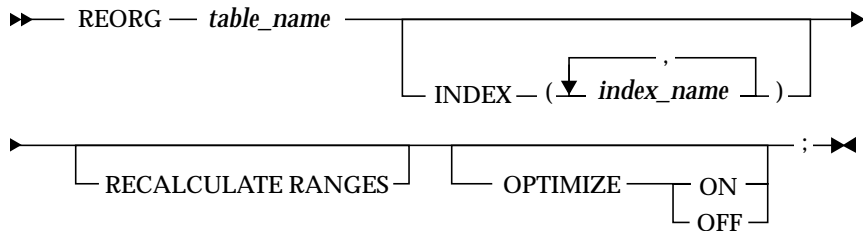
Command Element	Example	Convention
Values and parameters	<i>table_name</i>	Items that you replace with an appropriate name, value, or expression are in <i>italic</i> type style.
Optional items	[]	Optional items are enclosed by square brackets. Do not type the brackets.
Choices	ONE TWO	Choices are separated by vertical lines; choose one if desired.
Required choices	{ONE TWO}	Required choices are enclosed in braces; choose one. Do not type the braces.
Default values	<u>ONE</u> TWO	Default values are underlined, except in graphics where it is in bold type style.
Repeating items	name, ...	Items that can be repeated are followed by a comma and an ellipsis. Separate the items with commas.
Language elements	() , ; .	Parentheses, commas, semicolons, and periods are language elements. Use them exactly as shown.

Syntax Diagrams

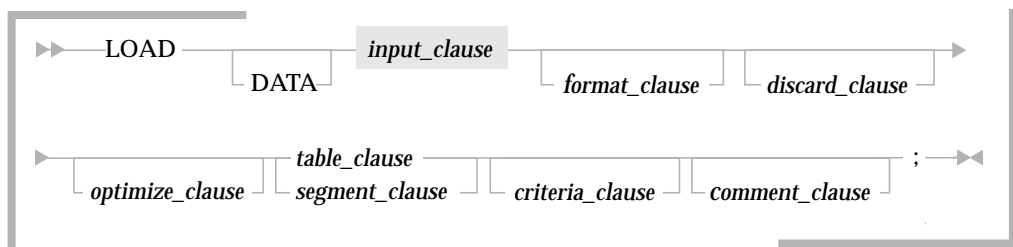
This guide uses diagrams built with the following components to describe the syntax for statements and all commands other than system-level commands:

Component	Meaning
	Statement begins.
	Statement syntax continues on next line. Syntax elements other than complete statements end with this symbol.
	Statement continues from previous line. Syntax elements other than complete statements begin with this symbol.
	Statement ends.
	Required item in statement.
	Optional item.
	Required item with choice. One and only one item must be present.
	Optional item with choice. If a default value exists, it is printed in bold .
	Optional items. Several items are allowed; a comma must precede each repetition.

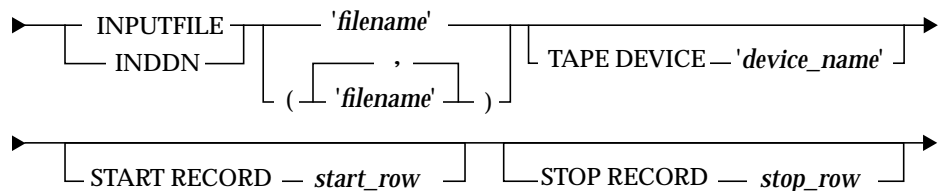
The syntax elements shown above are combined to form a diagram as follows:



Complex syntax diagrams such as the one for the following statement are repeated as point-of-reference aids for the detailed diagrams of their components. Point-of-reference diagrams are indicated by their shadowed corners, gray lines, and reduced size:



The point-of-reference diagram is then followed by an expanded diagram of the shaded portion—in this case, the `input_clause`:



Keywords and Punctuation

Keywords are words reserved for statements and all commands except system-level commands. When a keyword appears in a syntax diagram, it is shown in uppercase. You can write a keyword in upper- or lowercase, but you must spell the keyword exactly as it appears in the syntax diagram.

Any punctuation that occurs in a syntax diagram must also be included in your statements and commands exactly as shown in the diagram.

Identifiers and Names

Metavariables serve as placeholders for identifiers and names in the syntax diagrams and examples. A metavariable can be replaced by an arbitrary name, identifier, or literal, depending on the context. Metavariables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a metavariable appears in a syntax diagram, an example, or text, it is shown in *lowercase italic*.

The following syntax diagram uses metavariables to illustrate the general form of a simple SELECT statement:

►— SELECT — *column_name* — FROM — *table_name* —►

When you write a SELECT statement of this form, you replace the metavariables *column_name* and *table_name* with the name of a specific column and table.

Customer Support

Please review the following information before contacting the Customer Support Center at Red Brick Systems.

Support Solutions Warehouse

The Support Solutions Warehouse is the Customer Support Center's external web site, an online resource that registered Red Brick customers can use to:

- Submit new cases.
- Read release notes.
- Find answers to frequently asked questions (FAQs).
- Search the Problems and Solutions database.

To use the Support Solutions Warehouse, point your web browser to the following URL and enter your registered username and password:

<http://www.redbrick.com/RBCustomer/index.htm>

If you do not have a registered username and password, contact the Customer Support Center by telephone, fax, or e-mail.

General and Technical Questions

If you have general sales-related questions or technical questions about Red Brick products or services, contact Red Brick Systems as follows:

Telephone

General Questions (408) 399-3200 or 1 (800) 777-2585

Technical Questions (408) 399-7100 or 1 (800) 727-1866

FAX

General Questions (408) 399-3277

Technical Questions (408) 399-3297

Internet e-mail

General Questions info@redbrick.com

Technical Questions support@redbrick.com

World Wide Web www.redbrick.com

Existing Cases

If you want to inquire about the status of an existing case, please have the case number ready. The case number will always be given to you by the support engineer who logs the case or first contacts you. This number is used to keep track of all the activities performed during the resolution of each problem.

New Cases

If you want to log a new case, please have the following information ready:

- Red Brick Warehouse version
- Platform and operating-system version
- Error messages returned by Red Brick Warehouse or the operating system
- Concise description of the problem, including any commands or operations performed prior to the occurrence of the error message
- List of Red Brick Warehouse and/or operating-system configuration changes made prior to the occurrence of the error message

If you think the problem concerns client-server connectivity, please have the following additional information ready:

- Name and version of the client tool in use
- Version of Red Brick ODBC Driver in use, if applicable
- Name and version of client network and/or TCP/IP stack in use
- Error messages returned by the client application
- Warehouse and client locale specifications

Troubleshooting Tips

You can often reduce the time it takes to close your case by providing the smallest possible reproducible example of your problem. The more you can isolate the cause of the problem, the more quickly the support engineer can help you resolve it.

- For SQL query problems, try removing columns or functions, or restating WHERE, ORDER BY, or GROUP BY clauses until you can isolate the part of the statement causing the problem.
- For TMU load problems, verify the datatype mapping between the source file and the target table to ensure compatibility. Try loading a small test set of data to determine whether the problem concerns volume or data format.
- For connectivity problems, verify that the network is up and running by issuing the *ping* command from the client to the host. If possible, try another client tool to see if the same problem arises. Try running the *rbwconnect.verify* script to check that Red Brick Warehouse Connect is running and functioning properly. Check the *rbwconnect.log* file for errors.

Documentation Questions and Comments

If you have questions or comments about the Red Brick Warehouse documentation, please contact the Technical Publications Department at Red Brick Systems as follows:

Telephone	(408) 399-3200 or 1 (800) 727-1866
Internet e-mail	docs@redbrick.com

Introduction to the Table Management Utility

The Table Management Utility (TMU) is the Red Brick® Warehouse component that is used for those activities related to loading data into the database and maintaining the tables, indexes, and relational integrity of the database. While the primary function of the TMU is to load and index large amounts of data very quickly, it also performs the following tasks:

- Unloading data from a database in order to move data, either an entire table or selected rows, or to edit the data for use with the TMU or with other applications.
- Rebuilding tables and indexes after the table has been substantially modified by incremental loads or by insert, update, or delete statements.
- Backing up the database to prevent data loss; restoring it if data loss occurs.
- Generating DDL (CREATE TABLE) or TMU LOAD DATA statements from an existing table for use with unloaded data.
- Upgrading databases to provide upward compatibility with new versions of Red Brick Warehouse.

This chapter contains the following sections:

- TMU Operations and Functionality
- TMU Control Files and Statements
- TMU Options

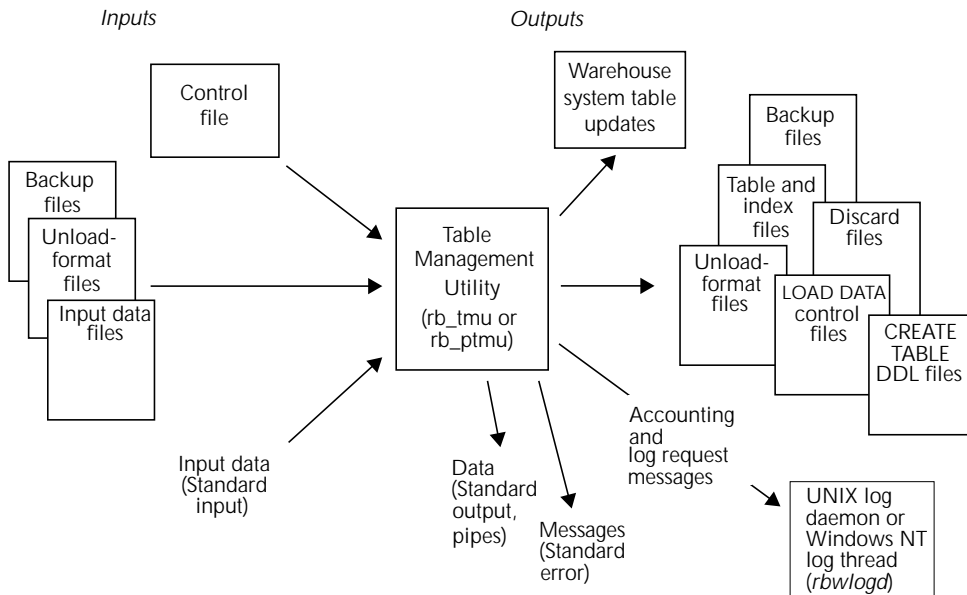
TMU Operations and Functionality

The TMU is a program that runs independently of the warehouse server; it is invoked from the operating-system command line and uses the same configuration information as other components of Red Brick Warehouse.

Before invoking the TMU, you must first write a control file, using the TMU control language, that specifies the task to be done and provides the information needed to do that task. Then you run the TMU, naming the control filename as input. The TMU reads the control file and carries out the task, reading its input from tape, disk, or standard input, and modifying the database or writing output files to tape or disk as directed. At the same time, it writes messages for system logging and accounting purposes. The TMU supports a variety of tape, disk, and data formats.

Note: Tape support is not available for Windows NT systems.

The following figure illustrates the TMU and its input and output options.

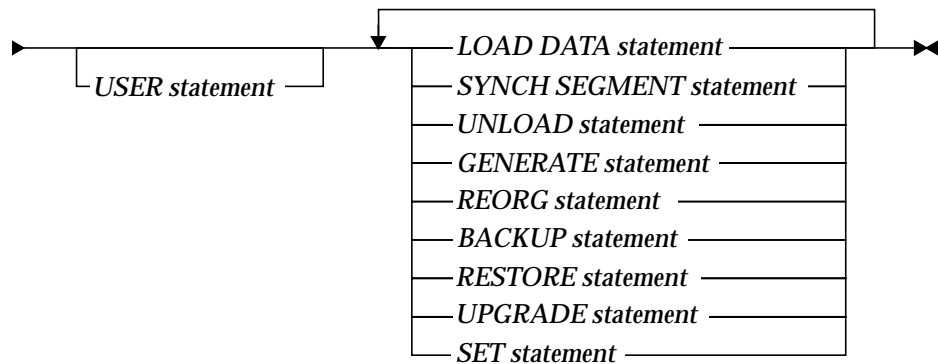


TMU Control Files and Statements

A TMU control file contains one or more statements that specify the functions to be performed and the information needed by the TMU to perform those functions. A single control file can contain multiple control statements of the same or different types; for example:

- A **USER** statement that provides a database username and password.
- Various types of control statements that correspond to the functions to be performed.
- **SET** statements that control the TMU execution environment.

The syntax for a TMU control file is:



Each of the above control statements is briefly described in the following sections.

Termination

Note that each control statement must be terminated with a semicolon (;) as shown in the detailed syntax diagrams for each statement in the remaining chapters. If multiple control statements are included in a single control file, each one must be terminated by a semicolon.

Comments

Comments in a control file can be either enclosed in C-language-style delimiters (`/*...*/`), in which case they can span multiple lines, or preceded by two dashes (`--`) and terminated by end-of-line, in which case they are limited to a single line.

Locales and Multibyte Characters

Most of a TMU control statement—LOAD DATA, UNLOAD, SYNCH, REORG, and so on—must be specified with ASCII characters, regardless of the database locale. However, the TMU does support multibyte characters for database object names and for some special characters used in those statements, and a locale can be specified for a TMU input file that differs from the database locale. Messages returned by the TMU are displayed in the language of the database locale unless the RB_NLS_LOCALE environment variable for the current user overrides that locale. In all other cases, TMU operations use the locale of the database.

USER Statement

A USER statement provides a database user name and password, thereby allowing you to invoke the TMU without entering a username and password on the command line or interactively in response to a prompt.

Only one USER statement can occur in a control file and it must be the first statement in the file. If a username and password are provided on the command line, those values override a USER statement present in the control file and a warning is issued that the USER statement was overridden.

For information about USER statements, refer to “USER Statement for Username and Password” on page 2-9.

LOAD DATA and SYNCH Statements

A LOAD DATA statement provides the control information needed to load data into a database. This information includes the LOAD DATA keywords, the source of data, its format, its locale, what to do with records that cannot be loaded, and how to map the input data record fields into the database table columns; it does not include the data itself.

If data is loaded into an offline segment of a table, that segment must be synchronized with the rest of the table after the data is loaded; this synchronization is performed with a SYNCH statement. Note that the SYNCH statement is used only in conjunction with load operations into offline segments.

For information about LOAD DATA and SYNCH control files, refer to Chapter 3, “Loading Data into a Warehouse Database.” For additional information about load operations with the *rb_cm* copy management utility, refer to Chapter 8, “Moving Data with the Copy Management Option.”

UNLOAD Statements

An UNLOAD statement provides information needed to unload data from a warehouse table in any of several formats in order to move the data or to use it with another tool. An UNLOAD statement contains the UNLOAD keyword and other relevant information such as the name of the table to be unloaded, a description of the desired output format, and where to write the output files. Data can be unloaded in the order determined by either a table scan or an index; either a complete table can be unloaded, or only those rows that meet the specified criteria.

In cases where you are unloading data that will later be loaded into another table, you can include instructions in the UNLOAD statement for the TMU to automatically generate an SQL CREATE TABLE statement that corresponds to the table to be unloaded and a TMU LOAD DATA statement that corresponds to its data. These automatically generated statements provide templates that with little or no modification allow you to create a table and load it with the unloaded data. (This functionality is also available in the GENERATE statement.)

For information about UNLOAD control files, refer to Chapter 4, “Unloading Data from a Table.” For information about unloading data for use with the *rh_cm* copy management utility, refer to Chapter 8, “Moving Data with the Copy Management Option.”

GENERATE Statements

A GENERATE statement provides the information needed to automatically generate an SQL CREATE TABLE or TMU LOAD DATA statement based on an existing table. Using a GENERATE statement allows you to separate the task of generating the CREATE TABLE or LOAD DATA statements from the task of unloading the data—to generate these statements without actually unloading the data.

For information about GENERATE control files, refer to Chapter 5, “Generating CREATE TABLE and LOAD DATA Statements.”

REORG Statements

A REORG statement instructs the TMU to “reorganize ” a table, which includes enforcing referential integrity and rebuilding any specified indexes to improve internal storage. (Referential integrity is the relational property that each foreign key value in a referencing table exists as a primary key value in the referenced table.) A REORG statement includes the REORG keyword, a table name, and index names and instructions for rebuilding them.

For information about REORG control files, refer to Chapter 6, “Reorganizing Tables and Indexes.”

BACKUP and RESTORE Statements

The TMU incremental backup and restore option has been discontinued in Red Brick Warehouse Version 5.1 with the introduction of the SQL-BackTrack™ for Red Brick Warehouse system. The TMU incremental backup and restore option is available only through the life of Version 5.1 to those users who purchased the option prior to the release of Version 5.1.

A BACKUP statement provides the information needed to use the TMU to perform full or incremental backups based on changes to the database (rather than changes to files), and a RESTORE statement provides the information needed to restore a damaged database or segment from TMU backup files.

For a brief discussion about backup/restore policies and for information about TMU BACKUP and RESTORE control files, refer to Chapter 7, “Performing Backup and Restore Operations.”

UPGRADE Statements

An UPGRADE statement instructs the TMU to upgrade an existing warehouse database so that it is compatible with a newer version of Red Brick Warehouse. Not all versions require that databases be upgraded; for those that do require an upgrade, the information needed to upgrade a database is built into the UPGRADE command in the new Red Brick Warehouse software.

The syntax for the TMU UPGRADE statement is as follows:

```
➤ UPGRADE _____ ; ➡  
                |____ DDLFILE 'filename' ____|
```

where DDLFILE '*filename*' indicates that as part of the upgrade procedure, the TMU is to generate a file containing DDL statements. Such files are not always required and their contents are specific to each release.

Refer to the *Migration Guide* and to the release notes that accompany each release of Red Brick Warehouse for the following information:

- Whether a database needs to be upgraded when a new release of Red Brick Warehouse is installed.
- Specific syntax for the UPGRADE statement for that release, including details about the contents of any required DDL files.
- Changes made by an UPGRADE operation for that release.

For general instructions on how to install new software and plan an upgrade for a production database, refer to the *Installation and Configuration Guide* for your platform.

SET Statements

Various options are available that allow you to customize certain aspects of TMU behavior for a specific session. TMU SET statements for these options can be included in a control file to override global configuration parameters set in the configuration file (*rbw.config*). For example, you might want to use a SET statement to change the default location and amount of temporary space to be used by a specific load operation.

For information about these SET statements, refer to “SET Statements and Parameters to Control Behavior” on page 2-11.

TMU Options

The following TMU options are available for enhanced performance and functionality. If your site is licensed to use an option, separate instructions are provided that describe how to enable it; no option can be used unless it is enabled with your unique license key.

Parallel TMU

The Parallel TMU, available as the PTMU option, speeds up load operations on tables that have multiple indexes and referential integrity constraints because it performs index building, data conversion, and referential integrity checks in parallel as data records are loaded.

Note: The PTMU option is not available for Windows NT systems.

Most of the information in this reference guide applies to both the TMU and the PTMU; in cases where differences in behavior or syntax exist, these differences are specified. For general guidelines about PTMU usage, refer to “Suggestions for Effective PTMU Operations” on page 2-22. For information about implementation and performance on specific platforms, refer to the *Installation and Configuration Guide* for that platform.

Enterprise Copy Management

The Enterprise Copy Management option, also available as part of the Enterprise Control and Coordination option, facilitates the movement and synchronization of data among multiple warehouse databases found throughout an enterprise. In addition to combining high-speed loading and unloading operations, it can also move data over a network. This option is described in Chapter 8, “Moving Data with the Copy Management Option.”

Auto Aggregate

The Auto Aggregate option provides the capability to automatically aggregate new input data with the data already in a table. For example, if you are loading daily sales amounts for each store into the Sales table, you can automatically add the new amount for each store to the amount already in the table to keep a running total, or aggregate. Aggregate operators include ADD, SUBTRACT, MIN, and MAX. This capability, combined with the ability to formulate an accept/reject criteria for incoming data, allows you to perform aggregation on a selective basis. This option is described in Chapter 3, “Loading Data into a Warehouse Database,” with a detailed example in Appendix A, “Example: Using the TMU in AGGREGATE Mode.”

Running the TMU and PTMU

Before you can use the TMU or the Parallel TMU (PTMU), you must prepare a control file containing statements that define the tasks to be performed; these statements are described in subsequent chapters. When the control file is ready—either a newly created one or an existing one that has been modified—and any required input files are ready, you can run the TMU or the PTMU, as described in this chapter.

This chapter contains the following sections:

- User Access and Required Permission
- Syntax for rb_tmu and rb_ptmu Programs
- Exit Status Codes
- Step-by-Step Procedure and Examples
- USER Statement for Username and Password
- SET Statements and Parameters to Control Behavior
- Suggestions for Effective PTMU Operations

Use of the PTMU is similar to use of the TMU, except as noted in the syntax on page 2-3 and the specific suggestions for PTMU use on page 2-22. The PTMU is a separate option that must be installed and enabled with a license key.

Note: The PTMU is not available for Windows NT systems.

User Access and Required Permission

To use the TMU or PTMU, the user must have the required permissions with respect to both the operating system and the database.

Operating System Access

The executable files for the TMU and PTMU are named *rb_tmu* and *rb_ptmu*, respectively, and they are located in the *bin* subdirectory in the *redbrick* directory. These programs run under the *redbrick* user ID and all files that they create are owned by the *redbrick* user.

Note: Throughout Red Brick technical publications, the *redbrick* directory (or *redbrick_dir* in examples) is used to indicate the directory into which the Red Brick Warehouse software was installed, and the *redbrick* user ID is used to indicate the warehouse administrator user ID, which is the operating-system user ID used to install the Red Brick Warehouse software. If your site installed the software in another location or with another user ID, then substitute that location or user for *redbrick* wherever you see references to the *redbrick* directory or *redbrick* user.

If you run either the TMU or the PTMU (*rb_tmu* or *rb_ptmu*) from a user other than *redbrick*, you must ensure that the *redbrick* user ID has read access to the control file and input files and write access to the directories in which the table, index, discard, and generated files will be written. For example, if the administrative user at your site has the username *redbrick* and you are running the TMU under your username *calvin*, then you must make sure that the *redbrick* user has the necessary permissions to read, write, and execute the required files. If you have used another name for the administrative user—for example, *dilbert*—you must make sure *dilbert* has the necessary permissions on the required files.

Database Access

You can specify the database to be accessed on the command line when you invoke the TMU. If you do not specify a database on the command line, the TMU uses the database specified by the *RB_PATH* environment variable.

The database user ID you supply the TMU—on the command line, in response to a prompt, or with a *USER* statement in the control file—must have the necessary object privileges and/or task authorizations to perform the TMU operation. The password also can be supplied on the command line, in response to a prompt, or in a *USER* statement.

Syntax for *rb_tmu* and *rb_ptmu* Programs

The syntax to invoke the TMU and PTMU, entered at an operating-system command line prompt, is as follows:

```
rb_tmu [options] control_file [db_username [db_password]]
```

```
rb_ptmu [options] control_file [db_username [db_password]]
```

options

Any or all of the following, in any order:

- | | |
|--|--|
| <p>-interval <i>nrows</i>
or
-i <i>nrows</i></p> | <p>Optional; directs the TMU to print a progress message to the system message file for every <i>n</i> rows of data that are loaded. Because the TMU processes rows in small batches, it reports the number of rows processed at the end of the batch in which an interval occurs; hence the number reported might not be exactly <i>nrows</i>. Frequent intervals slow down the load process.</p> <p>If an interval is not specified, no progress messages are printed.</p> |
| <p>-database <i>db_name</i>
or
-d <i>db_name</i></p> | <p>Optional; logical database name defined in the <i>rbw.config</i> file. This option overrides RB_PATH. If no database is specified, the value of the RB_PATH environment variable is used.</p> |
| <p>-timestamp
or
-t</p> | <p>Optional; timestamp information is appended to all information and error messages issued by the TMU. The timestamp format is localized in accordance with the locale of the operating system, whereas the messages use the locale of the database (or the locale specified by the RB-NLS_LOCALE environment variable).</p> |

control_file

Pathname of file containing the TMU control statements.

db_username, db_password

Optional; database username and password (not operating-system user account and password). The username can be either single-byte or multibyte characters; the password must be single-byte characters. If you do not supply these arguments with the command or in the control file, the TMU prompts for them before executing the control file. If you supply a username and password both with the command and in the control file, the values specified with the command override those in the control file.

Usage Notes

- To display the TMU or PTMU syntax, enter `rb_tmu` or `rb_ptmu` with no options at the system prompt; for example:

```
% rb_tmu
Usage: /redbrick_dir/bin/rb_tmu [<Options>] <control_file>
[<username> [<password>]]
Options:
-i or -interval <nrows>      Display progress every <nrows> rows.
-t or -timestamp              Append a timestamp to all TMU
messages.
-d or -database <database>   Database to use.
```

- If the TMU or PTMU is interrupted, it exits immediately after closing any open tables.

Exit Status Codes

Upon exiting, the TMU and PTMU return the highest status code encountered during processing. For example, if the TMU generates only warning messages, it returns an exit status code of 1; if, however, it generates both warning and fatal messages, it returns an exit status code of 3. You can use these exit status codes to control user-implemented applications that run the TMU or PTMU. The following table defines the meaning of each exit status code:

Status	Meaning
0	Information or statistics messages might have been issued during execution, but no warning, error, or fatal messages were issued.
1	Warning messages cause an exit status of 1.
2	Error messages cause an exit status of 2.
3	Fatal messages cause an exit status of 3.

An exit status of 2 or 3 cause the TMU or PTMU to stop execution.

Step-by-Step Procedure and Examples

Use the following procedure to set up your environment and invoke the TMU (or PTMU).

1. Log in as the *redbrick* user. (If you run the TMU as any user other than *redbrick*, you must verify that the *redbrick* user has the necessary access to all locations used for input, output, discard, and generated files, as described on page 2-2.)
2. Make sure the system is configured as you want it:
 - Verify that the RB_HOST environment variable is set to the correct warehouse daemon (UNIX) or service (Windows NT).
 - Verify that the RB_CONFIG environment variable is set to the directory containing the *rbw.config* file.
 - Verify that the RB_PATH environment variable is set to the correct database; if it is not set to the database that you want to access, you must use the -d option to provide the logical database name when you invoke the TMU.

Note: On Windows NT systems, the default value of these environment variables is determined by the warehouse service selected by RB_HOST in the Registry.

3. Invoke the TMU specifying the file containing the TMU control statements. You can enter your database username and password on the command line or with a USER statement within the control file, or you can supply it in response to the TMU prompt.

Example

The following example illustrates how to set the RB_PATH variable and run the TMU on the Aroma database with a control file named *aroma.tmu*, sending progress messages with timestamps to the terminal every 10,000 rows.

UNIX At a Korn or Bourne shell prompt, enter:

```
$ RB_PATH=AROMA; export RB_PATH
$ rb_tmu -i 10000 -timestamp aroma.tmu curly secret
```

UNIX At a C shell prompt, enter:

```
% setenv RB_PATH AROMA
% rb_tmu -i 10000 -timestamp aroma.tmu curly secret
```

Windows NT At a Windows NT shell prompt, enter:

```
c:\db1> set RB_PATH=AROMA
c:\db1> rb_tmu -i 10000 -timestamp aroma.tmu curly secret
```

The TMU first logs in user *curly* with password *secret* to the database referenced by the RB_PATH variable. The TMU executes the control statements contained in *aroma.tmu* located in the current directory. Progress messages are issued approximately every 10,000 rows (based on row batch boundaries, as described on page 2-3). Timestamps are appended to all messages.

Example

This example illustrates how to invoke the TMU and specify a database with the -d option; at a shell prompt, enter:

```
rb_tmu -d AROMA aroma.tmu
```

The TMU checks the control file for a USER statement. If it does not find one, it prompts for username and password before executing the control statements in the file *aroma.tmu* on the Aroma database located in the directory defined in the *rbw.config* file.

Example

This example illustrates how to invoke the PTMU and specify an interval with a timestamp; at a shell prompt, enter:

```
rb_ptmu -i 10000 -timestamp aroma.tmu curly secret
```

Example

This example illustrates how to capture TMU informational and error messages in a file. In this example, the messages will be written to the file named *load_messages*.

UNIX At a Korn or Bourne shell prompt, enter:

```
$ rb_tmu file.tmu system secret > load_messages 2>&1
```

UNIX At a C shell prompt, enter:

```
% rb_tmu file.tmu system secret >& load_messages
```

UNIX If you do not want to display messages on the terminal or write them to a file, you can redirect the system *stderr* output to */dev/null* to prevent the creation of very large files. This practice is not recommended because you will not be able to detect any problems that occur. You can also redirect messages through a filter such as UNIX *grep* to filter out repetitive informational messages.

Windows NT At a Windows NT shell prompt, enter:

```
c:\db1> rb_tmu file.tmu system secret > load_messages 2>&1
```

Example

This example illustrates how to use the output from another program—in this case, *zcat*, a decompression program—as the input for the TMU. At a shell prompt, enter:

```
zcat cpressd_file | rb_tmu mydb.tmu system secret
```

This command first executes the decompression program to decompress the file *cpressd_file* and pipes the output into the TMU. The TMU uses this output as its input when it executes the *mydb.tmu* control file. A separate file is not needed to hold the decompressed data, which reduces the temporary storage requirements. The control file must specify standard input ('-') as its input file.

Example

This example illustrates another way to use standard input for the input data, allowing a single control file (*mydb.tmu*) to process different data input files (one of which is *market.txt*). At a shell prompt, enter:

```
rb_tmu mydb.tmu system secret < market.txt
```

The TMU uses the file *market.txt* as the input when it executes the *mydb.tmu* control file; another input file can be named the next time the *mydb.tmu* control file is used. The control file must specify standard input ('-') as the input source.

For more information about file redirection and pipes, refer to your operating-system documentation.

Example

This example, based on UNIX named pipes and the *tee* command, illustrates how to run multiple instances of the TMU to read an input file once and load multiple tables. Similar capabilities are available on Windows NT systems, using named pipes and third-party software.

Assume the daily input data is in a file named *Sales.txt* and it is stored in a table named *Sales*; this same input data is also used to generate the aggregate sales data stored in tables named *Sales_Monthly* and *Sales_Quarterly*.

1. Modify the load script that loads the *Sales* table to read its input from the standard input (*stdin*). The modified script is in a file named *Sales.stdin.tmu*.
2. Modify the load scripts that load the aggregate tables to read their input from named pipes *pipeM* and *pipeQ*. These modified scripts are in files named *SalesMonthly.pipeM.tmu* and *SalesQuarterly.pipeQ.tmu*.
3. Create two named pipes *pipeM* and *pipeQ* with the UNIX *mkfifo* utility:

```
% mkfifo pipeM pipeQ
```
4. Start two instances of the TMU in the background; they will use the load scripts modified in step 2, which read input from the two pipes, as control files.

Note: In the C shell, the *>&* characters direct the output from each *rb_tmu* process to a separate file instead of to the terminal; the *&* character runs the process in the background. You must use the corresponding characters for the UNIX shell that you are using.

- ```
% rb_tmu MonthlySales.pipe_m.tmu system manager >& pipe_mout &
```
- ```
% rb_tmu QuarterlySales.pipe_q.tmu system manager >& pipe_qout &
```
5. Read the input data with the UNIX *cat* command, pipe the standard output to the two pipes using the UNIX *tee* command, and pipe the *tee* standard output to a third instance of the TMU—which uses the modified *Sales* load script as a control file:

```
% cat Sales.txt | tee pipeM pipeQ | rb_tmu Sales.stdin.tmu \
```

```
system manager >& sales_out
```

By using the named pipes and the *tee* command, you are reading the input file only once, but loading it into three tables with a single operation in the same time it takes to load a single table. You could also run step 5 in the background so you can monitor all three output files.

USER Statement for Username and Password

If you prefer not to enter a database username and password on the command line, you can include a USER statement at the beginning of a control file. A control file can include only one USER statement, and it must be the first statement in the file.

The USER statement syntax is:

► — USER — *db_username* — PASSWORD — *db_password* — ; — ►

USER *db_username*

Specifies the database user under whose name the TMU is invoked. The username can be either a literal value—for example, *john*, *smith2*, or *elvis*—or an environment variable. The database username can be composed of either single-byte or multibyte characters.

If the username is an environment variable, the corresponding username is determined from the environment. If the environment variable is not defined, a warning is issued; and if a valid user name was not supplied when the TMU was invoked, an error occurs.

UNIX If the username begins with \$, the value is taken as an environment variable—for example, \$DBADMIN.

Windows NT If the username is surrounded by % symbols or begins with \$, the value is taken as an environment variable—for example, %DBADMIN% or \$DBADMIN.

PASSWORD *db_password*

Specifies the password for the database user. The password can be either a literal value or an environment variable; it must be composed of single-byte characters.

If the password is an environment variable, the corresponding password is determined from the environment. If the environment variable is not defined, a warning is issued; and if a valid password was not supplied when the TMU was invoked, an error occurs.

UNIX If the password begins with \$, the value is taken as an environment variable—for example, \$SECRET.

Windows NT If the password is surrounded by % symbols or begins with \$, the value is taken as an environment variable—for example, %SECRET% or \$SECRET.

Examples

The following USER statement uses literal values:

```
user dbadmin password secret;
```

The following USER statements use environment variables:

UNIX

```
user $MY_DBNAME password $MY_PSSWD;
```

Windows NT

```
user %MY_DBNAME% password %MY_PSSWD%;  
user $MY_DBNAME password $MY_PSSWD;
```

The following USER statement uses both a literal value and an environment variable:

UNIX

```
user dbadmin password $MY_PSSWD;
```

Windows NT

```
user dbadmin password %MY_PSSWD%;  
user dbadmin password $MY_PSSWD;
```

SET Statements and Parameters to Control Behavior

Along with the various functional statements in a control file, you can include SET statements to specify certain aspects of TMU and PTMU behavior for a specific TMU session. For example, for a specific load operation, you might want to specify a different directory for temporary space than the one specified in the *rbw.config* file.

The following parameters for the TMU and PTMU can be controlled with SET statements:

Parameter	Controls
LOCK WAIT, NO WAIT	TMU behavior when the data-base or target tables are locked.
TMU_BUFFERS	Size of the buffer cache used by the TMU.
INDEX_TEMPSPACE_DIRECTORY, INDEX_TEMPSPACE_THRESHOLD, INDEX_TEMPSPACE_MAXSPILLSIZE	TMU use of temporary space, both memory and disk.
*TMU_CONVERSION_TASKS, *TMU_INDEX_TASKS	Parallel processing of data-conversion and index-building tasks during the load procedure.
*TMU_SERIAL_MODE	Use of serial processing.
DATEFORMAT	Valid alternative date format for input data.
RBW_LOADINFO_LIMIT	Number of rows of historical information stored in the RBW_LOADINFO system table.

* These parameters affect only the PTMU.

As noted in the following descriptions of the SET statements, some of them override analogous global settings in the *rbw.config* file. A SET statement in a control file affects TMU behavior only during the TMU session using that control file. If a control file contains multiple TMU statements—for example, a single file containing three LOAD DATA statements followed by a REORG statement—a SET statement after the first LOAD DATA statement applies to all subsequent statements. After that session, the option value reverts to the value specified in the *rbw.config* file; if no value is specified in the *rbw.config* file, it reverts to its default value.

Lock Behavior

The TMU automatically locks the database or the affected tables during its operations. If the database or table is already locked, then whether the TMU returns immediately or waits for the lock to be released depends on the behavior selected with the SET LOCK statement.

Syntax

To specify the behavior to be used for a specific session when locked tables are encountered, enter a SET LOCK statement in the TMU control file using the following syntax:

► — SET LOCK —

WAIT
NO WAIT

 — ; — ►

WAIT

Default behavior. If the database or table is already locked, the TMU waits until any existing locks are released and then completes the operation.

NO WAIT

If the database or table is already locked, the TMU returns with a message saying that the operation failed because the database or table was locked. (The default behavior for the warehouse server is also WAIT.)

Note: In cases where waiting for a lock might result in a deadlock, the lock request is refused and control is returned to the lock requestor. Note that the possibility of deadlocks occurs only when the LOCK TABLE or LOCK DATABASE command is used. A deadlock cannot be caused by the automatic locking operations of the TMU. For more information about deadlocks, refer to the *Warehouse Administrator's Guide*.

Example

The following example illustrates how to set the lock behavior with a SET LOCK statement:

```
set lock no wait;
```


Buffer Cache Size

TMU performance is affected by the size of its buffer cache; however, keep in mind that performance is affected not only by buffer cache size, but also by system load and other factors related to the database.

Red Brick Systems recommends that you use the default settings for the TMU buffer cache. After careful analysis of your hardware, software, and user environment, however, you might determine that changes to the buffer cache size would improve performance.

Syntax

To specify TMU buffer cache size for a specific session, enter a SET statement in the TMU control file; for all sessions, edit the TUNE parameter in the *rbw.config* file. The syntax is as follows:

► SET ——— TMU BUFFERS ——— *num_blocks* ——— ; —►

► TUNE ——— TMU_BUFFERS ——— *num_blocks* ———►

num_blocks

Specifies the number of 8-kilobyte blocks. It must be an integer between 128 and 8,208. The default buffer cache size is 128 blocks.

Note that a SET statement can increase, but can never decrease, the current buffer cache size during a given session. For example, if the current buffer size is 1024 blocks, set either in the *rbw.config* file or by a previous SET statement in the TMU control file, the size cannot be reduced to 512 with a SET statement.

Examples

The following examples illustrate how to specify buffer cache size.

SET command: `set TMU BUFFERS 1024;`

rbw.config file entry: `TUNE TMU_BUFFERS 1024`

Temporary Space Management

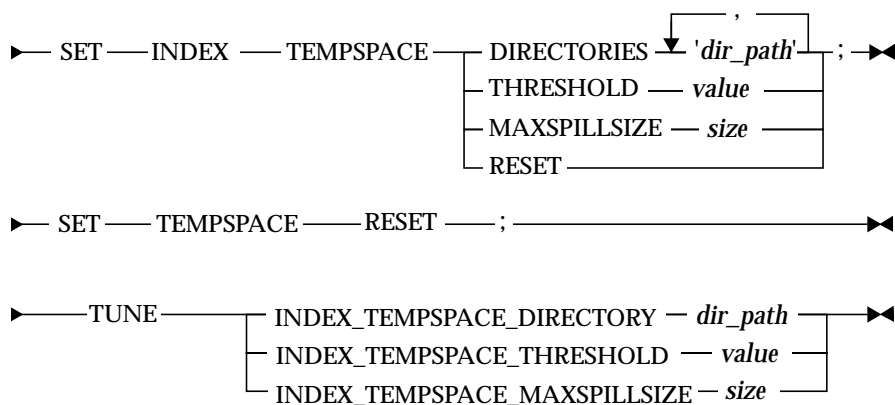
As data is loaded and indexed, intermediate results are stored in memory until they reach a threshold value, at which point they are written (spilled) to disk. The following parameters control how temporary space—both memory and disk—is used.

- INDEX TEMPSPACE DIRECTORIES, which specifies temporary space directories to be used by the TMU for index-building operations.
- INDEX TEMPSPACE THRESHOLD, which specifies the size at which index-building operations spill to disk.
- INDEX TEMPSPACE MAXSPILLSIZE, which specifies the maximum size to which a spill file can grow.

This section describes both the SET commands and the TUNE entries in the *rbw.config* file for these parameters. For more information about using these parameters, which affect not only TMU and PTMU operations but also SQL DDL statements, refer to the *Warehouse Administrator's Guide*.

Syntax

To specify INDEX TEMPSPACE parameters for a specific session, enter a SET statement in the TMU control file; for all sessions, edit the TUNE parameters in the *rbw.config* file. The syntax is as follows:



DIRECTORY 'dir_path', DIRECTORIES 'dir_path', ...

Specifies a directory or a set of directories that are to be used for temporary files; *dir_path* must be a full pathname. To define a set of directories using entries in the *rbw.config* file, enter multiple lines. The order in which the directories are specified has no effect because the order in which they are used is random (determined internally) and no user control is possible.

UNIX The default directory is */tmp*.

Windows NT The default directory is *c:\tmp*.

THRESHOLD value

Specifies the amount of memory used before writing intermediate results from an index-building operation to disk. For operations involving multiple indexes, this threshold value is allocated equally among the indexes being built.

The size must be specified as kilobytes (K) or megabytes (M) by appending K or M to the number. Note that no space is allowed between the number and the unit identifier (K, M). For example: 1024K, 500M.

The threshold value must be specified before the corresponding MAXSPILLSIZE value is specified; it must precede the MAXSPILLSIZE entry in the *rbw.config* file.

A value of 0 causes files to be written to disk after the first 200 rows or index entries.

The default threshold for index-building operations is 10 megabytes (10M)

MAXSPILLSIZE size

Specifies the total maximum amount of temporary space per operation. For an operation involving multiple indexes, this space is allocated equally among the indexes being built.

The size must be specified as kilobytes (K), megabytes (M), or gigabytes (G) by appending K, M, or G to the number. Note that no space is allowed between the number and the unit identifier (K, M, G). For example: 1024K, 500M, 8G.

The default MAXSPILLSIZE value is 1 gigabyte (1G). The maximum MAXSPILLSIZE value is 2047 gigabytes.

RESET

Resets the index-building TEMPSPACE parameters to the values specified in the *rbw.config* file. If a parameter name is not specified, all index-building TEMPSPACE parameters are reset to their default values for this session.

Usage Notes

In addition, use the following guidelines when setting index-building temporary space parameters:

- Always set the THRESHOLD value before setting the MAXSPILLSIZE value.
- Remember that the INDEX TEMPSPACE parameter settings in the *rbw.config* file affect not only TMU index-building operations but also SQL index-building operations.

Examples

The following examples illustrate SET commands that can be used to change parameters for a specific session:

```
SET INDEX TEMPSPACE THRESHOLD 2M;  
SET INDEX TEMPSPACE MAXSPILLSIZE 3G;
```

UNIX

```
SET INDEX TEMPSPACE DIRECTORIES '/disk1/itemp',  
                                '/disk2/itemp';
```

Windows NT

```
SET INDEX TEMPSPACE DIRECTORIES 'd:\itemp', 'e:\itemp';
```

The following example illustrates how to reset the INDEX_TEMPSPACE parameters to the values specified in the *rbw.config* file:

```
SET INDEX TEMPSPACE RESET;
```

The following examples illustrate entries in the *rbw.config* file that apply to all sessions:

```
TUNE INDEX_TEMPSPACE THRESHOLD 20M  
TUNE INDEX_TEMPSPACE_MAXSPILLSIZE 8G
```

UNIX

```
TUNE INDEX_TEMPSPACE_DIRECTORY /disk1/itemp  
TUNE INDEX_TEMPSPACE_DIRECTORY /disk2/itemp  
TUNE INDEX_TEMPSPACE_DIRECTORY /disk3/itemp
```

Windows NT

```
TUNE INDEX_TEMPSPACE_DIRECTORY d:\itemp  
TUNE INDEX_TEMPSPACE_DIRECTORY e:\itemp  
TUNE INDEX_TEMPSPACE_DIRECTORY f:\itemp
```

Parallel Processing Tasks (PTMU Only)

As data is loaded by the PTMU, it can use multiple tasks (even on a single CPU) to perform the data conversion and index-building portions of the load operation; you can control the amount of parallel processing for both data conversion and for index-building, based on your site resources and workload requirements. For more information about how parallel processing is used to load data, refer to “Data Processing During Load Operations” on page 3-4.

Note: The PTMU is not available for Windows NT systems.

Syntax

To set the PTMU parallel processing parameters for a single session, enter either or both SET statements in the PTMU control file; for all sessions, edit the TUNE parameters in the *rbw.config* file. The syntax is as follows:

```

▶———— SET TMU CONVERSION TASKS ——— num_tasks — ;————▶
▶———— SET TMU INDEX TASKS ————— num_tasks — ;————▶

▶———— TUNE TMU_ CONVERSION_ TASKS — num_tasks —————▶
▶———— TUNE TMU_INDEX_ TASKS ————— num_tasks —————▶

```

TMU CONVERSION TASKS

Tasks that convert input data to the platform-based internal format used to represent data; these tasks also ensure uniqueness and check referential integrity whenever such checks are performed. The number specified is the actual number of tasks used. The default value is one-half the number of processors on the machine (as determined from the hardware).

TMU INDEX TASKS

Tasks that make index entries into non-unique indexes, corresponding to the data being loaded. Each non-unique index can have at most one task associated with it; the number specified with this parameter is the maximum number of tasks that can be used to process all non-unique indexes. The actual number of tasks used will be the smaller of the number of non-unique indexes and the number specified with this parameter. The default value is one task per non-unique index.

Use this parameter if you want to use fewer index tasks than provided by the default value.

Note: The task that makes the entries into unique indexes is not affected by this parameter.

num_tasks

Integer that indicates the number of tasks.

Example

The following examples illustrate how to use the SET commands and TUNE entries.

To control parallel processing for a single PTMU session within a control file:

```
set tmu conversion tasks 5;
set tmu index tasks 8;
```

To control parallel processing for all PTMU sessions using the *rbw.config* file:

```
TUNE tmu_conversion_tasks 5
TUNE tmu_index_tasks 8
```

Example

To illustrate how the TMU CONVERSION TASKS parameter works, assume there are 8 processors on the system. By default, 4 of them will be used for conversion tasks. If you want to use more than 4 processors—perhaps the system is lightly loaded—set the TMU CONVERSION TASKS parameter to a number larger than 4 to increase the number of processors.

Example

To illustrate how the TMU INDEX TASKS parameter works, assume there are 5 non-unique indexes and TMU INDEX TASKS is set to 3. In this case, 3 tasks will be used, and some tasks will process multiple indexes in parallel. If there were 5 non-unique indexes and TMU INDEX TASKS were set to 6, 5 tasks would be used, one per non-unique index.

Serial Mode Operation (PTMU Only)

The PTMU can be forced to run in a serial mode in which no parallel processing is used; in this case, the PTMU is effectively running as the TMU. This capability is useful in cases where you do not want the resource consumption and overhead of parallel processing. You could use the TMU instead of the PTMU, but the ability to run the PTMU in serial mode allows you to combine operations in a single control file to be executed by the PTMU. Within the control file, you specify those operations that are to be executed in serial mode, with all other operations to be executed in parallel mode.

Note: The TMU SERIAL MODE parameter affects only those operations for which the PTMU uses parallel processing; it does not affect those operations for which the PTMU normally uses serial processing, as defined on page 2-22.

To control PTMU serial mode for a single session, enter a SET statement in the PTMU control file; for all sessions, edit the TUNE parameters in the *rbw.config* file. The syntax is as follows:

```

▶ SET TMU SERIAL MODE [ OFF | ON ] ;
▶ TUNE TMU_SERIAL_MODE [ OFF | ON ]

```

Example

Suppose you have a PTMU control file that performs multiple operations. Most operations are to be done in parallel, but you want the following operations to be performed in serial mode:

- Loading some small dimension tables where the overhead of parallel processing causes the operation to actually take longer in parallel mode than in serial mode.
- Loading a very large table that would, in parallel mode, preclude reasonable performance for other users, whereas in serial mode, the operation would complete in the required time and not impact other users.

By using the TMU SERIAL MODE parameter, you can combine all the operations in a single file to be processed by the PTMU, which will switch between parallel and serial mode as directed by the SET SERIAL MODE command included wherever needed in the control file.

Format of Datetime Values in TMU Statements

If you want to use an alternative date format (not ANSI SQL-92 datetime format) for a date constant specified in a TMU statement, you must use a TMU SET DATEFORMAT statement to specify the input format of such date when the format includes numeric month values and the default order of *mdy* is not used. (Examples of where you can use a date constant are in a LOAD DATA statement to load a constant into a date column or to specify a date value in an ACCEPT/REJECT clause or in an UNLOAD statement in a WHERE clause.) The SET DATEFORMAT statement must precede the LOAD DATA or UNLOAD statement that it applies to in the control file.

The following diagram shows the syntax for this statement.

►—— SET —— DATEFORMAT — '*format*' —;—►

format

Specifies the order of month, day, and year components for non-ANSI SQL-92 (alternative) date inputs, using a combination of the characters *m*, *d*, and *y*. The default format is *mdy*. This SET command uses the same format combinations as the SQL SET command; for more information about formats, refer to the SET DATEFORMAT command in the *SQL Reference Guide*.

Example

The following statement specifies that any constants in the TMU statement that follows will be assumed to be in *ymd* format (year, month, day); for example: 1997/11/30:

```
SET DATEFORMAT 'ymd' ;
```

The following statement specifies that any constants in the TMU statement that follows will be assumed to be in *dmy* format (day, month, year); for example: 30/11/1997:

```
SET DATEFORMAT 'dmy' ;
```


Load Information Limit

Information about each load operation is stored in the RBW_LOADINFO system table, one row per operation. The RBW_LOADINFO_LIMIT configuration parameter specifies the maximum number of rows that can be stored in that table, thereby allowing you to control the amount of historical load information recorded by the system.

The RBW_LOADINFO_LIMIT parameter can be specified only as an entry in the *rbw.config* file—there is no equivalent SET command—and it applies to all TMU sessions. The following diagram shows the syntax for this entry:

► DEFAULT — RBW_LOADINFO_LIMIT ——— value ———►

value

Specifies the number of rows of information to be stored. There is no upper limit on the value specified; however, it should be set to a reasonable number. The default value is 256 rows.

2

Usage Notes

- Setting this parameter to a value less than the current value causes the RB_DEFAULT_LOADINFO file to be truncated; however, the original file is saved as RB_DEFAULT_LOADINFO.save.
- The current value of RBW_LOADINFO_LIMIT is stored in the RBW_OPTIONS system table.

Example

To check the value of RBW_LOADINFO_LIMIT, query the RBW_OPTIONS table:

```
select option_name, value from rbw_options where option_name =
'RBW_LOADINFO_LIMIT';
```

OPTION_NAME	VALUE
RBW_LOADINFO_LIMIT	256

Suggestions for Effective PTMU Operations

After you invoke the PTMU, its use is the same as the use of the TMU, except as discussed in the following sections that apply only to the PTMU.

Note: The PTMU is not available for Windows NT.

Operations That Use Parallel Processing

The PTMU uses parallel processing for some operations and serial processing for others:

- Parallel processing is used only for online LOAD operations in INSERT, APPEND, and REPLACE mode.
- Serial processing is used for all the following operations:
 - LOAD operations in MODIFY or UPDATE mode
 - Offline LOAD operations
 - SYNCH, REORG, UNLOAD, BACKUP, RESTORE, and UPGRADE operations

If you are using the PTMU to load data to take advantage of parallel processing, do not use MODIFY mode when you can use APPEND or INSERT mode. For example, if you are loading data into an empty table, use INSERT mode instead of MODIFY mode; and if you are adding new rows to an existing table but not modifying any existing rows, use APPEND mode instead of MODIFY mode.

Discard Limits on Parallel Load Operations

Because the PTMU processes multiple input rows at the same time, its behavior might be different from the TMU when a load operation reaches the discard limit and terminates early. For example, if input row 500 is discarded, causing the maximum discard limit to be exceeded, the PTMU might already be processing rows 501, 502, and so on. If these rows are in the process of being discarded, messages will appear for these rows and they will appear in the discard file (if specified) even though they are beyond the limit. Any rows beyond the discard limit that have been inserted into the table will be removed before the load operation terminates.

The same behavior occurs if the load terminates prematurely for other similar reasons, such as exceeding the MAXROWS PER SEGMENT limit on the target table.

AUTOROWGEN with the PTMU

The Parallel TMU supports automatic row generation during parallel load operations; however, automatic row generation reduces the amount of parallelism that can be applied to the operation because the referential integrity checking and row generation must all be done by a single process. For the best performance, do not use automatic row generation for parallel loads; the decrease in performance can be significant.

If you are going to load a large amount of data that is expected to have only a few rows that will fail referential integrity checking, load the data specifying AUTOROWGEN OFF and naming a discard file. Then load the rows in the discard file specifying AUTOROWGEN ON. Note, however, this strategy is not appropriate if there will be a large number of discarded rows because the discard processing will significantly slow the load processing.

2

Multiple Tape Drives with the PTMU

Note: This capability is not available on all UNIX systems.

If you are using the PTMU with multiple tape drives (such as 8-mm drives), you can load data in sequence into a warehouse database without operator intervention. Include the following clause in the control file to specify the filename and each device name:

```
INPUTFILE 'filename' TAPE DEVICE 'device_name[,...]'
```

Examples

The following example illustrates how to describe and load multiple tape drives for the PTMU.

The following line in the control file loads data from tape devices *tx0* and *tx1*:

```
INPUTFILE 'myfile' TAPE DEVICE '/dev/rmt/tx0,/dev/rmt/tx1'
```

If the PTMU loads all the data from *tx0* and then from *tx1* and still has not reached the end of file, it pauses and requests that the next tape be loaded. The next tape should be loaded in the tape device *tx0*.

3480/3490 Multiple-Tape Drive with the PTMU

Note: Support for this tape drive is not available on all UNIX systems.

If you are using the PTMU with the 3480 or 3490 multiple-tape drive, include the following clause in the control file, specifying a device name and a range of cartridges for each tape device:

```
INPUTFILE 'filename' TAPE DEVICE 'device_name[(start-end)][,...]'
```

Example

The following line in the control file loads data from 3480/3490 tape devices *tf0* and *tf1*:

```
INPUTFILE myfile' TAPE DEVICE '/dev/rmt/tf0(1-3), /dev/rmt/tf1(1-3)'
```

If, after loading all the data on *tf1*, the PTMU has not reached the end of file, it pauses and requests that the next tape be loaded.

Data is loaded in the following order:

Load Order	Tape Device	Cartridge
Tape 1	tf0	1
Tape 2	tf1	1
Tape 3	tf0	2
Tape 4	tf1	2
Tape 5	tf0	3
Tape 6	tf1	3
Operator prompted for new cartridges		
Tape 7	tf0	1
Tape 8	tf1	1
Tape 9	tf0	2
...

Loading Data into a Warehouse Database

Data is loaded into a warehouse database with the Table Management Utility (TMU) and a control file containing a LOAD DATA statement.

This chapter contains the following sections:

- The LOAD DATA Operation
- Procedure for Loading Data
- Some Preliminary Decisions
- Writing a LOAD DATA Statement:
 - Input Clause
 - Format Clause
 - Locale Clause
 - Discard Clause
 - Optimize Clause
 - Table Clause
 - Segment Clause
 - Criteria Clause
 - Comment Clause
- Fieldtypes
- Datetime Format Masks for Datetime Fields
- Restricted Datetime Masks for Numeric Fields
- Writing a SYNCH Statement
- Format of Input Data
- Fieldtype Conversions
- LOAD DATA Syntax Summary

The LOAD DATA Operation

Before you can load data, the database and the tables to be loaded must already exist. Databases are built with a utility program (*rb_creator* on UNIX systems and *dbcreate* on Windows NT systems) and the user tables are defined with SQL CREATE TABLE statements. Primary key indexes are built automatically during the load process for each table that has a primary key, as are any existing user-defined indexes. For information about defining tables and indexes, refer to the *Warehouse Administrator's Guide* and the *SQL Reference Guide*.

Inputs, Outputs, and Options

Input to the TMU for a data loading operation consists of:

- A LOAD DATA control file.
- Tape or disk files containing the input data records.

Output from a data loading operation consists of:

- Table and index files that are part of the database.
- Status, error, and warning messages.
- One or more discard files (in ASCII or EBCDIC format) containing records or rows that could not be loaded because of format errors, data integrity violations, or referential integrity violations.

The TMU also updates various Red Brick Warehouse system tables, which contain the data format descriptions, table and index files, and other information needed by the TMU and the warehouse server.

The TMU automatically creates indexes on all primary keys, based on table definitions. It also builds and updates any additional user-created indexes that exist at the time of the load operation.

If the data is loaded into an offline segment, the data loading operation must be completed with a SYNCH SEGMENT statement, which synchronizes the newly loaded segment with the table and its indexes.

Auto Aggregate Option

The TMU Auto Aggregate option provides the capability to automatically and selectively aggregate new input data with the data already in a table. If your site has purchased the Auto Aggregate option, separate instructions are provided that describe how to enable it.

Automatic Row Generation Feature

The TMU Automatic Row Generation feature is useful for rapid loading of non-standard or “dirty” data. This option provides the following alternatives to discarding an input row that violates referential integrity:

- Generating and adding any new rows needed to preserve referential integrity into the referenced tables and then adding the original input row into the referencing table instead of discarding it.
- Loading the new row into the table being loaded after replacing values that violate referential integrity with default values that are already present in specified referenced tables.
- Combining these two behaviors on a table-by-table basis for each referenced table, sometimes adding new rows to referenced tables and sometimes loading modified rows into the table being loaded.

Enterprise Control and Coordination Option

The Enterprise Control and Coordination option and the Copy Management option both provide a copy management utility (*rb_cm*) that can be used to move data between different databases located throughout an enterprise.

The Enterprise Control and Coordination option also provides access to comment information related to each load operation that is stored in the RBW_LOADINFO system table. For more information about the Enterprise Control and Coordination option, refer to the *Warehouse Administrator's Guide*.

If your site has purchased either of these options, separate instructions are provided that describe how to enable it.

Data Processing During Load Operations

When data is loaded into a database from an input file, the load process includes several processing stages. By understanding what activities occur during each stage, you are better able to avoid bottlenecks and resource conflicts, thereby reducing the time required to load data.

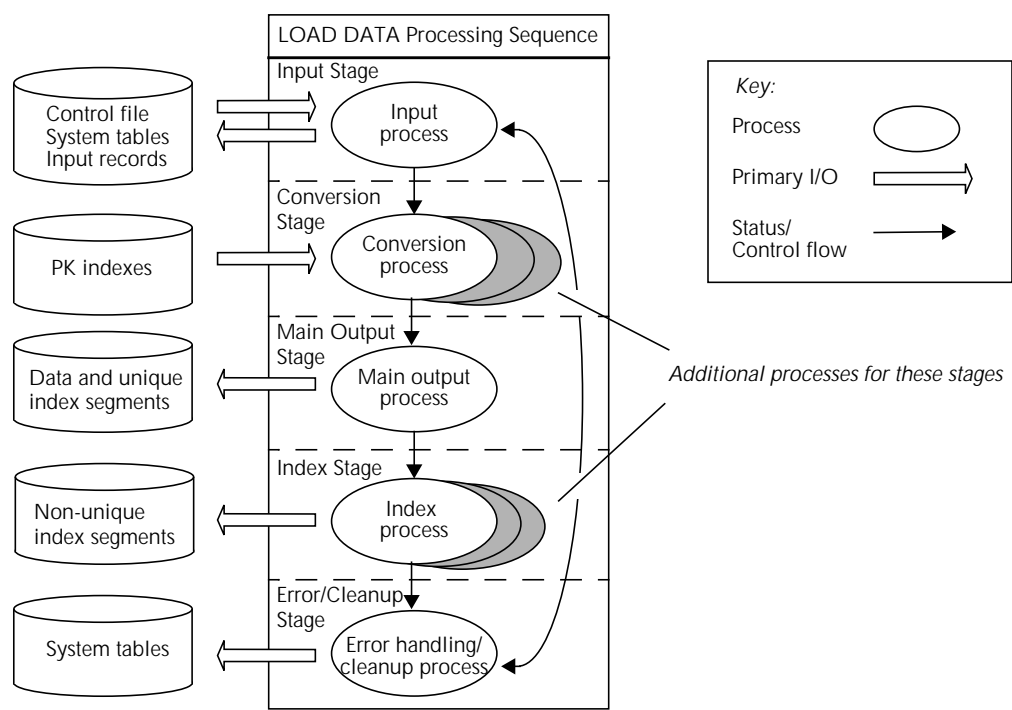
A load operation consists of the following stages.

- Input stage
 - Validates syntax of TMU control statement.
 - Locks table(s) and segment(s).
 - Reads input records, monitoring progress and status communicated by error handling/cleanup stage.
 - For PTMU, sets up additional processes for conversion and index stages.
- Conversion stage
 - Converts input records to internal row format and validates data.
 - Checks referential integrity (if Automatic Row Generation is off).
- Main output stage
 - Checks referential integrity (if Automatic Row Generation is on).
 - Writes row data to table.
 - Writes entries to unique indexes and verifies uniqueness.
- Additional index stage for non-unique indexes
 - Writes entries to non-unique indexes.
- Error handling and cleanup stage
 - Handles error processing.
 - Finishes building indexes.
 - Communicates progress and status back to input stage.

The TMU uses a single process that controls all stages; it processes small batches of rows, passing one batch through each stage before starting the next batch. The PTMU with its parallel processing capability improves performance in two ways:

- It uses separate processes for each stage, creating a pipeline in which batches of rows are passed from one stage to the next, with multiple batches of rows being processed simultaneously. Even on systems with a single CPU, multiple processes can take advantage of I/O and CPU overlap, which might result in reduced load time.
- On systems with multiple CPUs, the PTMU further improves the pipeline throughput by creating additional conversion and output processes. (The number of additional processes is user-definable.)

The following figure illustrates the sequence of stages in a load operation and the additional parallelism provided by the multiple processes of the PTMU. (The TMU uses a single process for all stages.)



Input Stage

During the input stage, the syntax of the LOAD DATA statement is checked and the table or segment is locked for exclusive use. You can specify whether you want the TMU to wait for a lock or to return immediately if the table is in use.

Conversion Stage

During the conversion stage, any necessary data conversion on each record is done, including conversion between character sets—for example, EBCDIC to ASCII or MS932 to EUC; conversion from the external character set to internal (binary) format; and decimal scaling. In this stage, referential integrity is checked (if Automatic Row Generation is off) and the data is validated by comparing it with the column datatype and checking for truncation, underflow, and overflow. The PTMU uses multiple conversion processes so performance improvements are significant when there is much conversion work to be done.

Main Output and Index Stages

During the main output stage, data is written to the table and entries are made in all unique indexes. In addition, if Automatic Row Generation is on, then referential integrity checks are also done during this stage, and any automatically-generated rows are inserted into referenced tables. In this stage, the PTMU uses only a single process to make all entries into each unique index.

During the index stage, entries are made into any non-unique indexes. The PTMU, by default, uses one index process per non-unique index, thereby speeding up this part of the load operation.

Error Handling/Cleanup Stage

The error handling/cleanup stage performs the error handling, which includes keeping track of rows loaded in case of interrupts, and cleans up after the processing has completed. It also monitors progress through the pipeline, providing feedback to the input stage to control the flow of records being processed. The PTMU uses only a single process for this stage.

Procedure for Loading Data

To load data into the tables of a warehouse database, use the following general procedure:

1. Determine the order in which the tables are going to be loaded, as described on page 3-9.
2. Determine whether you are going to load ordered or unordered data and order the data if desired, as described on page 3-9.
3. Determine the following information about your input data:
 - Source of your input data (disk, TAR or standard label tape, or standard input)
 - Record length (fixed or variable)
 - Record format (fixed or separated)
 - Record field order and type
 - Mapping between input fields and table columns
 - Character set (Locale; ASCII or EBCDIC)

The description of the input data is supplied in the Input clause, as described on page 3-20. Additional information about file and record formats is provided on page 3-106 and about datatype conversions from the input data to the server datatypes on page 3-111.

4. Determine the load mode to be used: APPEND, INSERT, MODIFY, REPLACE, or UPDATE. If you use MODIFY or UPDATE mode, determine whether to use the AGGREGATE option. The load mode is specified in the Format clause, as described on page 3-24.
5. Determine whether you want to use Automatic Row Generation to ensure that no rows are discarded for referential integrity violations, for some or all of the referenced tables, as described on “Maintaining Referential Integrity with Automatic Row Generation” on page 3-10. Also decide whether you want to use separate discard files for records discarded for data integrity and referential integrity violations. These choices are specified in the Discard clause, as described on page 3-33.
6. Determine whether you want to use optimize mode to load the data. Optimize mode is selected in the Optimize clause, as described on page 3-45.
7. Write the LOAD DATA control statements, one per table, in a file. Note that a single file can contain multiple control statements of different types.
8. If you are loading data into a segment of a table, determine whether you want to use an offline load operation. If you do, take the segment offline with an ALTER SEGMENT operation.
9. Run the TMU with a control file containing the LOAD DATA statements.

10. If you loaded data into an offline segment, synchronize the segment by running the TMU with a SYNCH statement, as described on page 3-104, and then bring the segment online with an ALTER SEGMENT operation.
11. To check the status of a load operation on a specific table, query the RBW_LOADINFO table. (Note that the Comment column contents are displayed only if the Enterprise Control and Coordination option is enabled.) For example:

```
select substr(tname,1,10) as tname, substr(status, 1,10) as
status, started, finished, substr(comment,1,10) as comment
from rbw_loadinfo where tname = 'SALES';
```

TNAME	STATUS	STARTED	FINISHED	COMMENT
SALES	COMPLETE	1997-08-26 01:48:54	1997-08-26 01:49:25	NULL

This chapter provides the information needed to write the LOAD DATA statements, the field specifications within the LOAD DATA statements, and the SYNCH statement for offline load operations.

For examples that illustrate how to convert data files from existing RDBMSs for use with Red Brick Warehouse, refer to Appendix B, “Loading Data from Existing Systems.”

Some Preliminary Decisions

Before you write the LOAD DATA statements to load data into the tables of your database, you should make the following decisions, which affect how you write the statements:

- In what order the tables will be loaded.
- Whether the input data is or should be ordered.
- Whether to use automatic row generation to maintain referential integrity.

Each of these topics is discussed in the following sections.

Determining Table Order

The TMU loads tables in the order of the LOAD DATA statements in the control file; each LOAD DATA statement corresponds to one table. To control the order in which tables are loaded, place the LOAD DATA statements in the file in the desired order.

Tables can be loaded in any order as long as any table referenced by a foreign key is loaded before the table containing that foreign key; that is, a referenced table must be loaded before the table that references it. For example, if the Sales table—the referencing table—contains three foreign keys, each of the three tables referenced by the foreign keys must be loaded before the Sales table can be loaded.

3

Ordering Input Data

You must decide whether to order the data in the input files, balancing any improvement in load time against the amount of time available to load data, the time spent ordering input data, and the difficulty of maintaining ordered data.

The initial load of input data into a table is somewhat faster with ordered data. However, for incremental loads of data into indexed tables, the optimized load mode makes data-order issues unimportant. With more than one STAR index, a combination of primary key and STAR indexes, or references to multi-column primary keys, it is usually not useful to attempt to order data.

If you want to order the data for an initial load, order the data for each referenced table by the primary key values. If you have a single STAR index on the referencing table, you might want to order the data in the key order of the STAR index definition, which can result in a more efficient index. To order the input data based on a single STAR index on the referencing table, order it so

that the data in the foreign key column(s) named first in the CREATE STAR INDEX statement changes the most slowly. Data in the foreign key column(s) named next changes the next most slowly, and so on. The order of data in each foreign key column must match the order of data in the corresponding primary key in the referenced tables. If you use multiple STAR indexes, then the difficulty of choosing an order for the input data increases and the benefits are reduced.

Within a single column in the key, data can be in any arbitrary order, provided that the order within that column is the same as the order in the corresponding primary key in the referenced table. For example, if the input data for the foreign key column of the referencing table is in descending sort order, the input data for the corresponding column in the referenced table must also be in descending sort order.

Maintaining Referential Integrity with Automatic Row Generation

The Automatic Row Generation option (AUTOROWGEN) allows the TMU to add any rows needed to preserve referential integrity. If this option is OFF, the TMU discards records that violate referential integrity; however, this behavior can be both time-consuming and frustrating in situations where the data being loaded is dirty, unfamiliar, or incomplete. This option offers the following alternatives, in addition to discarding rows, to maintain referential integrity:

- Generating and adding new rows to the referenced tables (ON mode).
- Modifying the row to be added to the table being loaded—the target table—by using a column default value for one or more of the foreign keys (DEFAULT mode).
- Combining these actions within a single load operation with a mixed-mode operation.

This flexibility allows you to choose how you want to maintain referential integrity on a table-by-table basis within a single load operation. Tables are locked automatically, for either read or write access as needed, at the beginning of the load operation.

The AUTOROWGEN option can be set for ON and OFF mode in the *rbw.config* file or in the Discard clause of a LOAD DATA statement; however, it can be set for DEFAULT or mixed-mode operation only in the Discard clause.

Discarding Records That Violate Referential Integrity

If the AUTOROWGEN option is OFF (the default behavior), all records that violate referential integrity are discarded or written either to the standard discard file or to files designated for referential integrity violations. (Separate files can be designated for violations of each referenced table.)

Adding Generated Rows to Referenced Tables

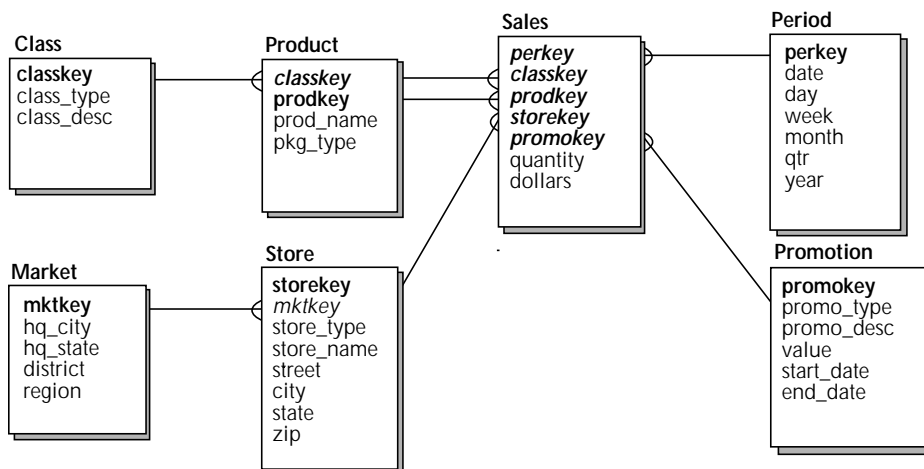
If the AUTOROWGEN option is ON, whenever an input row in the table being loaded contains a value in a foreign key column that is not present in the primary key column of the referenced table, a row is generated and added to the referenced table before the input row is added to the target table. This behavior cascades through any outboard tables that are in turn referenced by the referenced table.

In this mode, the referenced tables grow as rows are inserted into them; and if a table grows beyond its MAXROWS PER SEGMENT value, a REORG operation might be required on STAR indexes built on these foreign key columns.

The generated rows get their values from default values defined for each column when the table was created.

Example: AUTOROWGEN ON

This example illustrates the AUTOROWGEN ON option, which adds rows to the referenced table. Assume you are the warehouse administrator for the following database. (**Bold text indicates primary keys; *bold italic* indicates foreign keys.**)



The Sales table contains daily total sales for products sold in a chain of retail stores. Because all managers have authority to order goods for their own stores, frequently when you load the daily sales data, new products appear for which there are no supporting entries in the Product table. Your operations run more smoothly when you can complete the nightly load and go back the next day to complete the entries for these new items.

The Product table is defined as follows:

```
create table product (  
    classkey integer not null,  
    prodkey integer not null,  
    prod_name char(30) default 'new product',  
    pkg_type char(20),  
    constraint prod_pkc primary key (classkey, prodkey),  
    ...;
```

Each manager has a range of Prodkey values to assign to new products within the defined classes.

The LOAD DATA statement for the daily load operation on the Sales table sets the Automatic Row Generation option to ON.

```
load data  
    inputfile 'sales.txt'  
    recordlen 86  
    insert  
    discardfile 'sales_disc.txt'  
    autorowgen on  
    ...
```

When a record containing the sales dollars for a brand new product is encountered during the load process, the TMU inserts the record in the Sales table and adds a row containing the new Prodkey value into the Product table, filling in that row with any specified default values or NULL.

Assume the following record is encountered in the load process:

```
...:01/96/d22:7:789:78:...:236:...  
    |      |      |      |  
    Perkey |      | Storekey | Dollars  
           |      | New Prodkey value  
           Classkey
```


The value 789—which was assigned by a store manager who added a new item at his store—does not appear in the primary key of the Product table. The Automatic Row Generation option allows the TMU to insert the above information into the Sales table after adding the following row to the Product table:

classkey	prodkey	prod_name	pkg_type
7	789	new product	NULL

The task of replacing the default values with real values remains, but now the data is loaded and analysis can proceed. You can find any new products that were added by using a SELECT statement of the form:

```
select prodkey, product from product
where product = 'new product';
```

You now need to track down the missing information and update the Product table entry.

Modifying the Input Rows

If the Discard clause specifies AUTOROWGEN DEFAULT mode for a list of referenced tables, whenever an input row contains a value in a foreign key column that is not present in the primary key column of a referenced table in the list, the row is first modified by replacing the missing value with the default value for the foreign key column, and then it is added to the target table. In this mode, referenced tables in the list do not grow. This mode is useful for data that contains unknown values in foreign key columns that are not of critical importance to the application. It is also useful in cases where you do not want a referenced table to grow to exceed the MAXROWS PER SEGMENT value.

Example: AUTOROWGEN DEFAULT

This example illustrates the AUTOROWGEN DEFAULT mode in which referential integrity is preserved by modifying the input rows before they are loaded; it is based on the database in the previous example on page 3-11. Assume the Sales table is defined as follows, with a default value assigned to the Prodkey column:

```
create table sales (  
    perkey integer not null,  
    classkey integer not null,  
    prodkey integer not null default 0,  
    storekey integer not null,  
    promokey integer not null,  
    quantity integer,  
    dollars dec(7,2),  
    constraint sales_pkc primary key (perkey, classkey, prodkey,  
        storekey, promokey),  
    ... ;
```

Assume the load operation occurs on the Sales table with AUTOROWGEN DEFAULT mode specified for the Product table; note that for all other tables referenced by the Sales table, the default behavior is OFF mode.

```
load data  
    inputfile 'sales.txt'  
    recordlen 86  
    insert  
    discardfile 'sales_disc.txt'  
    autorowgen default (product)  
    ...
```

The following records, which contain Prodkey values not present in the Product table, are encountered in the load process:

...:01/96/d22:7:**789**:78:45:36:236.56:...
...:01/96/d22:7:**790**:78:46:42:168.72:...
...:01/96/d22:7:**791**:78:46:143:937.25:...

| Perkey | | | | Quantity Dollars
| | Storekey Promokey
| New Prodkey value
Classkey

No changes will be made to the Product table, but the first two records will be added to the Sales table as:

perkey	classkey	prodkey	storekey	promokey	quantity	dollars
1996-01-22	7	0	78	45	36	236.56
1996-01-22	7	0	78	46	42	168.72

Note, however, that the third record will be discarded—because its primary key value will be identical to that of the previous record. Any records that violate referential integrity with respect to any referenced tables other than the Product table will be discarded.

Adding Rows in Mixed Mode

The `AUTOROWGEN` option also allows you to combine the `ON` and `DEFAULT` behaviors in a mixed-mode operation. To combine behaviors, you must use the Discard Clause of a `LOAD DATA` statement to specify on a table-by-table basis whether rows should be added to the referenced or referencing table.

Note: Potential conflicts might arise in mixed-mode operation when a referenced table appears in a DEFAULT mode table list and it is also referenced by another table that appears in an ON mode table list. For an example of this behavior, refer to “Usage Notes” on page 3-42.

Example

This example illustrates the AUTOROWGEN mixed-mode operation; it is based on the database in the previous example on page 3-11. Assume the LOAD DATA statement for the Sales table contains the following Discard clause:

```
...  
discardfile 'sales_dscd' discards 100  
autorowgen on (store, promotion) default (product)  
...
```

As records are loaded into the Sales table, rows will also be added to the Store and Promotion tables as needed to maintain referential integrity; rows will also be added to the Market table if necessary because it is an outboard table referenced by the Store table. However, if a record to be loaded contains a foreign key value not found in the Prodkey column of the Product table, the record will be added to the Sales table using the default value (0) for the Sales table's Prodkey column. And if a record to be loaded into the Sales table contains a Perkey value not found in the Perkey column of the Period table, it will be discarded because the Period table is not in either table list and hence is controlled by AUTOROWGEN OFF mode.

Specifying the AUTOROWGEN Mode

The default behavior for a warehouse database is controlled as a system default by an entry in the *rbw.config* file; the syntax is as follows:

► OPTION AUTOROWGEN — **OFF** —————►
 └ ON ———┘

The AUTOROWGEN option also can be set for a specific load operation in the Discard clause of a LOAD DATA statement, as described in “Discard Clause” on page 3-33. Setting this option in the Discard clause provides more flexibility because you can specify the behavior (ON, OFF, DEFAULT) for each table referenced by the table being loaded.

Writing a LOAD DATA Statement

The LOAD DATA statement for the TMU specifies, in this order:

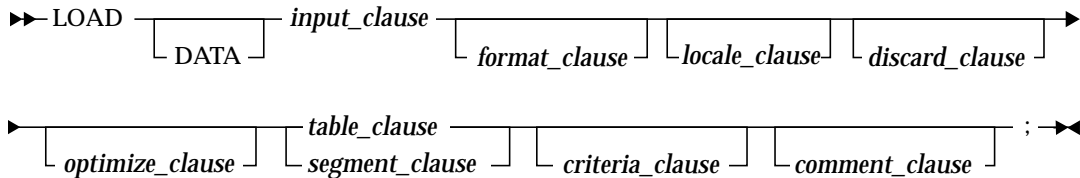
1. The files containing your input data, which can be tape files, disk files, or standard input.
2. The format of the input data (format specification).
3. The locale of the input data, if different from the database locale.
4. Optional discard instructions, which can include filenames and formats for discarded records and the automatic row generation option.
5. Optional optimization instructions that specify whether to build indexes in optimize mode and a discard file for discarded records.
6. The table into which the data is to be loaded and a mapping of data fields into table columns. Alternatively, an offline segment of a table can be specified.
7. Optional criteria that determine which input records should be loaded and which should be discarded.
8. Optional comment text that allows you to store information about a load operation or the data loaded. This information can be accessed only if the Enterprise Control and Coordination option is enabled.

Each LOAD DATA statement loads only one table. It can load data into all of the columns in a table or into a subset of the columns. The names of the columns to be loaded are specified together with a description of the source data, which is called a *field specification*. A field specification contains information about the datatype of the data field within the input record if the data is in an input file, or it contains information about the automatically generated input data if the TMU must produce the data.

A control file can contain multiple LOAD DATA statements for multiple tables, which are processed sequentially.

LOAD DATA Syntax

The syntax of the LOAD DATA statement is:



The LOAD DATA statement must end with a semicolon.

Note: The LOAD DATA statement syntax is repeated throughout this chapter to provide a point of reference for the detailed syntax for each clause.

Example

This example illustrates a LOAD DATA statement for a UNIX-based system that reads a fixed-format tape input file named *market.txt*, which uses a different locale than the warehouse uses, and stores the data in a table named Market, which has two columns, Mktkey and State.

```

Input clause  load data
               inputfile '/db/inputs/market.txt'
Format clause  tape device '/dev/rmt/0mn'
Locale clause  recordlen 7 replace
Discard clause discardfile '/db/aroma/discards'
               discards 1
Optimize clause optimize on discardfile '/db/aroma/mktdups.txt'
Table clause  into table market(
               mktkey integer external (4),
               state char(2)
               );

```

The LOAD DATA statement for a Windows NT system is similar except for file naming conventions:

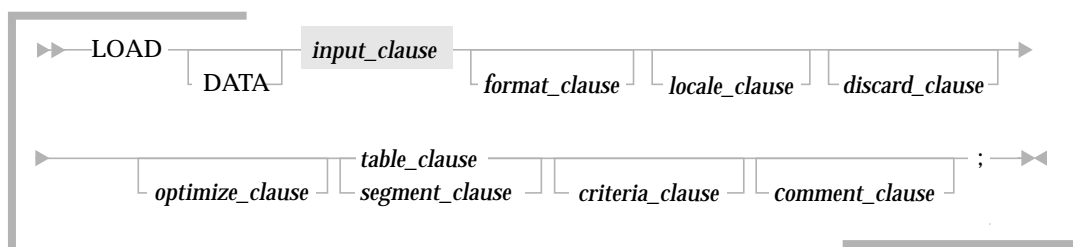
```
Input clause  load data
Format clause inputfile 'c:\db\inputs\market.txt'
Locale clause recordlen 7 replace
Discard clause nls_locale 'English_Canada.MS1252@Default'
              discardfile 'c:\db\aroma\discards'
Optimize clause discards 1
Table clause  optimize on discardfile 'c:\db\aroma\mktdups.txt'
              into table market(
              mktkey integer external (4),
              state char(2)
              );
```

The Input clause, Format clause, Locale clause, Discard clause, Optimize clause, Table clause, Criteria clause, and Comment clause are described in detail in the following sections. For convenient reference, a syntax summary for the LOAD DATA statement is shown at the end of this chapter.

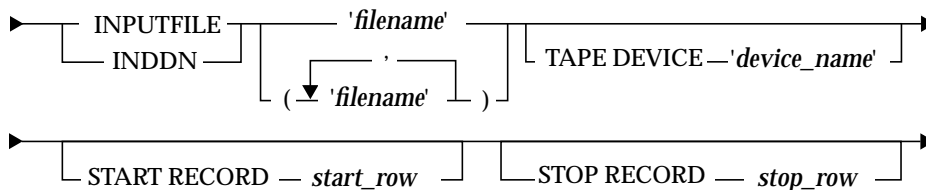
Input Clause

The TMU accepts input from tape drives, disk drives, and system standard input. The Input clause specifies the file or files containing the input data, the input device (for tape drives), and record numbers (for partial loads). Within a single LOAD DATA statement, files must be all tape files, all disk files, or all standard input.

The entire LOAD DATA statement syntax diagram is repeated here to provide a point of reference for the Input clause syntax:



The syntax for *input_clause* is:



INPUTFILE '*filename*'

Specifies the name of the file that contains the input data to be loaded into the table. The filename must satisfy operating system conventions for file specification. Note that the name must be enclosed in single quotes. A list of filenames must be enclosed in parentheses, with commas between each filename. (INDDN can be used instead of INPUTFILE.)

If input is standard input, the filename reference is '-':

```
INPUTFILE '-'
```

Note: If the LOAD DATA statement appears in a control file for the *rb_cm* copy management utility, INPUTFILE must be set to standard input.

If multiple input files are used, the list of names must be enclosed in parentheses and the names must be separated by commas:

```
load data
  inputfile ('market1.txt', 'market2.txt', 'market3.txt')
  start record 100
  ...
```

A filename can contain environment variables:

UNIX INPUTFILE '\$INPUT/file.1'

If a \$ character is part of the filename, it must be preceded by a backslash (\) character so it is not confused with an environment variable.

Windows NT INPUTFILE '%INPUT%\file.1'

If a % character is part of the filename, it must be preceded by a backslash (\) character so it is not confused with an environment variable.

For standard label tapes, the filename in the LOAD DATA statement can be upper- or lowercase; however, the filename on the tape must be uppercase. Standard label tapes support filenames of up to 17 characters. If the specified filename is longer than 17 characters, the TMU uses the first 17 characters.

For TAR tapes, filenames are case-sensitive and the case of the filename in the LOAD DATA statement must match the filename on the tape.

TAPE DEVICE 'device_name'

Specifies a tape device; it must be a rewind tape device. Use this clause if the input file(s) are on one or more tapes.

Note: Tape support is not available for Windows NT systems.

Tape format can be either TAR or ANSI standard-label tapes (fixed- or variable-record length, but not segmented records). The TMU checks to see if the tape conforms to the ANSI standard label tape format. If so, it treats the tape as an ANSI labeled tape. If a tape does not appear to be an ANSI labeled tape, then the TMU treats the tape as a TAR tape. For more information about tape formats, refer to “Format of Input Data” on page 3-106.

Each name in the filename list can be the name of a single file on a multi-file standard-label tape. However, the TMU does not support multiple TAR archive files on a tape; it reads only the first file.

If the input device is a tape drive and it is not ready, the TMU issues a message to the terminal to ready the device.

The tape device name must be enclosed in single quotes. The name can be specified as a literal or as an environment variable.

Note: The TAPE DEVICE parameter is not valid for a LOAD DATA statement appearing in a control file for the *rb_cm* copy management utility. Load input must come from standard input when the LOAD DATA statement is in a control file used by the *rb_cm* copy management utility.

Example

This example illustrates how a device name or an environment variable can be used to specify the tape device on a UNIX system.

If a tape contains a file named *data_file.1* and the tape is mounted on a tape device named */dev/rmt0*, then the input clause is:

```
INPUTFILE 'data_file.1' TAPE DEVICE '/dev/rmt0'
```

If the tape device is defined as an environment variable named TAPE, the Input clause is:

```
INPUTFILE 'data_file.1' TAPE DEVICE '$TAPE'
```

START RECORD, STOP RECORD

Specifies which records in the input file mark the beginning and end of loading. If the START RECORD keywords are specified, loading begins at the specified record. Earlier records are read and counted, but their contents are ignored. If the STOP RECORD keywords are specified, loading stops after the specified number of records.

Default values are:

```
START RECORD: 1  
STOP RECORD: end-of-all-files
```

The START RECORD and STOP RECORD clauses are useful in the following circumstances:

- A load operation ends prematurely (for example, because a discard limit specified with a Discard clause was reached) and you want to resume loading after the last record loaded: Use START RECORD to specify the next record to be loaded. The TMU issues messages that provide row numbers to use with START RECORD.
- An input file is too large: Use START RECORD and STOP RECORD to split the records into two load operations.

- You are unsure about the format or content of the input data and want to test the load script: Use STOP RECORD to stop the load operation after a few rows.

The TMU follows these rules when counting rows:

- It counts only the rows it sees. For example, if tape #1 is not used and the load starts with tape #2, then the first record is the first one on tape #2 (the first tape used).
- The number of rows counted is not reset between files or tapes. The number keeps incrementing until the end of the current LOAD DATA statement.
- If START RECORD is specified on data fields defined as a SEQUENCE fieldtype, the sequence value is incremented for each row skipped. For example, if you specify START RECORD 10 and are loading a column using SEQUENCE (2,2), then the first row loaded is input row 10 with sequence value 20.

Examples

In this example the input file, *market.txt*, has fixed-format records. The fields in *market.txt* start and end at the same position in each record; they are not separated by characters. The input file is located on a disk device, which is the default. The TMU starts counting records from the beginning of the file, but starts loading at record 100 and stops loading at record 200.

```

Input clause  load data
               inputfile 'market.txt'
               start record 100
Format clause  stop record 200
Locale clause  recordlen 7 replace
Discard clause discardfile 'English_Canada.MS1252@Default'
               discardfile 'mktdisc.txt'
Optimize clause discards 1
Table clause  optimize on discardfile 'mktdups.txt'
               into table market(
               mktkey integer external (4),
               state char (2)
               );

```

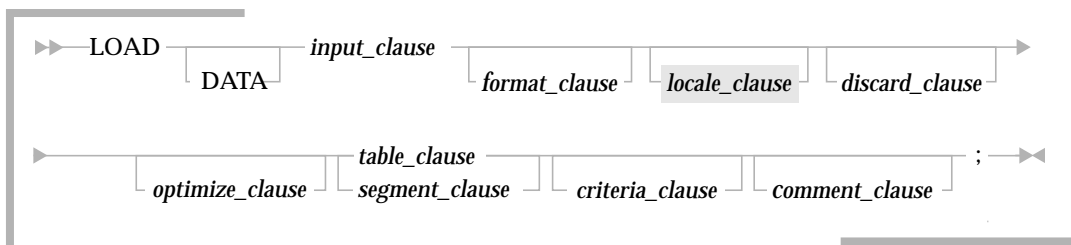
Format Clause

The Format clause is optional and specifies the format details of the input data:

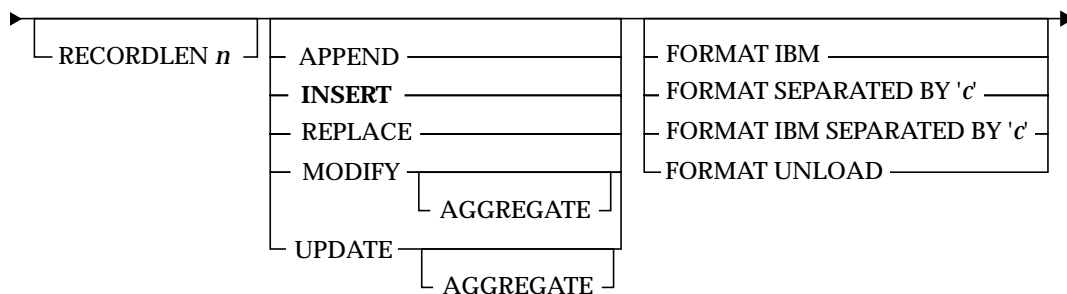
- Record length: fixed or variable
- Mode: append, insert, modify, replace, or update
- Logical data format: fixed, separated, or unloaded from the same or another warehouse database
- Character set: ASCII or EBCDIC

For more information about valid format combinations, refer to “Format of Input Data” on page 3-106.

The entire LOAD DATA statement syntax diagram is repeated here to provide a point of reference for the Format clause syntax:



The syntax for *format_clause* is:



RECORDLEN *n*

Number of bytes in each record; signifies fixed-length records without any newline character (or other record separator). If files contain binary data, this value is required. If files contain only character and external numeric data (for example, ASCII characters and numbers), this value should be omitted, signifying newline-separated data records; however, if it is included, *n* must include the newline character.

RECORDLEN is not allowed with FORMAT UNLOAD.

APPEND

Load mode used to insert additional rows of data into an existing table. Each new row must have a primary key value that does not already exist in the table; otherwise, the record is discarded. INSERT privilege on the table is required.

INSERT

Load mode used to load data into an empty table. If the table is not empty, then the load operation terminates. INSERT privilege on the table is required. The default mode is INSERT.

REPLACE

Load mode used to replace the entire contents of a table. INSERT and DELETE privileges on the table are required.

Caution: In REPLACE mode, the existing contents of a table are destroyed. Use this mode carefully.

MODIFY

Load mode used to insert additional rows or to update existing rows in a table. If the input row has the same primary key value as an existing row, the new row replaces the existing row; otherwise, it is added as a new row. Selected columns can be updated by using the DEFAULT and RETAIN keywords, as described on page 3-53. INSERT and UPDATE privileges on the table are required.

In MODIFY mode, the primary key column(s) must be present in the LOAD DATA statement.

UPDATE

Load mode used to update existing rows in an existing table. Each new row must have a primary key value that is already present in the table; otherwise, the record is discarded. Selected columns can be updated by using the DEFAULT and RETAIN keywords, as described on page 3-53. UPDATE privilege on the table is required.

In UPDATE mode, the primary key column(s) must be present in the LOAD DATA statement.

AGGREGATE

Indicates that aggregate operators are used on a column-by-column basis in MODIFY or UPDATE modes. In these modes, aggregate operators can be used on simple fields (as described on page 3-57) and on increment fields (as described on page 3-70). The aggregate operators available are: ADD, SUBTRACT, MIN, MAX, and INCREMENT.

In MODIFY AGGREGATE mode, if the input row's primary key matches an existing row in the table, the existing row is updated as defined for the specified aggregate operator. If the input row's primary key does not match an existing row in the table, the row is inserted. In this case, if the aggregate operator is ADD, SUBTRACT, MIN, or MAX, which require a value from a matching row in the table (there is no such row in this case), the input value is inserted; if the aggregate operator is INCREMENT, the increment value will be inserted.

In UPDATE AGGREGATE mode, if the input row's primary key does not match the primary key of a row already in the table, the input row is discarded. If it does match an existing row, the existing row is updated as defined for the specified aggregate operator.

For a detailed example of aggregate operation, refer to Appendix A, "Example: Using the TMU in AGGREGATE Mode."

Aggregate operators cannot be used on primary key columns. The MODIFY AGGREGATE and the UPDATE AGGREGATE modes are part of the Auto Aggregate option and must be enabled with a license key.

No FORMAT Keyword

Indicates by the omission of any format specification that records are in the ASCII character set and that their length is determined by the field lengths specified in the field specifications in the Table clause.

FORMAT IBM

Specifies that data is in the EBCDIC character set. CHARACTER and EXTERNAL fields are converted from EBCDIC to ASCII, and integer fields are converted to the byte ordering of the machine that is running Red Brick Warehouse. For details about specific EBCDIC to ASCII conversions, contact the Red Brick Customer Support Center.

A tape must have an ANSI standard label (in EBCDIC). The TMU accepts multiple-file tapes and multiple-reel files. Input files must be specified in the order in which they appear on the tape.

Specification of IBM format is typically used when the input file was prepared on an IBM mainframe system.

FORMAT SEPARATED by 'c'

Specifies that fields within a data record are separated by the character *c*, which must be a single-character literal and it must be different from the radix (decimal) point character. For example, data separated by an exclamation mark (!) is specified by:

```
format separated by '!'
```

and data separated by a tab character is specified by:

```
format separated by 'tab'
```

where *tab* represents the actual tab keystroke, which might appear as a blank space: ' '.

Caution: The separator character must be specified using the database locale character set, but it can be either a single-byte or multibyte character. If the character used as a separator in the input data cannot be expressed as a character in the database locale, then the input data cannot be interpreted correctly.

FORMAT IBM SEPARATED by 'c'

Specifies that data is in the EBCDIC character set and that fields within a data record are separated by the character *c*, which must be a single-character literal. It must be different from the radix (decimal) point character.

Caution: The separator character must be specified using the database locale character set, but it can be either a single-byte or multibyte character. If the character used as a separator in the input data cannot be expressed as a character in the database locale, then the input data cannot be interpreted correctly.

FORMAT UNLOAD

Specifies that the data to be loaded was unloaded in internal format from a warehouse database using a TMU UNLOAD statement. This format choice cannot be used to load data that was unloaded in external format. For more information about the UNLOAD statement, refer to Chapter 4, “Unloading Data from a Table.”

The RECORDLEN keyword and field specifications are not allowed when FORMAT UNLOAD is used.

Example

This example illustrates the use of the Format clause. The Market table contains existing data that is modified by the records in the input file, *market.txt*. The keyword **MODIFY** specifies that if a record in *market.txt* has the same primary key as a row in the Market table, the record in the file replaces the row in the table. If a row does not yet exist in the table, the TMU adds a new row.

Assume the *market.txt* file has fixed-format records; there is no separator character between fields and all of the records are the same length. **RECORDLEN** specifies the number of bytes the TMU reads for each record. **RECORDLEN** is calculated by summing the number of bytes in the Mktkey and State fields and adding one more byte for the newline character. The TMU loads a new record every seven bytes.

```

Input clause      load data
Format clause     ├── inputfile 'market.txt'
                  ├── recordlen 7 modify
Locale clause     ├── nls_locale 'English_Canada.MS1252@Default'
Discard clause    ├── discardfile 'mktdisc.txt'
                  ├── discards 1
Optimize clause   ├── optimize on discardfile 'mktdups.txt'
Table clause      ├── into table market(
                  │   mktkey integer external (4),
                  │   state char (2)
                  └── );

```

Example

This example illustrates the use of a Format clause to load data in **UNLOAD** format. Note that there are no format specifications other than the keyword **UNLOAD** and no field specifications.

```

Input clause      load data
Format clause     ├── inputfile 'market.txt'
                  ├── insert format unload
Discard clause    ├── discardfile 'mktdisc.txt'
                  ├── discards 1
Optimize clause   ├── optimize on discardfile 'mktdups.txt'
Table clause      ├── into table market;

```


Locale Clause

The unique combination of a language and a location is known as a *locale*. A locale specification consists of four components: *language*, *territory*, *character set*, and *collation sequence*. For example, the default locale for Red Brick Warehouse is:

```
English_UnitedStates.US-ASCII@binary
```

where

- English = language
- United_States = territory
- US-ASCII = character set, or code page
- binary = collation sequence

Examples of other locales are:

```
Japanese_Japan.MS932@Binary  
German_Austria.Latin1@Default  
French_France.Latin1@Default  
Spanish_Mexico.Latin1@Spanish
```

Note: Any collation sequence value other than *Binary* implies a linguistic sort; the value *Default* resorts to the sort definition specified by the CAN/CSA Z243.4.1 Canadian ordering, which covers English and several Western European languages.

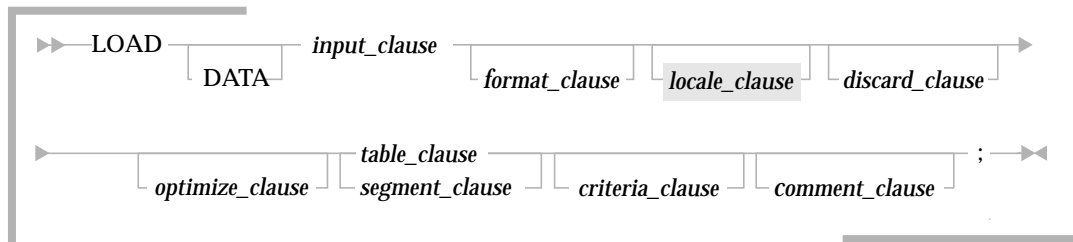
3

For more information about locales in warehouse databases, refer to the *Warehouse Administrator's Guide*.

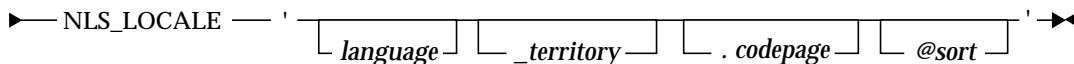
Although the TMU uses the database locale for most of its processing, a different locale can be specified for a TMU input file. In this way, data can be automatically converted from one character set to another as it is loaded into a database table. The locale of the input file, if different from the database locale, is specified with the NLS_LOCALE keyword in the Locale clause of the LOAD DATA statement. If the Locale clause is omitted, the input locale is assumed to be the same as the database locale.

Caution: The Locale clause refers only to the contents of the input file itself; all information specified in TMU control files must be specified in the database locale. Specifically, this means that separator, radix, and escape characters must be specified using the database locale character set. If the character used as a separator or radix point in the input data cannot be expressed as a character in the database locale, then the input data cannot be interpreted correctly.

The entire LOAD DATA statement syntax diagram is repeated here to provide a point of reference for the Locale clause syntax:



The syntax for *locale_clause* is:



NLS_LOCALE

Identifies the locale of the input data when the character set of the input data differs from that of the database locale. The locale specification also determines which character is interpreted as the decimal (radix) point in the input data, but it can be overridden by a RADIX POINT definition, which is specified in the Table clause as part of a DECIMAL field description.

language_territory.codepage@sort

Specifies all or part of the locale for the input file. The locale specification must be enclosed in single quotes. It is not necessary to specify all four parts of a locale; however, it is strongly recommended that if the locale is not fully specified, the language should be one of the specified components. Otherwise, the unspecified components might default to incompatible values. For a list of compatible components, refer to Appendix C, “Defined Locales.”

Note the following rules regarding default values for unspecified locale components; the same default values and precedence rules apply to input file locales as to locales specified with the RB_NLS_LOCALE environment variable.

- If only the language is specified, the omitted components are set to the default values for that language. For example, if the locale is set to

Japanese

the complete locale specification will be as follows:

Japanese_Japan.JapanEUC@Binary

For a list of default components for each language, refer to Appendix C, “Defined Locales.”

- If only the territory is specified, the language defaults to English, the character set to US-ASCII, and the sort to Binary. For example, if the locale is set to:

`_Japan`

the complete, but *impractical*, locale specification will be as follows:

`English_Japan.US-ASCII@Binary`

- Similarly, if only the character set is specified, the language defaults to English, the territory defaults to UnitedStates, and the sort component defaults to Binary.
- Finally, if only the sort component is specified, the language defaults to English, the territory defaults to UnitedStates, and the character set defaults to US-ASCII.

Note: It is not necessary to specify all the separator characters (the underscore, the period, and the @ character) in a partial locale specification. Only the character that immediately precedes the component(s) is required—such as the underscore character (`_`) in the previous territory example.

Usage Notes

- For all discard files specified in the Discard clause, the input file locale is used; however, for a discard file specified in the Optimize clause, the database locale is used.
- For all ACCEPT/REJECT processing in the Criteria clause, the database locale is used.
- The locale for TMU messages is either the database locale or the locale specified for the current user with the `RB_NLS_LOCALE` environment variable.
- Before specifying a character set for the input file that is different from the character set for the database, make sure that conversion between those character sets is supported. For a complete list of supported languages and character sets, refer to Appendix C, “Defined Locales.”

Caution: Red Brick Warehouse provides no recovery mechanism when data loss or data corruption occurs because of incompatible character sets.

Examples

The following example illustrates the use of the Locale clause in a LOAD Data statement, where the language is English, the territory is Canada, the character set (code page) is MS1252, and the sort order is Default, a Canadian sort order definition.

```
Input clause      load data
                  inputfile 'market.txt'
Format clause    recordlen 7 modify
Locale clause    nls_locale 'English_Canada.MS1252@Default'
Discard clause   discardfile 'mktdisc.txt'
                  discards 1
Optimize clause  optimize on discardfile 'mktdups.txt'
Table clause     into table market(
                  mktkey integer external (4),
                  state char (2)
                  );
```

The following LOAD DATA statement includes a locale specification for a Japanese Shift-JIS input file:

```
load data
  inputfile 'temp.data'
  recordlen 20
  replace
  nls_locale 'Japanese_Japan.MS932@Binary'
  discardfile 'temp.discards'
into table t1 (
  col1 position(2) decimal external(12),
  col2 position(15) character(5));
```

Discard Clause

If the TMU rejects any records because of data conversion, data content, or referential integrity errors or because records do not meet the ACCEPT/REJECT criteria in the Criteria clause, it places these records in one or more discard files.

Note: In loads performed in optimize mode, as described on page 3-45, duplicate rows are discarded to the discard file specified in the Optimize clause rather than the discard file specified in the Discard clause.

The optional Discard clause is used to specify the following information about discard files:

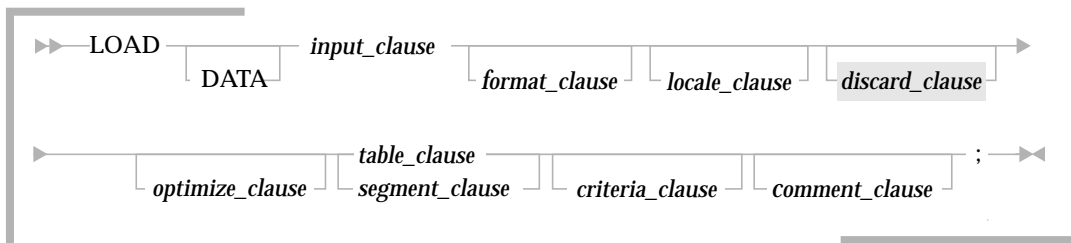
- Discard filenames.
- Whether to separate records discarded for referential integrity violations from those discarded for data integrity violations (data conversion, data content, or data that does not satisfy the ACCEPT/REJECT criteria).
- Whether to further separate records violating referential integrity by specifying separate discard files for the referenced tables. (If a record contains multiple referential integrity violations, it is written to the file specified for each violated dimension.)

The Discard clause also specifies whether referential integrity is to be preserved using the Automatic Row Generation option rather than by discarding rows. The default behavior for this option is set with the AUTOROWGEN parameter in the *rbw.config* file; if it is not set in the *rbw.config* file, the default behavior is OFF. The default behavior can be changed for all tables or for one or more specific tables in the Discard clause of the LOAD DATA statement for that table.

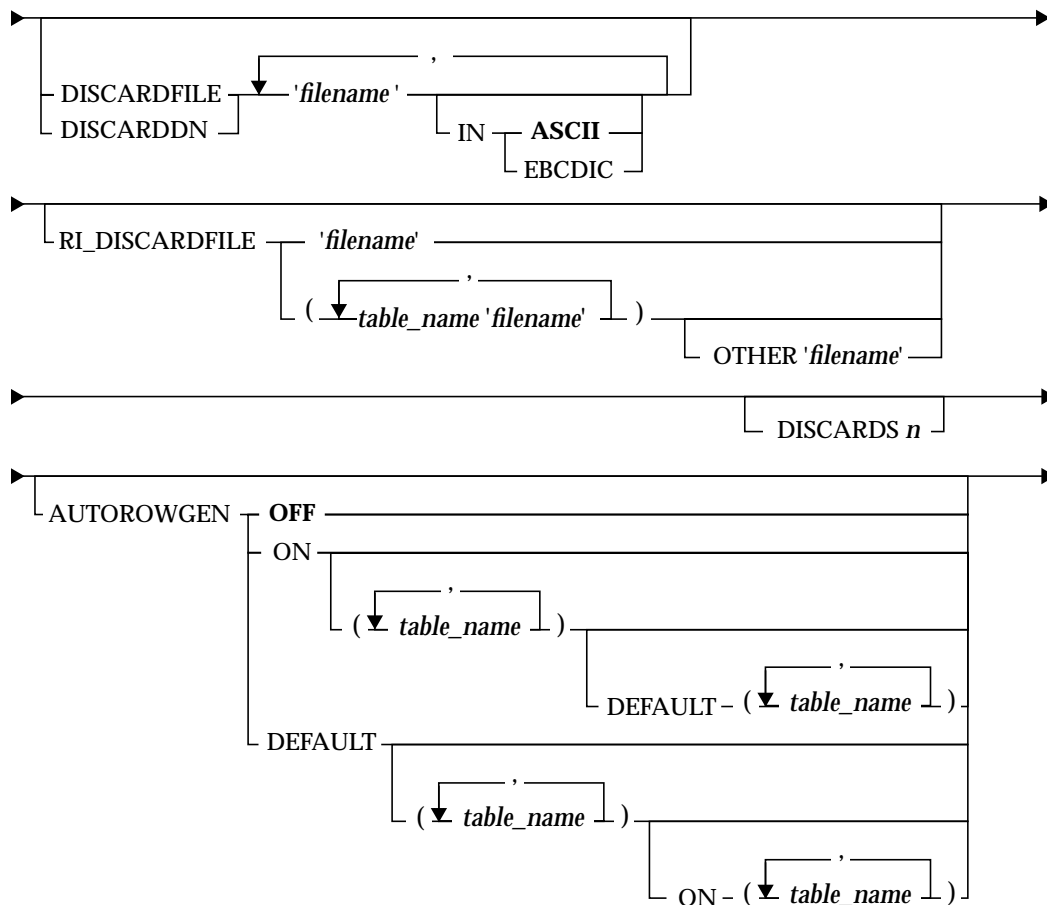
Each input record is first converted to internal row format and the data integrity is checked. If an error occurs during this phase, the record is discarded and written to the standard discard file, if one is specified. If no error occurs during this phase, referential integrity checks are performed on each referenced table.

Syntax

The entire LOAD DATA statement syntax diagram is repeated here to provide a point of reference for the Discard clause syntax:



The syntax for `discard_clause` is:



DISCARDFILE 'filename'

The name of the file(s) to which the TMU will write discarded input records. Two files (but no more than two) can be specified here so that both ASCII and EBCDIC versions of the discard file can be written. The filename(s) must satisfy operating-system file-specification conventions and must be enclosed in single quotes. (DISCARDN can be used instead of DISCARDFILE.)

All discarded records are written to this file(s) unless the RI_DISCARDFILE clause is present and specifies separate files for records that violate referential integrity.

Note: The user *redbrick* must have write permission for the discard file.

ASCII, EBCDIC

Optional; specifies the character set of the output file. This clause applies only to EBCDIC input data in an ASCII locale. If ASCII is specified—or no value is specified, the output file is written in the character set of the input locale. The output file can be in EBCDIC only if the input file is in EBCDIC.

RI_DISCARDFILE 'filename'

Optional; name of the file to which to discard the records that violate referential integrity. This clause cannot be used on a table that does not reference other tables.

The filename must satisfy operating-system file-specification conventions and must be enclosed in single quotes.

Note: You cannot specify ASCII/EBCDIC format for these files; they always are written in the character set of the input locale.

DISCARDS *n*

Specifies the maximum number of records that can be discarded—for any reason—before the TMU terminates execution of the control file. If the control file contains multiple control statements, any remaining statements are not executed. If *n* is 0, there is no maximum: The TMU completes execution of the control file regardless of how many records are discarded.

The discard limit applies to total records discarded from all input files in a single LOAD DATA statement. If multiple input files are specified, the number of records discarded is not reset for each file.

In load operations performed in optimize mode, as described on page 3-45, the actual number of discards might exceed the discard limit *n* because duplicate rows are not detected until all input rows have been processed and counted.

If the Discard clause is omitted or if DISCARDS *n* is omitted from the Discard clause, the default value is 0.

If no filenames are specified in the Discard clause but DISCARDS *n* is present, execution of the control file stops after *n* discards, but the discarded records are not saved in a file.

Example

In this example, the TMU stores rejected records in the file *prd_dscd.txt* in ASCII format. If more than 10 records are rejected, the TMU terminates execution of the control file and any remaining records are not loaded. Records loaded before the termination remain in the table.

```

      Input clause      load data
      Format clause    ----- inputfile 'prod.txt'
                        ----- format separated by ':'
      Discard clause   ----- discardfile 'prd_dscd.txt'
                        discards 10
      Optimize clause  ----- optimize on discardfile 'prd_dups.txt'
      Table clause    ----- into table product(
                                classkey integer external (2)
                                prodkey integer external (2),
                                prodname char (30)
                                pkg_type (20)
                                );

```

Example

In this example, the TMU stores records rejected for data integrity violations in the file *prod_di.txt* and it stores records rejected for referential integrity violations in the file *prod_ri.txt*.

```

      Input clause      load data
      Format clause    ----- inputfile 'prod.txt'
                        ----- format separated by ':'
      Discard clause   ----- discardfile 'prod_di.txt'
                        ----- ri_discardfile 'prod_ri.txt'
                        discards 10
      Optimize clause  ----- optimize on discardfile 'prd_dups.txt'
      Table clause    ----- into table product(
                                classkey integer external (2)
                                prodkey integer external (2),
                                prodname char (30)
                                pkg_type (20)
                                );

```


table_name 'filename'

A table name-filename pair that names a table referenced by a foreign key in the table being loaded and a file to which to discard the records that violate referential integrity with respect to the referenced table. The use of these pairs allows referential integrity violations to be resolved and reprocessed more easily than if all discards are stored in a single file.

These name pairs provide a separate discard file for each named table. If a single record violates referential integrity with respect to multiple referenced tables, that record will be written to the file associated with each of those tables.

Multiple pairs can be specified. If some but not all referenced tables are listed here, records that violate referential integrity with respect to tables missing from the list are written either to the file following the OTHER keyword, or if that keyword is missing, then to the standard discard file (following the DISCARDFILE keyword).

The filename(s) must satisfy operating-system file-specification conventions and must be enclosed in single quotes; pairs must be separated by commas.

OTHER 'filename'

Optional; specifies a file to which to discard any records that violate referential integrity with respect to referenced tables not named in the table_name-filename pairs. If a table_name-filename pair list is present and this clause is omitted, then any records that violate referential integrity with respect to tables missing from the list are written to the standard discard file (following the DISCARDFILE keyword).

The filename must satisfy operating-system file-specification conventions and must be enclosed in single quotes.

Example

In this example, the TMU stores records rejected for data integrity violations in the file *orders_di.txt*. It stores those rejected for referential integrity violations against the referenced tables Supplier and Deal in the files *sup_ri.txt* and *deal_ri.txt*, respectively. Any other records discarded for referential integrity are stored in the file *misc_ri.txt*.

```

Input clause  load data
Format clause  inputfile 'aroma_orders.txt'
Discard clause  format separated by '*'
                discardfile 'orders_di.txt'
                ri_discardfile (supplier 'sup_ri.txt',
                                deal 'deal_ri.txt') other 'misc_ri.txt'
                discards 10
Optimize clause  optimize on discardfile 'orders_dups.txt'
Table clause    into table orders(
                order_no integer external,
                perkey integer external,
                supkey integer external,
                dealkey integer external,
                order_type char (20),
                order_desc char (40)
                close_date date 'YYYY-MM-DD',
                price dec external (7,2)
                );

```

AUTOROWGEN

Specifies the behavior of the TMU when input records violate referential integrity; that is, when a foreign key value in an input record does not match a primary key value in the referenced table. The behavior can be specified on a table-by-table basis, allowing a combination of OFF, ON, and DEFAULT modes.

Whenever a list of referenced tables is present, the AUTOROWGEN option uses a mixed-mode operation, applying the specified mode to the tables in the list. All other directly referenced tables are treated as if AUTOROWGEN were set to OFF; that is, any records that violate referential integrity with respect to such a table are discarded.

The AUTOROWGEN option requires that all referenced tables into which generated rows are to be inserted have defined MAXROWS PER SEGMENT values.

Example

In this example, the TMU halts execution if it discards a record; however, instead of discarding records for referential integrity violations, the TMU attempts to generate the necessary rows for the referenced tables.

```

Input clause  load data
Format clause inputfile 'aroma_product.txt'
Discard clause format separated by '*'
              discardfile 'product.disc.txt'
              discards 1
              autorowgen on
Optimize clause optimize on discardfile 'prod_dups.txt'
Table clause  into table product(
              classkey integer external (2)
              prodkey integer external (2),
              prodname char (30)
              pkg_type (20)
              );

```

AUTOROWGEN OFF

Records violating referential integrity are written to the discard file.

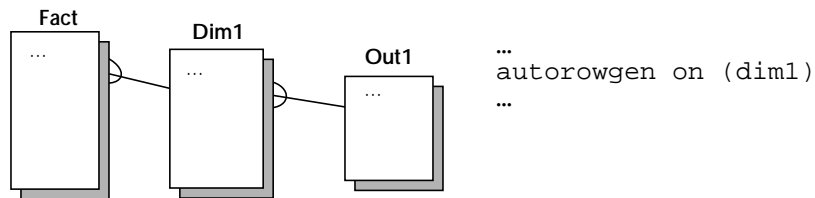
AUTOROWGEN ON

New rows are automatically generated (using the column default values) and added to the referenced tables in order to fully satisfy referential integrity, and the input record is added to the table being loaded. If no list of tables follows the ON keyword, this behavior applies to all tables; however, if a list of tables follows the keyword, this behavior applies only to the tables in the list and tables referenced directly or indirectly (outboard tables) by those tables.

When AUTOROWGEN is set to ON and a potential referential integrity violation cascades through a series of referenced tables, new rows are added until referential integrity is satisfied. If any of the rows necessary to preserve referential integrity cannot be added—for example, because conflicting modes apply to a referenced table or because of permission/authorization violations—the record is discarded.

Example

Assume the following tables and AUTOROWGEN setting:



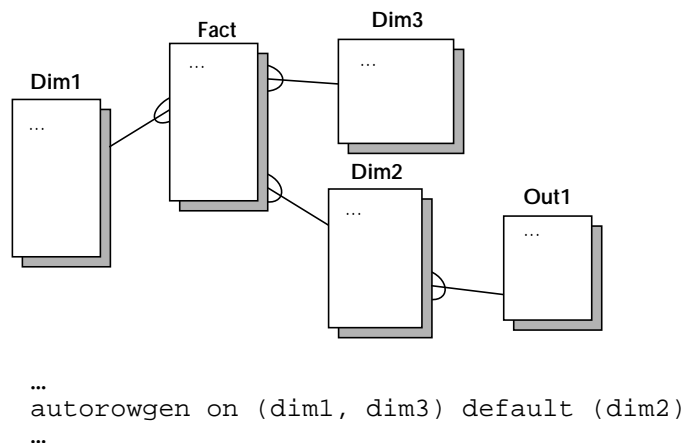
If a record to be inserted into table Fact violates referential integrity with respect to table Dim1, a new row with the new primary key value will be added to table Dim1. If this row will violate referential integrity with respect to table Out1, a new row will be added to Out1 to satisfy referential integrity. Then the needed row can be added to Dim1 and the record that started the whole process can be added to Fact.

AUTOROWGEN DEFAULT

If a referential integrity violation is detected, the foreign key value in that record is changed to the foreign key column's default value and a new row is inserted into the table being loaded. If no list of table names is present, this behavior applies to all tables; however, if a list of table names follows the DEFAULT keyword, this behavior applies only to the tables in the list.

Example

Assume the following tables and AUTOROWGEN setting:



If a record to be inserted into the Fact table violates referential integrity with respect to the Dim2 table, a new row—in which the foreign key value that violated referential integrity is replaced by the foreign key column's default value—will be added to the Fact table.

table_name

Specifies a table directly referenced by the table to be loaded. A table cannot appear in more than one list. The behavior for a table in a list is determined as follows:

- For any table in a list following ON: If a referential integrity violation occurs involving a table in this list or a table referenced directly or indirectly by a table in this list, a new row will be added to the referenced table.
- For any table in a list following DEFAULT: If a referential integrity violation occurs involving a table in this list, the record is modified and added to the referencing table.
- If a conflict arises because a row necessary for referential integrity cannot be added, the input record is discarded.

Example

In this example, the TMU halts execution if it discards a record (because of “discards 1”); however, instead of discarding records for referential integrity violations, the TMU attempts to generate the necessary rows for the referenced tables Dim1 and Dim2 and to modify input rows that reference Dim3.

```

Input clause      load data
Format clause     ──── inputfile 'market.txt'
Discard clause    ──── recordlen 7
                  ──── discardfile 'mktdisc.txt'
                  ──── discards 1
Optimize clause   ──── autorowgen on (dim1, dim2) default (dim3)
Table clause      ──── optimize on discardfile 'mktdups.txt'
                  ──── into table market(
                  ──── mktkey integer external (4),
                  ──── state char (2)
                  ──── );

```

Usage Notes

The following information applies to the use of the AUTOROWGEN option in the Discard clause.

Default Values

Whenever a referential integrity violation is detected and a row is to be inserted with a default value—in either the referenced table (ON mode) or in the referencing table (DEFAULT mode), the default values used are determined from default values defined for each column when the table was created. The default values can be literals, NULL, or system values such as CURRENT_USER, CURRENT_DATE, CURRENT_TIME, or CURRENT_TIMESTAMP. These default values also have specific interactions and restrictions with the column attributes NOT NULL and UNIQUE as defined in the *SQL Reference Guide*.

A default value assigned to a column can be changed with the ALTER TABLE statement.

You can determine the default values for each column in a table by selecting from the RBW_COLUMNS system table as follows:

```
select name, defaultvalue from rbw_columns
       where tname = 'TABLENAME' ;
```

Table Locks

Whether a referenced table is locked for read or write access during a load operation depends on the AUTOROWGEN mode. Locks are obtained at the beginning of the load operation and held throughout the operation. First, write locks are obtained on all referenced tables into which a write might occur to maintain referential integrity; then read locks are obtained on all referenced tables that must be read to verify referential integrity.

Note: All required locks are obtained automatically by the TMU; the user does not need to lock any tables manually.

During a load operation—for all AUTOROWGEN modes—the table being loaded is locked for write access. In addition, the various modes—whether specified in the *rbw.config* file or the LOAD DATA statement—require additional locks on referenced tables as follows:

- OFF mode: Directly referenced tables require a read lock; other tables are not locked.

- **ON mode:** If no list of tables follows the ON keyword, all referenced tables and all tables referenced directly or indirectly by those tables are locked for write access.

If a list of tables follows the ON keyword, tables in the list and all tables referenced directly or indirectly by those tables are locked for write access. Any directly referenced tables not present in the list are locked only for read access.

- **DEFAULT mode:** If no list of tables follows the DEFAULT keyword, all directly referenced tables are locked for read access. Indirectly referenced tables require no locks.

If a list of tables follows the DEFAULT keyword, those tables are locked for read access.

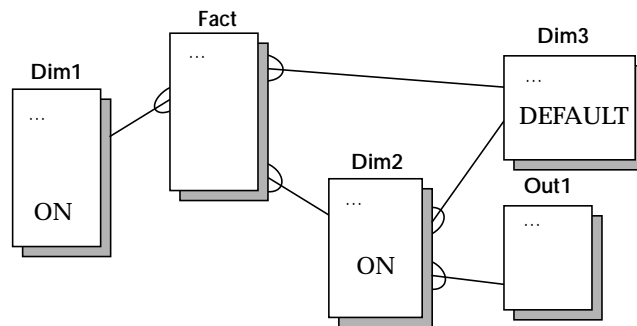
- Whenever lists of tables are present, any directly referenced table not in the list is locked for read access.

Conflicts in Mixed-Mode Operation

In mixed-mode operation, with both ON- and DEFAULT-mode table lists present, the LOAD DATA statement might specify potentially conflicting behavior. If such conflicts occur, a warning message is issued and the record is discarded.

Example

Assume the database contains the following tables:



and the LOAD DATA statement for the Fact table contains the following Discard clause:

```
discardfile 'fact_dscd' discards 100
autorowgen on (dim1, dim2) default (dim3)
```

Assume that a record to be loaded into the Fact table requires (for referential integrity) that a row be added to the Dim2 table, which in turn requires (for referential integrity) that a row be added to the Dim3 table. However, according to the AUTOROWGEN clause, referential integrity violations in which Dim3 is referenced directly are to be resolved by replacing the foreign key value with the default column value before adding the row to the Fact table. Because of this conflict, the TMU discards the record.

DEFAULT Mode and Simple Star Schemas

In a simple star schema, the primary key is composed of all the foreign key columns and only those columns. In DEFAULT mode, the same value—the default value for the column—will be used for each row to be entered in the referencing table. Because each row must contain a unique primary key, repeated use of the same value might cause records to be discarded as duplicates rather than entered into the referencing table. This behavior is illustrated by the example on page 3-14.

Troubleshooting

If automatic row generation is in ON or DEFAULT mode but the TMU is unable to generate rows in a referenced table:

- Verify that the database user ID running the TMU has INSERT privilege on the table.
- Verify that the referenced table does not specify both NOT NULL and DEFAULT NULL for any column: This combination on a single column prevents automatic row generation.
- Verify that a MAXROWS PER SEGMENT value is set for each referenced table.

If a load operation terminates before any rows are loaded, verify that the *redbrick* user has write permission on any files named in the Discard clause.

For additional information about using the automatic row generation feature, refer to “Maintaining Referential Integrity with Automatic Row Generation” on page 3-10.

Optimize Clause

The Optimize clause applies only to those operations that use the REPLACE, INSERT, or APPEND modes in the Format clause; it cannot be used with the MODIFY or UPDATE modes.

The Optimize clause specifies how the indexes are to be updated during a TMU incremental load operation. If OPTIMIZE OFF mode is used (non-optimize mode), indexes are updated when each input row is inserted into the data file. If OPTIMIZE ON mode is used (optimize mode), the index entries can be inserted as a batch operation. The batch operation is faster because it requires fewer I/O operations.

The default mode is non-optimize mode, which provides better performance when the data being loaded contains many duplicate rows. If your data does not contain a large number of duplicate rows, you might prefer to use the optimize mode for data-loading activities.

In the optimize mode, new index nodes in B-TREE and STAR indexes are built using fill factors specified in the *rbw.config* file or found in the RBW_INDEXES system table. For information about fill factors, refer to the *Warehouse Administrator's Guide*.

In optimize mode, space is allocated for index entries for all records, including duplicate records, and that space is not reclaimed for immediate reuse when the duplicate records are discarded. If the data you are loading contains many duplicate records, this behavior has several side effects:

- Indexes built in optimize mode are larger than indexes built in non-optimize mode.
- The MAXROWS PER SEGMENT limit might be reached before that many rows are actually loaded into a table segment because records are counted before duplicates are eliminated.
- The actual number of discarded rows might exceed the number *n* set as the maximum number of discards in the Discard clause.
- The actual number of rows that can be loaded in optimize mode might be less than the number calculated to fit in the space available, even if the discarded duplicate records are corrected and reloaded in non-optimize mode.

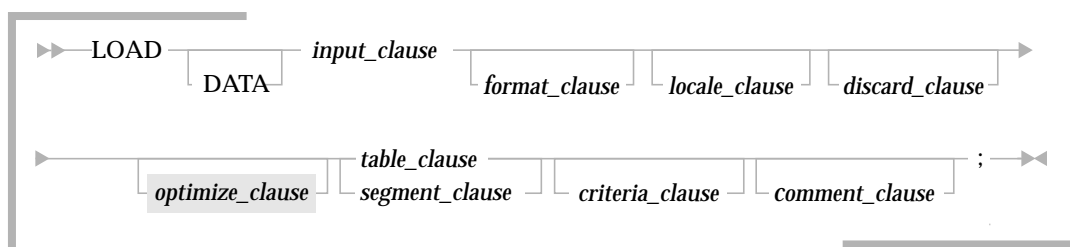
- When duplicate rows are discarded, the first row encountered in the table is kept and subsequent duplicates are discarded. Note that this row may or may not be the first row encountered in the load process (because of the way space is reused and the batch processing that occurs during index-building operations).

Checking for duplicate rows consumes both time and memory when the load process is done in optimize mode; when the number of duplicate rows is very large, the amounts consumed can be very significant. Red Brick Systems strongly recommends that you do not use optimize mode for load processes in either of the following cases:

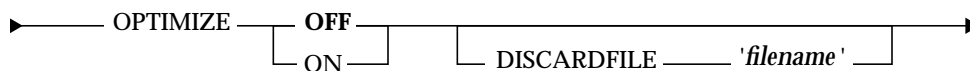
- The target table has a UNIQUE index other than the primary key index (multiple UNIQUE indexes), and more than 5,000 records in the input data will be discarded because their key values are duplicates of existing rows.
- The target table has a single UNIQUE index (the primary key index), the input data contains more than 1 million records, and more than 10% of these records will be discarded because their key values are duplicates of existing rows.

You can set the optimize mode globally in the *rbw.config* file for all load operations. If the optimize setting in the *rbw.config* file is what you want for a given LOAD DATA statement, then you do not need to include the Optimize clause in the LOAD DATA statement.

The entire LOAD DATA statement syntax diagram is repeated here to provide a point of reference for the Optimize clause syntax:



The syntax for *optimize_clause* is:



ON, OFF

Turns optimize mode on or off. This setting overrides the global setting in the *rbw.config* file for this LOAD DATA statement. (The default behavior is OFF.)

In a TMU control file containing multiple LOAD DATA statements, an Optimize clause applies only to the LOAD DATA statement that contains it. Any LOAD DATA statement that does not contain an Optimize clause uses the optimize setting in the *rbw.config* file.

DISCARDFILE 'filename'

Specifies the file into which duplicate records are discarded. A duplicate record is a record that contains a value that is the same as the value in an existing row for a column that is declared UNIQUE in the CREATE TABLE statement and is indexed—either a primary key index or a user-defined index.

This file contains rows in the same format as those rows discarded during an UNLOAD EXTERNAL operation on a table. The format of duplicate records discarded in optimize mode differs from the format of records discarded for referential integrity violations, data conversion, or format errors. You can specify a separate discard file for these duplicate records in each table's LOAD DATA statement so that records of different formats are not mixed together in a single discard file. If no discard file is specified, then duplicate records are discarded to the same file as other discarded records. Note that discarded records are not always written to the discard file in the same order as the input records.

Numeric and datetime data are formatted according to the ANSI SQ-92 rules for these datatypes. Data formats are not localized; however, multibyte characters in data and table and column names are preserved.

If a discard file is specified and optimize mode is off, the discard file clause is ignored.

3

Examples

In this example, the TMU discards duplicate records and saves them in the named file *mktdups.txt*. Any records discarded for other reasons (referential integrity violations, data conversion errors) are saved (in a different format) in the file *mktdisc.txt*.

```

Input clause      load data
Format clause     └─ inputfile 'market.txt'
Discard clause    └─ recordlen 7
                  └─ discardfile 'mktdisc.txt'
Optimize clause   └─ discards 1
Table clause      └─ optimize on discardfile 'mktdups.txt'
                  └─ into table market(
                      mktkey integer external (4),
                      state char (2)
                      );

```

Loading Data into a Warehouse Database

Optimize Clause

In this example, the TMU discards duplicate records and saves them in the file *mktdisc.txt*, because no other discard file is specified.

```
Input clause  load data
Format clause  \ inputfile 'market.txt'
Discard clause \ recordlen 7
               \ discardfile 'mktdisc.txt'
               \ discards 1
Optimize clause \ optimize on
Table clause   \ into table market(
               \ mktkey integer external (4),
               \ state char (2)
               \ );
```

Table Clause

The Table clause specifies the table into which the data is to be loaded. It can include:

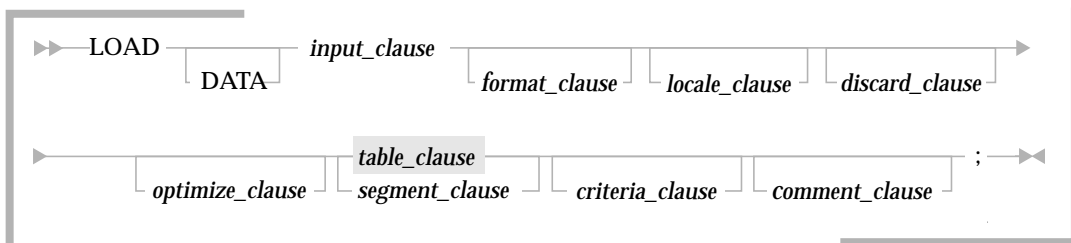
- Table column names
- Field specifications, which describe either:
 - Fields within the records of the input file.
 - Instructions for the TMU to retain the existing value, to use a default value, or to generate data, such as a sequence of numbers, for some columns.

Each field or group of fields maps to a column within the specified table. For example, in the following Table clause, Dollars and Mktkey are columns in the Sales table. The field description *decimal external (7,2)* describes the data to be loaded into the Dollars column. The field description *integer external (2)* describes the data to be loaded into the Mktkey column:

```
into table sales (
  dollars decimal external (7, 2),
  mktkey integer external (2))
```

The mapping of input datatypes to warehouse datatypes is defined on page 3-111. For all formats except FORMAT UNLOAD, one or more field specifications is required; for UNLOAD, field specifications are not allowed.

The entire LOAD DATA statement syntax diagram is repeated here to provide a point of reference for the Table clause syntax:



The diagram illustrates the syntax for the `INTO TABLE` statement. It shows the following components and their relationships:

- INTO TABLE**: The starting keyword of the statement.
- table_name**: The name of the table to be created or replaced, indicated by a line from `INTO TABLE` to `table_name`.
- (**: The opening parenthesis, indicated by a line from `table_name` to **(**.
- col_name**: The first column name, indicated by a line from **(** to `col_name`.
- AS - \$pseudocolumn**: An optional clause for creating a pseudocolumn, indicated by a line from `col_name` to `AS - $pseudocolumn`.
- \$pseudocolumn**: The name of the pseudocolumn, indicated by a line from `AS - $pseudocolumn` to `$pseudocolumn`.
- RETAIN**: A keyword for retaining the table structure, indicated by a line from `col_name` to **RETAIN**.
- DEFAULT**: A keyword for setting default values, indicated by a line from `col_name` to **DEFAULT**.
- simple_field**: A field name, indicated by a line from `col_name` to **simple_field**.
- concat_field**: A field name, indicated by a line from `col_name` to **concat_field**.
- constant_field**: A field name, indicated by a line from `col_name` to **constant_field**.
- sequence_field**: A field name, indicated by a line from `col_name` to **sequence_field**.
- increment_field**: A field name, indicated by a line from `col_name` to **increment_field**.
-)**: The closing parenthesis, indicated by a line from `col_name` to **)**.

<i>table name</i>
<p> <code>ALTER TABLE</code> <i>table name</i> <code>ADD</code> <i>column name</i> <i>data type</i> [<i>constraints</i>];</p>

col_name and ***\$pseudocolumn***

A pseudocolumn is not a real column in the table; it is used to store data temporarily for one of the following reasons:

- For future use with the CONCAT option.
- To discard an input field that is not to be used in the table.
- As a reference in a Criteria clause.

If a table column is omitted from the Table clause, existing rows retain the current value for that column and new rows are loaded with the default value for the column. In MODIFY or UPDATE mode, the primary key column must be present in the table clause.

Examples

In this example, you want to concatenate the three fields City, State, and Zip into the Address column. You also want to store the ZIP code as an individual column, but not the city or state. Define pseudocolumns for the city and state information. These pseudocolumns temporarily hold the city and state information until the TMU processes the CONCAT field for that record.

```
load data ...
  into table address(
    $city char (10),
    $state char (2),
    zip char (10),
    address
      concat ($city, $state, zip)
  );
```

In this example, assume you do not want to load the Zip field into the table. Define that field as a pseudocolumn and its data will not be stored:

```
load data ...
  into table address(
    city char (10),
    state char (2),
    $zip char (10),
  );
```

AS \$pseudocolumn

Stores the input value in the specified column in the table as well as in the specified pseudocolumn for reference in a Criteria clause. You must specify the AS \$pseudocolumn clause in either of two situations:

- A Criteria clause is present and the input data is in separated format.
- A Criteria clause is present and the input data is in fixed format.

Under these circumstances, you must specify the input column with AS \$pseudocolumn, and you must use \$pseudocolumn in the Criteria clause to refer to the input field because there is no way to refer to that column by specifying its position.

```
load data ...
  format separated by '/'
  ...
  into table market
  (
    ...
    cycle as $cycle integer external
    ...
  )
  reject $cycle >= 30
;
```

When loading fixed-format input data and specifying a Position clause, you need not specify AS *\$pseudocolumn* because you can refer to the input data by its position. You can specify both a pseudocolumn for use in the Criteria clause and a table column for the same input field.

```
load data ...  
  into table market  
  (...  
    cycle position (67),  
    $cycle position (67)  
  ...)  
  reject $cycle >= 30  
;
```

RETAIN

Causes an existing row, when being updated, to retain its current value for the column corresponding to this field; and causes a new row to store the default value for the column. (Default column values are defined by CREATE TABLE statements.) This behavior applies to both AGGREGATE and non-aggregate modes.

The RETAIN keyword is not allowed with primary key columns or pseudocolumns.

DEFAULT

Causes the default value for the column corresponding to this field (or NULL if no default value is defined) to be stored in the column. To load the column's default value into both new and existing rows, include a field specification with the DEFAULT keyword for that column. The DEFAULT keyword is not allowed for pseudocolumns.

Note: If the DEFAULT keyword is used for a column that is defined as NOT NULL DEFAULT NULL, then the load operation terminates.

Selective Column Updates with **RETAIN** and **DEFAULT**

The following table defines the TMU behavior during load operations with respect to the type of field specification and the load mode specified in the Format clause of the LOAD DATA statement.

Field Specification	Load Modes (Format Clause)		
	APPEND, INSERT, or REPLACE	UPDATE*	MODIFY*
No column name /field specification	Load column default value.	Retain current column value.	New row: Load column default value.
RETAIN keyword	Error (#1362)		Existing row: Retain current column value.
DEFAULT keyword	Load column default value.		
Simple specifier, CONCAT, CONSTANT, SEQUENCE, INCREMENT	Load field input value.		

* In UPDATE or MODIFY mode, the primary key columns must be present in the Table clause of the LOAD DATA statement.

Example

In this example, the Market table is being loaded with new data that reflects a geographic reorganization of the districts and regions to which each headquarters city belongs.

The CREATE TABLE and basic LOAD DATA statements for the Market table are as follows:

CREATE TABLE statement

```
create table market (
  mktkey integer not null,
  hq_city char (20),
  hq_state char (5),
  district char (14),
  region char (10),
  primary key (mktkey))
maxrows per segment 128
```

LOAD DATA statement

```
load data
inputfile 'aroma_market.txt'
recordlen 45
replace
discardfile 'aroma_discards'
discards 1
into table market (
  mktkey integer external(2),
  hq_city char(20),
  hq_state char(2),
  district char(13),
  region char(7) );
```

The new LOAD DATA statement retains the information in the Hq_city and Hq_state columns but loads the new definitions in the District and Region columns. Note that even if the Hq_city and Hq_state field specification lines were omitted, the values currently in the table would be retained.

```
load data
inputfile 'aroma_mkt_upd.txt'
recordlen 45
modify
discardfile 'mkt_upd_discards'
discards 1
into table market (
  mktkey integer external(2),
  hq_city retain,
  hq_state retain,
  district char(13),
  region char(7) );
```

These lines
could be ———→
omitted. ———→

Field Specifications (Simple, Concat, Constant, Sequence, Increment)

Specifies the data to be loaded into the column. Fields can be simple, concatenated, constant, sequenced, or incremented.

- A simple field specifies the datatype of the field in the input record that is to be loaded into the column.
- A concatenated field specifies the concatenation of fields in the input record to be loaded into the column.
- A constant field specifies a constant value that is to be loaded into the column. The TMU generates the value; it does not exist in the input file.
- A sequential field specifies a sequence of numbers to be loaded into the column. The TMU generates the numbers; they do not exist in the input file.
- An increment field specifies a value to be added to existing column values. The TMU generates the value to be added; it does not exist in the input file.

Example

This example illustrates the Table clause with a LOAD DATA statement that reads a separated-format input file, *customer.txt*. The records in *customer.txt* replace the data in the Customer table.

The Table clause lists the names of the columns in the Customer table and describes the data being loaded. The field specification for the Custkey column describes a sequence field. The TMU generates numbers starting from 1000 and loads them into the Custkey column.

The field specifications for Name, City, and Zip specify simple fields. The TMU gets the data for these columns from the input file. The fieldtype descriptions of the simple fields specify the datatype of the input data, which are mapped to the datatype of the column.

Loading Data into a Warehouse Database

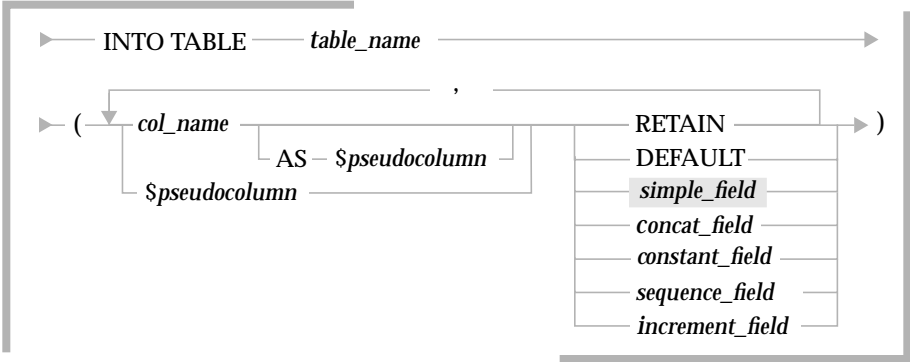
Table Clause

```
Input clause  load data
               \
Format clause  \ inputfile 'customer.txt'
               \
Discard clause \ replace
               \ format separated by ':'
Optimize clause \ discardfile 'aroma_discards'
               \ discards 5
Table clause   \ optimize on discardfile 'aroma_dups.txt'
               \ into table customer(
               \   custkey sequence (1000, 1),
               \   name char,
               \   city char,
               \   zip integer
               \ );
```

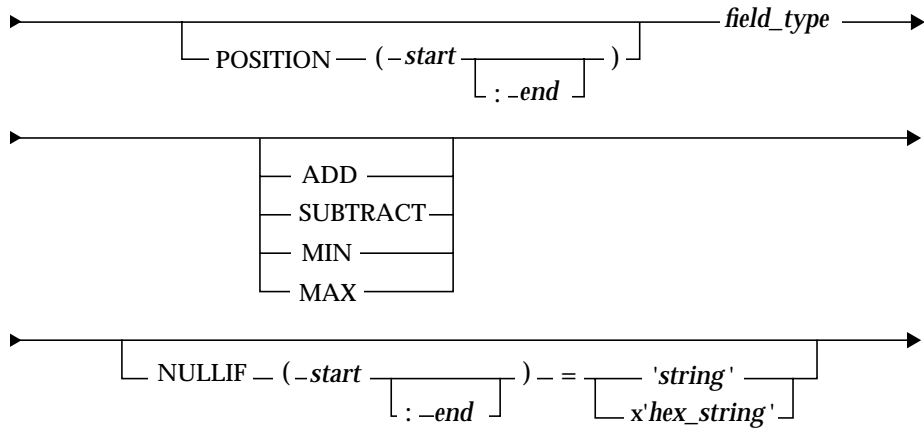
Note: The lengths of the fieldtypes are not defined, not even for the character fields. Length definitions are unnecessary because the TMU uses the separator character ':' to determine the length of each field in the input file.

Table Clause: Simple Fields

A simple field specifies the datatype of the field in the input record that is to be loaded into the column. The entire Table clause syntax diagram is repeated here to provide a point of reference for the simple field syntax:



The syntax for *simple_field* is:



3

POSITION
 Specifies the offset in bytes from the beginning of the record. This option is used for fixed-format files only; do not use this option with the SEPARATED BY keywords. The position of the first field in a record is 1. If no position is specified, then the position of a field is one greater than the last byte of the previous field.

Position refers to the position in the record, not within a specific field. Therefore, when you specify *start:end*, remember to use the position of the data in the record. In the following example, the field to be loaded into the Weight column starts at position 30 and ends at position 32. Other fields or blanks precede this field in the input file:

```
weight position (30:32) integer external (3)
```

If *:end* is specified, the length of the field is $end - start + 1$ bytes. This length overrides any length specified in the fieldtype specification.

For the CHARACTER fieldtype, if no length is specified with the POSITION keyword or in the fieldtype description (for example, char (15)), then the column width defined for the table is used.

For DECIMAL (EXTERNAL, PACKED, ZONED), INTEGER EXTERNAL, FLOAT EXTERNAL, DATE, TIME, TIMESTAMP, and M4DATE fieldtypes, a length must be specified either with the POSITION keyword or with the length parameter in the fieldtype specification.

The POSITION keyword is ignored for data in separated format because the field lengths are variable.

field_type

Specifies the datatype of the input field, for example, integer external. For information about fieldtypes, refer to page 3-82.

NULLIF

Provides a way to load a column with NULL. If the data in the specified position is equal to the value of the string, then the column value for the corresponding row is set to NULL.

As in a Position clause, *start:end* refers to the position of the data within the record, not to the position within the field. In the following example, the field for the City column contains 20 bytes, starting at position 25 and ending at position 44. The TMU stores a null indicator in the City column if the first three bytes in the that field match the starting *San*:

```
city position (25:44) char (20) nullif (25:27) = 'San'
```

If *:end* is not specified, the length of the string is used.

The string must be specified using the database locale character set.

x'hex_string' indicates a string in hexadecimal format; the character x must be present.

NULLIF is not allowed with separated-format files. If the data is in separated format, a column is loaded with NULL when there is no data between the separator characters surrounding the field.

Aggregate Operators

The aggregate operators ADD, SUBTRACT, MIN, and MAX can be used only with the MODIFY AGGREGATE or UPDATE AGGREGATE modes; they cannot be used with primary key columns or “true” pseudocolumns (not the AS \$pseudocolumns) or non-numeric columns.

If a value in the specified field of the input record or a value in the specified column of the table is NULL, then the result of the aggregation operation (ADD, SUBTRACT, MIN, or MAX) is NULL.

ADD

Adds the value in the input record to the corresponding value in the table column.

SUBTRACT

Subtracts the value in the input record from the corresponding value in the table column.

MIN

Keeps the smaller of the value in the input record and the value in the corresponding table column.

MAX

Keeps the larger of the value in the input record and the value in the corresponding table column.

Example

This example illustrates a LOAD DATA statement that reads a fixed-format file. The Position clause specifies the starting byte of each field relative to the offset in bytes from the beginning of the record. The field that maps to the Perkey column starts at position 4 and ends at position 8 followed by 3 spaces. The next field maps to the Prodkey column and starts at position 12, and the field after that maps to the Custkey column and starts at position 17.

```
load data
inputfile 'orders.txt'
recordlen 39
modify
discardfile 'orders_discards'
discards 1
into table orders(
  perkey position (4) integer external (5),
  prodkey position (12) integer external (2),
  custkey position (17) integer external (2),
  invoice sequence (1000,1)
);
```

Notice that the field specification for Invoice does not contain a Position clause. The values to be stored in the Invoice column are generated by the TMU. Because the values do not exist in the input file, a Position clause is not relevant.

The following is sample data from the *orders.txt* file. The dashes (-) represent spaces; actual data would contain spaces.

Perkey	Prodkey	Custkey
---10045---	12---	56
---10046---	13---	57
---10047---	14---	58

Example

This example shows a NULLIF condition on the destination column, City. If a value starting at position 3 and ending at position 5 in *market.txt* is *San*, the TMU stores a null indicator in the City column.

```
load data
inputfile 'market.txt'
discardfile 'market_discards'
into table market(
  mktkey integer external (2),
  city char (20) nullif (3:5) = 'San'
);
```


Example

In this example, the TMU adds the dollar values in the input record to the existing values in the corresponding row of the Sales table. Because UPDATE AGGREGATE is specified, new rows are not added to the table; each record must have a primary key value that is already present in the Sales table. An aggregate mode must be specified because ADD is part of the Auto Aggregate option.

```
load data
inputfile 'sales.txt'
update aggregate
discardfile 'sales_discards'
into table sales(
    perkey integer external (5),
    prodkey integer external (2),
    mktkey integer external (2),
    dollars decimal external (7,2) add
);
```

Example

In this example, the TMU compares a value in the Dollars field of the input record to the value in the corresponding row of the Dollars column and retains the largest dollar value. Records containing values that do not yet exist in the table are added as new rows. An aggregate mode must be specified because MAX is part of the Auto Aggregate option.

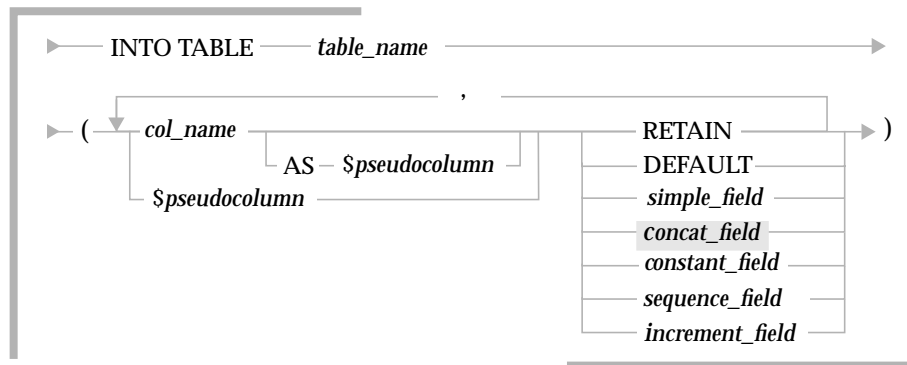
```
load data
inputfile 'sales.txt'
modify aggregate
discardfile 'sales_discards'
into table sales(
    perkey integer external (5),
    prodkey integer external (2),
    mktkey integer external (2),
    dollars decimal external (7,2) max,
    weight integer external (3)
);
```

For a detailed example of aggregate operation, refer to Appendix A, “Example: Using the TMU in AGGREGATE Mode.”

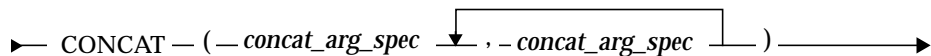
Table Clause: Concatenated Fields

A concatenated field loads a column of CHARACTER data with the concatenation of several CHARACTER data input fields.

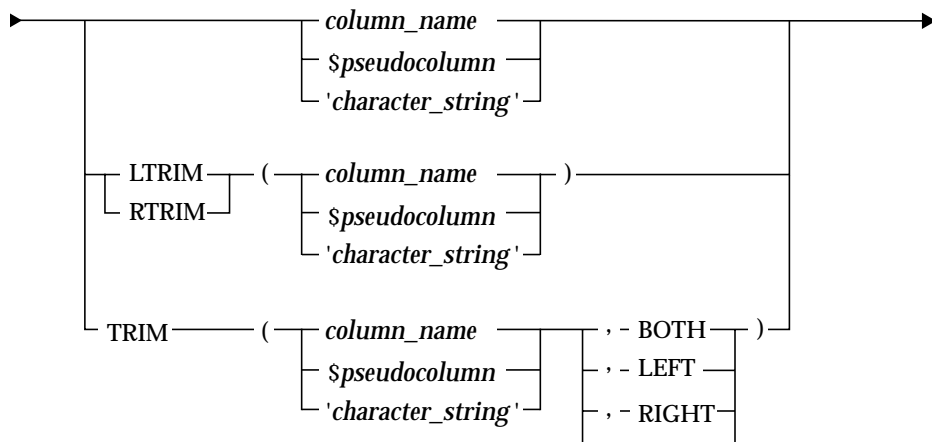
The entire Table clause syntax diagram is repeated here to provide a point of reference for the concatenated field syntax:



The syntax for *concat_field* is:



The syntax for *concat_arg_spec* is:



column_name, \$pseudocolumn, character_string

Specifies the input fields to be concatenated. Definitions for these input fields (except character strings) must precede the concatenated field definition; no forward references are allowed.

If any of the fields to be concatenated contains NULL for a given row, then the result of the concatenation is NULL.

Character strings must be specified using the database locale character set.

LTRIM, RTRIM, TRIM

Remove preceding blanks, trailing blanks, or both, respectively, before concatenating the input fields.

Example

In this example, two fields are concatenated and stored in a column. All of the fields in the *product.txt* file are loaded into the Product table. The values in the Aroma and Acid fields in *product.txt* are stored in the Aroma and Acid columns and are also joined in the Body column with the ampersand character (&) used as a separator. The LTRIM option removes preceding blanks.

```
load data
inputfile 'product.txt'
replace
format separated by ':'
discardfile 'product_discards'
discards 100
into table product (
  prodkey integer external (2),
  product char (12),
  aroma char (8),
  acid char (7),
  body
  concat (ltrim (aroma), '&', ltrim (acid))
);
```

Example

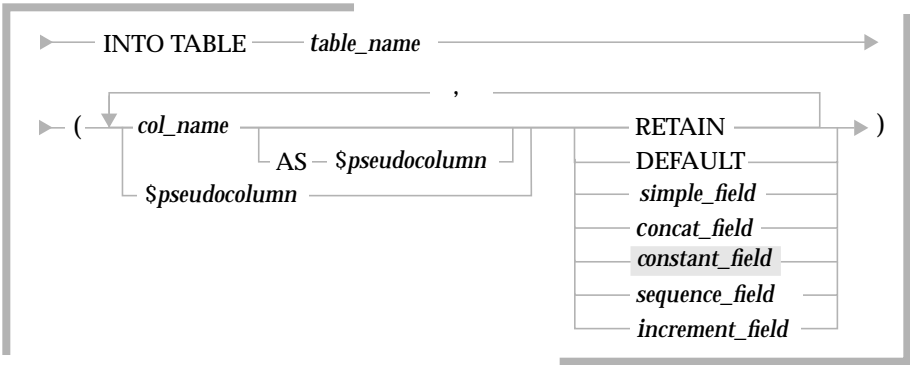
This example illustrates the use of concatenated fields and pseudocolumns. Several fields are read and stored in the table, but the *Str* field is not stored as a separate column; instead, it is saved as a pseudocolumn for use in a concatenated column. The concatenated column named *Prd_fr_st_pc* in the *Nba_basic* table holds the string formed by concatenating four fields read earlier, including the one stored as a pseudocolumn.

```
load data
inputfile 'nba_basic'
replace
format separated by '|'
discardfile 'nba_basic.dsc'
discards 100
into table nba_basic(
    fill_l char,
    prd_key char,
    frm_key char,
    $str char,
    pck char,
    pr_fr_st_pc
        concat(prd_key, frm_key, $str, pck),
    mp char,
    pcbo_tot int external,
    pcbo_rx int external
);
```

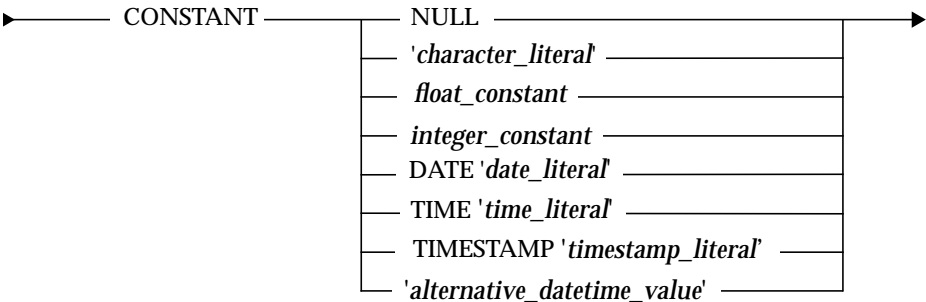
Table Clause: Constant Fields

A constant field loads a column of any datatype with a constant value legal for that datatype.

The entire Table clause syntax diagram is repeated here to provide a point of reference for the constant field syntax:



The syntax for *constant_field* is:



character_literal, float_constant, integer_constant, date_literal, time_literal, timestamp_literal, alternative_datetime_value

Specifies the value to be inserted into the specified column. The value supplied must be type-compatible with the designated output column, as defined in “Fieldtype Conversions” on page 3-111. For information about legal literal and constant values, refer to the *SQL Reference Guide*.

Both ANSI SQL-92 datetime datatypes and the defined alternative datetime formats are valid. If an ANSI SQL-92 datetime keyword is present, then the

literal that follows the keyword must be ANSI SQL-92 format. If you use an alternative datetime value with numeric months and a format other than *mdy*, you must include a SET DATEFORMAT command in the TMU control file. For more information about the TMU SET DATEFORMAT command, refer to “Format of Datetime Values in TMU Statements” on page 2-20.

Note: You can also use a simple field and a datetime format mask to specify a non-standard datetime value, as described on page 3-93.

Character literals must be specified using the database locale character set. Decimal constants must be specified with a decimal radix.

Examples

In this example, the TMU generates the value 999 and stores it in the Dollars column for each record loaded into the Orders table.

```
load data
inputfile 'orders.txt'
replace
discardfile 'orders_discards'
discards 10
into table orders(
    invoice integer external (5),
    perkey integer external (5),
    prodkey integer external (2),
    custkey integer external (2),
    dollars constant 999
);
```

In this example, the TMU generates a constant date of March 10, 1998, and constant time and timestamp values and stores them in the corresponding columns for each record loaded into the Period table.

```
load data
inputfile 'period.txt'
replace
format separated by '*'
into table period (
    perkey integer external (5),
    date_col constant date '1998-03-10',
    time_col constant time '03:15:30',
    timestamp_col constant timestamp '1998-03-10 3:15:30'
);
```

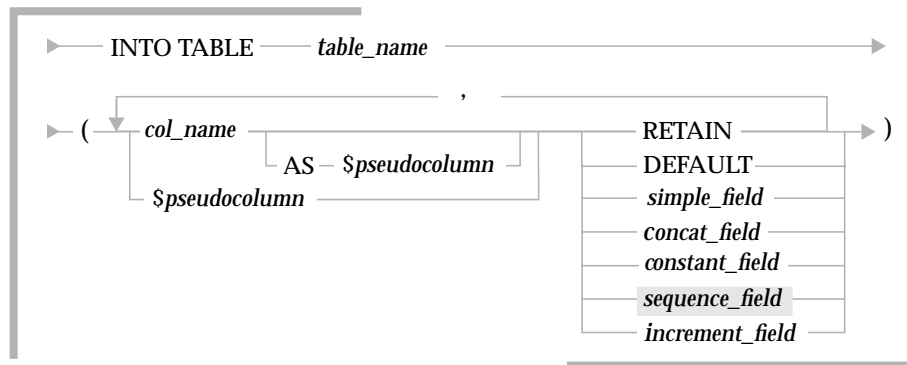
In this example, the LOAD DATA statement includes a date constant with numeric months in a *dmy* format, so the TMU control file must include a SET DATEFORMAT command to specify the alternative date format used in the LOAD DATA statement.

```
set dateformat 'dmy';
load data
inputfile 'period.txt'
replace
discards 1
into table period(
  perkey integer external (5),
  date_col constant '10-03-1998', -- March 10, 1998
  time_col constant '03:15:30 PM',
  timestamp_col constant '10-03-1998 03:15:30 PM');
```

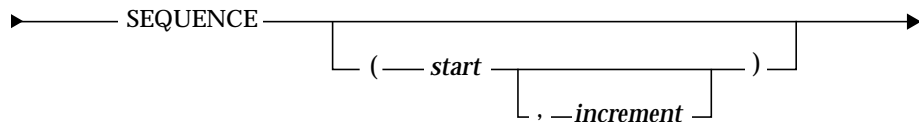
Table Clause: Sequence Fields

A sequence field loads a numeric column with a sequentially computed integer value.

The entire Table clause syntax diagram is repeated here to provide a point of reference for the sequence field syntax:



The syntax for *sequence_field* is:



start, increment

Specifies the starting value and the value by which to increment. These values can be any negative or positive integer, including 0. The increment value is applied to each new row, whether that row is skipped, loaded, or discarded because of an error. The default value for both *start* and *increment* is 1.

Caution: Verify that the incremented values will not overflow the range of the datatype of the destination column.

Example

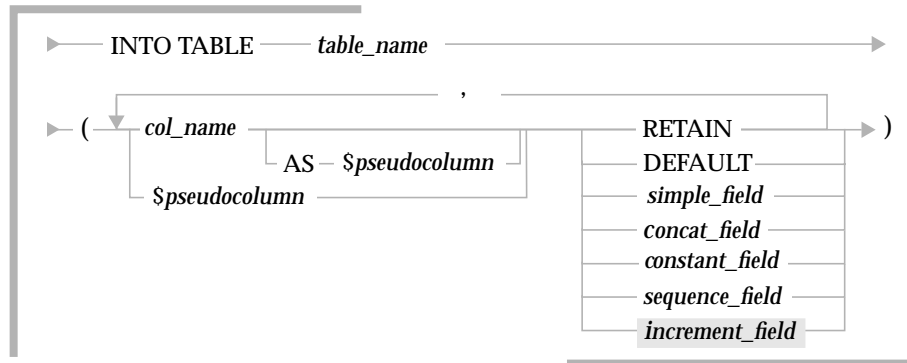
In this example, the TMU automatically generates numbers starting from 1000 and loads them into the Invoice column of the Orders table. The numbers increment by 1 for each new row in the table.

```
load data
inputfile 'orders.txt'
append
discardfile 'orders_discards'
discards 10(
into table orders
    invoice sequence (1000, 1),
    perkey integer external (5),
    prodkey integer external (2),
);
```

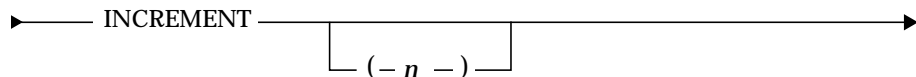
Table Clause: Increment Fields

An increment field adds a value to the value in the column.

The entire Table clause syntax diagram is repeated here to provide a point of reference for the increment field syntax:



The syntax for *increment_field* is:



n

Specifies the increment (decrement) amount. It must be a positive or negative numeric constant. The default value of *n* is 1.

If the value in the specified column is NULL, the result of the increment operation on that row is also NULL.

You must specify either the UPDATE AGGREGATE or the MODIFY AGGREGATE format in the Format clause. The INCREMENT mode is part of the Auto Aggregate option and must be enabled with a license key.

The INCREMENT keyword cannot be used with pseudocolumns (true pseudocolumns or “AS *\$pseudocolumns*”).

Example

This example illustrates the use of the increment field and the UPDATE AGGREGATE mode. The TMU updates the Weight column by adding the value 15 to the values already existing in the column. Because UPDATE AGGREGATE is specified, each record in the *sales.txt* file must have a primary key value that exists in the Sales table; otherwise, the record is discarded. An aggregate mode must be specified because Increment fields use the Auto Aggregate option.

```
load data
inputfile 'sales.txt'
recordlen 32
update aggregate
discardfile 'sales_discards'
discards 1
into table sales(
    perkey integer external (5),
    prodkey integer external (2),
    mktkey integer external (2),
    dollars decimal external (7,2),
    weight increment (15)
);
```

Segment Clause

The Segment clause can be used instead of a Table clause to specify a segment of a table into which to load data. To load data into a single segment, the following conditions must be met:

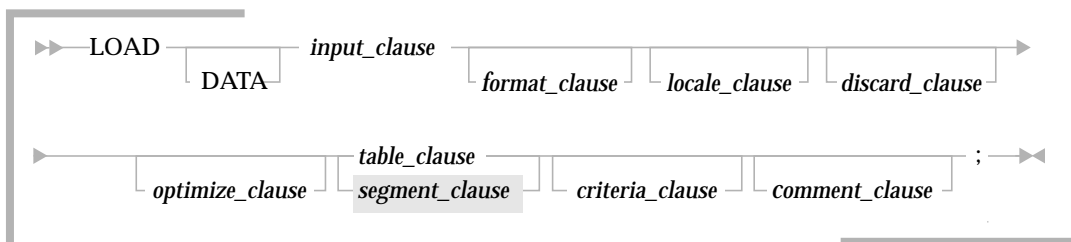
- The segment must be attached to a table.
- The segment must be offline.
- The Format clause mode must be APPEND, INSERT, or REPLACE.
- The AUTOROWGEN option in the Discard clause must be OFF.

Note: This clause cannot be used for a single-segment table.

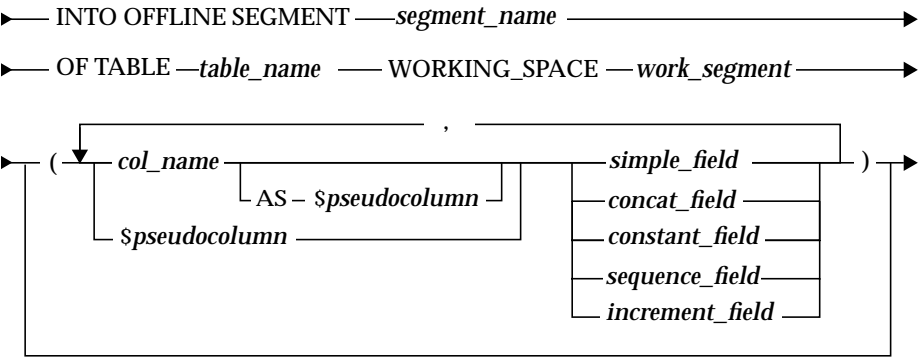
After data is loaded into an offline segment, the partial indexes must be synchronized with the existing indexes on the table with a SYNCH SEGMENT operation.

The offline load operation is always done in OPTIMIZE mode, regardless of settings in the Optimize clause or the *rbw.config* file.

The entire LOAD DATA statement syntax diagram is repeated here to provide a point of reference for the Segment clause syntax:



The syntax for *segment_clause* is:



Note: The column names, pseudocolumns, and field specifications must be omitted if and only if the input data is in UNLOAD format, as specified in the Format clause.

segment_name

Specifies the offline row data segment into which data is to be loaded. The segment must be attached to the table specified by *table_name* and must be offline.

The segment must also meet the conditions specified by the mode in the Format clause. The scope of the mode is limited to the segment being loaded. For example, INSERT fails if the segment is not empty, but does not fail if rows are present in other segments; and REPLACE deletes all rows in the specified segment but not rows in any other segment.

OF TABLE *table_name*

Specifies the table to which the segment is attached.

WORKING_SPACE *work_segment*

Specifies an unattached segment that contains adequate space to hold control information as the new data is loaded; this segment can be relatively small. In most cases, the following values are adequate:

INITSIZE, EXTENDSIZE: Default values
 MAXSIZE: 50 or 100 kilobytes

Use a larger MAXSIZE value under any of the following conditions:

- The table to be loaded has many indexes.
- The data to be loaded has many duplicates.
- The INDEX TEMPSPACE THRESHOLD value is small relative to the amount of data.

col_name, pseudocolumn, field specifications

Same as for Table clause; defined on page 3-50 and page 3-55.

Synch Operation

After data is loaded into the offline segment, the partial indexes built in the work segment must be synchronized, or merged, with the table's indexes with a SYNCH OFFLINE SEGMENT operation, as described in "Writing a SYNCH Statement" on page 3-104.

Example

This example illustrates a LOAD DATA statement to load data into an offline segment, followed by a SYNCH operation to synchronize the offline segment with the rest of the table.

```
load data
  inputfile 'sales_96_data'
  append
  discardfile 'discards_sales_96' discards 3
  into offline segment s_lq96 of table sales
    working_space work01 (
      perkey date (10) 'MM/Y*/d01',
      prodkey integer external (2),
      mktkey integer external (2),
      dollars integer external (3)
    );
  synch offline segment s_lq96 with table sales
  discardfile 'discards_synch';
```

Criteria Clause

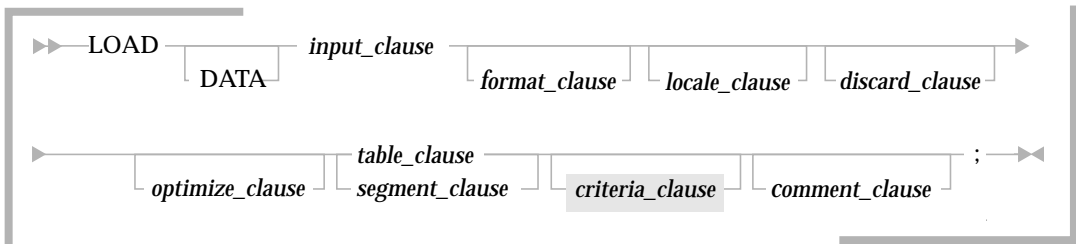
The Criteria clause allows you to specify that a comparison be made on each input record or on each row of data. The result of the comparison—true or false—causes the record to be loaded or discarded, depending on whether the Criteria clause specifies ACCEPT or REJECT. You can use this clause to ensure that the correct data is loaded or that rows are not aggregated more than once when a load operation in an AGGREGATE mode is interrupted.

The Criteria clause can be used for comparisons on numeric, character, and datetime datatype columns.

Caution: The Criteria clause uses the collating sequence and the character set from the database locale for all processing.

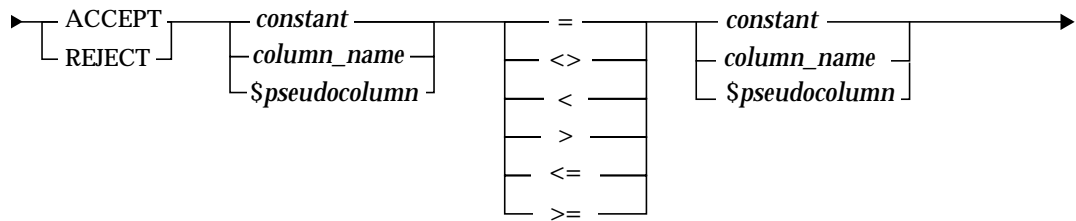
Syntax

The entire LOAD DATA statement syntax diagram is repeated here to provide a point of reference for the Criteria clause syntax:

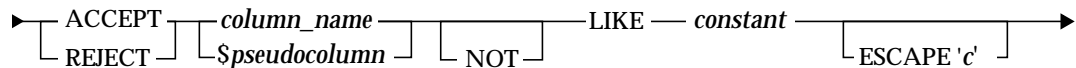


3

The syntax for `criteria_clause` on a numeric or datetime column is as follows:



The syntax for `criteria_clause` on a character column is as follows:



ACCEPT

Specifies that each row of input data that meets the comparison criteria (that is, it evaluates to TRUE) is loaded into the table; all others, including those containing NULL indicators, are discarded.

REJECT

Specifies that each row of input data that meets the comparison criteria (that is, it evaluates to TRUE) is rejected and discarded; all others, including those containing NULL indicators, are loaded.

constant

Any numeric, character, or datetime literal with which each input record or row of data is compared. Character and datetime constants, by definition, must be enclosed in single quotes (' ').

The constant datatype must be the same datatype as, or compatible with, the datatype of the column or pseudocolumn with which it is compared. For example, numeric constants cannot be compared with character or datetime columns.

Numeric, character, and datetime constants must conform to the definition of literals, as defined in the *SQL Reference Guide*. (Both ANSI SQL-92 datetime datatypes and the defined alternative datetime formats are valid.)

Character literals must be specified in the character set of the database locale. Decimal constants must be specified with a decimal radix.

column_name

References any numeric, character, or datetime column in the specified table. Referencing a column causes the TMU to check existing data in the table before loading a record. For example, based on the following ACCEPT clauses, the TMU loads a record only if the corresponding value existing in the Sales column of the table is equal to the constant value of 1000:

```
accept SALES = 1000
```

In this case, the syntax of *criteria_clause* is:

```
ACCEPT column_name = constant
```

The column reference can be used only in UPDATE and MODIFY modes; it cannot be used in APPEND, INSERT, or REPLACE modes (because the corresponding row does not yet exist in these modes). When a column reference is used in MODIFY mode, the ACCEPT/REJECT clause is applied only

when a record corresponds to an existing row, as identified by the primary key. When a record does not correspond to an existing row, no comparison is done and the record is loaded—in other words, no comparison is done for insert operations in MODIFY mode.

\$pseudocolumn

Refers to the contents of a numeric, character, or datetime field in the input record. Referencing a pseudocolumn causes the TMU to check the data in the input record, first checking for referential integrity and then making the Criteria clause comparison, before loading a record. For example, based on the following REJECT clause, the TMU rejects all records with values in the sales field that are less than 100:

```
reject $SALES < 100
```

In this example, the syntax of *criteria_clause* is:

```
REJECT $pseudocolumn < constant
```

A *pseudocolumn* must refer to a numeric, character, or datetime pseudocolumn defined in the field specification.

For more information about pseudocolumns, refer to page 3-51.

LIKE, NOT LIKE

Compares column or field values with a character string. The column or pseudocolumn referenced must be of CHARACTER datatype.

The percent (%) wildcard character matches any character string. The underscore (_) wildcard character matches any one character in a fixed position.

ESCAPE 'c'

The ESCAPE keyword, which can be used only with a LIKE or NOT LIKE comparison, defines a character (c) to serve as an escape character so that the wildcard characters can be treated as character literals rather than control characters. Use the ESCAPE keyword whenever the pattern to be matched contains a percent or underscore character.

Caution: The escape character must be specified using the database locale character set and can be either a single-byte or multibyte character.

Usage Notes

- In determining whether a row meets the comparison criteria, a three-valued logic is used: TRUE, FALSE, and UNKNOWN, with the NULL indicator evaluated as UNKNOWN. This behavior can be confusing; for example, a row containing a NULL indicator in the Dollars column is rejected for both of the following criteria:

```
accept dollars >= 1000
accept dollars < 1000
```

- Only one ACCEPT/REJECT Criteria clause can be present in each LOAD DATA statement. For example, all of the following ACCEPT/REJECT clauses are valid:

```
accept dollars >= 1000
accept dollars < 1000
reject dollars > 2000
```

but only one of them can appear in a single LOAD DATA statement.

Examples

The following examples illustrate valid Criteria clauses that compare column values to constants. (This kind of Criteria clause can be used only in UPDATE, MODIFY, or AGGREGATE mode.)

```
accept DISTRICT = 475
reject BATCH_ID > 100
accept CITY = 'Los Angeles'
accept AUTH = 'Y'
reject SALE_DATE <= date '1995-10-16'
-- ANSI SQL-92 datetime format
reject SALE_DATE <= '10-16-1995'
-- Alternative datetime format
accept SALE_TIMESTAMP >= timestamp '1995-10-16 13:13:13'
-- ANSI SQL-92 datetime format
```

The following examples illustrate valid Criteria clauses that compare input field values to constants:

```
accept $CITY = 'Los Angeles'
accept $TIME_COL >= '13:13:13'
reject $TIME_COL >= time '08:35:40'
accept $TIMESTAMP_COL >= timestamp '1995-10-16 12:13:13'

reject $CITY < 'Los Angeles' /* rejects records where input
value occurs before "Los Angeles" in alpha-sorted list */
```

The following example illustrates a valid Criteria clause that compares the input field value with the column value for each row:

```
accept $DISTRICT = DISTRICT
```

The following examples illustrate the use of the LIKE and NOT LIKE operators in a Criteria clause:

```
reject zip not like '950%'
-- rejects any zip codes that do not begin with '950'

accept city like 'Hamb_rg'
-- accepts cities like 'Hamburg, Hamberg, and so on.'

reject sales_pct like '%Monthly \%' escape '\'
-- rejects any string that ends with 'Monthly %'
```

The following example illustrates how to compare an input value with a constant. A record is rejected if the value in the field (in the input file) for the Cycle column is greater than or equal to 100. Values in that field are stored both in the Cycle column of the Sales table and in the \$cycle pseudocolumn for reference in the Criteria clause.

Input clause	load data
Format clause	inputfile 'sales.txt'
Discard clause	update
	discardfile 'sales_discards'
	discards 1
	optimize on
Table clause	into table sales
	(perkey position (7) integer external (5),
	prodkey position (15) integer external (2),
	mktkey position (20) integer external (2),
	cycle position (25) integer external (3),
	\$cycle position (25) integer external (3))
Criteria clause	reject \$cycle >= 100;

Alternatively, the two lines in the previous example that describe the field at position 25 can be combined into a single line:

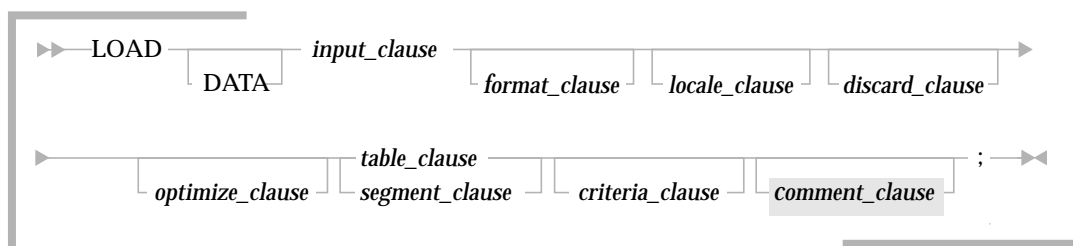
```
cycle as $cycle position (25) integer external (3))
```

Comment Clause

The Comment clause contains a user-defined text string that describes the load operation or the data being loaded. This information is then stored in the RBW_LOADINFO system table to provide a historical record regarding the loading of data into the specified table. The information can be retrieved by querying the RBW_LOADINFO system table.

The Comment clause and access to the corresponding information in the RBW_LOADINFO system table is available only with the Enterprise Control and Coordination option, which must be enabled with a license key.

The entire LOAD DATA statement diagram is repeated here to provide a point of reference for the Comment clause syntax:



The syntax for *comment_clause* is:



COMMENT

Causes the text string that follows to be stored in the RBW_LOADINFO system table in the Comment column for the row that describes this load operation.

'character_string'

Specifies the comment to be inserted into the RBW_LOADINFO system table. Up to 256 bytes (plus the single quotes) can be included. For information about legal character literal values, refer to the *SQL Reference Guide*.

The character string must be specified using the database locale character set.

Example

In this example, a comment is included that describes the source of the data being loaded. This information, together with the other information stored in RBW_LOADINFO, provides a useful record of load activity on the Sales table.

```
load data
inputfile 'sales.txt'
update aggregate
discardfile 'sales_discards'
into table SALES(
    perkey integer external (5),
    prodkey integer external (2),
    mktkey integer external (2),
    dollars decimal external (7,2),
    weight integer external (3) add
)
comment 'East coast, Q2-96,input file sales.txt';
```

If you want to check load activity on the Sales table, for example, to verify that a specific batch of data was loaded, you can query the RBW_LOADINFO system table as follows:

```
select *
from RBW_LOADINFO
where tname = 'SALES'
order by started;
```

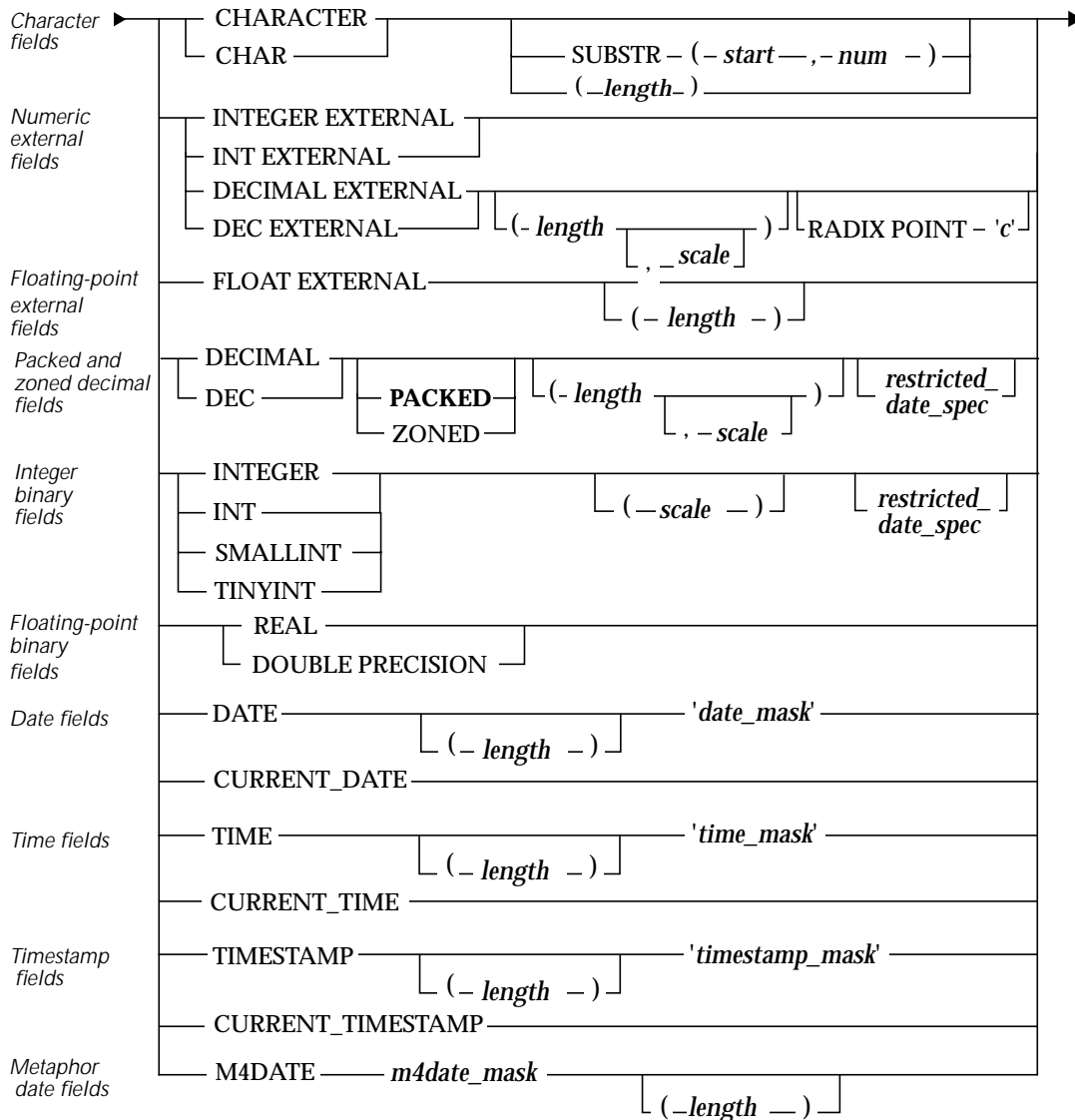
The information returned in this example is ordered by the timestamp at the start of the load operation (the column named “STARTED”) and includes the comment entered in the LOAD DATA statement, as well as other information for load operations on the Sales table.

Note: The RBW_LOADINFO system table contains a row for each of the last 256 LOAD DATA operations. To retrieve data in any specific order, include an ORDER BY clause in the SELECT statement. If the Enterprise Control and Configuration option is not enabled, information from the Comment column is not returned when the RBW_LOADINFO table is queried.

Fieldtypes

A fieldtype specifies the datatype of the input data in a simple field, as described on page 3-57. The TMU converts this datatype into the datatype defined for the column in the CREATE TABLE statement. The two datatypes must be compatible, as defined on page 3-111.

The syntax for *field_type* is:



Each fieldtype is defined in the following sections, with examples of each type.

length

Specifies the number of bytes in the input field. The length parameter is not allowed for data in SEPARATED format.

With fixed-format input data for the CHARACTER fieldtype, if no length is specified with the POSITION keyword or in the fieldtype description, then the column width defined for the table is used.

With fixed-format input data for DECIMAL (EXTERNAL, PACKED, ZONED), INTEGER EXTERNAL, FLOAT EXTERNAL, DATE, TIME, TIMESTAMP, and M4DATE fieldtypes, a length must be specified either with the POSITION keyword or with the length parameter in the fieldtype specification.

scale

Specifies the number of digits to the right of the decimal (radix) point. If a scale factor is not specified, the default value is zero. The scaled input is then converted to a value of the target datatype.

If the target datatype is INTEGER, TINYINT, or SMALLINT, all digits to the right of the decimal (radix) point must be zero; otherwise, a truncation error occurs and the record is discarded.

If the target datatype is FLOAT, DOUBLE PRECISION, or REAL, the scaled input is converted directly to the appropriate floating point value. If the scaled input exceeds the maximum allowed value for the target floating point datatype, then an overflow error occurs and the record is discarded.

If the target datatype is DECIMAL or NUMERIC, the decimal point of the input data (specified or default) is aligned with the decimal (radix) point of the target column (as specified by *scale* in the column definition). If the precision of the integer part of the scaled input value exceeds the precision of the target column, an overflow error occurs and the record is discarded. If the fractional part of the scaled input value exceeds the precision of the target datatype fraction and the excess least significant digits of the input data are non-zero, then a truncation error occurs and the record is discarded.

Character Fieldtype

CHARACTER, CHAR

Identifies a character string. A CHARACTER field can contain any character in the computer's character set.

The total length (in bytes) is specified by *length*. If the length of a target character column exceeds the length of the source field in the input record, then the input characters are left-justified with spaces to fill the excess. If the length of a target character column is less than the length of the source field, then the source field is truncated. If the length is not specified, the column definition in the CREATE TABLE statement determines the length.

SUBSTR (*start*, *num*)

Indicates that only a subset of a character field is to be loaded and specifies the starting position and the number of characters counted in characters, not bytes. (The POSITION keyword specification uses bytes.) This functionality is intended for use with multibyte character sets.

Tip: TMU performance is better if you are able to load substrings of fixed-format character data with the POSITION keyword rather than the SUBSTR function. With multibyte characters, however, you must use the SUBSTR function to extract a string because POSITION is byte-based whereas SUBSTR is character-based.

Examples

```
character          -- length specified by column definition
char (10)
character (24)
char (24) substr (1, 5)
```

The following example illustrates the use of the SUBSTR keyword in a LOAD DATA statement to load partial character strings into Col2 of the Sales table. The numbers in parentheses define the starting character position and the number of characters in the substring.

```
load data
...
into table sales (
    col1 decimal external radix point ',',
    col2 char substr(1,5)
);
```

For example, if the input data to Col2 is the string *California*, only the substring *Calif* will be loaded into the column.

Numeric External Fieldtypes

For numeric external (ASCII) fieldtypes in fixed-format input, the length must be specified either with the POSITION keyword or with the *length* parameter.

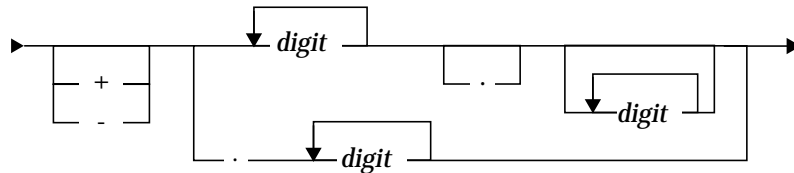
For numeric external fieldtypes (INTEGER EXTERNAL, DECIMAL EXTERNAL, and FLOAT EXTERNAL), leading and trailing blanks are allowed in a data input file.

INTEGER EXTERNAL, INT EXTERNAL

Identifies a string of characters representing a number in $[\pm]digits$ format. These numbers cannot exceed 38 digits. The *length* and *scale* parameters, as defined on page 3-83, can be used to represent the total length in bytes (the number of numeric digits, or precision) and the position of the decimal respectively.

DECIMAL EXTERNAL, DEC EXTERNAL

Identifies a string of characters representing a decimal number in the following format:



3

The *length* and *scale* parameters, as defined on page 3-83, can be used to represent the total length in bytes (the number of numeric digits, or precision) and the position of the decimal respectively. These numbers cannot exceed 38 digits.

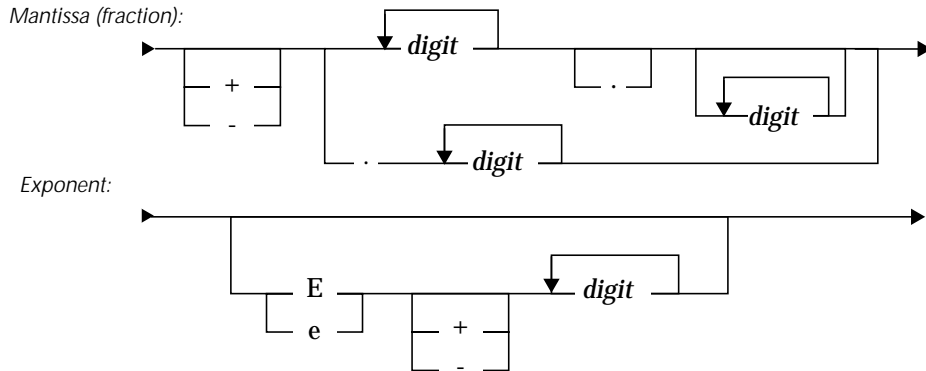
RADIX POINT 'c'

Specifies any single-byte or multibyte character that is used to indicate the radix in numeric data. If the input records contain separated data, the radix character must be different from the character specified as the separator character in the Format clause. If no radix point is specified, the default radix of the input locale is used.

Caution: If specified, the radix character must be specified using the database locale character set, but it can be either a single-byte or multibyte character. If the character used as a radix point in the input data cannot be expressed as a character in the warehouse database, then the input data cannot be interpreted correctly.

FLOAT EXTERNAL

Identifies a string of characters representing a floating point number in the following format:



The *length* parameter can be used to represent the total length of the field in bytes.

Examples

```
int external          --length specified by POSITION clause*
integer external (8)
decimal external      --length specified by POSITION clause*
decimal external (5)
decimal external (5,2)
float external        --length specified by POSITION clause*
float external (8)
```

*If the input records are in separated format, the length can be determined implicitly.

Packed and Zoned Decimal Fieldtypes

For packed and zoned decimal fieldtypes in fixed-format input, the length must be specified either with the POSITION keyword or with the *length* parameter. The *scale* parameter, as defined on page 3-83, can be used to specify the position of the decimal.

DECIMAL, DEC, DECIMAL PACKED

Identifies decimal numbers in packed format. These numbers cannot exceed 38 digits.

If present, the length value specifies the number of bytes in the input record field, which yields a numeric value with a precision of $(2 * \text{length}) - 1$. Each digit occupies 1/2 byte, with 1/2 byte reserved to indicate the sign (+ or -) of the value, so the maximum length is 20 bytes.

DECIMAL ZONED

Identifies decimal numbers in an IBM-zoned decimal representation. This format is supported only with the FORMAT IBM clause. These numbers cannot exceed 38 digits.

If present, the length value specifies the number of bytes in the input record field, which corresponds to the number of numeric digits (precision) in the input value. Each digit occupies one byte, so the maximum length is 38 bytes.

restricted_date_spec

Provides a restricted datetime format mask that allows packed or zoned decimal input data to be loaded into datetime columns. For more information about the restricted datetime masks, refer to “Restricted Datetime Masks for Numeric Fields” on page 3-101.

Examples

```
decimal          -- packed; length specified by POSITION clause
dec              -- packed; length specified by POSITION clause
decimal packed   -- length specified by POSITION clause
dec packed (5,2)
decimal zoned     -- length specified by POSITION clause
decimal zoned (5,2)
decimal zoned (5)
decimal packed (8) date 'YYYYMMDD'
```

If the TMU LOAD DATA script references a packed decimal input fieldtype of DECIMAL PACKED (6,3) for conversion to a database datatype of DECIMAL (5,2), the following conversions or errors occur:

Input Value	Result
473220	473.22
819077	Truncation error; record discarded.
2323478320	Overflow error; record discarded.

Integer Binary Fieldtypes

For integer binary fieldtypes, an optional scale parameter can be specified, as defined on page 3-83. The length is implied by the fieldtype.

INTEGER, INT

Identifies a four-byte binary integer in two's-complement representation.

SMALLINT

Specifies a two-byte binary integer in two's-complement representation.

TINYINT

Specifies a one-byte binary integer in two's-complement representation.

restricted_date_spec

Provides a restricted datetime format mask that allows integer binary input data to be loaded into datetime columns. For more information about the restricted datetime masks, refer to "Restricted Datetime Masks for Numeric Fields" on page 3-101.

Examples

```

int
integer (3)
smallint
smallint (2)
tinyint
integer date 'YYYYMMDD'

```

If the TMU LOAD DATA script references a binary integer input fieldtype of INT (3) for conversion to a database datatype of DECIMAL (5,2), the following conversions or errors occur:

Input Value	Result
473220	473.22
819077	Truncation error; record discarded
2123478320	Overflow error; record discarded

Floating-Point Binary Fieldtypes

REAL

Specifies four-byte floating-point numbers. This fieldtype is not supported if the FORMAT IBM keywords are included in the Format clause.

DOUBLE PRECISION

Identifies an eight-byte floating-point number. This fieldtype is not supported if the FORMAT IBM keywords are included in the Format clause.

Examples

```
real  
double precision
```

Datetime Fieldtypes

DATE, TIME, TIMESTAMP

Identifies character data that will be processed and stored as date, time, and timestamp information. For datetime fieldtypes in fixed-format input, the length must be specified either with the POSITION keyword or with the *length* parameter.

Unlike other fieldtypes, DATE, TIME, and TIMESTAMP fields are each composed of subfields. The *date_mask*, *time_mask*, and *timestamp_mask* elements represent format masks that must be defined in the field specification to specify which subfields are used and their order and length. For more information about format masks, refer to “Datetime Format Masks for Datetime Fields” on page 3-93.

The following table defines the legal subfields for each datetime fieldtype; required fields for each fieldtype are indicated in **bold**.

Fieldtype	Legal Subfields
DATE	year and Julian date; or year , month, date
TIME	hour , minute, second, fractional second
TIMESTAMP	year , Julian date, hour , minute, second, fractional second, or year , month, date, hour , minute, second, fractional second

CURRENT_DATE

Specifies that the value to be used is the time at which the row is actually loaded; remember that the date value changes at midnight.

CURRENT_TIME

Specifies that the value to be used is the time at which the row is loaded.

CURRENT_TIMESTAMP

Specifies that the value to be used is the value of CURRENT_TIME and CURRENT_DATE at the time the row is loaded.

M4DATE

String of characters representing a date in the format specified by *format*.

M4DATE *m4date_mask*

M4DATE *m4date_mask*(*length*)

The length must be specified with the POSITION keyword or with the *length* parameter, which specifies the total length in bytes.

The TMU converts this string into the Metaphor DIS date format and stores it as an integer when loading a table. The format must be one of the following:

Format	Example: April 10, 1996
YYDDD or YYYYDDD	96/100 1996/100
YYMD or YYYYMD	960410 1996/4/10
MDYY or MDYYYY	4/10/96 04101996
DMYY or DMYYYY	10/4/96 10041996

where:

D One or two digits specifying the day of the month.

DDD Three digits specifying day of the year.

M One or two digits specifying the month of the year.

YY and YYYY Two and four digits respectively specifying a year. A two-digit year *nn* is interpreted as 19*nn*.

The day, month, and year fields can be contiguous or can be separated by a single blank, slash (/), hyphen (-), period (.) or comma (,). If the fields are contiguous, then each day and month representation must contain two digits.

Note: All formats defined for the M4DATE fieldtype can be represented by legal format masks for datetime literals, which means input data containing dates in a M4DATE format can be loaded into a DATE column without any modification of the input data. Because the datetime scalar functions cannot be used on the integer columns loaded from M4DATE fields, use of M4DATE fields is not recommended for new databases.

Datetime Format Masks for Datetime Fields

A format mask for a datetime fieldtype is created by concatenating legal subfield format specifiers, using either a fixed- or variable-length subfield or a combination of both. For example, a DATE format mask composed of month, day, and year subfields might look like one of these:

'MMDDYYYY'	DATE format mask, fixed format input:
'M*/D*/Y*'	DATE format mask, variable format input
'MM D* YYYY'	DATE format mask, both fixed and variable
'm8d16y1997'	DATE format mask, constant date (Aug 16, 1997); date constants can also be defined with a CONSTANT field specification, as described on page 3-65.

Examples

```

date (8) 'MMDDYYYY'
date (8) 'DDMMYYYY'
date 'y1996m8d17'
    
```

Note: Binary integer and packed or zoned decimal input data require the use of restricted format masks, as described in “Restricted Datetime Masks for Numeric Fields” on page 3-101.

Subfield Components

The following table defines each subfield component, its default value, and its specifier in the format mask. Examples for fixed- and variable-length subfields are provided in the sections that follow.

Subfield Component	Default Value	Mask Specifier	Subfield Range and Interpretation
Year	Required subfield. No default; must be specified	Y	1 to 9999; number of Ys specify number of digits to be read.
		y?Y* or y?YY	00-49 imply 2000 to 2049; and 50-99 imply 1950-1999.
		ynY* or ynYY	Century is fixed: 1 to 99, specified by <i>n</i> . Year expressed by 1 or 2 digits.
		yn	Year is fixed: 1 to 9999, specified by <i>n</i> .
Julian Day (day of year)		J	1 to 366; number of Js specify number of digits to be read. January 1st is 1.
		jn	Day is fixed: 1 to 366, specified by <i>n</i> . Date value stored is adjusted for leap years.
Month	1	M	1 to 12; number of Ms specify number of digits to be read.
		mn	Month is fixed: 1 to 12, specified by <i>n</i> .
		Mon	*Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec
		Month	*January, February, March, April, May, June, July, August, September, October, November, December
Day	1	D	1-31; number of Ds specify number of digits to be read. Further constrained by month and year according to rules in Gregorian calendar.
		dn	Day is fixed: 1-31, specified by <i>n</i> .

Subfield Component	Default Value	Mask Specifier	Subfield Range and Interpretation
Hour	Required subfield. No default value; must be specified	H	0 to 23; number of <i>Hs</i> specify number of digits to be read.
		A AM	*Optional AM/PM subfield specifier (not case sensitive). <i>A</i> implies data contains either A or P. <i>AM</i> implies data contains AM or PM. 12-hour times are converted to 24-hour times.
		hn	Hour is fixed: 0 to 23, specified by <i>n</i> .
Minute	0	I	0 to 59; number of <i>Is</i> specify number of digits to be read.
		in	Minute is fixed: 0 to 59, as specified by <i>n</i> .
Second	0	S	0 to 59; number of <i>Ss</i> specify number of digits to be read.
		sn	Second is fixed: 0 to 59, as specified by <i>n</i> .
Second fraction	0	F	0 to 999999; number of <i>Fs</i> specify number of digits to be read, as well as the scaling factor of this component, which is $10^{-\text{\#of Fs}}$ For example, <i>F</i> indicates tenths of a second, <i>FF</i> indicates hundredths of a second.
		fn	Second fraction is fixed and specified in microseconds: 0 to 999999, specified by <i>n</i> .

* Interpretation of these subfields (Month, Mon, and AM, but not A) is locale-specific.

In a fixed-length subfield, the last letter of the subfield mask character repeats once for each character of the input. Fixed-length subfields cannot contain blanks.

In a variable-length subfield, the subfield specifier is followed by an asterisk (*). For Mon* and Month*, enough characters are processed to complete the month name; in the case of the numeric subfields, characters are processed until the first non-digit character. Leading blanks are ignored. After the specified number of characters have been processed, zero or more digits can follow.

An underscore (_) can be used to indicate the end of the subfield and that the next character should be ignored. The underscore can also be used as a wildcard to skip bytes; it must be repeated for each byte to be skipped.

Regardless of the format specified for the subfields, the number of bytes processed is limited by the *length* parameter for the datetime field.

Examples of Subfield Masks

Fixed-length subfields:

YYYY	Indicates 4 digits for year.
MMDDYYYY	Indicates 2 digits each for month and day and 4 digits for year.
____Mon	Indicates the 4 bytes (represented by 4 underscores) preceding the 3-character month should be ignored.

Variable-length subfields:

D*/M*/YYYY	Indicates 1 or more digits for day and month; 4 digits for year; subfields separated by a slash (first non-digit character).
Mon d1 y?Y*	Indicates short month; day is 01; 1- or 2-digit year; subfields separated by spaces.

Combination of fixed- and variable-length subfields, where the numbers beneath each subfield correspond to the explanations that follow:

'Month* D*, YY*_HH:II:SS.FFFF* '
(1) (2) (3) (4) (5) (6) (7)

- (1) Skip any blanks, read full month name, and check for a blank.
- (2) Skip any blanks, read one or more digits for date, and check for a comma.
- (3) Skip any blanks and read two or more digits for year; ignore the non-digit character following the year.
- (4) Read two digits for hours; no white space or other non-digits allowed; check for a colon.
- (5) Read two digits for minutes; no white space or other non-digits allowed; check for a colon.
- (6) Read two digits for seconds; no white space or other non-digits allowed; check for a period.
- (7) Skip any blanks, read four or more digits for fractional seconds, and ignore anything that follows.

Examples of Input Fields and Format Masks

The following table contains various types of input fields and suggests masks to read them.

To read fields like this:	Use a format mask like this:
June 29 May 14 April 1	'Month D* y1996'
June 29 49 May 14 96 April 1 1	'Month D* y?Y*' (years: 2049, 1996, 2001) 'Month D* Y*' (years: 0049, 0096, 0001) 'Month D* y19Y*' (years: 1949, 1996, 1901)
Tue Jun 4 16:50:49 PDT 1996 Tue Jun 25 16:50:49 PDT 1996	For timestamp: '___ _Mon D* HH:II:SS___ _YYYY' For date: '___ _Mon D* _____YYYY'
Tue Jun 4 1996 Sat Jun 15 1996	'___ _Mon* D* YYYY'
Tue Jun 04 16:50:49 PDT 1996	For timestamp: '___ _Mon DD HH:II:SS___ _YYYY' For time: '_____HH:II:SS_____'
Jun 29, 1996 Jun 7, 1996	'Mon D*,_YYYY' or 'Mon D*,Y*''
Jun 29 1996 Jun 9 1996 Jun 2 1996	'Mon D* YYYY*'
01/15/96 08:26 AM (stored as 1996-01-15 08:26:00) 11/15/01 06:15 pm (stored as 2001-11-15 18:15:00)	'MM/DD/y?Y* HH:II AM'
1995/060 (stored as 1995-03-01) 1996/060 (stored as 1996-02-29) 1991/366 (rejected) 1980/366 (stored as 1980-12-31)	'YYYY/J*' or 'YYYY/JJJ'
1997年01月06日	'YYYY_ _MM_ _DD_ _' (where the multibyte characters for year, month, and day are skipped to produce 01-06-1998 (January 6, 1998).

Example: Loading Datetime Data

This example illustrates how data in various formats is loaded into DATETIME columns.

A table named Datetime is defined as follows:

```
create table datetime (  
  d1 date,  
  d2 date,  
  d3 date,  
  ts1 timestamp(0),  
  t1 time,  
  ts2 timestamp(0))
```

The data for the Datetime table is in a file named *datetime_inputs* with fields separated by an asterisk (*). The first three records of input data in *datetime_inputs* look like this:

```
96/12/25*December 25 96*07042359*11:59:00PM*Tue Jun 29 16:40:55 PDT 1996  
06/12/25*December 25 6*07042359*11:59:00AM*Tue Jun 29 16:40:55 PDT 1996  
6/12/25*December 25 6*07042359*12:59:00AM*Tue Jun 29 16:40:55 PDT 1996  
|           |           |           |           |  
to d1       to d2       to ts1      to t1       to ts2
```

A LOAD DATA statement to load this data follows; note that the current date will be loaded into column D3:

```
load data  
inputfile 'datetime_inputs'  
replace  
format separated by '*'  
into table datetime (  
  d1 date 'y19Y*/M*/D*', -- Date subfields separated by /  
  d2 date 'Month D* y?Y*', -- Date subfields separated by space  
  d3 current_date, -- Rows loaded with date at time of load  
  ts1 timestamp(8) 'y1996MMDDHHII', -- Fixed format mask  
  t1 time 'HH:II:SSAM', -- Time subfields separated by :  
  ts2 timestamp '_ _ _ _ Mon DD HH:II:SS_ _ _ _ Y*'  
  -- First 4 characters and 5 characters  
  -- between S and Y are ignored  
);
```

If the data is loaded on July 1, 1996, the information stored in the Datetime table looks like this:

d1	d2	d3	ts1	t1	ts2
1996-12-25	1996-12-25	1996-07-01	1996-07-04 23:59:00	23:59:00	1996-06-29 16:40:55
1906-12-25	2006-12-25	1996-07-01	1996-07-04 23:59:00	11:59:00	1996-06-29 16:40:55
1906-12-25	2006-12-25	1996-07-01	1996-07-04 23:59:00	00:59:00	1996-06-29 16:40:55

Suppose you wanted to load a specific date—Aug. 16, 1996—instead of the date in the input record into the D2 column. The LOAD DATA statement to load this data uses a constant field with a date value as follows:

```
load data
inputfile 'datetime_inputs'
replace
format separated by '|'
into table datetime (
    d1 date 'y19Y*/M*/D*', -- Date subfields separated by /
    d2 date '1996-08-16', -- Rows loaded with 1996-08-16
    d3 current_date, -- Rows loaded with date at time of load
    ts1 timestamp(8) 'y1996MMDDHHII', -- Fixed format mask
    t1 time 'HH:II:SSAM', -- Time subfields separated by :
    ts2 timestamp '_ _ _ _Mon DD HH:II:SS_ _ _ _Y*'
        -- First 4 characters and 5 characters
        -- between S and Y are ignored
);
```

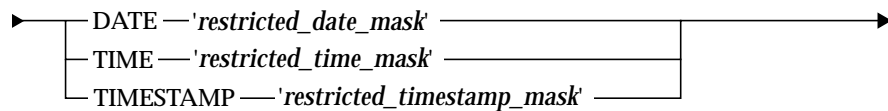
If the data is loaded on July 1, 1996, the information stored in the Datetime table looks like this:

d1	d2	d3	ts1	t1	ts2
1996-12-25	1996-08-16	1996-07-01	1996-07-04 23:59:00	23:59:00	1996-06-29 16:40:55
1906-12-25	1996-08-16	1996-07-01	1996-07-04 23:59:00	11:59:00	1996-06-29 16:40:55
1906-12-25	1996-08-16	1996-07-01	1996-07-04 23:59:00	00:59:00	1996-06-29 16:40:55

Restricted Datetime Masks for Numeric Fields

The TMU can load binary integer or packed or zoned decimal input data into datetime columns when the input fields are described by a restricted datetime format mask. For example, if an input record contains a date value for February 14, 1998, represented as 19980214 in a packed decimal field, the TMU can extract the date from the input field and store it in a DATE column.

The syntax for *restricted_date_spec* (in the *fieldtype* specification on page 3-82) is:



***restricted_date_mask, restricted_time_mask,
restricted_timestamp_mask***

Specify the subfields present in the input fields. These masks are defined like the masks for the DATE, TIME, and TIMESTAMP input fields on page 3-93, with the following restrictions (because the inputs are numeric, fixed-format):

- No separators (slashes) are allowed.
- No variable-width fields are allowed: The mask cannot contain asterisks (*).
- No alphabetic representations are allowed: The mask cannot contain “Mon,” “Month,” or AM/PM specifiers.
- A mask for a decimal input field must represent exactly the number of digits that can occur in decimal input.
- A mask for an integer input field cannot represent more digits than can occur in the number of bytes represented by the fieldtype:

Integer Fieldtype	Maximum Digits in Mask
INTEGER (4 bytes)	10
SMALLINT (2 bytes)	5
TINYINT (1 byte)	3

Note: Scale values are ignored when restricted date masks are used. A length value—from a length argument or a Position clause—is required with packed or zoned decimal fieldtypes and must be consistent with the format mask.

The underscore (`_`) character can be used in the mask to indicate that an input digit should be ignored.

Input Data

The input data must meet the following requirements:

- No negative numbers are allowed. If the field contains a negative number, the row is discarded.
- For packed or zoned decimal input, the number of digits present in the input data must be the same as the number of digits indicated by the mask. If they are not the same, an error occurs and the TMU does not begin the load operation.
- For binary integer input:
 - If the input data contains more digits than indicated by the mask, the row will be discarded.
 - If the input data contains fewer digits than indicated by the mask, the input is considered to have leading zeroes. For example, a mask of 'YYYYMMDD' used for input data 980101 causes the data to be stored as 00980101.

Examples

The following examples illustrate valid and invalid restricted masks.

Restricted Date Mask	Examples	Comments
YYYYMMDD	19980214	Valid: 4 digits of year, 2 digits each for month and day.
y1998MMDD	0214 00000214	Valid: Year fixed at 1998, followed by 2 digits each of month and day.
y?YYJJ	98046	Valid: Imply century from 2-digit year, followed by 3 digits of Julian date.
y?Y*	–	Invalid because it contains a variable-length year field.
MM/DD/YYYY	–	Invalid because it contains separators (/).

This example illustrates how to use the restricted date mask in a LOAD DATA statement. The Period table, created by the CREATE TABLE statement on the left side, contains a DATE datatype column named Date_Col. The LOAD DATA statement on the right side loads input records that contain date information stored as binary integers into the Date_Col column in the Period table.

CREATE TABLE statement

```
create table period (  
  perkey integer not null,  
  month char (15),  
  year integer,  
  quarter integer,  
  tri integer,  
  date_col date,  
  primary key (perkey))  
maxrows per segment 256
```

LOAD DATA statement

```
load data  
  inputfile 'aroma_period.txt'  
  replace  
  discardfile 'aroma_discards'  
  discards 1  
into table period (  
  perkey integer external (4),  
  month char (4),  
  year integer external (4),  
  quarter integer external (1),  
  tri integer external (10),  
  date_col integer date 'YYYYMMDD'  
) ;
```

In this example, the input records contain the date information in the format 'YYYYMMDD' (for example, 19971225) stored as a binary integer. The TMU extracts the date information from the binary input and stores it as a DATE datatype in the Date_Col column.

Writing a SYNCH Statement

If data is loaded into an offline segment, you must complete the load operation by synchronizing the segment with the table and its indexes before the segment can be brought online and made available for use. Synchronization is necessary only for offline load operations: If the segment into which data was loaded was online at the time of the load, synchronization is not necessary.

Note: The SYNCH operation acquires an exclusive lock on the target table but this operation is much quicker than an online load of the table.

To perform this synchronization, run the TMU with a control file containing a SYNCH OFFLINE SEGMENT statement. You can include this statement in the same control file as the LOAD DATA statement or you can put it in a separate control file. At the end of the synchronization operation, the work segment used for the offline load is detached from the table and available for reuse.

If you decide, after loading data into the segment, that you want to remove the newly loaded data rather than incorporate it into the table, you have two choices:

- Delete all the data in the segment with the ALTER SEGMENT...CLEAR statement; this choice is appropriate if the segment was empty or if you do not want the data that was in the segment before the load operation.
- Delete only the newly loaded data with the UNDO LOAD option to the SYNCH SEGMENT statement. This choice is useful for segments that contained data you want to preserve before the offline operation was done.

The syntax for a SYNCH OFFLINE SEGMENT statement is:

```

▶▶—— SYNCH OFFLINE SEGMENT —— segment_name — WITH TABLE——▶
▶—— table_name ——┐
                    │—— DISCARDFILE —— ' discard_filename ' ——┐
                    │—— UNDO LOAD ——┐
                    └──────────────────────────────────────────┘ ; —▶▶
  
```

segment_name

Specifies the name of an offline segment that contains newly loaded data not yet synchronized with the owning table and its indexes.

table_name

Names the owning table of *segment_name*.

DISCARDFILE *discard_filename*

Specifies a file to which all duplicate rows are discarded. This ASCII file contains rows in the same format as those rows discarded during an optimized load or an UNLOAD EXTERNAL operation on a table. For more information about discard files, refer to page 3-47.

If this clause is omitted, no file is written.

UNDO LOAD

Synchronizes the segment with the table and its indexes by deleting all the rows that were just added to the segment. This operation is useful in cases where you discover you loaded the wrong data or where a lot of rows are discarded unexpectedly and you want to start over. It removes all evidence of the previous offline load operation, leaving intact the rows that were in the segment prior to the offline load.

Caution: The UNDO LOAD option does not work in REPLACE mode.

Example

This example illustrates a control file containing a LOAD DATA statement and a SYNCH SEGMENT statement to synchronize the newly loaded offline segment with the rest of the table.

```
load data
  inputfile 'sales_96_data'
  append
  discardfile 'discards_sales_96' discards 3
  into offline segment s_lq96 of table sales
  working_space work01 (
    perkey date (10) 'MM/Y*/d01',
    prodkey integer external (2),
    mktkey integer external (2),
    dollars integer external (3)
  ) ;
synch offline segment s_lq96 with table sales
  discardfile 'discards_synch';
```

3

Because the SYNCH operation requires an exclusive lock on the table, you might prefer to use a separate control file for that operation so you can perform it at a time when users are not accessing the table.

After the SYNCH operation, you must use ALTER SEGMENT...ONLINE before you can access the segment.

Format of Input Data

The TMU supports a wide variety of input data formats; however, not all platforms support all formats. The TMU accepts both disk and tape input and system standard input; tape files can be ANSI standard label or TAR (Tape ARchive) formats. Record format can be either fixed or separated.

Note: Tape input is not supported on Windows NT systems.

Not all combinations of data, record format, and tape formats are valid. The following table defines the valid combinations.

Input Device	File Format	Records	Data
Disk files standard input (stdin)	Standard flat files	Fixed format	ASCII EBCDIC
		Separated format	ASCII
Tape Devices: 4 mm DAT 8 mm (Exabyte) 1/4" cartridge* 9-track reel 3480/3490 18-track cartridge**	TAR	Fixed format	ASCII EBCDIC
		Separated format	ASCII
	ANSI standard label	Fixed and variable length	ASCII EBCDIC

*Supported only for TAR tapes.

**Supported only by variable-block-length device drivers for ANSI Standard Label tapes.

The TMU also provides limited support for IBM standard label tapes. It can read IBM standard label tapes with fixed-length records in EBCDIC FB format; however, it cannot read variable-length (VB or VBS) tapes. The filenames on the tape must be uppercase.

In addition to fixed and separated record formats, the TMU also supports an internal storage format, UNLOAD, which is used to load data files written with a TMU UNLOAD control file. UNLOAD-format tapes can be written in either TAR or standard label format.

The TMU handles disk and tape files differently with respect to file format, record length and format, and data type, as described in the following sections.

Disk Files

Disk files can contain either fixed-format or separated-format records. If the SEPARATED keyword is not present in the LOAD DATA statement, the TMU assumes a fixed format.

Fixed-Format Records

For fixed-format records, all records are a fixed length and all fieldtypes are allowed. The TMU determines the record length from the RECORDLEN clause in the LOAD DATA statement according to the following rules:

- If the RECORDLEN value is greater than the sum of field lengths specified in the field specification, the RECORDLEN value is used as record length.
- If the RECORDLEN value is less than the sum of field lengths specified in the field specification, a warning message is issued and the load process stops.
- If the RECORDLEN clause is missing, each record is read until a newline is encountered; binary data is not permitted.
- If the RECORDLEN clause is missing and a record is shorter than the sum of field lengths specified in the field specification, a warning message is issued and the row is written to the discard file.
- If the RECORDLEN clause is missing and a record is longer than the sum of field lengths specified in the field specification, data beyond that length is discarded.

To read EBCDIC in fixed-record format, include the FORMAT IBM clause in the LOAD DATA statement. This clause forces CHARACTER and EXTERNAL fields to be converted from EBCDIC to ASCII, and INTEGER fields to be converted to the byte-ordering of the native machine.

Examples

The following examples illustrate how to read fixed-format disk files.

The RECORDLEN value equals the sum of the field lengths:

```
LOAD DATA
  INPUTFILE 'mkt.txt'
  RECORDLEN 126
  ...
```

The RECORDLEN value equals the sum of the field lengths; data is in EBCDIC:

```
LOAD DATA
  INPUTFILE 'mkt.txt'
  RECORDLEN 126
  FORMAT IBM
  ...
```

The data is newline-delimited; notice that RECORDLEN is not present:

```
LOAD DATA
  INPUTFILE 'mkt.txt'
  ...
```

Separated-Format Records

For separated-format records, the TMU determines the length of each field in the record by the separator character defined in the SEPARATED clause of the LOAD DATA statement. The end of each record is indicated by the newline character (or by the end of file for the last record in the file).

Only character and external fieldtypes are allowed; length values and POSITION keywords are ignored.

If a RECORDLEN clause is present with separated-format records, then the record is read until either a newline is read or the number of characters specified by RECORDLEN is read, whichever comes first.

If records in separated format are longer than 8192 bytes, then a RECORDLEN clause, which specifies the maximum length of a record, must be used.

Examples

These examples illustrate LOAD DATA statements to read separated-format disk files or system standard input.

The fields are separated by a comma and the record is terminated by a newline character:

```
LOAD DATA
  INPUTFILE 'mkt.txt'
  FORMAT SEPARATED BY ','
  ...
```


The fields are separated by a slash and the record length is 126 bytes:

```
LOAD DATA
  INPUTFILE 'mkt.txt'
  RECORDLEN 126
  FORMAT SEPARATED BY '/'
  ...
```

The input is read from standard input, fields are separated by a colon, and the record is terminated by a newline character:

```
LOAD DATA
  INPUTFILE '-'
  FORMAT SEPARATED BY ':'
  ...
```

Tape Files on UNIX Systems

Tape files can be read from TAR or ANSI standard label tapes, as described in the following sections.

TAR Tapes

TAR tape files are handled like disk files for both fixed-format and separated-format records.

The TMU can read TAR tape files that span multiple tape volumes; however, the TMU does not support multiple TAR archives on a single tape.

3

Example

This example illustrates a LOAD DATA statement to read TAR tape files.

The fields are separated by a comma and the record is terminated by a newline character:

```
LOAD DATA
  INPUTFILE '/disk1/mkt.txt'
  TAPE DEVICE '/tape_dev'
  FORMAT SEPARATED BY ','
  ...
```

ANSI Standard Label Tapes

ANSI standard label tapes can contain either fixed-length or variable-length records. The TMU determines the record length from the tape label; if the RECORDLEN clause is present, it is ignored.

If the SEPARATED clause is present, the TMU assumes separated-format records. It ignores the label and reads each tape record, scanning for the field-separator character defined in the TMU LOAD DATA statement. If there are fewer fields than specified in the command, the TMU issues an error message and discards the row. If there are more fields, it ignores the remaining fields in the tape record.

If the SEPARATED clause is not present, the TMU assumes fixed-format data. It determines the format from the label and reads both fixed-length and variable-length records according to the format described in the tape label. If the record is shorter than expected, it issues an error message and discards the record. If the record is longer, it ignores the remaining characters in the record.

Spanned format tapes are not supported, and a single tape record is not separated into multiple table rows.

To read EBCDIC format data with either fixed-format or separated-format records, use the FORMAT IBM or FORMAT IBM SEPARATED BY 'c' clauses respectively.

Examples

These examples illustrate commands to read ANSI standard label tape files.

The record length is determined by the label and the field lengths are specified in the statement:

```
load data
  inputfile 'mkt.txt'
  tape device 'tape_dev'
  ...
```

The record length is determined by the label; the fields are separated by a comma; and the data is in EBCDIC:

```
load data
  inputfile 'mkt.txt'
  tape device 'tape_dev'
  format ibm separated by ','
  ...
```

Fieldtype Conversions

The TMU performs conversions between compatible fieldtypes and datatypes, converting the data in each field in the input record to the datatype of the corresponding column in the table.

A CHARACTER field is compatible only with the datatype CHARACTER. Although all numeric fields are compatible with any numeric datatype, the conversion can yield unexpected results, as specified in the table on page 3-112.

Datetime input data (in either datetime or binary input fields) is compatible only with other datetime datatypes, as defined in the table on page 3-113.

Rows are discarded in the following cases:

- The data in an input field is not compatible with the datatype of the output table column.
- The value of a numeric input field exceeds the maximum possible value of the output table column.

The following table defines how blanks and empty fields in CHARACTER or NUMERIC EXTERNAL fields are mapped to column values:

Field	Column Value
NUMERIC EXTERNAL field with blanks	Default column value
Empty NUMERIC EXTERNAL field	NULL
CHARACTER field with blanks	Blank
Empty CHARACTER field	Default column value

The following table defines the legal conversions and the results that occur for non-datetime datatypes. Rows in this table represent input-record fieldtypes declared in the TMU LOAD DATA statement. Columns in this table represent the datatypes used in warehouse tables and declared with the CREATE TABLE command. The entry in each table cell defines what can happen when input data of a given fieldtype is loaded into a table column of a given datatype.

Input-Record Fieldtypes	Table Datatypes						
	Char	Integer	Smallint	Tinyint	Decimal	Real	Float, Double
Character, CONSTANT 'str', Concat	C	N/A	N/A	N/A	N/A	N/A	N/A
Integer External	N/A	O	O	O	O	S	S
Decimal External	N/A	O,D,S	O,D,S	O,D,S	O,D,S	S	S
Float External, CONSTANT f	N/A	O,S	O,S	O,S	O,S	O,S	O,S
Decimal, Packed Decimal, Zoned Decimal	N/A	O,D	O,D	O,D	O,D	S	S
Integer CONSTANT i SEQUENCE i	N/A	None	O	O	O	S	OK
Smallint	N/A	OK	None	O	O	O,S	OK
Tinyint	N/A	OK	OK	None	O	OK	OK
Real	N/A	O,S	O,S	O,S	O,S	None	OK
Double	N/A	O,S	O,S	O,S	O,S	O,S	None
CONSTANT NULL	None	None	None	None	None	None	None
M4Date	N/A	Date	N/A	N/A	N/A	N/A	N/A

Abbreviations used in this table are:

- N/A Not allowed; load process is stopped.
- None No conversion required.
- C Character-to-character, left-justified, space-fill or truncation on right.
Truncation does not cause the record to be discarded.
- OK No overflow or loss of significance possible.
- O Overflow possible; overflow causes the record to be discarded.
Overflow might occur when data is loaded into a DECIMAL column
because the precision is not set large enough to include both the
whole number and its decimal component.
- S Loss of significance possible; not a fatal error.

- D

Decimal point alignment. Truncation of digits to the right of the decimal point possible; truncation causes the record to be discarded.
- Date

The date represented is converted to the number of days since January 1, 1970. Dates prior to this date are represented as negative numbers.

The following table defines the legal datatype conversions for datetime datatypes:

Input-Record Fieldtypes *	Table Datatypes		
	DATE	TIME	TIMESTAMP
DATE	None	N/A	Dateparts: from input data Timeparts: midnight
TIME	N/A	None	Dateparts: 1900-01-01 Timeparts: from input data
TIMESTAMP	Dateparts	Timeparts	None

* including binary numeric input data

Abbreviations used in the preceding table are:

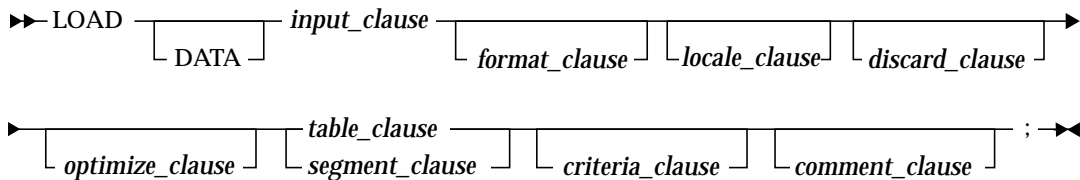
- N/A

Not allowed; load process is stopped.
- None

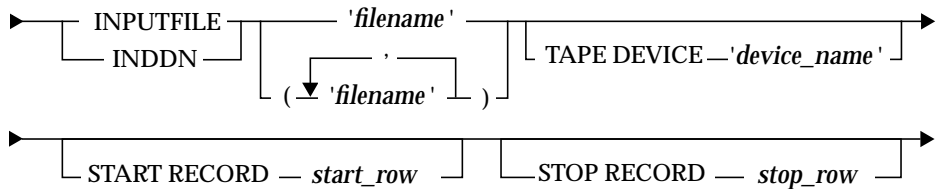
No conversion required.

LOAD DATA Syntax Summary

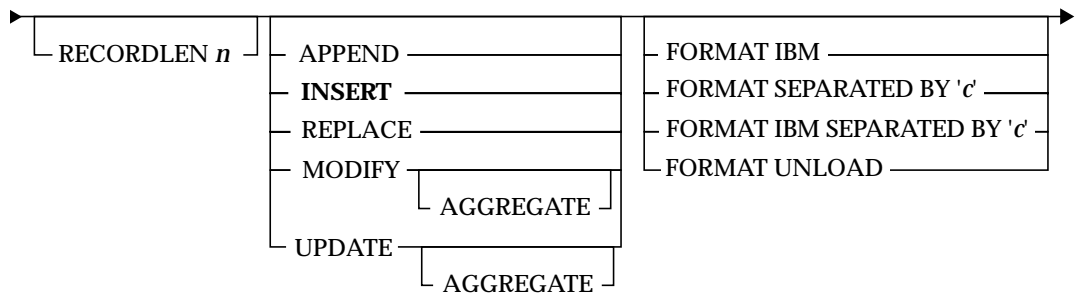
The following syntax diagrams provide the complete syntax for the TMU LOAD DATA statement.



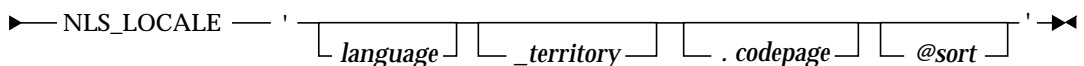
input_clause



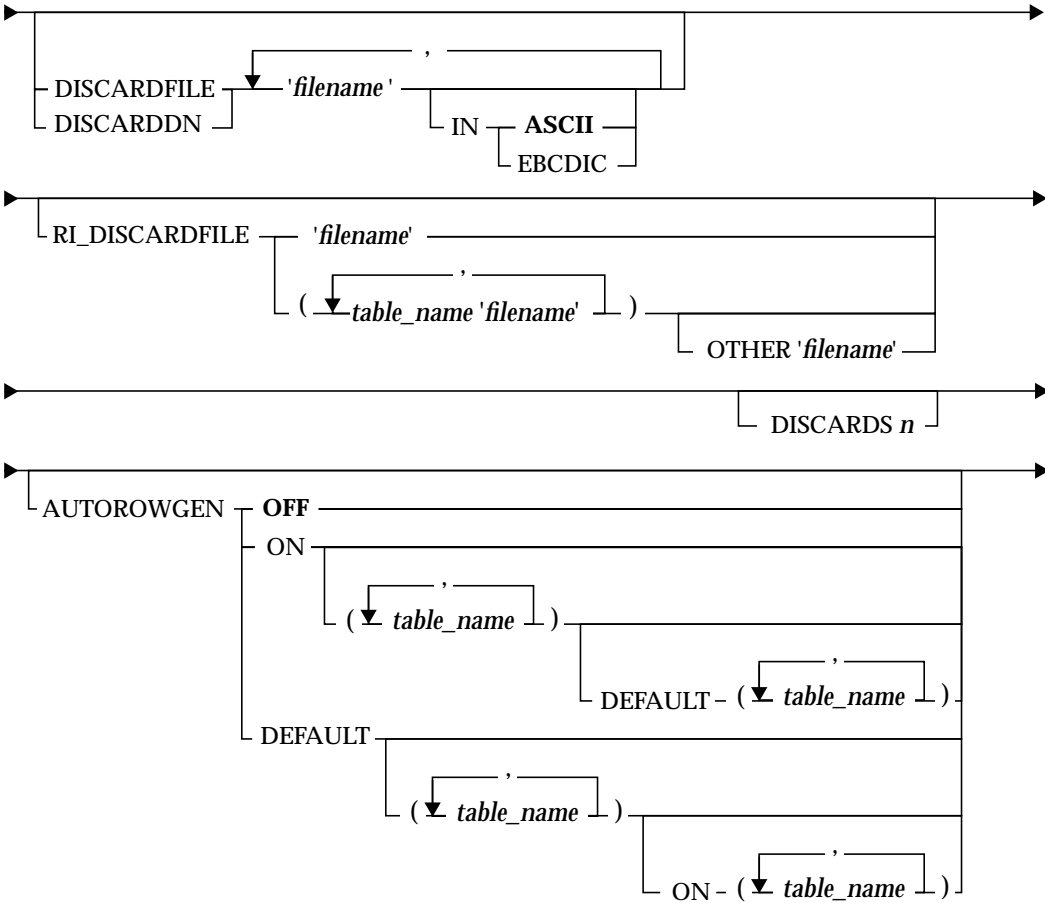
format_clause



locale_clause



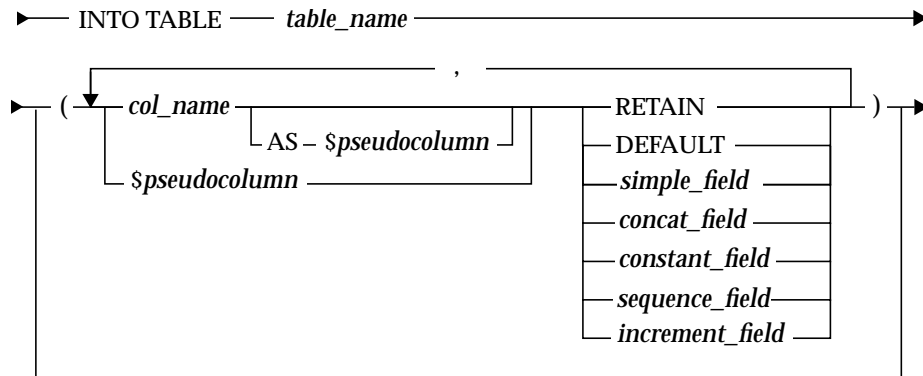
discard_clause



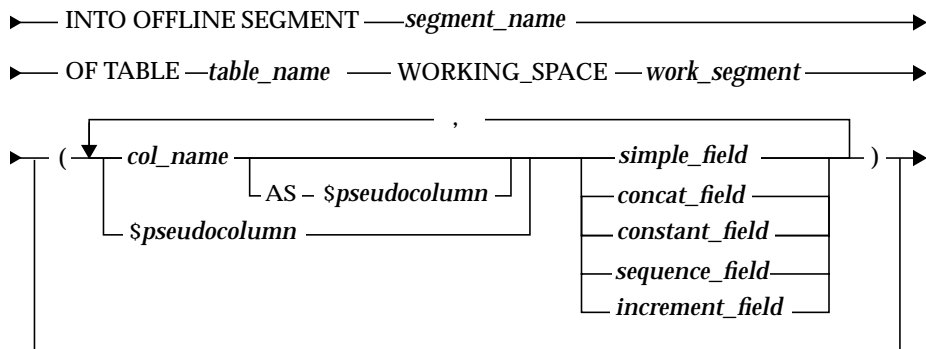
optimize_clause



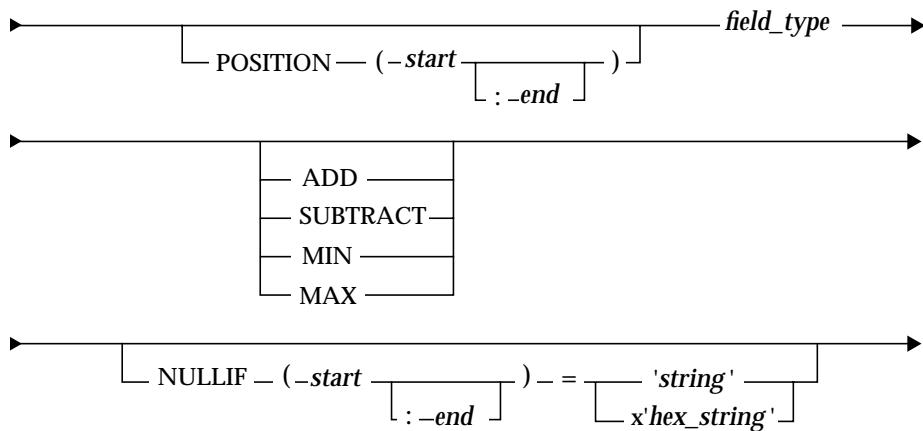
table_clause



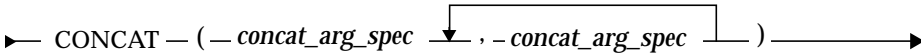
segment_clause



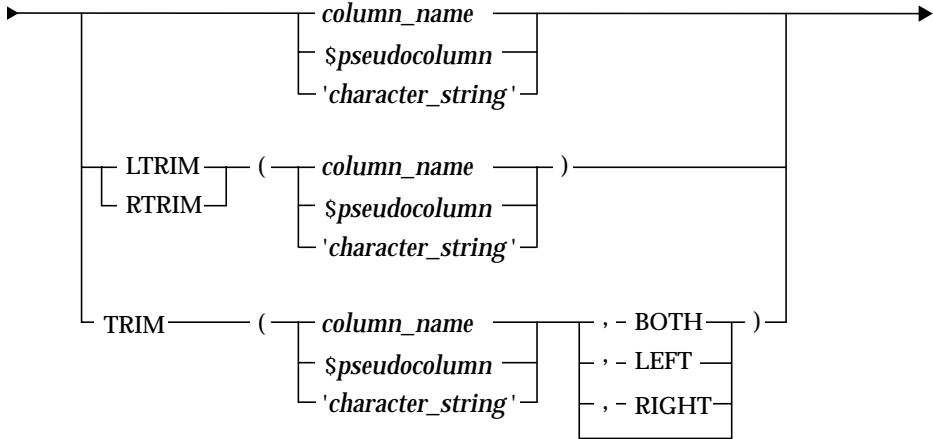
simple_field



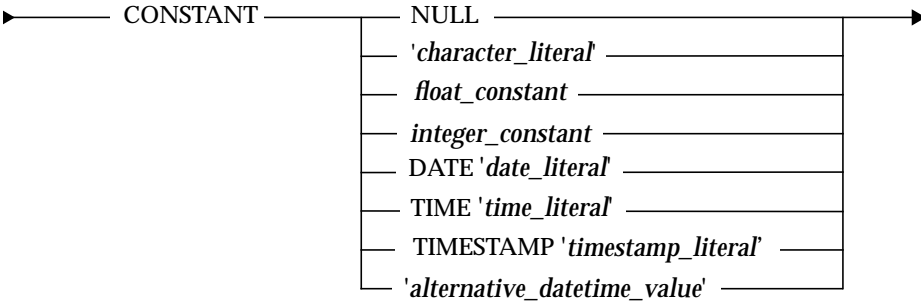
concatenated_field



where *concat_arg_spec* is:

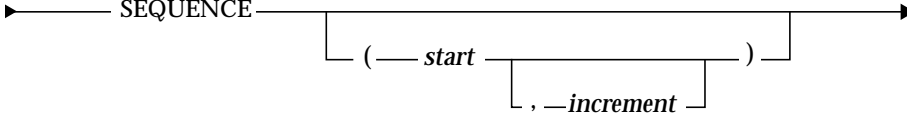


constant_field

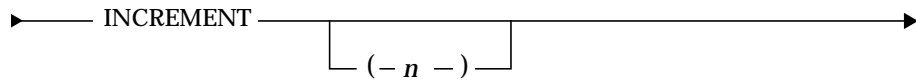


3

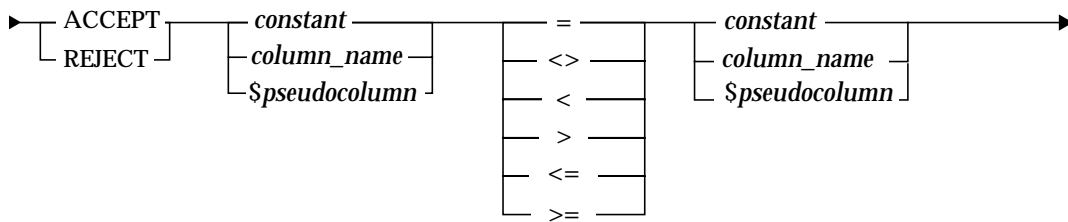
sequence_field



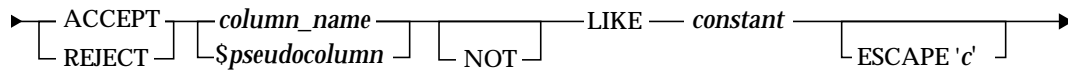
increment_field



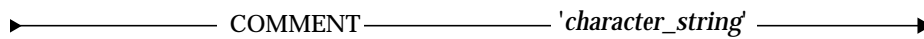
criteria_clause on non-character column

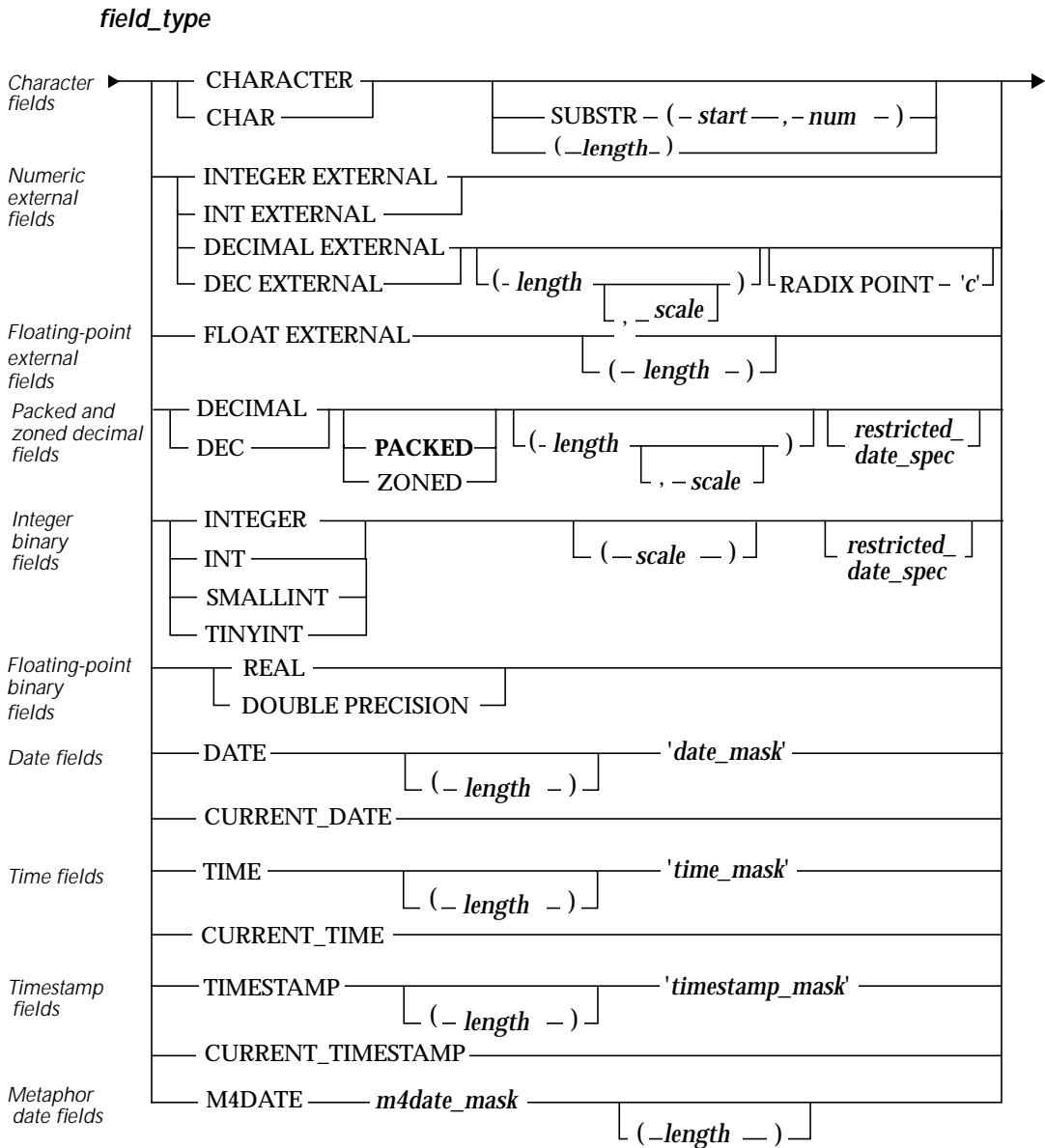


criteria_clause on character column

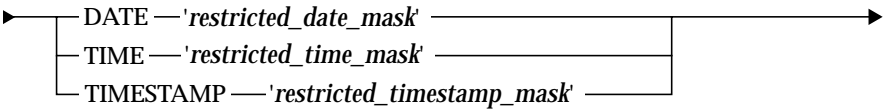


comment_clause





restricted_date_spec





Unloading Data from a Table

You can unload data from a warehouse table to a file, a magnetic tape, or standard output with the TMU UNLOAD statement. This data can later be reloaded by the TMU or used by some other application. You might also find a TMU selective UNLOAD operation to be faster than an SQL query in some cases.

This chapter contains the following sections:

- The UNLOAD Operation
- UNLOAD Syntax
- Unloading or Loading Internal-Format Data
- Unloading or Loading External-Format Data
- Converting a Table to Multiple Segments
- Moving a Database
- Loading External-Format Data into Third-Party Tools
- Unloading Selected Rows
- Example: UNLOAD Statements and External Format Data

The UNLOAD Operation

The TMU UNLOAD operation is a flexible one that you can use for many purposes, which include:

- Moving a table or a database from one system to another.
- Maintaining better performance on frequently updated databases by periodically (quarterly or annually) unloading, and then reloading the data. This operation reorganizes the database for more efficient data storage.
- Archiving a segment before removing its rows from a table.
- Loading data into tools provided by some other vendor.

You can unload an entire table or just the data in a specified segment. The TMU can also perform a selective unload; by specifying constraints in a WHERE clause in the UNLOAD statement, you can select the rows to be unloaded.

You can specify whether the rows of a table are unloaded in the order of the data (by doing a relation scan of the table) or in the order of one of the table's indexes.

You can also “pipe” the output data to another program for additional processing, such as compressing or filtering.

To facilitate reloading unloaded data, the TMU can automatically generate CREATE TABLE and LOAD DATA statements corresponding to the table and data being unloaded as part of the UNLOAD operation. This capability is also available with a TMU GENERATE statement as described in Chapter 5, “Generating CREATE TABLE and LOAD DATA Statements.”

The TMU unload capability is used by the *rb_cm* copy management utility. For more information about this utility, refer to “The *rb_cm* Utility” on page 8-2.

To execute an UNLOAD statement, you must be a member of the DBA system role, be the owner of the table, or have SELECT privilege for the table.

You can unload data in two formats: an internal binary format and an external character-based format. Data unloaded in the internal format can be reloaded only into a Red Brick Warehouse database on the same platform. Data unloaded in the external format can be reloaded into a Red Brick Warehouse database on the same or a different platform.

Internal Format

Unloading a table to the internal format creates a binary output file. The TMU quickly reloads internal-format files; however, the files must be reloaded only on a system on the same platform. For example, you cannot unload a table to the internal format on an HP 9000 and reload it on an IBM RISC System/6000, or vice versa.

To reload an internal-format file, use the `FORMAT UNLOAD` option in your `LOAD DATA` statement. Loading data in this format is quicker than loading from an external-format unload file or another all-character flat file.

External Format

Unloading a table to the external format creates an output file that can be reloaded on the same or a different platform. Because external-format files are character-based, you can also read and edit the files, if necessary. You can also use the files with other applications. For example, you can import the data unloaded into an external-format file into a desktop spreadsheet application.

When data is unloaded in external format, multibyte characters are preserved in data and table and column names, but data is not localized. Numeric and datetime data are formatted according to ANSI SQL-92 rules for these datatypes.

When unloading data in the external format, the TMU generates the following character-based files:

- A file containing the data. To name the file, specify the `OUTPUTFILE` keyword and filename.
- An optional file containing a generated `CREATE TABLE` statement, which can be used to create a table to hold the unloaded data. To produce this file, specify the `DDLFILE` keyword and filename. This file can also be produced separately using the `GENERATE CREATE TABLE` statement.
- An optional file containing a generated `LOAD DATA` statement, which can be used to reload the data. To produce this file, specify the `TMUFILE` keyword and filename. This file can also be produced separately using the `GENERATE LOAD DATA` statement.

To reload the external-format unloaded data, invoke the TMU using the automatically generated TMU control file (named with the `TMUFILE` keyword), which contains the `LOAD DATA` statement.

Data Conversion to External Format

The following table defines how data from a table is mapped into an external data file.

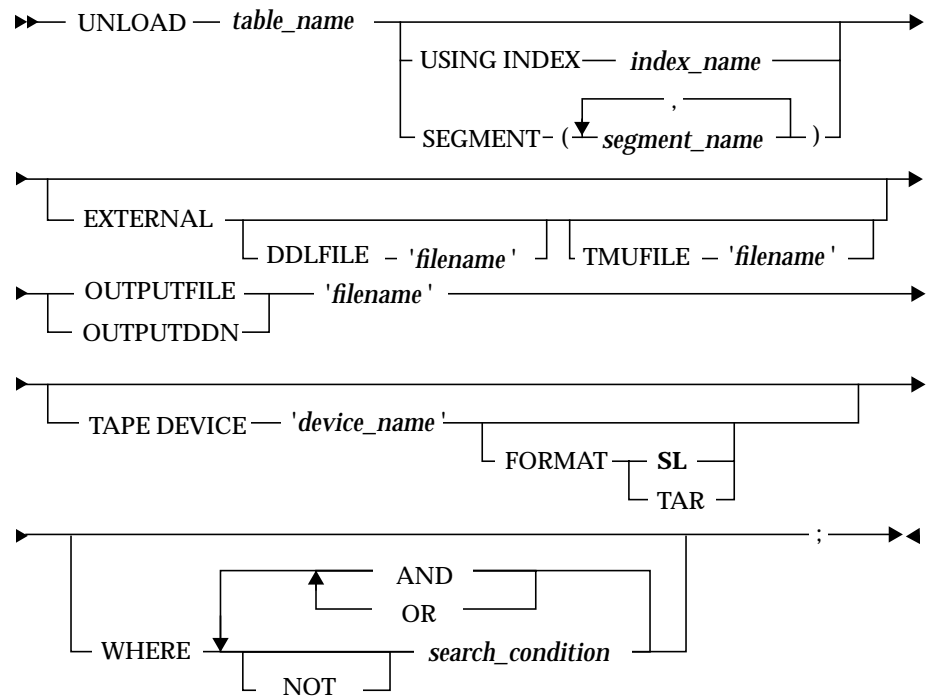
Datatypes	Number of Bytes	Format	Notes
Tinyint	4	[0 -]ddd	-128 to 127
Smallint	6	[0 -]dddddd	-32768 to 32767
Integer	11	[0 -]ddddddddddd	-2147483648 to 2147483647
Decimal	1-38	[0 -] [digits] . [digits]	Number of digits on either side of decimal point depends on precision and scale of column. It does not vary from row to row.
Float	20	[-] d.ddd...E[+ -]dd	May lose precision
Double	31	[-] d.ddd...E[+ -]dd	May lose precision
Date	10	YYYY-MM-DD	Uses ANSI SQL format
Time (0)	8	HH:II:SS	Uses ANSI SQL format
Time (non-zero)	15	HH:II:SS.FFFFFFFF	Uses ANSI SQL format
Timestamp (0)	20	YYYY-MM-DD HH:II:SS	Uses ANSI SQL format
Timestamp (non-zero)	26	YYYY-MM-DD HH:II:SS.FFFFFFFF	Uses ANSI SQL format

In the external data file, each column is preceded by a single character indicator, which indicates whether a NULL is present in that column for a given row. If the indicator is blank, the value follows. If the indicator is %, the field is NULL and filled with blanks. A newline character follows each record to make it easier for third-party tools to load the file.

For an example of an external data file, refer to “Example: UNLOAD Statements and External Format Data” on page 4-16.

UNLOAD Syntax

The syntax for the TMU UNLOAD statement is as follows:



table_name

Table to be unloaded. If any data segments are offline, all segments must be unloaded by an UNLOAD statement with a SEGMENT clause.

USING INDEX index_name

Specifies the index to use for the unload operation; the index order determines the order in which the rows of data are unloaded. If no index is specified, the data is unloaded by a table scan.

The index can be any index on the table except a TARGET index. You can determine an index name from the RBW_INDEXES system table.

If this clause is used, all segments of the index must be online.

SEGMENT *segment_name*

Allows unloading of specific segments. One or more segments can be unloaded. If data is unloaded by segments, the data in each segment is unloaded in row order by scanning the segment of the table. No index order can be specified.

The specified segment can be any segment attached to the table being unloaded. The segment can be either online or offline for the unload operation. This clause must be used for offline data segments.

EXTERNAL

Specifies that the unloaded data will be in plain text format in the database locale character set. If you do not specify EXTERNAL, the data will be unloaded in internal (binary) format. For a description of the external data format, refer to page 4-4.

DDLFILE '*filename*'

Specifies the name of the file to which the TMU automatically writes a CREATE TABLE statement for the table during an unload-to-external operation. The file does not include any segment information. You can then use this file to create a table to hold the data on any platform. If you do not specify the DDLFILE keyword and filename, the file is not created.

filename can be a relative pathname or a full pathname and can include environment variables. Enclose *filename* in single quotes.

This file must be written to disk; it cannot be written to a tape device.

TMUFILE '*filename*'

Specifies the name of the file to which the TMU automatically writes a LOAD DATA statement during an unload-to-external operation. After unloading the data, you can use this file as the control file when you invoke the TMU to reload the data.

Before reloading the data, you might need to modify this file to correctly specify the load mode, the input file name, and if loading from a tape, the tape device name. If the unloaded table data is written to a tape device, this file will contain a template TAPE DEVICE clause. If the unloaded table data is written to a disk file, no TAPE DEVICE clause will be included.

filename can be a relative pathname or a full pathname and can include environment variables. Enclose *filename* in single quotes.

This file must be written to disk; it cannot be written to a tape device.

OUTPUTFILE '*filename*'

Specifies the file to which the table data is written during an unload operation.

This file can be written to disk, tape, standard output, or piped to another program or filter. The filename can be a relative pathname or a full pathname and can include environment variables. Enclose *filename* in single quotes.

If the output is standard output, the filename reference is '-':

```
OUTPUTFILE '-'
```

Note: If the UNLOAD statement appears in a control file used by the *rb_cm* copy management utility, OUTPUTFILE must be set to standard output.

If the output from a table or segment is piped to another program, the filename reference is '| *command*' where *command* is an operating system program to which the output should be piped. For example, the following statement unloads the Sales table using external format and pipes the output to the *compress* program, which compresses and writes the data to a file named *outdata_sales*:

```
unload sales external OUTPUTFILE '|' compress > outdata_sales'
```

Because not all operating-system error cases are detected by the TMU, you should verify that the program completed successfully.

TAPE DEVICE '*device_name*'

Specifies the tape device, if the data is to be written to a tape file. Enclose *device_name* in single quotes.

If the table data is to be written to a disk file, do not use the TAPE DEVICE keyword and *device_name*.

Note: Tape support is not available on Windows NT systems.

FORMAT

Specifies the format of the tape output file: SL (ANSI Standard Label) or TAR format. SL is the default. If you unload in SL format and the output tape is not already an ANSI Standard Label tape, the TMU prompts for a volume ID (volume serial number) to be used to label the tape.

A TAR file cannot exceed 8,589,934,591 bytes, a limit imposed by the IEEE-POSIX standard. If the table to be unloaded exceeds this limit, use a Standard Label tape. You can unload multiple files (tables) to a single TAR archive by writing a single control file that contains multiple UNLOAD statements. You cannot, however, use a WHERE clause in an UNLOAD statement (a selective unload) that writes to a TAR file.

WHERE *search_condition*

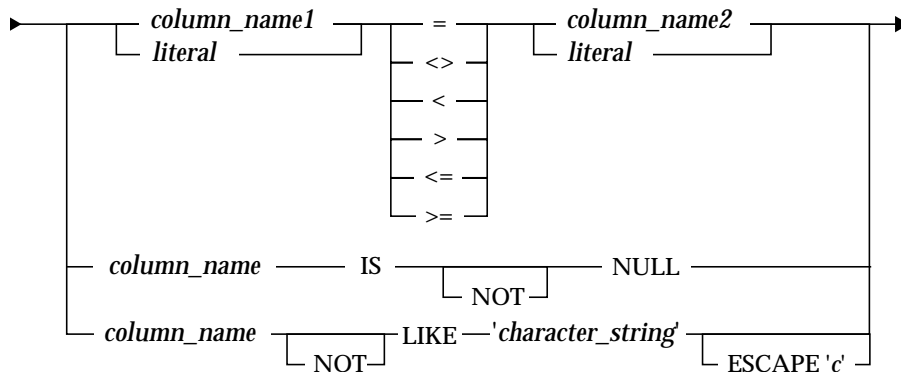
Specifies the rows to be unloaded; only those rows that satisfy the search condition are unloaded. The supported search conditions are a subset of those supported for SQL queries. Search conditions can be grouped with parentheses to force an evaluation order.

A WHERE clause can be combined with a segment list to limit the “scope” of the unload operation. If specific segments are listed, the search condition applies only to those segments. If no specific segments are listed, the search condition applies to the entire table.

Note: If you know that the search condition will choose rows from only a specific set of segments (for example, if the search condition contains a constraint on the segmenting key), you can achieve fastest performance by listing only those segments in a SEGMENT clause in the UNLOAD statement.

The constraints on character data in the WHERE clause are evaluated according to the collation sequence defined by the database locale.

The following syntax diagram shows how to construct a *search_condition* for use in a WHERE clause an UNLOAD statement.



column_name1, column_name2

Names of columns in the table to be unloaded. A column can be compared with another column, tested for NULL, or compared with a constant value.

literal

Specifies a fixed sequence of characters, a numeric constant, or a datetime constant. These character, datetime, and numeric literals must correspond to the literal language elements, as defined in the *SQL Reference Guide*. These literals must be specified in the database locale character set; datetime constants must be expressed in ANSI SQL-92 format; and decimal constants must use a decimal radix.

'character_string'

Specifies either a completely specified character string or a character pattern containing one or more wildcards; the character string must be enclosed in single quotes (' '). For a complete definition, refer to the description of the LIKE predicate in the *SQL Reference Guide*. The character string must be specified in the database locale character set.

ESCAPE 'c'

Defines a character *c* to serve as an escape character so that wildcard characters (% and _) can be used within the preceding string constant.

Caution: The escape character must be specified in the database locale character set and can be either a single-byte or multibyte character.

Unloading or Loading Internal-Format Data

To unload a table in the internal format (default):

1. Create or open a TMU control file and write an UNLOAD statement that writes the table or segment(s) to disk or tape.
2. Invoke the TMU, using the control file that you created in step 1.

The TMU creates a file that contains the unloaded data in internal format.

To reload internal-format data into a table:

1. If you are creating a new table, create it with the SQL CREATE TABLE command, either one that you write or one that the TMU generated.

Note: The CREATE TABLE statement must create either the same table as the table from which the data was unloaded or a table with the same number, type, and order of columns.

2. Prepare a control file containing a LOAD DATA statement that specifies FORMAT UNLOAD and the name of the file containing the unloaded data.

Note: A LOAD DATA statement with FORMAT UNLOAD cannot contain field specifications.

3. Invoke the TMU, specifying the control file you created in step 2.
4. Create any needed indexes, synonyms, and views. As the table is loaded, the TMU automatically builds primary key indexes and updates other existing indexes.

You can also use pipes to accomplish the unload/load process without using an intermediate tape or disk file. For more information about using pipes, refer to your operating-system documentation.

Example

This example shows how to unload a table using internal format. The Market table is unloaded into the *market.output* file on a disk. The data is then reloaded into the same table using a LOAD DATA statement that specifies FORMAT UNLOAD.

To unload the Market table in the internal format and place the contents into the file *market.output*:

1. Create or open a TMU control file and write an UNLOAD statement. In this example, the file is named *unloadmkt.tmu*.

```
unload market
outputfile 'market.txt' ;
```

2. Invoke the TMU, specifying *unloadmkt.tmu* as the control file:

```
rb_tmu unloadmkt.tmu db_username db_password
```

The contents of the Market table are now written in an internal format to the file *market.txt*.

To reload the Market table with internal-format data:

1. In a file, prepare a LOAD DATA statement that specifies FORMAT UNLOAD. In this example, the file is named *loadmkt.tmu*.

```
load data
inputfile 'market.txt'
format unload
into table market;
```

Note: The output file specified in the UNLOAD statement is used as the inputfile of the LOAD DATA statement.

2. Invoke the TMU from the command line, specifying *loadmkt.tmu* as the control file.

```
rb_tmu loadmkt.tmu db_username db_password
```

3. Create any needed indexes, synonyms, and views.

Unloading or Loading External-Format Data

To unload a table in the external format:

1. Create or open a TMU control file and write an UNLOAD statement with the EXTERNAL keyword that writes the table or segments(s) to disk or tape.

If you are going to reload the data into a new table, include the DDLFILE and TMUFILE keywords and filenames so that templates for CREATE TABLE and LOAD DATA statements are automatically generated. Alternatively, you can generate these statements at another time with the TMU GENERATE statement.

2. Invoke the TMU, using the control file containing the UNLOAD statement that you created in step 1.

The TMU creates a file that contains the unloaded data in external format. If you specified the TMUFILE and DDLFILE clauses, it also creates files containing the automatically generated LOAD DATA and CREATE TABLE statements.

Note: The CREATE TABLE statement does not include any segment information or a MAX ROWS PER SEGMENT clause; you can edit the file to include this information if necessary.

To reload external-format data into a warehouse table:

1. If you are creating a new table, create it with the SQL CREATE TABLE command, either one that you write or one that the TMU generated automatically with a TMU UNLOAD or GENERATE statement executed on the table from which the data was unloaded.

Note: The CREATE TABLE statement must create either the same table as the table from which the data was unloaded or a table with the same number, type, and order of columns.

2. Review the file containing the automatically generated TMU LOAD DATA statement to be sure the input file name is correct. If loading from tape, edit the TAPE DEVICE clause to specify the correct device name.
3. Invoke the TMU, specifying the file that contains the LOAD DATA statement as the TMU control file.
4. Create any needed indexes, synonyms, and views. As the table is loaded, the TMU automatically builds primary key indexes and updates other existing indexes.

Example

This example shows how to unload a table into a file using the external format. The Sales table is unloaded to the *sales.output* file. The data is then reloaded using the automatically generated TMU file.

To unload the Sales table in the external format and place the contents into the file *sales.output*:

1. Create or open a file and write an UNLOAD statement with the EXTERNAL keyword. In this example, the file is named *unloadsales.tmu*.

```
unload sales external
ddlfile 'sales.create'
tmufile 'sales.load'
outputfile 'sales.output';
```

2. Invoke the TMU from the command line, using the file *unloadsales.tmu* as the control file.

```
rb_tmu unloadsales.tmu db_username db_password
```

The TMU creates a file named *sales.create*, which contains a CREATE TABLE statement that you can use to re-create the table; a file named *sales.load*, which contains the LOAD DATA statements for reloading the data; and a file named *sales.output*, which contains the data in external format.

To reload the Sales table in a new database:

1. Create the table using the DDL file *sales.create*. For example, if you are using the RISQL Entry Tool, you can execute the file as follows.

```
RISQL> run sales.create ;
```

2. Modify the *sales.load* file, which contains the LOAD DATA statement, to correctly specify the input tape device.
3. Invoke the TMU from the command line, specifying *sales.load* as the control file.

```
rb_tmu sales.load db_username db_password
```

4. Create any needed indexes, synonyms, and views.

Converting a Table to Multiple Segments

If a table resides in a single segment, you can use the unload operation to split the data among additional new segments as follows:

1. Unload the table to disk or tape.
2. Create a new table with multiple segments.
3. Reload the data, using UNLOAD format.

Moving a Database

To move a database, create a file containing an UNLOAD statement for each table in the database. You can specify the internal format for fastest performance or the external format for increased flexibility. However, if you move a database to a system on a different platform, you must specify the external format.

If you unload the data in the external format, either include the TMUFILE parameter so that the TMU generates a file containing LOAD DATA statements needed to reload the data or use the GENERATE LOAD DATA command to create the appropriate TMU file.

If you unload the data in the internal format, you must create or generate a control file containing the LOAD DATA statements. You can use the GENERATE LOAD DATA command to create the appropriate TMU file.

Be sure to write the LOAD DATA statements in the order the tables must be loaded. For information about determining table order, refer to “Determining Table Order” on page 3-9.

For more information about moving a database, refer to the *Warehouse Administrator's Guide*.

Loading External-Format Data into Third-Party Tools

After unloading a table in the external format, you can use the information in the automatically generated TMU file to load the data into products that accept portions of fixed input. The TMU file contains the LOAD DATA statements for reloading and provides information about the positions of the columns.

For example, you can load data into IBM DB2 using the DB2 LOAD utility or into Microsoft Excel using the Excel parse function.

When loading data into Excel, note that Excel does not accept ANSI SQL date formats. You can use the parse function to get date components and then use a date function to turn the components into an Excel date. You must also use the Excel parse function to interpret null-indicator characters to extract columns with null values. Because the data is in the external format, you can look at the data so that you can set up Excel to handle its format.

Unloading Selected Rows

To unload only selected rows of a table, create an UNLOAD statement that contains a WHERE clause specifying which rows to unload. Only those rows that satisfy the column constraints in the WHERE clause are written to the unload file.

A WHERE clause can be combined with a segment list to limit the “scope” of the unload operation. If specific segments are listed, the search condition applies only to those segments. If no specific segments are listed, the search condition applies to the entire table. If you know that the WHERE clause will only unload rows from specific segments, the unload operation is faster if you list just those segments in a SEGMENT clause of the UNLOAD statement.

To write the rows selected by the WHERE clause to a TAR file, you must first insert them into a temporary table or unload them to a disk file: You cannot use a WHERE clause in an unload operation to a TAR file (because each header block in the TAR file must know the length of the file that follows it, which is not known in the case of a selective unload operation.)

Example

Assume you want to unload the 1996 sales data from the Sales table in the Aroma database. The following UNLOAD statement unloads the rows for 1996 from the Sales table, based on the Perkey column. The rows are written in external format to a file named *1996_sales_data*.

```
unload sales
external outputfile '1996_sales_data'
where perkey >= 96001 and perkey <= 96053;
```

Unloading Data from a Table

Example: UNLOAD Statements and External Format Data

Example

Assume you want to unload the 1996 sales data for the northern region from the Sales table. The following UNLOAD statement unloads the rows for 1996 for the northern region from the Sales table, based on the Perkey and Mktkey columns. The rows are written in internal format to a file named *1996_northern_sales_data*.

```
unload sales
  outputfile '1996_northern_sales_data'
  where perkey >= 96001 and perkey <= 96053
    and ( mktkey = 6
      or mktkey = 7
      or mktkey = 8 );
```

Example: UNLOAD Statements and External Format Data

This example illustrates the external format generated by the TMU, as well as the automatically generated CREATE TABLE and LOAD DATA statements for the Market table in the Aroma database.

The original Market table is created by the following statement:

```
create table market (
  mktkey integer not null,
  hq_city char(20),
  hq_state char(20),
  district char(20),
  region char(20),
  constraint mkt_pkc primary key (mktkey));
```

This TMU UNLOAD statement

```
unload market
  external
  ddlfile 'market_create.risql'
  tmufile 'market_load.tmu'
  outputfile 'market.txt';
```

produces a file named *market.txt* that contains unloaded data from the Market table in the external format that looks like this:

```
000000000001 Atlanta      GA      Atlanta      South
000000000002 Miami       FL      Atlanta      South
000000000003 New Orleans  LA      New Orleans   South
000000000004 Houston     TX      New Orleans   South
000000000005 New York    NY      New York      North
...
```

It also produces a file named *market_create.risql* that contains this SQL statement to recreate the Market table:

```
CREATE TABLE MARKET (  
    MKTKEY INTEGER NOT NULL UNIQUE,  
    HQ_CITY CHARACTER(20),  
    HQ_STATE CHARACTER(20),  
    DISTRICT CHARACTER(20),  
    REGION CHARACTER(20),  
    PRIMARY KEY(MKTKEY));
```

It also creates a file named *market_load.tmu* that contains the following LOAD DATA statement:

```
LOAD DATA INPUTFILE 'market.txt'  
RECORDLEN 97  
INSERT  
INTO TABLE MARKET (  
    MKTKEY POSITION(2) INTEGER EXTERNAL(11) NULLIF(1)='% ',  
    HQ_CITY POSITION(14) CHARACTER(20) NULLIF(13)='% ',  
    HQ_STATE POSITION(35) CHARACTER(20) NULLIF(34)='% ',  
    DISTRICT POSITION(56) CHARACTER(20) NULLIF(55)='% ',  
    REGION POSITION(77) CHARACTER(20) NULLIF(76)='% ');
```

Note that the table in the LOAD DATA statement and the CREATE TABLE statement is MARKET; if you are going to use these statements to create a new table, you might want to edit them to change the name. Similarly, input filenames or tape device names often must be edited to make them correspond to the actual physical locations.

Note also the use of the NULLIF keyword and the % character in the LOAD DATA statement to indicate whether a column in the unloaded table contained NULL. For example, if the value in the District column for New Orleans were NULL, the unloaded data would look like this:

```
000000000001 Atlanta      GA      Atlanta      South  
000000000002 Miami       FL      Atlanta      South  
000000000003 New Orleans  LA      %            South  
000000000004 Houston     TX      New Orleans  South  
000000000005 New York    NY      New York     North  
...
```

Unloading Data from a Table

Example: UNLOAD Statements and External Format Data



Generating CREATE TABLE and LOAD DATA Statements

The TMU can automatically generate CREATE TABLE and LOAD DATA statements based on existing tables; these statements can be used as templates for creating and loading new tables. These statements can also be generated as part of the UNLOAD process; however, the GENERATE statements provide more flexibility.

This chapter contains the following sections:

- Generating CREATE TABLE Statements
- Generating LOAD DATA Statements
- Example: GENERATE Statements and External-Format Data

Generating CREATE TABLE Statements

If you need to write a CREATE TABLE statement for a new table to hold unloaded data or you want to create a template for a new table that is similar to an existing table, you can use the TMU GENERATE CREATE TABLE command to generate one instead of generating it as part of the UNLOAD operation. You can either use the statement as generated or edit it to make any necessary changes (for example, modifying filenames or table names or adding segment information or MAXROWS or MAXROWS PER SEGMENT values).

To execute a GENERATE statement, you must have SELECT privileges on the table.

Syntax

The syntax for the TMU GENERATE CREATE TABLE statement is as follows:

```
►— GENERATE CREATE TABLE FROM — table_name —————►  
►— DDLFILE ——— 'filename' ————— ; —————►
```

table_name

Name of an existing table for which a CREATE TABLE statement is to be generated.

DDLFILE '*filename*'

Specifies the name of the file to which the TMU writes the generated CREATE TABLE statement. The file does not include any segment information. You can then use this file to create a table on any platform.

filename can be a relative pathname or a full pathname and can include environment variables. Enclose *filename* in single quotes.

filename can also begin with a single vertical bar (“|”) character, followed by a command string. This special format causes the TMU to direct the generated output data to a system pipe rather than to a file. The generated CREATE TABLE statement serves as input to the command string, which is run as a shell command.

Example

This example illustrates how to generate a CREATE TABLE statement for the existing table named `Product` and write the generated statement to the disk file named `create_product.risql` in the current directory.

```
generate create table from product
ddlfile 'create_product.risql';
```

Example—UNIX

This example illustrates how you can use system pipes and a remote shell command (*rsh*) to create the table on a remote host. A CREATE TABLE statement is generated for the existing table named `Sales`. Instead of writing the generated statement to a disk file, however, the generated statement is passed to a system pipe and executed on a remote shell (*rsh*) command on a UNIX host named *north1*.

The remote shell executes the UNIX command *cat* to copy the remote shell input to a file named *sales.create*. The *cat* command is enclosed in quotes since it contains the special character “>” to redirect output. In this single operation, a CREATE TABLE statement is automatically generated and copied to a disk file on a remote host.

```
generate create table from sales
ddlfile '| rsh north1 "cat > sales.create"';
```

Example—UNIX

As in the previous example, generate a CREATE TABLE statement for the table named `Sales` and pass the output to a remote shell on a UNIX host named *north1*. As the UNIX remote shell command (*rsh*), pass the generated CREATE TABLE statement directly to the RISQL Entry Tool running on the *north1* host to create the table in an existing warehouse database. In this single operation, a replica of the `Sales` table is created on the remote host.

```
generate create table from sales
ddlfile '| rsh north1 risql user password';
```

Note: The combination of the GENERATE CREATE TABLE statement and the remote shell capability illustrated in this example is particularly useful with the *rb_cm* copy management utility. A similar GENERATE statement can be included in the *rb_cm* unload control file before the UNLOAD statement, causing the remote table to be created immediately before data is copied to that table. For more information about this utility, refer to Chapter 8, “Moving Data with the Copy Management Option.”

Generating LOAD DATA Statements

If you need to write a LOAD DATA statement to load unloaded data or you want to create a template to load similar data into a new table, you can use the TMU GENERATE LOAD DATA command to generate one instead of generating it as part of the UNLOAD operation. The GENERATE LOAD DATA statement allows you to specify a name for the target table and a name for the input file; you can either use the statement as generated or edit it to make any necessary changes.

To execute a GENERATE statement, you must have SELECT privileges on the table.

Syntax

The syntax for the TMU GENERATE LOAD DATA statement is as follows:

```

▶▶ GENERATE LOAD [ DATA ] FROM — table_name —————▶
▶ [ INTO — new_table_name ] [ INPUTFILE — 'new_filename' ] —▶
▶ [ TAPE DEVICE — 'device_name' ] [ EXTERNAL ] —————▶
▶ TMUFILE ——— 'filename' —————▶ ; ▶▶
  
```

table_name

Name of an existing table for which a LOAD DATA statement is to be generated.

INTO *new_table_name*

Specifies an alternative table name to be used in the INTO clause of the generated LOAD DATA statement. If not specified, *table_name* is used in the INTO clause. The table named *new_table_name* does not have to exist at the time the LOAD DATA statement is generated, but it must exist when the generated LOAD DATA statement is used.

INPUTFILE '*new_filename*'

Specifies the filename to be used in the INPUTFILE clause of the generated LOAD DATA statement. If not specified, the string "???" is used in the INPUTFILE clause, requiring the user to edit the generated file later to specify a valid input filename.

TAPE DEVICE '*device_name*'

Specifies that a TAPE DEVICE clause be included in the generated LOAD DATA statement and the device name to be used in that clause. Use this clause to generate a LOAD DATA statement to load data from a tape drive. If not specified, no TAPE DEVICE clause is included in the generated statement.

Note: Tape support is not available on Windows NT systems.

EXTERNAL

Specifies that the generated LOAD DATA statement be written with the field specifications necessary to reload the table from an external unload-format file. If EXTERNAL is not specified, the generated LOAD DATA statement will be written to conduct an internal-format load—that is, it will specify FORMAT UNLOAD and will not include field specifications.

TMUFILE '*filename*'

Specifies the name of the file to which the TMU writes the generated LOAD DATA statement for the table. This file can then be used unchanged as input to the TMU to load or reload the table from an internal or external unload-format file. The generated file can also be used as a template and edited so that the generated field specifications match some other input format.

filename can be a relative pathname or a full pathname and can include environment variables. Enclose *filename* in single quotes.

filename can also begin with a single vertical bar (“|”) character, followed by a command string. This special format causes the TMU to direct the generated output data to a system pipe rather than to a file. The generated LOAD DATA statement serves as input to the command string run as a shell command.

Example

This example generates a LOAD DATA statement based on an existing table named Product for a new table named Newproduct (the INTO clause). The generated LOAD DATA statement will read input data from a file named *product_unload* (the INPUTFILE clause), which is in EXTERNAL format. The generated statement is written to a file named *load_newproduct* in the current directory (the TMUFILE clause).

```
generate load data from product
into newproduct
inputfile 'product_unload'
external
tmufile 'load_newproduct';
```

Example

This example generates a LOAD DATA statement based on an existing table named Market; the generated statement also loads data into a table named Market (by the absence of the INTO clause). The data for the new table comes from the system standard input (a filename in the INPUTFILE clause of '-') as an internal-format unload file. The generated statement is written to the file named *copy_market* in the current directory.

```
generate load data from market
  inputfile '-'
  tmufile 'copy_market';
```

Example: GENERATE Statements and External-Format Data

This example illustrates the CREATE TABLE and LOAD DATA statements for the Store table in the Aroma database that are generated by the GENERATE statement, and the external-format data produced by an UNLOAD statement.

The original Store table is created by the following statement:

```
create table store (  
    storekey integer not null,  
    mktkey integer not null,  
    store_type char(10),  
    store_name char(30),  
    street char(30),  
    city char(20),  
    state char(5),  
    zip char(10),  
    constraint store_pkc primary key (storekey),  
    constraint store_fk1 foreign key (mktkey)  
        references market (mktkey))  
maxrows per segment 2500;
```

The following TMU GENERATE statement

```
generate create table from store ddlfile 'recreate_store':
```

produces a file named *recreate_store* that contains this SQL statement to create the Store table:

```
CREATE TABLE STORE (  
    STOREKEY INTEGER NOT NULL UNIQUE,  
    MKTKEY INTEGER NOT NULL,  
    STORE_TYPE CHARACTER(10),  
    STORE_NAME CHARACTER(30),  
    STREET CHARACTER(30),  
    CITY CHARACTER(20),  
    STATE CHARACTER(5),  
    ZIP CHARACTER(10),  
    PRIMARY KEY(STOREKEY),  
    CONSTRAINT STORE_FK1 FOREIGN KEY(MKTKEY)  
        REFERENCES MARKET (MKTKEY) ON DELETE NO ACTION);
```

Note that the GENERATE statement does not produce segment information or a MAXSEGMENTS or MAXROWS PER SEGMENT clause for the CREATE TABLE statement; you can edit the file to provide the necessary information.

The following TMU GENERATE statement

```
generate load data from store
into new_store
inputfile 'store_data'
external tmufile 'store_load';
```

creates a file named *store_load* that contains the following LOAD DATA statement:

```
LOAD DATA INPUTFILE 'store_data'
RECORDLEN 136
INSERT
NLS_LOCALE 'English_UnitedStates.US-ASCII@Binary'
INTO TABLE NEW_STORE (
  STOREKEY POSITION(2) INTEGER EXTERNAL(11) NULLIF(1)='% ',
  MKTKEY POSITION(14) INTEGER EXTERNAL(11) NULLIF(13)='% ',
  STORE_TYPE POSITION(26) CHARACTER(10) NULLIF(25)='% ',
  STORE_NAME POSITION(37) CHARACTER(30) NULLIF(36)='% ',
  STREET POSITION(68) CHARACTER(30) NULLIF(67)='% ',
  CITY POSITION(99) CHARACTER(20) NULLIF(98)='% ',
  STATE POSITION(120) CHARACTER(5) NULLIF(119)='% ',
  ZIP POSITION(126) CHARACTER(10) NULLIF(125)='% ');
```

Note that you can specify a new target table name and an input filename in the GENERATE LOAD DATA statement. Note also the use of the % character in the LOAD DATA statement to indicate whether a column in the unloaded table contained NULL.

If you were to unload the data in external format from the Store table, it would look like this:

```
00000000001 00000000014 Small      Roasters, Los Gatos
1234 University Ave          Los Gatos          CA    95032

00000000002 00000000014 Large      San Jose Roasting Company
5678 Bascom Ave              San Jose          CA    95156

00000000003 00000000014 Medium      Cupertino Coffee Sup-
ply          987 DeAnza Blvd          Cupertino          CA
97865

00000000004 00000000003 Medium      Moulin Rouge Roast-
ing          898 Main Street          New Orleans        LA
70125

00000000005 00000000010 Small      Moon Pennies
98675 University Ave          Detroit          MI    48209

...
```

Reorganizing Tables and Indexes

The TMU REORG operation is used to maintain referential integrity and to improve internal storage of a table and its indexes.

This chapter contains the following sections:

- The REORG Operation
- REORG Syntax
- Usage Notes

The REORG Operation

The TMU REORG statement performs two functions:

- It checks referential integrity, if applicable for the target table, and deletes any rows that violate it. (Referential integrity is the relational property that each foreign key value in a table exists as a primary key value in the referenced table.)
- It performs an internal reorganization of one or more of the table's indexes (all types) to improve the internal storage of this information and thereby the performance when the index is used to access data. It can rebuild all indexes or selectively rebuild one or more named indexes.

If no changes are made to the database except by complete loads of data (and you do not use a RESTORE...FORCE operation to restore individual segments), then you do not need to perform a REORG operation.

If a table changes over time, you might need to reorganize it and any other tables and indexes affected by changes to that table. You might also need to use a REORG operation to rebuild the affected indexes if you use a database restore operation to restore individual segments of a table or index.

Whenever modifications to a database affect more than about 30% of the data, you should run the TMU with a REORG statement for any tables directly modified and for any tables whose foreign keys reference modified tables. Periodically rebuilding such tables and indexes with a REORG statement ensures referential integrity and optimal performance.

Certain operations can invalidate STAR indexes. For example, increasing the MAXROWS PER SEGMENT or the MAXSEGMENTS parameter on a table can invalidate STAR indexes on tables that reference the altered table. These operations always generate a warning message that says STAR indexes based on the altered table might be invalid, in which case the affected STAR index(es) needs to be reorganized. You can either reorganize affected indexes when the message is issued or schedule the REORG operation for a more convenient time. However, any non-query (INSERT, UPDATE, or DELETE) operation against a table that has an invalid index results in an error message that says the index must be reorganized. You must perform a REORG operation before the table can be accessed.

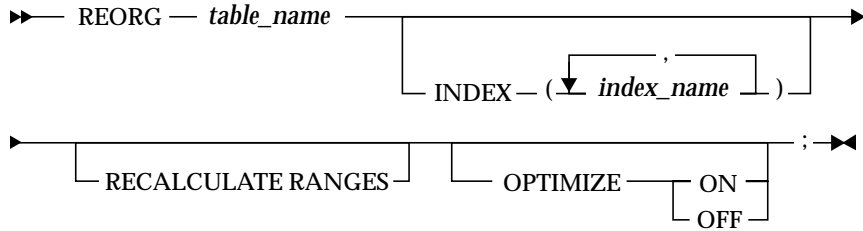
Note: You can also rebuild an index with the DROP INDEX and CREATE INDEX commands instead of using a REORG statement. If the parallel index creation option is used with the CREATE INDEX command, this approach is often faster, particularly on multi-processor systems. However, if the table contains duplicate rows, the CREATE INDEX statement will fail, whereas the REORG operation (with OPTIMIZE off) will remove duplicate rows and successfully build an index.

The REORG statement locks the table for exclusive use.

To use the REORG statement, you must be a member of the DBA system role or be the owner of the table.

REORG Syntax

The syntax for the TMU REORG statement is as follows:



table_name

Specifies the table to be reorganized.

INDEX *index_name*

Indicates that specific indexes are to be rebuilt. If this clause is not present, all indexes defined on the table are rebuilt. Index names for user-created indexes are specified in the CREATE INDEX statements.

The name for a system-generated primary key index is the string “_PK_IDX” appended to the table name. For example, the primary index for the Market table is MARKET_PK_IDX.

RECALCULATE RANGES

Specifies that the ranges for any STAR index rebuilt with this REORG statement are to be recalculated to split index entries evenly among the segments for the index. Use this option when you have changed a MAXROWS PER SEGMENT or a MAXSEGMENTS value for a table that participates in the STAR index.

If an index list is provided in a REORG command that includes this option, at least one index named must be a STAR index.

For information about segment ranges, refer to the *SQL Reference Guide*.

OPTIMIZE ON, OFF

Specifies the index or indexes are to be rebuilt in optimize mode. This option overrides the optimize mode set in the *rbw.config* file. If this clause is not present in the REORG statement, the default behavior is determined by the *rbw.config* file.

For a REORG operation on a unique index that might contain duplicate key values, use non-optimize mode (OPTIMIZE OFF). The operation will fail in optimize mode (OPTIMIZE ON) in such a case.

Usage Notes

The REORG operation does not save discarded rows; you cannot specify that discarded rows be saved in a file.

Referential integrity is preserved for databases, unless you use the OVERRIDE REFCHK option in a DELETE statement or use the REPLACE mode in a LOAD DATA statement on a table when rows in that table are referenced by another table. In either of these cases, rows can be deleted from a referenced table in violation of referential integrity. To restore referential integrity, you must perform a REORG operation on any tables that reference the table from which rows were deleted. The REORG operation deletes any rows that reference deleted row, thus restoring referential integrity.

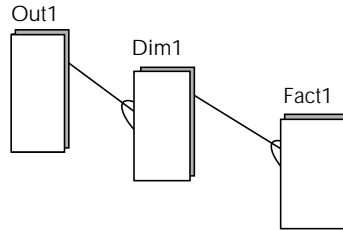
Note: Be sure that you perform the REORG operation on the *referencing* table, not the table from which the rows were deleted (the *referenced* table).

Delete operations performed by a REORG operation do not cascade to referencing tables, so if you reorganize a referenced table, you must reorganize each table that references the reorganized table until there are no more tables that reference a reorganized table.

After a RESTORE...FORCE operation, a REORG operation might fail with an error message regarding data integrity errors. Do not use the table until these errors are resolved. Use *tblchk*, the warehouse table verification utility to resolve these errors when directed to do so by an error message or by Customer Support Services. This utility is located in *redbrick_dir/util/service* and is described by a README file in that directory.

Example

This example illustrates how rows are deleted from referencing tables in a REORG operation. Assume the Fact1 table references the Dim1 table, which in turn references the Out1 table:



After some rows are deleted from Out1, a REORG operation is performed on Dim1. Any rows in Dim1 that reference rows that were deleted from Out1 (that, violate referential integrity) are deleted by the REORG operation to preserve referential integrity. However, rows in Fact1 that reference deleted rows in Dim1 are not deleted by the REORG operation on Dim1. To delete these rows and preserve referential integrity, you must also perform a REORG operation on Fact1.



Performing Backup and Restore Operations

This chapter describes a general policy for warehouse backup and restore operations and details of the discontinued TMU-based incremental backup and restore option in the following sections:

- Backup/Restore Policy
- TMU BACKUP and RESTORE Operations
- BACKUP Syntax
- RESTORE Syntax
- Database Backup and Restore Procedures
- Segment Restore Procedures

Backup/Restore Policy

Every warehouse administrator should have a recovery plan in case a database is damaged and becomes unusable. If your database is modified by incremental loads or insert, update, or delete operations, or if you are unable to retain all of the input files used to create the database, then you should back it up periodically in case of system or software failure. In determining how often to back up a database, you must balance the frequency of backups with amount of unbacked-up data at risk and the time required for a backup.

You have several choices in implementing a backup and restore policy:

- The recommended solution is SQL-BackTrack for Red Brick Warehouse, a database backup and restore system designed specifically for Red Brick Warehouse, for those operating system/platforms for which it is available. For more information about this option, refer to the *SQL-BackTrack for Red Brick Warehouse User's Guide*.
- If SQL-BackTrack is not available for your site, you can use the file-oriented backup and restore facilities provided for that operating system; for example, the UNIX-based *dump* and *restore* commands or similar utility programs available for the Windows NT operating system.

With the introduction of SQL-BackTrack for Red Brick Warehouse option, the TMU incremental backup and restore option can no longer be purchased but is supported only for a limited time to customers who purchased it prior to the release of Version 5.1. If your site has purchased this option, separate instructions that explain how to enable it are provided.

The remaining sections in this chapter describe the TMU BACKUP and RESTORE statements and their use.

TMU BACKUP and RESTORE Operations

The TMU-based incremental backup and restore feature is similar to the UNIX *dump* and *restore* commands, but it has the ability to perform either full backup and restore operations or incremental operations that take advantage of the internal structure of a warehouse database. An incremental backup saves only those blocks in the database that have changed since the last backup. Data is restored from the last full backup, plus any incremental backups done after the full backup.

You should also consider the fact that while it requires more time to do a full backup than an incremental backup, it requires less time to restore from a current full backup than from an older full backup and the incremental backups.

When you back up a database, you always back up across an entire database, backing up everything or just the changed portions of the database. However, you can restore either an entire database (to the specified level) or, in some cases, only selected segments of the database.

Backup levels and timestamps for backup and restore operations are maintained in the RBW_SEGMENTS table.

Backup Level

Each block of data stored on the disk has a backup level stored in it so that a level *n* backup will back up all blocks with a backup level greater than or equal to *n*. Up to 10 backup levels (0-9) are supported. As disk data blocks are created or modified by data manipulation, the blocks are flagged to indicate that they need to be backed up by the next backup, whatever its level. When a backup occurs, any data blocks at or above the specified backup level are backed up and marked with that backup level. If the next backup uses a higher level, only those blocks changed since the last backup will be backed up.

With segmented storage, only segments that have been modified are scanned for changed blocks. This fact provides significant performance advantages when changes are confined to specific segments rather than distributed over all segments in a large database.

A backup level value for each segment is stored in the BACKUPLEVEL column of the RBW_SEGMENTS system table; this number indicates that all blocks within the segment have been backed up to this level. A backup level of NULL indicates the segment has never been backed up. (Unattached segments are never backed up.) The same information is available for PSUs within a segment in the BACKUPLEVEL column of the RBW_STORAGE system table.

Note: The row for the system table segment, named RBW_SYSTEM, contains the value 15, which indicates that the segment is backed up for every backup operation.

Examples

To determine what segments were backed up by a specific backup operation, enter the following select statement:

```
select * from rbw_segments
where backuplevel = n;
```

where *n* is the level of the backup operation in question.

To determine which segments need to be backed up, enter the following select statement:

```
select * from rbw_segments
where backuplevel > n;
```

where *n* is the level of the last backup operation.

Backup Locking Operations

A TMU backup operation locks the database against any write operations. Read-only operations continue while the backup is processing, but write operations are locked out until the backup is complete. You can set the WAIT option to cause the backup operation to wait until current locks on the tables are released.

Database Locale

When a database is backed up (fully or incrementally), the database locale is stored with the data. When the database is restored, the locale of the backup must match the locale of the database; otherwise the restore operation fails and an error message is displayed.

Backup and restore operations display messages that contain timestamp values; these values are localized according to the locale of the operating system.

BACKUP Syntax

The syntax for the TMU BACKUP statement is:

```
➤➤ — BACKUP TO —                     'filename'                     —                     TAPE DEVICE — 'device_name'                                         LEVEL n                     ; ➤➤
```

7

filename

Specifies the filename for the backed-up data when backup is to a disk file.

TAPE DEVICE 'device_name'

Specifies the tape device to be used when backup is to a tape.

Note: Tape support is not available on Windows NT systems.

LEVEL n

Specifies the backup level. Level 0, the default, is a full backup; any other number is an incremental backup.

Usage Notes

Backup operations do not back up indexes under construction and issue a warning message to that effect.

Backup operations cannot complete during INSERT, UPDATE, DELETE, or ALTER TABLE operations or while the database is locked. Use the TMU SET LOCK WAIT/NO WAIT command to control whether the backup operation waits for locks to be released or returns without performing the operation.

Caution: After using ALTER SEGMENT...CHANGE PATH, you must perform a full (level 0) backup before you can perform an incremental backup.

Most operating systems limit disk files to a maximum size of 2 gigabytes. Therefore, you cannot use disk-based backup files to back up databases that occupy more than about 90% of this amount, but must use tape-based backups instead. You might find it practical to use tape-based full backups with disk-based incremental backups when the size of the incremental backups are less than the 2-gigabyte limit.

Examples

The following examples illustrate various database backup statements:

UNIX

```
backup to '/disk1/db_bup/072496'; # full backup on 7/24/96
```

```
backup to '/disk1/db_bup/072596' level 1;  
# incremental backup on 7/25/96
```

```
backup to tape device '/dev/rmt0' level 0;  
# Full backup on 8/16/96
```

Windows NT

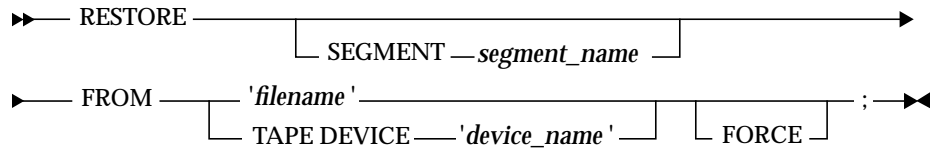
```
backup to 'e:\db_bup\072496'; # full backup on 7/24/96
```

```
backup to 'e:\db_bup\072596' level 1;  
# incremental backup on 7/25/96
```

```
backup to 'e:\db_bup\081696' level 0;  
# Full backup on 8/16/96
```

RESTORE Syntax

The syntax for the TMU RESTORE statement is:



7

You can restore either the whole database or a single segment.

SEGMENT *segment_name*

Specifies a single segment to be restored. To perform a single-segment restore, the segment must exist in the database, and the definition of the owning table must not have changed since the backup.

For more information about restoring a segment, refer to “Segment Restore Procedures” on page 7-12.

Note: The system segment, which contains the system tables, cannot be restored except as part of a full restore operation.

filename

Specifies the filename of the backed-up data when data is to be restored from a disk file.

TAPE DEVICE '*device_name*'

Specifies the tape device to be used when the restore operation is from tape.

Note: Tape support is not available on Windows NT systems.

FORCE

Forces a restore of a segment that has changed (by adding, updating, or deleting rows or changing ranges) since the backup that is being used for the restore operation. Use only when directed to do so by Customer Support Services.

Usage Notes

If the segment has changed (because rows were added, updated, or deleted from the table or because segment ranges were changed) since the backup level from which you are trying to restore, the changes will be lost when the restore operation restores the segment to its backed-up state. To prevent such an

inadvertent loss of data, the TMU issues a message stating that you cannot complete the operation without using the FORCE keyword in the RESTORE command. The FORCE keyword causes the RESTORE command to override the built-in consistency checks and might leave the database in an inconsistent state.

If the segment has not changed since the backup being used, no message is issued and you do not need to use the FORCE option.

You must perform a REORG operation to rebuild indexes after the entire restore operation is complete if you used the FORCE option during any of the incremental restore operations.

Note: If you used the FORCE option, run the *tblchk* utility to verify that the table is intact before you perform the REORG operation. If *tblchk* reports any discrepancies in the table, do not fix them until directed to do so by Customer Support Services.

Examples

The following examples illustrate various database restore statements, based on the backup statements on page 7-6:

UNIX

```
restore from '/disk1/db_bup/072496';
# Restore full backup from file 072496

restore from '/disk1/db_bup/072596';
# Restore incremental backup from file 072596

restore segment t2 from '/disk1/db_bup/072696';
# Restore segment from file 072696

restore from tape device '/dev/rmt0' ;
# Restore full backup for 8/16/96 from tape
```

Windows NT

```
restore from 'e:\db_bup\072496';
# Restore full backup from file 072496

restore from 'e:\db_bup\072596';
# Restore incremental backup from file 072596

restore segment t2 from 'e:\db_bup\072696';
# Restore segment from file 072696

restore from 'e:\db_bup\081696' ;
# Restore full backup for 8/16/96
```

The following examples illustrate the use of the FORCE option. Assume you are restoring to a level-2 backup of an individual segment and that segment changed after the level-1 backup. The TMU will issue a message before the level-0 and level-1 restores, asking you to use the FORCE option for those restore operations. Rewrite the RESTORE statements for the level-0 and level-1 restores to include the FORCE keyword.

UNIX

```
restore segment t2 from '/disk1/db_bup/072696' force;  
# Restore segment from file 072696, a full backup.  
  
restore segment t2 from '/disk1/db_bup/072796' force;  
# Restore segment from file 072796, a level-1 backup.  
  
restore segment t2 from '/disk1/db_bup/072896';  
# Restore segment from file 072896, a level-2 backup.
```

Windows NT

```
restore segment t2 from 'e:\disk1\db_bup\072696' force;  
# Restore segment from file 072696, a full backup.  
  
restore segment t2 from 'e:\disk1\db_bup\072796' force;  
# Restore segment from file 072796, a level-1 backup.  
  
restore segment t2 from 'e:\disk1\db_bup\072896';  
# Restore segment from file 072896, a level-2 backup.
```

Database Backup and Restore Procedures

The general procedure for TMU-based backup operations is:

1. Run backup at level 0 to complete a full backup and to provide the base for any restore operations.
2. Periodically run backups at level n , where n is equal to or greater than the last backup. If equal to the last backup, it replaces that backup. If greater than the last backup, only new blocks or blocks modified since the most recent backup at the next lower level are backed up.

In addition to regularly-scheduled full backups, also perform a full backup when the ALTER SEGMENT...CHANGEPATH option is used or whenever a database object is deleted.

The general procedure for restore operations is:

1. Restore from level 0.
2. Restore from level n , where n is greater than the last restore and was backed up at a later date than the previously restored backup.

The following examples illustrate two different backup strategies. In the first system, the daily backups are shorter. In the second system, each successive level 1 backup becomes longer, but the restore is simpler, requiring only one incremental restore. The second system also requires only two tapes, so if one daily backup is bad, it does not impact the subsequent backups.

Example

This example illustrates a system where a complete backup occurs once a week with daily incremental backups.

1. Run a backup at level 0 over the weekend to have a fresh full backup to start the week.
2. On Monday evening, run a backup at level 1.
3. On Tuesday evening, run a backup at level 2.
4. On Wednesday evening, run a backup at level 3.
5. On Thursday evening, run a backup at level 4.
6. On Friday evening, run a backup at level 5.

If the database needs to be restored to Tuesday's level for any reason, do the following:

1. Restore the most recent level 0 backup (this will destroy the database).
2. Restore the most recent level 1 backup (which must have been created after the level 0 backup).
3. Restore the most recent level 2 backup (which must have been created after the level 1 backup).

The database will now be back to its state as of Tuesday evening.

If the database needs to be restored to Friday's level, then you need to restore the most recent level 0, 1, 2, 3, 4, and 5 backups in that order (and with each level backup having been created after the previous level).

Example

This example illustrates a system where a complete backup occurs once a week, and daily incremental backups back up everything that has changed since the last complete (weekly) backup.

1. Run a backup at level 0 over the weekend to have a fresh full backup to start the week.
2. On Monday evening, run a backup at level 1.
3. On Tuesday evening, run a backup at level 1.
4. On Wednesday evening, run a backup at level 1.
5. On Thursday evening, run a backup at level 1.
6. On Friday evening, run a backup at level 1.

If the database needs to be restored to Tuesday's level for any reason, do the following:

1. Restore the most recent level 0 backup. (This action will destroy the database.)
2. Restore Tuesday's level 1 backup.

The database is now restored.

If the database needs to be restored to Friday's level, you need to restore the weekend's level 0 backup and Friday's level 1 backup.

Segment Restore Procedures

If you need to restore a database from TMU-based backup files, first decide whether you can use a segment restore operation. You cannot restore a single segment in the following cases:

- The segment does not exist in the database. If the segment does not exist—for example, if you inadvertently dropped a table and the associated segment was also dropped—you cannot restore the segment. Instead, you must either:
 - a. Perform a full restore, which is the recommended solution; or
 - b. Re-create the missing segment and then restore it. You must define the segment exactly as it was when it was backed up. The restore operation does not detect any differences in definition of filenames or size parameters; if differences exist, you can damage the database so that a full restore is required.
- The table description for the table containing the segment has changed. If the description has changed—for example, if columns were added or dropped from the table—do not use the segment restore procedure. Instead, you must perform a full restore.
- The number of rows in the segment has changed. If you have inserted or deleted rows, the number of rows has probably changed; do not use the segment restore procedure.

If none of the above conditions applies in your case so that a segment restore is possible, determine whether an up-to-date database backup for that segment exists:

- If you have an up-to-date backup, you can restore the segment with no loss of data.
- If you do not have an up-to-date backup, you can restore the segment so that the database is left in a consistent, usable state, even though the unbacked-up changes are lost.

Caution: When restoring a segment, the TMU cannot detect and prevent restoring from an old backup that is the right level. For example, if you do a full backup on Sunday nights and incremental backups on all other nights (level 1-6), the TMU cannot detect or prevent a restore operation on Wednesday that uses the level-2 (Tuesday night) restore from a previous week. Such an operation would result in database corruption. Use care in labeling and storing backup media to prevent such an occurrence.

If you want to provide users with partial access to a table or index with a damaged segment while you are restoring the damaged segment, use the ALTER SEGMENT command to take the damaged segment offline before you perform the restore operation. You can set the PARTIAL_AVAILABILITY options for tables and indexes for all users in the *rbw.config* file to provide access to the partially available table or index. Alternatively, an individual user can select partial access by setting the PARTIAL_AVAILABILITY option from the command line. After you have restored the segment, bring it back online before performing any needed REORG or DROP/CREATE INDEX operations.

Use the following table to determine how to restore a damaged segment.

Contents of Damaged Segment		Procedure with No Up-to-Date Backup	Procedure with Up-to-Date Backup
User table	Data	1. RESTORE SEGMENT (with FORCE [†]) to restore the segment. 2. REORG table (all indexes) to refresh or rebuild index entries for any lost data. ^{††} 3. REORG any referencing tables to maintain referential integrity. Repeat for each level of referencing tables, working from outboard tables up to and including the last referencing table. ^{††}	RESTORE SEGMENT. No other action needed.
	Indexes	DROP and CREATE INDEX or REORG ^{†††} affected index.	
System table		Contact Red Brick Customer Support Center.	

[†]If you use the FORCE option, run *tblchk*, the warehouse table verification utility before you perform a REORG operation. This utility is located in *redbrick_dir/util/service* and is described by a README file in that directory. Do not run *tblchk* with the *-f* option unless you are directed to do so by Customer Support Services.

^{††}Data added, deleted, or changed since last backup is lost.

^{†††}A TMU REORG operation for a single index might be faster than dropping and creating the restored index, but it locks the table for exclusive use, whereas dropping and creating the restored index allows continued—but somewhat slower—access for users while the index is being rebuilt.



Moving Data with the Copy Management Option

In an enterprise environment with multiple Red Brick data warehouses linked together by networks, the need might arise to move data among warehouses in order to synchronize the data in equivalent tables. For example:

- A database administrator at the regional level needs to periodically consolidate data from by various stores into a larger regional database.
- A database administrator for a small department needs to periodically refresh the department database with current corporate data.
- A retail organization might want to share sales data for specific items with the suppliers of those items.

The Enterprise Control and Coordination and the Copy Management Option both include a utility named *rb_cm*, which is used to move data between warehouse databases across networks.

This chapter contains the following sections:

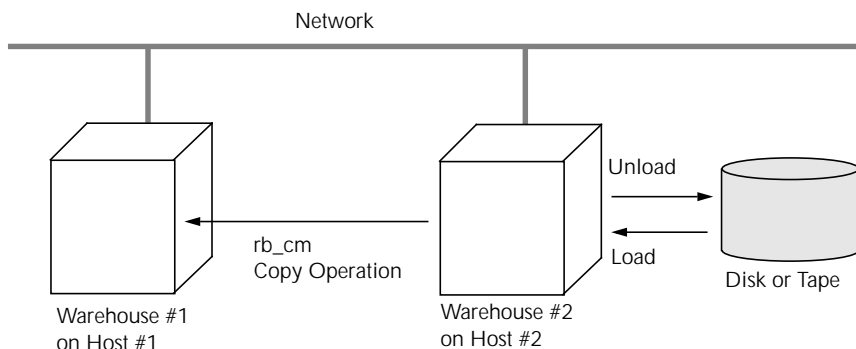
- The *rb_cm* Utility
- *rb_cm* Syntax
- TMU Control Files for Use with *rb_cm*
- Examples of *rb_cm* Operations
- Verifying Results of *rb_cm* Operations

The *rb_cm* Utility

The Table Management Utility (TMU) can perform high performance unloads and loads to and from physical storage. The *rb_cm* utility provides an interface that allows you to combine the following tasks into a single operation:

- Extract data from a warehouse database using a TMU UNLOAD statement.
- Move the data across a network (rather than to and from physical storage).
- Load the extracted data into a warehouse database using a TMU LOAD statement.

The following figure illustrates the difference between an *rb_cm* copy operation and the LOAD and UNLOAD commands:



The *rb_cm* utility can copy data between any two tables—in the same database, in different databases, in different databases in different warehouses, or in different databases on different platforms—as long as the column datatypes are compatible between the source and destination tables.

The *rb_cm* utility supports all of the existing TMU load and unload functionality. For example, *rb_cm* can unload data from the source table in internal or external format. It can unload by column value (selective unload) or by segment, and load in APPEND, INSERT, MODIFY, REPLACE, or UPDATE mode. These features give you substantial flexibility in copying data between tables.

You can issue an *rb_cm* command from either the computer on which the source table is located or the computer on which the destination table is located. The following sections discuss the requirements for running the *rb_cm* utility.

System Requirements

If you are performing a copy operation over a network, either from the local computer to the remote computer or from the remote computer to the local computer, the system requirements are as follows:

- Both computers must be on the same network.
- Both computers must have Red Brick Warehouse installed, and either the Enterprise Control and Coordination option or the Enterprise Copy Management option must be enabled on both computers.
- You must be able to establish communication from the local host to the remote host as follows:

UNIX From the local host where you issue the command, you must be able to access the remote shell daemon on the remote host.

Windows NT From the local host where you issue the command, you must be able to access the Red Brick Service on the remote host.

- Your environment on the remote host must have the *rb_tmu* executable in your path.

UNIX Your *.cshrc* or *.profile* file on the remote host should include lines to set the PATH environment variable to include *redbrick_dir/bin*.

Windows NT The Red Brick Service on the remote host automatically uses the correct path.

- **Windows NT** The Red Brick Copy Management Service must be started on the remote host.

If you are copying data between tables on the same computer, there are no special system requirements other than having either the Enterprise Control and Coordination option or the Enterprise Copy Management option enabled.

If you are issuing the *rb_cm* command from a system other than the source or destination machine, you must be able to access the remote shell on the source machine, and you must also be able to access the remote shell on the destination machine from the source machine.

Database Security Requirements

In order to copy table data using the *rb_cm* utility, the user running *rb_cm* must have the necessary authorizations on both the source and destination databases. The user running *rb_cm* must have:

- One of the following on the source database:
 - The DBA system role
 - Ownership of the source table
 - The ACCESS_ANY task authorization
 - SELECT privilege on the source table
- One of the following on the destination database:
 - The DBA system role
 - Ownership of the destination table
 - The MODIFY_ANY task authorization
 - INSERT, DELETE, or UPDATE privileges on the destination table. The specific privileges needed depend on the type of load that the *rb_cm* utility performs. The following table summarizes this requirement:

Load Mode at Destination	Required Permission		
	INSERT	DELETE	UPDATE
APPEND	Yes	No	No
INSERT	Yes	No	No
MODIFY	Yes	No	Yes
REPLACE	Yes	Yes	No
UPDATE	No	No	Yes

rb_cm Syntax

The syntax of the *rb_cm* utility is as follows:

```

source      rb_cm  [-s unload_host] [-c unload_config_path] [-h unload_rbhost]
(unload)    →      [-d unload_database] [-f filter_cmd]
parameters  unload_control_file  unload_username  unload_password

destination →      [-s load_host] [-c load_config_path] [-h load_rbhost]
(load)      [-d load_database] [-f filter_cmd]
parameters  load_control_file  load_username  load_password

[-p]
```

Note: The prefix *unload* refers to the source of the data; the prefix *load* refers to the destination.

8

-s unload_host, load_host

Specifies the computer hostname of the computer on which the corresponding source or destination table is located. These parameters are optional.

The default is the hostname of the computer from which you issue the *rb_cm* command.

-c unload_config_path, load_config_path

Specifies the pathname to the directory containing the *rbw.config* file for the corresponding source or destination warehouse host.

UNIX If the `RB_CONFIG` environment variable is set, this command-line argument is optional and, if present, overrides the environment variable. If `RB_CONFIG` is not set, this argument is required.

Windows NT The value of `RB_CONFIG` is taken from the Registry, based on the value of `RB_HOST`. If desired, another path for a different configuration file can be specified.

If no value is specified, the default is the value of `RB_CONFIG` on the corresponding source or destination computer.

-h unload_rbhost, load_rbhost

Specifies the logical name for the warehouse API daemon or thread (*rbwapid*) for the corresponding source or destination warehouse host.

UNIX If the RB_HOST environment variable is set, this command-line argument is optional and, if present, overrides the environment variable. If RB_HOST is not set, this argument is required.

Windows NT The value of RB_HOST is determined by its value in the Registry; if desired, another value for RB_HOST can be specified.

The default is the value of RB_HOST on the corresponding source or destination computer.

-d *unload_database, load_database*

Specifies the logical database names of the corresponding source and destination databases.

UNIX If the RB_PATH environment variable is set, this command-line argument is optional and, if present, overrides the environment variable. If RB_PATH is not set, this argument is required.

Windows NT The value of RB_PATH is taken from the Registry, based on the value of RB_HOST. If desired, another logical database name can be specified

The default is the value of RB_PATH on the corresponding source or destination computer.

-f *filter_cmd*

Specifies a user-supplied filter program. This parameter is optional. The program can be an executable or a script; the only restrictions are that it accept input from standard input and send output to standard output.

unload_control_file

Indicates the full pathname of the file containing the UNLOAD statement. This file must be located on the computer specified by *unload_host*. For more information, refer to “TMU Control Files for Use with *rb_cm*” on page 8-8.

unload_username, unload_password

Specifies the database username and password under which the unload portion of the *rb_cm* operation is performed.

load_control_file

Indicates the full pathname of the file containing the LOAD statement. This file must be located on the computer specified by *load_host*. For more information, refer to “TMU Control Files for Use with *rb_cm*” on page 8-8.

load_username, load_password,

Specifies the database username and password under which the load portion of the *rb_cm* operation is performed.

-p

Specifies the parallel TMU (*rb_ptmu*) to execute the LOAD statement contained in *load_control_file*. (There is no benefit to running the parallel TMU for an unload operation so this is always performed by the standard TMU).

Note: Arguments in the *rb_cm* command must be in the order shown in the syntax.

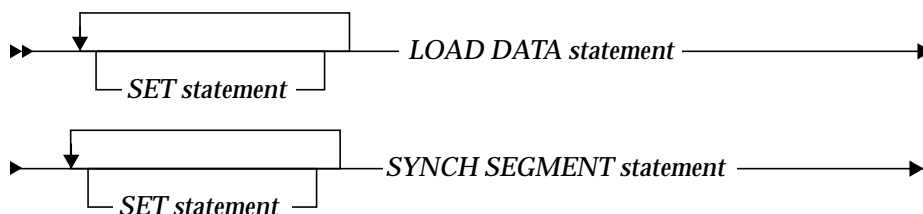
TMU Control Files for Use with rb_cm

The *rb_cm* utility works by directing the output from a TMU UNLOAD statement to a TMU LOAD statement. Before running *rb_cm*, therefore, you must prepare compatible TMU LOAD and UNLOAD control files.

The syntax for an UNLOAD control file for use with *rb_cm* is as follows:



The syntax of a LOAD DATA and/or a SYNCH OFFLINE SEGMENT control file for use with *rb_cm* is as follows:



Note that you can include only a single UNLOAD statement in an UNLOAD control file and only a single LOAD DATA statement in a LOAD DATA control file. Each statement must be terminated by a semicolon(;).

In both LOAD and UNLOAD control files, multiple control statements must be separated by a semicolon (;). Comments can be either enclosed in C-style delimiters (*/*...*/*), in which case they can span multiple lines, or preceded by two hyphens (--) and terminated by end-of-line, in which case they are limited to a single line.

LOAD and UNLOAD Statements

The LOAD and UNLOAD statements are required for their respective control files. The syntax of these statements differs in one respect from the syntax of LOAD and UNLOAD statements used with the TMU directly: You cannot specify a disk file or tape device as the destination of the unload operation and you cannot specify a disk file or a tape device as the data source of the load operation. You must specify standard output as the unload destination and standard input as the load source.

For LOAD and SYNCH statement syntax, refer to Chapter 3, “Loading Data into a Warehouse Database. For UNLOAD statement syntax, refer to Chapter 4, “Unloading Data from a Table.”

Specifying INTERNAL Format

If you are using the *rb_cm* utility to copy data between tables stored on the same platform type (for example, if both source and destination platforms are Sun SPARC systems), unload the data using the internal format. Internal format is a binary data format that requires less time to load than external-format data.

Internal format is the default unload format and does not need to be specified explicitly in the UNLOAD statement. The corresponding LOAD statement must include the FORMAT UNLOAD keywords, however, to indicate the data to be loaded is internal-format data.

Specifying EXTERNAL Format

If you are using the *rb_cm* utility to copy data between tables stored on platforms of different types (for example, if the source platform is Digital UNIX and the destination platform is Sun SPARC), you must unload the data into an external character format. External format produces plain text data in the database locale character set, data that is compatible across different platform types.

To unload data in the external format, include the EXTERNAL keyword in the UNLOAD statement. To load external format data, you must define all the fields in the input data records, which increases the complexity of the LOAD statement. However, you can have the TMU automatically generate a LOAD statement for the specific data and table you are unloading, either as part of the UNLOAD process or with a separate TMU GENERATE statement.

For example, if you create a file with the following GENERATE command and run the TMU for this file, the TMU will produce a file named *load_control_file*:

```
GENERATE LOAD FROM unload_table
INPUTFILE '-'
EXTERNAL
TMUFILE 'load_control_file' ;
```

The *load_control_file* will contain the necessary LOAD statement with all the column datatypes defined. You can edit this file to include any other TMU directives that may be required, such as a SYNCH statement or SET option.

SYNCH Statement

If data is copied into an offline segment of a table, that segment must be synchronized with the rest of the table. To perform this synchronization, write a control statement that contains a SYNCH statement, including the segment and table names. This statement is used only in conjunction with load operations into offline segments.

For information about the SYNCH statement, refer to “Writing a SYNCH Statement” on page 3-104.

SET Statements

The SET statements provide controls for:

- TMU behavior when the database or target tables are locked (SET LOCK option). If you are using *rb_cm* for unattended copies in multi-user environments, it is important to have the SET LOCK option set to WAIT.
- Size of the buffer cache used by the TMU (SET BUFFERCOUNT option).
- Temporary space used by the TMU for index-building operations (SET INDEX_TEMPSPACE options).

You can include all of these SET statements in a LOAD control file and all of them except the INDEX_TEMPSPACE options in an UNLOAD control file.

A SET statement affects load or unload behavior only during the *rb_cm* operation using the control file that contains that SET statement. After that session, the option value reverts to the value specified in the *rbw.config* file; if no value is specified in the *rbw.config* file, the option value reverts to its default.

For more information about these SET statements, refer to “SET Statements and Parameters to Control Behavior” on page 2-11.

Examples of *rb_cm* Operations

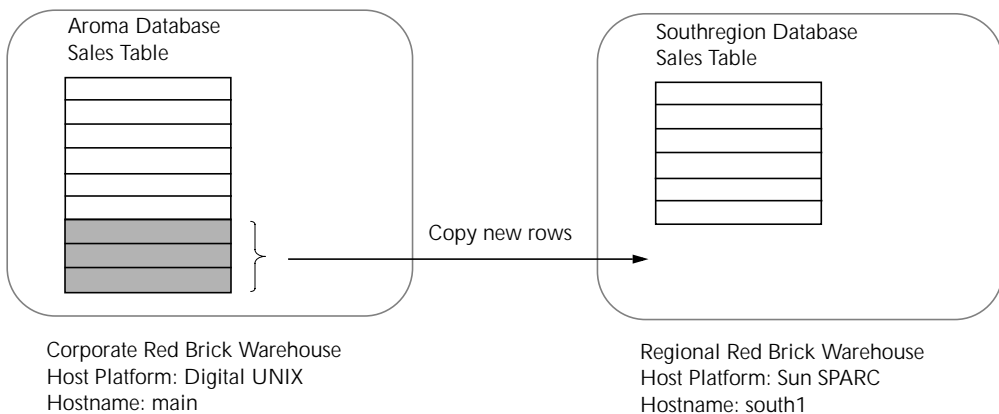
This section presents two scenarios in which data needs to be copied between tables and gives examples of the required *rb_cm* commands. These scenarios illustrate:

- Copying data between different computers.
- Copying data between tables in the same warehouse on the same computer.

Example: Copying Data Between Different Computers

Suppose that the Aroma database is located in a data warehouse for sales and marketing. A regional marketing team maintains a version of the Aroma database named Southregion, which contains only the data relevant to their region. The regional team can make changes to the Southregion database and run long queries against it without affecting users outside the team.

Periodically, the regional team wants to copy any new rows relevant to their region from the Sales table in the Aroma database to the Sales table in the Southregion database. They can do this using *rb_cm* with a selective unload operation. The following figure summarizes this scenario:



To perform the operation described above, the administrator needs to carry out the following tasks:

1. Set up the UNLOAD control file.
2. Set up the LOAD control file.
3. Issue the *rb_cm* command.

All of these steps are described below. Note that the first two steps only need to be done once—before the initial copy operation. Subsequent copies can reuse the same control files.

Setting Up the UNLOAD Control File

The administrator sets up an UNLOAD control file named *unload_new_sales* on the host computer. The UNLOAD statement contained in this file must unload to external format because the source and destination computers in this example have different architectures. The UNLOAD statement must also do a selective unload of the relevant rows. The following UNLOAD statement fulfills both of these requirements:

```
UNLOAD SALES
EXTERNAL
OUTPUTFILE '-'
WHERE PERKEY = 94050
      AND (MKTKEY = 1
          OR MKTKEY = 2
          OR MKTKEY = 3
          OR MKTKEY = 4);
```

This UNLOAD statement does a selective unload of those rows that are relevant to the south region (where Mktkey is equal to 1, 2, 3, or 4) and are new (where Perkey is equal to the most recent value).

Setting Up the LOAD DATA Control File

The administrator sets up a LOAD control file named *load_new_sales* on the destination computer. The GENERATE command can be used to obtain a LOAD DATA statement as follows:

```
GENERATE LOAD DATA FROM SALES INPUTFILE '-' EXTERNAL TMUFILE
'load_new_sales';
```

The administrator enters this statement in a TMU control file and runs the TMU. The TMU creates a file named *load_new_sales* containing the following LOAD statement:

```
LOAD DATA INPUTFILE '-'
RECORDLEN 62
INSERT
INTO TABLE SALES (
  PERKEY POSITION(2) INTEGER EXTERNAL(11) NULLIF(1)='% ',
  PRODKEY POSITION(14) INTEGER EXTERNAL(11) NULLIF(13)='% ',
  MKTKEY POSITION(26) INTEGER EXTERNAL(11) NULLIF(25)='% ',
  DOLLARS POSITION(38) DECIMAL EXTERNAL(12) NULLIF(37)='% ',
  WEIGHT POSITION(51) INTEGER EXTERNAL(11) NULLIF(50)='% ');
```

With the present example, the administrator needs to modify the *load_new_sales* file by changing the INSERT keyword to APPEND. This directs the TMU to append the rows to the destination table. The administrator might also add an OPTIMIZE ON clause to improve performance. The LOAD statement now looks as follows:

```
LOAD DATA INPUTFILE '-'
RECORDLEN 62
APPEND
OPTIMIZE ON
INTO TABLE SALES (
    PERKEY POSITION(2) INTEGER EXTERNAL(11) NULLIF(1)='% ',
    PRODKEY POSITION(14) INTEGER EXTERNAL(11) NULLIF(13)='% ',
    MKTKEY POSITION(26) INTEGER EXTERNAL(11) NULLIF(25)='% ',
    DOLLARS POSITION(38) DECIMAL EXTERNAL(12) NULLIF(37)='% ',
    WEIGHT POSITION(51) INTEGER EXTERNAL(11) NULLIF(50)='% ');
```

Note: The NULLIF clause at the end of each field specification is used by the TMU when it loads the data; an extra position before each field is reserved for a null indicator.

8

Running *rb_cm*

After the control files have been written, a user with the necessary privileges can issue an *rb_cm* command using those files. The *rb_cm* command can be issued from either the source host computer (main) or the destination host computer (south1).

If the *rb_cm* command is issued from main, the source host computer, the command for this operation (formatted for readability) might look like this:

```
% rb_cm \
source  _____ -s main -c redbrick_dir -h RB_HOST -d Aroma \
parameters          $RB_CONFIG/util/unload_new_sales maindba secret \
destination _____ -s south1 -c /south1_redbrick_dir -h RB_HOST -d Southregion \
parameters          '$RB_CONFIG'/util/load_new_sales southdba cryptic
```

Note the following details about the *rb_cm* command:

- The -c, -h, and -d options are present for both source and destination and will override the corresponding environment variables. These options are not required if the corresponding variables are set.
- You cannot use the RB_CONFIG environment variable as part of an explicit pathname with the *rb_cm* utility for a source or destination that is a Windows NT system.

Moving Data with the Copy Management Option

Examples of *rb_cm* Operations

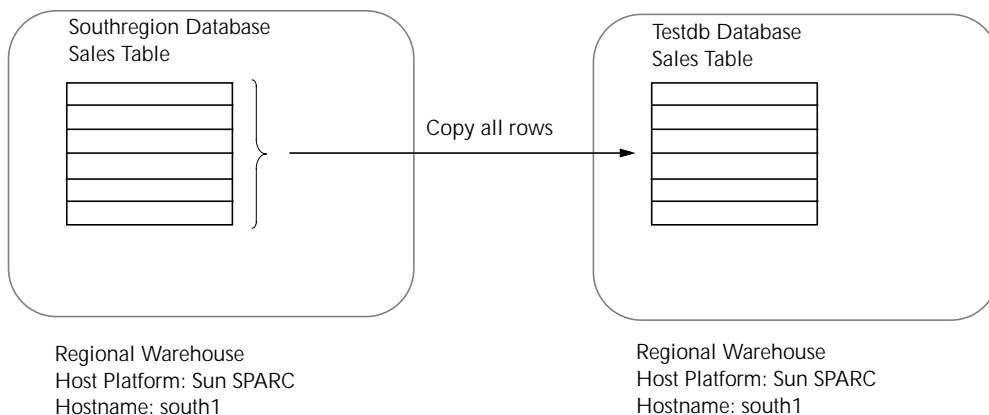
- If you use the `RB_CONFIG` environment variable to specify a control file on the remote computer, you must use appropriate escape characters so that it is passed to the remote computer and not interpreted on the local computer. So if you issued an equivalent command from the destination host computer, the command (formatted for readability) would look like this:

```
% rb_cm \  
source      — -s main -c redbrick_dir -h RB_HOST -d Aroma \  
parameters  ' $RB_CONFIG' /util/unload_new_sales maindba secret \  
destination — -s south1 -c /south1_redbrick_dir -h RB_HOST -d Southregion \  
parameters  $RB_CONFIG /util/load_new_sales SouthDBA cryptic
```

Note: The above examples are broken into multiple lines for clarity; when you enter the *rb_cm* command, it should all be entered as a single line.

Example: Copying Data Between Tables on the Same Computer

Now suppose that the same regional marketing team from the last example keeps a copy of their Southregion database named Testdb, to which they can make numerous updates and simulate various scenarios. Periodically they need to replace the modified data in Testdb with the actual data stored in the Southregion database. The following figure illustrates this scenario for a single table:



To perform this operation the administrator must set up the `LOAD` and `UNLOAD` control files and then run an appropriate *rb_cm* command.

Setting Up the UNLOAD Control File

The administrator sets up an UNLOAD control file named *unload_south_sales*. This file contains the following UNLOAD statement:

```
UNLOAD
SALES
OUTPUTFILE '-' ;
```

This UNLOAD statement unloads data in internal (binary) format. This is possible because both UNLOAD and LOAD operations take place on the same platform.

Setting Up the LOAD Control File

The administrator sets up an LOAD control file named *load_south_sales*. This file contains the following LOAD statement:

```
LOAD
INPUTFILE '-'
REPLACE
FORMAT UNLOAD
OPTIMIZE ON
INTO TABLE SALES;
```

This LOAD statement replaces all the rows in the sales table with the loaded data.

Running *rb_cm*

After the control files are set up, a user with the required privileges can issue an *rb_cm* command to copy the data. This command might look like this:

```
% rb_cm \
-d Southregion $RB_CONFIG/util/unload_south_sales SouthDBA cryptic \
-d Testdb /mktg/local/test/util/load_south_sales TestDBA cryp007tic
```

Note the following about the *rb_cm* command:

- The *-s* option is not present so the default source and destination is the machine on which the *rb_cm* command is issued.
- The *-c* and *-h* options are not present; both databases are in the same warehouse and use the warehouse API and configuration file defined by the *RB_HOST* and *RB_CONFIG* variables.
- You cannot use the *RB_CONFIG* environment variable as part of an explicit pathname with the *rb_cm* utility for a source or destination that is a Windows NT system.

Verifying Results of rb_cm Operations

To verify that all of the rows were successfully copied by the *rb_cm* utility, query the RBW_LOADINFO table in the destination database. This system table holds information on each load operation performed against the database, including loads that are issued as part of an *rb_cm* operation. This information includes the times at which the load started and completed, the number of rows inserted into the table, and the status of the load. For more information on the RBW_LOADINFO system table, refer to the *Warehouse Administrator's Guide*.

Example

This example illustrates how to query the RBW_LOADINFO system table to determine load information:

```
RISQL> select substr (tname,1,12) as table_name,  
> substr(username, 1,10) as user_name,  
> substr(string(started), 1, 19) as load_start,  
> substr(string(finished), 1, 19) as load_finish,  
> substr (status, 1, 6) as status  
> from rbw_loadinfo  
> where tname = 'SALES';
```

TABLE_NAME	USER_NAME	LOAD_START	LOAD_FINISH	STATUS
SALES	SYSTEM	1996-04-05 00:33:24	1996-04-05 00:33:26	NULL
SALES	SYSTEM	1996-04-06 00:35:38	1996-04-06 00:35:41	NULL

```
RISQL>
```

Example: Using the TMU in AGGREGATE Mode

This example illustrates one way in which the TMU Auto Aggregate option can be used to generate and maintain quarterly and yearly aggregates from daily input data.

This chapter contains the following sections:

- Background
- Strategy
- Load Procedure: Refresh Loads
- Load Procedure: Daily Loads
- Results

Background

Daily input data is extracted from another system used for day-to-day operations. This data has the following time granularities:

- Monthly totals for all months prior to the current month
- Month-to-date totals for the current month
- Daily totals for the current month

In addition to these time granularities, analysis done with the warehouse database requires quarter-to-date and year-to-date aggregates. The TMU can generate these figures automatically and update them as needed as part of the data loading procedures.

Example: Using the TMU in AGGREGATE Mode
Strategy

The warehouse database is updated daily with information that includes the new daily total and month-to-date total, as well as possible restated amounts for previous daily totals. Twice a month the entire warehouse database is completely reloaded (refreshed) from the operational system.

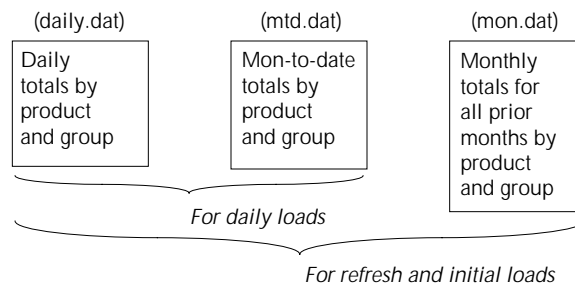
The example assumes the current date is May 4, 1996.

Strategy

The first two tasks are deciding how to capture restated daily values and devising a period key strategy to handle the desired time aggregate levels.

The fact that the daily inputs might contain restated values for previous days requires extra care in computing the quarter-to-date and year-to-date aggregates: Simply adding new daily totals to these aggregates does not capture any restated daily totals for days already included in the aggregate totals. The solution to this problem is to use the daily month-to-date totals from the operational data, and each day subtract the previous day's month-to-date totals from the two aggregates and add the new month-to-date totals, thereby capturing any restated daily totals. (Restatements for other than the current month are captured by the semi-monthly refreshes.)

This plan requires that input data for the daily updates be split into two files, one containing daily totals and one containing month-to-date totals so that only the month-to-date totals can be used to compute the aggregates. The semi-monthly refreshes (and the initial load) are done with three files: the daily and month-to-date files used for the daily updates, and a file containing monthly data for prior months. These files and their use are illustrated in the following figure:



After a refresh, the quarter-to-date and year-to-date aggregates are computed using the Auto Aggregate option. Splitting daily data into two files and refresh data into three files as described keeps the month-to-date totals separate for the aggregations done daily and after the refreshes. (Remember that you need both the current day's and the previous day's month-to-date records.)

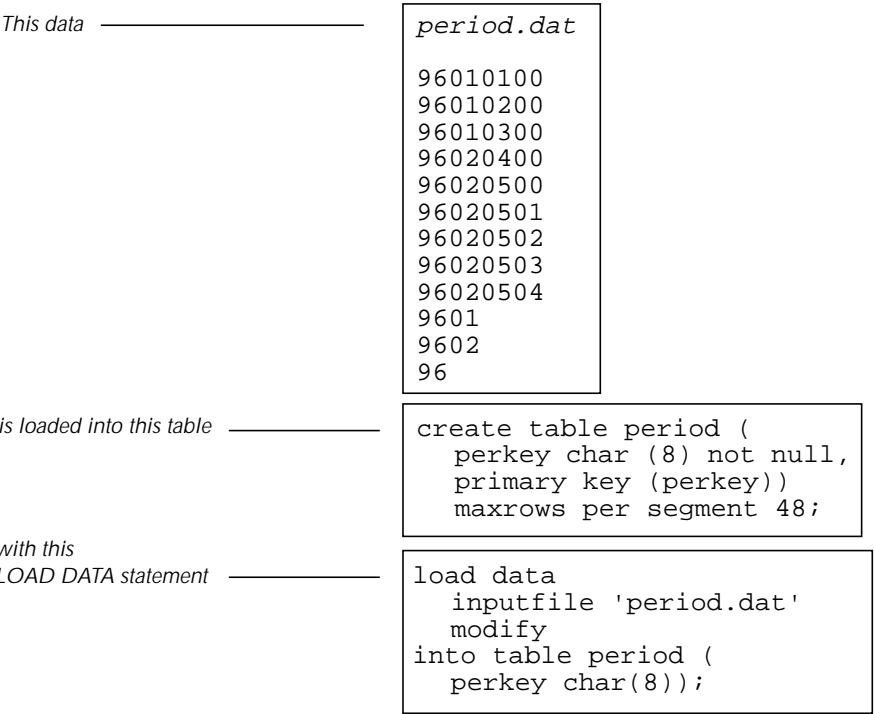
A period key of eight characters is used, two characters each for year, quarter, month, and date: YYQQMMDD. With this format, the various levels of aggregation are represented as follows:

Aggregation	Format	Explanation
Daily	YYQQMMDD	All 8 positions filled.
Monthly totals	YYQQMM00	All 8 positions filled.
Month-to-date	YYQQMM00	All 8 positions filled.
Quarter-to-date	YYQQ_ _ _ _	4 characters followed by 4 spaces.
Year-to-date	YY_ _ _ _ _ _	2 characters followed by 6 spaces.

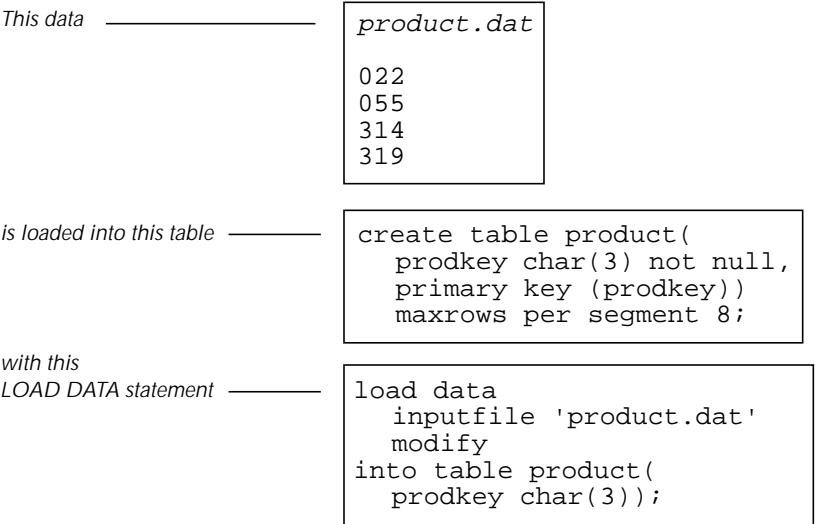
The Dimension Tables

The dimension tables in this example—Period, Product, and Market—are created and loaded with data as follows:

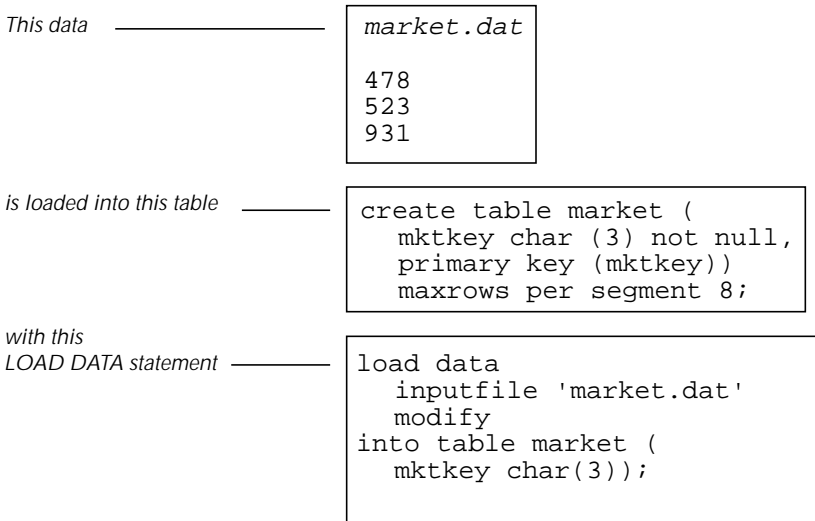
Period table:



Product table:



Market table:



A

The Sales Table

The Sales table is created as follows:

```
create table sales(  
  perkey char (8) not null,  
  prodkey char(3) not null,  
  mktkey char(3) not null,  
  dollars integer,  
  primary key (perkey, prodkey, mktkey),  
  foreign key (perkey) references period (perkey),  
  foreign key (prodkey) references product(prodkey),  
  foreign key (mktkey) references market (mktkey));
```

The data for the Sales table comes from three input files:

<i>month.dat</i>	Contains monthly total sales dollars for January–April, 1996 for each month for each product in each market.
<i>mtd.dat</i>	Contains total sales dollars sold month-to-date (May 3, 1996) for each product in each market.
<i>daily.dat</i>	Contains total sales dollars sold for May 1–3, 1996, for each day for each product in each market.

Example: Using the TMU in AGGREGATE Mode
Strategy

Monthly data for
Jan-Apr '96

```

/* month.dat */
96010100022478132
96010100022523048
96010100022931019
96010100055478026
96010100055523423
96010100055931025
96010100314478721
96010100314523096
96010100314931516
96010100319478318
96010100319523741
96010100319931925
96010200022478012
96010200022523036
96010200022931428
96010200055478076
96010200055523011
96010200055931066
96010200314478030
96010200314523741
96010200314931852
96010200319478045
96010200319523098
96010200319931016
96010300022478231
96010300022523311
96010300022931056
96010300055478753
96010300055523072
96010300055931455
96010300314478622
96010300314523944
96010300314931823
96010300319478456
96010300319523029
96010300319931047
96020400022478036
96020400022523891
96020400022931038
96020400055478059
96020400055523761
96020400055931648
96020400314478089
96020400314523078
96020400314931051
96020400319478512
96020400319523526
96020400319931096
    
```

perkey mktkey
 prodkey sales

Month-to-date
data for May '96

```

/* mtd.dat */
96020500022478055
96020500022523106
96020500022931066
96020500055478124
96020500055523164
96020500055931212
96020500314478124
96020500314523103
96020500314931107
96020500319478093
96020500319523094
96020500319931088
    
```

perkey mktkey
 prodkey sales

Daily data
May 3 '96

```

/* daily.dat */
96020501022478044
96020501022523033
96020501022931022
96020501055478055
96020501055523066
96020501055931099
96020501314478088
96020501314523077
96020501314931065
96020501319478045
96020501319523006
96020501319931008
96020502022478004
96020502022523068
96020502022931041
96020502055478023
96020502055523082
96020502055931091
96020502314478005
96020502314523009
96020502314931013
96020502319478021
96020502319523052
96020502319931049
96020503022478007
96020503022523005
96020503022931003
96020503055478046
96020503055523016
96020503055931022
96020503314478031
96020503314523017
96020503314931029
96020503319478027
96020503319523036
96020503319931031
    
```

perkey mktkey
 prodkey sales

4 products
(022, 055, 314, 319),
3 markets
(478, 523, 931)

A

Load Procedure: Refresh Loads

The following procedure is used to load the Sales table for each semi-monthly refresh (and the initial load).

(Steps 1–3 load the detail level records.)

1. Load the *month.dat* file containing monthly totals for previous months.
2. Load the *mtd.dat* file with the month-to-date totals for each product and market.
3. Load the *daily.dat* file with the daily totals for the current month.

(Steps 4–5 produce quarterly and quarter-to-date records.)

4. Load data for prior months (January–April) and compute prior quarters' aggregates (Q1: January–March) using the *month.dat* file.
5. Load current month's to-date data (May) and compute the quarter-to-date (April and May to date) aggregate using the *mtd.dat* file.

(Steps 6–7 produce the year to date aggregates.)

6. Load prior months' data (January–April) and compute the year-to-date aggregate for prior months using the *month.dat* file.
7. Load current month's data (May) and finish year-to-date aggregate by including current month using the *mtd.dat* file.

For Refresh Load, Steps 1–3:

The following LOAD DATA statements are used to load data into the Sales table initially and for each semi-monthly refresh thereafter.

```
/*initial load */
/*loading month records */
load data
  inputfile 'month.dat'
  modify
into table sales(
  perkey char(8),
  prodkey char(3),
  mktkey char(3),
  dollars integer external (3));
```

```
/*loading month-to-date records */
load data
  inputfile 'mtd.dat'
  append
into table sales(
  perkey char(8),
  prodkey char(3),
  mktkey char(3),
  dollars integer external (3));

/*loading daily records */
load data
  inputfile 'daily.dat'
  append
into table sales(
  perkey char(8),
  prodkey char(3),
  mktkey char(3),
  dollars integer external (3));
```

For Refresh Load, Steps 4-5:

The following LOAD DATA statements are used to produce the quarterly and quarter-to-date aggregates initially and for each semi-monthly refresh thereafter.

```
/* load monthly data and compute aggregates for full quarters*/
load data
  inputfile 'month.dat'
  modify aggregate
into table sales(
  perkey position(1:4) char (8),
  prodkey position(9:11) char(3),
  mktkey position(12:14) char(3),
  dollars position(15) integer external(3) add);

/* load month-to-date data; compute aggregate for qtr-to-date*/
load data
  inputfile 'mtd.dat'
  modify aggregate
into table sales(
  perkey position(1:4) char (8),
  prodkey position(9:11) char(3),
  mktkey position(12:14) char(3),
  dollars position(15) integer external(3) add);
```

A

For Refresh Load, Steps 6–7:

The following LOAD DATA statements are used to produce the yearly and year-to-date aggregates initially and for each semi-monthly refresh thereafter.

```
/*load monthly data and compute year-to-date for prior months*/
load data
  inputfile 'month.dat'
  modify aggregate
into table sales(
  perkey position(1:2) char (8),
  prodkey position(9:11) char(3),
  mktkey position(12:14) char(3),
  dollars position(15) integer external(3) add);

/* load current month and complete year-to-date aggregate*/
load data
  inputfile 'mtd.dat'
  modify aggregate
into table sales(
  perkey position(1:2) char (8),
  prodkey position(9:11) char(3),
  perkey position(12:14) char(3),
  dollars position(15) integer external(3) add);
```

Load Procedure: Daily Loads

The following procedure is used to load the Sales table each day with the daily updates.

(Steps 1–2 load new and modified, or restated, detail data.)

1. Load the *daily.dat* file containing daily totals for the current month, using MODIFY mode (to capture any restatements).
2. Load the *mtd.dat.new* file with the month-to-date totals as of the current date, using MODIFY mode.

(Steps 3–4 adjust the quarterly and quarter-to-date figures for any restated totals.)

3. Subtract out yesterday's month-to-date total from the quarterly and quarter-to-date figures by loading the *mtd.dat.old* file using MODIFY AGGREGATE mode and subtracting the dollars column.
4. Add in today's new month-to-date total to the quarterly and quarter-to-date figures by loading the *mtd.dat.new* file, using MODIFY AGGREGATE mode and adding the dollars column.

- (Steps 5–6 adjust the yearly and year-to-date figures for any restated totals.)
5. Subtract out yesterday's month-to-date total from the yearly and year-to-date figures by loading the *mtd.dat.old* file using MODIFY AGGREGATE mode and subtracting the dollars column.
 6. Add in today's new month-to-date total to the yearly and year-to-date figures by loading the *mtd.dat.new* file, using MODIFY AGGREGATE mode and adding the dollars column.

For Daily Load, Steps 1–2:

The following LOAD DATA statements are used to load the daily detail data.

```
/*loading month-to-date records*/
load data
  inputfile 'mtd.dat.new'
  modify
into table sales(
  perkey char(8),
  prodkey char(3),
  mktkey char(3),
  dollars integer external (3));

/*loading daily records*/
load data
  inputfile 'daily.dat'
  modify
into table sales(
  perkey char(8),
  prodkey char(3),
  mktkey char(3),
  dollars integer external (3));
```

A

For Daily Load, Steps 3–4:

The following LOAD DATA statements are used to calculate the quarter and quarter-to-date aggregates, including adjustments for any restated totals.

```
/*after initial loads, daily updates*/
/*using yesterday's mtd.dat file */
/*(you will always have to keep the prior day's data)*/
/* monthly totals and aggregates for previous quarters */
load data
  inputfile 'mtd.dat.old' */yesterday's file/*
  modify aggregate
into table sales(
  perkey position(1:4) char (8),
  prodkey position(9:11) char(3),
  mktkey position(12:14) char(3),
  dollars position(15) integer external(3) subtract);
```

Example: Using the TMU in AGGREGATE Mode

Results

```
/* current month for current quarter-to-date data */
load data
  inputfile 'mtd.dat.new' /*today's file*/
  modify aggregate
into table sales(
  perkey position(1:4) char (8),
  prodkey position(9:11) char(3),
  mktkey position(12:14) char(3),
  dollars position(15) integer external(3) add);
```

For Daily Load, Steps 5–6:

The following LOAD DATA statements are used to calculate the yearly and year-to-date aggregates, including adjustments for any restated totals.

```
/*aggregates of year-to-date data - previous months*/
load data
  inputfile 'mtd.dat.old' /*yesterday's file*/
  modify aggregate
into table sales(
  perkey position(1:2) char (8),
  prodkey position(9:11) char(3),
  mktkey position(12:14) char(3),
  dollars position(15) integer external(3) subtract);

/*aggregates of year-to-date data - current month*/
load data
  inputfile 'mtd.dat.new' /*today's file*/
  modify aggregate
into table sales(
  perkey position(1:2) char (8),
  prodkey position(9:11) char(3),
  mktkey position(12:14) char(3),
  dollars position(15) integer external(3) add);
```

Results

You update the database with a new daily file (*daily.dat*) and a new monthly file (*mtd.dat*) for May 4, with some restated daily totals. The following figure shows the new data. (Only the first row for May 1, the restated totals—three items on May 1—and the data for May 4 from the *daily.dat* file are shown.)

Month-to date data for May '96
(mtd.dat)

96020500022478100
96020500022523142
96020500022931093
96020500055478182
96020500055523233
96020500055931303
96020500314478206
96020500314523176
96020500314931361
96020500319478256
96020500319523208
96020500319931168

Daily data for May 4 '96
(daily.dat)

96020501022478044
...
96020501314931265
96020501319478154
96020501319523060
...
96020504022478045
96020504022523036
96020504022931027
96020504055478058
96020504055523069
96020504055931091
96020504314478082
96020504314523073
96020504314931054
96020504319478054
96020504319523060
96020504319931080

Restated totals {

The quarter-to-date and year-to-date figures change as shown, reflecting not only the new sales for May 4th, but also the restated sales for previous days.

May 3rd aggregates

RISQL> select * from sales where perkey = '9602';				
PERKEY	PROD	MKTG	DOLLARS	
9602	022	478		91
9602	022	523		997
9602	022	931		104
9602	055	478		183
9602	055	523		925
9602	055	931		860
9602	314	478		213
9602	314	523		181
9602	314	931		158
9602	319	478		605
9602	319	523		620
9602	319	931		184
RISQL> select * from sales where perkey = '96';				
PERKEY	PROD	MKTG	DOLLARS	
96	022	478		466
96	022	523		1392
96	022	931		607
96	055	478		1038
96	055	523		1431
96	055	931		1406
96	314	478		1586
96	314	523		1962
96	314	931		2349
96	319	478		1424
96	319	523		1488
96	319	931		1172

May 4th aggregates

RISQL> select * from sales where perkey = '9602';				
PERKEY	PROD	MKTG	DOLLARS	
9602	022	478		136
9602	022	523		1033
9602	022	931		131
9602	055	478		241
9602	055	523		994
9602	055	931		951
9602	314	478		295
9602	314	523		254
9602	314	931		412
9602	319	478		768
9602	319	523		734
9602	319	931		264
RISQL> select * from sales where perkey = '96';				
PERKEY	PROD	MKTG	DOLLARS	
96	022	478		511
96	022	523		1428
96	022	931		634
96	055	478		1096
96	055	523		1500
96	055	931		1497
96	314	478		1668
96	314	523		2035
96	314	931		2603
96	319	478		1587
96	319	523		1602
96	319	931		1252

A

Example: Using the TMU in AGGREGATE Mode
Results



Loading Data from Existing Systems

Data for the Red Brick Warehouse server can be obtained from either flat file formats or from existing relational databases. In general, obtaining the data from flat files results in a shorter, less complex production process. Flat files must be sorted and then converted into one of the several file formats that the Red Brick Warehouse Table Management Utility (TMU) accepts. If data cannot be obtained from flat file sources, then the following descriptions can help in extracting data from several popular relational database management systems.

This chapter contains the following sections:

- Oracle's ORACLE
- Oracle's Rdb
- Metaphor's DBS 2xx
- IBM's DB2

Oracle's ORACLE

To extract data from an Oracle database, use SQL*Plus to create one flat file per table. All of the commands described here are best edited into a command file that can be submitted in batch to SQL*Plus.

1. Configure SQL*Plus to suppress column headers and page numbers on every page by using the commands:

```
set termout off
set echo off
set heading off
set pagesize 0
set feedback off
```

2. Specify data-specific formatting commands to control the format of the output flat file:

```
set linesize 43
column key1 format 99
column key2 format 9999
column key3 format 99999999
column rowid format a18
```

3. Direct the output of SQL*Plus to the file *MYFILE.LIS* with the command:

```
SPOOL MYFILE.LIS
```

4. Spool the contents of table *MYTABLE* to *MYFILE.LIS* with the command:

```
select key1, key2, key3, rowid from MYTABLE;
```

Use the UNIX *od* (octal dump) or *more* command to look at the contents of *MYFILE.LIS* in order to determine the column offsets to each field in the file. (If the Oracle database is on a VAX system, then you can use the VMS DUMP utility to look at the file contents.)

Oracle's Rdb

To dump an Rdb database to a flat file, use the Rdb Maintenance Utility (RMU). Invoke RMU with a command of the form shown below:

```
$ RMU/UNLOAD/RMS_RECORD_DEF=FILE=file_name - database_file_name -
table_name - output_file_name
```

Examine the *output_file_name* to prepare the LOAD DATA statement for the TMU. For more information, refer to the documentation for the Rdb Maintenance Utility.

Metaphor's DBS 2xx

If the data destined for Red Brick Warehouse is currently resident on a Metaphor DBS 2xx database server, the DIS/Metaphor COPY OUT TO TAPE command can be used to produce a 9-track tape of each table to be transferred. This tape will be in the Metaphor/DIS MTTF format. The tape can then be converted to fixed record format using the Metaphor/DIS Table Unformatter utility as described in the *IBM Data Interpretation System Database Administration* manual.

The fixed record tapes can then be sorted as necessary and the resulting files used as input to the TMU. This procedure also applies to Britton-Lee and Sharebase database systems.

IBM's DB2

To extract data from an IBM/DB2 database, use the DSNTIAUL unload utility. This method of using the DSNTIAUL unload utility to extract a database from DB2 does not use REORG/UNLOAD. *Do not* use REORG/UNLOAD because it does not produce the desired formats.

1. Unload data from DB2 using DSNTIAUL utility described in IBM's *DB2 Utilities* manual. See attached JCL example. The resulting flat file(s) may then be sorted as necessary and written to tape as a fixed blocked file using AL label types. The resulting tapes can be processed by the TMU.
2. Format of the output is described in LOAD statements when SYSPUNCH DD is included in the JCL.

A sample JCL stream for extracting data from a DB2 database is shown below.

```
//JOB CARD
***JOBPARM S=SYSA,R=MCS
***SETUP R=1
***SETUP OUTPUT=VARIOUS
***
*****
***
*** JCL FOR DSNTIAUL UNLOAD UTILITY
*** THE FOLLOWING JCL CODE USES THE DSNTIAUL
*** UTILITY, WHICH IS AN UNCONVENTIONAL
*** METHOD OF UNLOADING A DATABASE FROM AN IBM DB2.
*** THE SUDS DATABASE CONSISTS OF FOUR (4) TABLES:
*** PRODUCT, MARKET, PERIOD, AND FACT.
***
*****
***
//STEP1 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//STEPLIB DD DISP=SHR,DSN=DSN220MC.DSNEXIT39
// DD DISP=SHR,DSN=DSN220MC.DSNLOAD
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
DSN SYSTEM(DB2X)
RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB22) LIB('DSN220MC.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DSN=SUDS.PRODUCT.UNLOAD,
// DISP=(NEW,KEEP,KEEP),DCB=DEN=4,
// LABEL=(1,SL,EXPDT=98000),
// UNIT=TAPE,
// VOL=(,RETAIN,,35,SER=RBST01)
//SYSPUNCH DD SYSOUT=*
//SYSIN DD *
RBSSAS.PROD01HVDC
//STEP2 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
```

Loading Data from Existing Systems

IBM's DB2

```
//STEPLIB DD DISP=SHR,DSN=DSN220MC.DSNEXIT
// DD DISP=SHR,DSN=DSN220MC.DSNLOAD
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
    DSN SYSTEM(DB2X)
    RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB22) LIB('DSN220MC.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DSN=SUDS.MARKET.UNLOAD,
// DISP=(NEW,KEEP,KEEP),DCB=DEN=4,
// LABEL=(2,SL,EXPDT=98000),
// UNIT=TAPE,
// VOL=(,RETAIN,,35,REF=*.STEP1.SYSREC00)
//SYSPUNCH DD SYSOUT=*
//SYSIN DD *
    RBSSAS.MKT01HVDC
//STEP3 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//STEPLIB DD DISP=SHR,DSN=DSN220MC.DSNEXIT
// DD DISP=SHR,DSN=DSN220MC.DSNLOAD
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
    DSN SYSTEM(DB2X)
    RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB22) LIB('DSN220MC.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DSN=SUDS.PERIOD.UNLOAD,
// DISP=(NEW,KEEP,KEEP),DCB=DEN=4,
// LABEL=(3,SL,EXPDT=98000),
// UNIT=TAPE,
// VOL=(,RETAIN,,35,REF=*.STEP1.SYSREC00)
//SYSPUNCH DD SYSOUT=*
//SYSIN DD *
    RBSSAS.PER01HVDC
//STEP4 EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//STEPLIB DD DISP=SHR,DSN=DSN220MC.DSNEXIT
// DD DISP=SHR,DSN=DSN220MC.DSNLOAD
//SYSTSPRT DD SYSOUT=*
//SYSTSIN DD *
    DSN SYSTEM(DB2X)
    RUN PROGRAM(DSNTIAUL) PLAN(DSNTIB22) LIB('DSN220MC.RUNLIB.LOAD')
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSREC00 DD DSN=SUDS.FACT.UNLOAD,
// DISP=(NEW,KEEP,KEEP),DCB=(LRECL=78,BLKSIZE=16380,DEN=4),
// LABEL=(4,SL,EXPDT=98000),
// UNIT=TAPE,
// VOL=(,RETAIN,,35,REF=*.STEP1.SYSREC00)
//SYSPUNCH DD SYSOUT=*
//SYSIN DD *
    RBSSAS.FACT01HVDC
//*
```

Defined Locales

This appendix identifies the languages, territories, character sets, and sort components supported by the Red Brick products. The following table lists the *logical* combinations of these components; however, any combination can be used in a locale specification.

The values shown in boldface are the default values for the corresponding language when an incomplete locale is specified. For example, the default character set for German is Latin1.

Language	Territory	Character Set	Sort
English	UnitedStates	US-ASCII Latin1 MS1252 UTF-8	Binary Default
	Australia		
	Canada		
	UnitedKingdom		
German	Germany	Latin1 MS1252 UTF-8	Default Binary
	Austria		
	German-Switzerland		
French	France	Latin1 MS1252 UTF-8	Default Binary
	French-Canada		
	French-Switzerland		

Defined Locales

Language	Territory	Character Set	Sort
Spanish	Spain	Latin1 MS1252 UTF-8	Spanish TraditionalSpanish Binary
	Argentina		
	Chile		
	Mexico		
Portuguese	Portugal	Latin1 MS1252 UTF-8	Default Binary
	Brazil		
Italian	Italy	Latin1 MS1252 UTF-8	Default Binary
Norwegian	Norway	Latin1 MS1252 UTF-8	Danish Binary
Swedish	Sweden	Latin1 MS1252 UTF-8	Swedish Binary
Danish	Denmark	Latin1 MS1252 UTF-8	Danish Binary
Finnish	Finland	Latin1 MS1252 UTF-8	Swedish Binary
Japanese	Japan	JapanEUC MS932 UTF-8	Binary

Notes

- The component strings in this table must be spelled exactly as shown, but they are not case-sensitive.
- In the Sort column, any value that is not “Binary” is a linguistic sort definition. “Default” refers to the sort definition specified by the CAN/CSA Z243.4.1 Canadian ordering standard, which covers English and several Western European languages.

- *All* character sets include US-ASCII as a subset, so any of the listed character sets can be used when the language is English; however, the character sets listed for each language are the most appropriate choices.
- Character set conversions can reliably be performed between any two character sets listed for a given language. Conversions outside the scope of each language row in the table are not supported. For example, characters can be converted from Latin1 to MS1252 but not from Latin1 to JapanEUC.
- The MS932 character set, listed for Japanese, is a superset of Shift-JIS.
- Unicode is not a supported character set, nor are any shifted encoding schemes.

Defined Locales



Index

Symbols

%, TMU NULL indicator 4-4, 4-17, 5-8
%, TMU wildcard character 3-77
_, TMU wildcard character 3-77

A

ACCEPT keyword, TMU 3-76
ACCESS_ANY task authorization 8-4
ADD aggregate operator, TMU 3-59
AGGREGATE mode, loading data 3-26
AGGREGATE operators, TMU 3-59
APPEND mode, loading data 3-25
AS \$pseudocolumn, TMU 3-51
Auto Aggregate option, TMU
described 1-8, 3-2
example A-1 to A-13
usage 3-59
Automatic Row Generation, TMU
described 3-3, 3-10 to 3-16
example 3-39, 3-41
syntax and usage 3-38 to 3-44
AUTOROWGEN parameter
See Automatic Row Generation

B

backup operations, TMU 7-2 to 7-13
See also restore operations
backup level 7-3
BACKUP statement syntax 7-5
during index creation 7-5
licensing option 7-2
locks during 7-4
overview 1-6
procedure 7-10
sample scenarios 7-10
BACKUP statement (TMU), syntax 7-5
binary datetime inputs 3-101 to 3-103

blanks in input data 3-111
buffer cache, TMU 2-13

C

cache size, buffer (TMU) 2-13
cases, tracked by technical support xix
cautions
backup levels for TMU restore 7-12
Criteria clause and locale 3-75
escape character and locale 3-77
incompatible character sets 3-31
Locale clause scope 3-29
overflow on increment fields 3-68
radix and locale 3-85
REPLACE mode for TMU 3-25
separator character restrictions 3-27
UNDO LOAD and REPLACE
mode 3-105
character data, sort methods C-2
CHARACTER fieldtype, TMU 3-84
character set, conversions C-3
columns, determining default values 3-42
Comment clause, LOAD DATA
statement 3-80 to 3-81
comments, TMU control file 1-3
comparisons, TMU LOAD DATA
statement 3-75
compressed files
input to TMU 2-7
output from TMU 4-7
CONCAT fields, LOAD DATA
statement 3-62 to 3-64
constant dates, loading 3-65, 3-93, 3-100
CONSTANT fields, LOAD DATA
statement 3-65 to 3-66
control files, TMU 1-3

- conventions
 - syntax diagrams xv
 - syntax notation xiv
- copy management, *See* `rb_cm` utility
- CREATE TABLE control file
 - from GENERATE statement 5-2
 - from UNLOAD statement 4-6
- Criteria clause, LOAD DATA statement
 - comparisons, 3-valued logic 3-78
 - locale use 3-31
 - syntax and usage 3-75 to 3-79
- CURRENT_DATE keyword, TMU 3-91
- CURRENT_TIME keyword, TMU 3-91
- CURRENT_TIMESTAMP keyword, TMU 3-91
- Customer Support Center xviii
- D**
- damaged segment, restoring with TMU 7-12
- data
 - conversion to EXTERNAL format 4-4
 - loading 1-4, 3-1 to 3-119
 - loading from other databases B-1 to B-4
 - loading into third-party tools 4-14
 - unload formats 4-2
 - unloading 1-5, 4-1 to 4-16
- database option (-d), TMU 2-3
- databases
 - backup operations, TMU 7-2 to 7-13
 - loading data 3-1 to 3-119
 - locking by TMU 2-12
 - moving 4-14
 - restore operations, TMU 7-2 to 7-13
 - upgrading to new release, general 1-7
- datatypes, conversions during load
 - process 3-111 to 3-113
- DATE fieldtype, TMU 3-91
- dates, loading constant dates 3-65, 3-93, 3-100
- datetime inputs
 - binary and packed/zoned
 - decimal 3-101 to 3-103
 - Datetime fieldtypes 3-91 to 3-100
 - format masks 3-93
 - restricted format masks 3-101
- DB2 database, input from B-3
- DECIMAL fieldtype, TMU 3-85, 3-87
- DECIMAL ZONED fieldtype, TMU 3-87
- Discard clause, loading data 3-33 to 3-44
- discard files, TMU
 - all discards 3-35
 - locale 3-31
 - optimized load discards 3-47
 - referential integrity discards 3-35, 3-37
 - types and use 3-33
- discarded rows during load 3-111
- DISCARDFILE keyword, TMU 3-35, 3-47
- disk file formats, input data 3-107
- disk spill files, INDEX TEMPSPACE
 - parameters 2-14
- documentation
 - list of Red Brick Systems xii
 - support xx
- DOUBLE PRECISION fieldtype, TMU 3-90
- duplicate records
 - discarding 3-47
 - optimize mode 3-45
 - REORG operations 6-4
- E**
- EBCDIC data format 3-26, 3-106
- e-mail addresses, for Red Brick Systems xviii
- empty input fields 3-111
- Enterprise Control and Coordination
 - option 1-8
 - See also* `rb_cm` utility
 - copy management 8-1 to 8-16
 - load information 3-3
- Enterprise Copy Management option, *See* `rb_cm` utility
- environment variables
 - general use with TMU 1-4
 - RB_CONFIG with TMU 2-5
 - RB_PATH with TMU 2-3
- error codes 2-4
- ESCAPE keyword
 - Criteria clause 3-77
 - UNLOAD statement 4-9
- exit status codes 2-4

external data format, TMU
 conversion rules 4-4
 example 4-16, 5-7
 with rb_cm utility 8-9
 EXTERNAL keyword, TMU UNLOAD
 statement 4-6

F

fieldtypes, TMU input
 records 3-82 to 3-100
 CHARACTER 3-84
 conversions during load 3-111 to 3-113
 CURRENT_DATE 3-91
 CURRENT_TIME 3-91
 CURRENT_TIMESTAMP 3-91
 DATE 3-91
 DECIMAL 3-85, 3-87
 DECIMAL ZONED 3-87
 DOUBLE PRECISION 3-90
 FLOAT EXTERNAL 3-86
 INTEGER 3-85, 3-89
 length of field 3-83
 M4DATE 3-92
 REAL 3-90
 scale of field 3-83
 SMALLINT 3-89
 TIME 3-91
 TIMESTAMP 3-91
 TINYINT 3-89

file formats, input data
 disk files 3-107
 fixed record 3-107
 separated record 3-108
 tape files 3-109

file redirection, TMU 2-7

filesize, TAR limit 4-7

fixed-format records, input data 3-107

FLOAT EXTERNAL fieldtype, TMU 3-86

FORCE keyword, TMU restore
 operation 7-7

Format clause, LOAD DATA
 statement 3-24 to 3-28

FORMAT keyword, TMU
 IBM format 3-26
 IBM SEPARATED by 'c' format 3-27
 SEPARATED by 'c' format 3-27
 UNLOAD format 3-27

format masks, datetime
 datetime fieldtypes 3-93
 numeric fieldtypes 3-101

G

GENERATE statements (TMU)
 CREATE TABLE syntax and
 usage 5-2 to 5-3
 example 5-7
 example with rb_cm 8-12
 LOAD DATA syntax and
 usage 5-4 to 5-6

I

IBM standard label tapes 3-106

INCREMENT fields 3-70 to 3-71

incremental backup, *See* backup operations,
 TMU

INDEX_TEMPSPACE parameters
 DIRECTORY 2-15
 MAXSPILLSIZE 2-15
 THRESHOLD 2-15
 TMU control 2-14

indexes
 default names 6-3
 rebuilding with REORG 6-1
 STAR, invalid, cause of 6-2

Input clause, LOAD DATA
 statement 3-20 to 3-23

input data
 CONCAT fields 3-62 to 3-64
 CONSTANT fields 3-65 to 3-66
 fieldtypes 3-82 to 3-100
 file formats 3-106
 INCREMENT fields 3-70 to 3-71
 ordered 3-10
 record formats 3-106
 SEQUENCE fields 3-68 to 3-69
 simple fields 3-57 to 3-61
 unused fields 3-50

input files, LOAD DATA statement 3-20

input locale, for TMU
 defined 3-29

INSERT mode, loading data 3-25

INTEGER fieldtype, TMU 3-85, 3-89

internal data format, TMU
 for unloaded data 4-3
 with rb_cm utility 8-9
 interrupted load 3-22
 interval option (-i), TMU 2-3
 INTO OFFLINE SEGMENT keyword,
 TMU 3-72
 invalid STAR indexes, cause of 6-2

K

keywords, in syntax diagrams xvii

L

length of field, TMU input records 3-83
 LIKE, NOT LIKE, TMU wildcards 3-77
 LOAD DATA control file
 from GENERATE statement 5-4
 from UNLOAD statement 4-6
 with rb_cm utility 8-9
 LOAD DATA statement 3-18
 See also loading data
 ACCEPT criteria 3-76
 ADD aggregate operator 3-59
 AGGREGATE mode 3-26
 APPEND mode 3-25
 AUTOROWGEN keyword 3-38
 clauses, main
 Comment 3-80 to 3-81
 Criteria 3-75 to 3-79
 Discard 3-33 to 3-44
 Format 3-24 to 3-28
 Input 3-20 to 3-23
 Locale 3-29 to 3-32
 Optimize 3-45 to 3-48
 Segment 3-72 to 3-74
 Table 3-49 to 3-71
 CONCAT fields 3-62 to 3-64
 CONSTANT fields 3-65 to 3-66
 creating with GENERATE 8-9
 fieldtypes 3-82 to 3-100
 INCREMENT fields, input
 data 3-70 to 3-71
 input files 3-20
 INSERT mode 3-25
 MAX aggregate operator 3-59
 MIN aggregate operator 3-59

LOAD DATA statement (*continued*)

 MODIFY mode 3-25
 NLS_LOCALE keyword 3-29
 NULLIF keyword 3-58
 POSITION keyword 3-57
 pseudocolumns 3-50, 3-64, 3-77
 rb_cm example 8-12, 8-15
 rb_cm, use with 8-8
 RECORDLEN keyword 3-24, 3-107
 REJECT criteria 3-76
 REPLACE mode 3-25
 RETAIN keyword 3-52
 SEQUENCE fields 3-68 to 3-69
 simple fields 3-57 to 3-61
 SUBSTR keyword 3-84
 SUBTRACT aggregate operator 3-59
 syntax 3-18
 syntax summary 3-114 to 3-119
 trim options 3-63
 UPDATE mode 3-25
 loading data 3-1 to 3-119
 Auto Aggregate example A-1 to A-13
 datatype conversions 3-111 to 3-113
 discard files 3-35 to 3-37, 3-47
 discarded rows 3-2, 3-47, 3-111
 failure or interruption 3-22
 from other databases B-1 to B-4
 inputs and outputs 3-2
 into segments 3-72
 load information 3-80
 offline load 3-72
 overview 1-4
 processing flow 3-4 to 3-5
 RBW_LOADINFO system table 3-80
 status, checking 3-8
 terminating with NOT NULL DEFAULT
 NULL 3-52
 unused input fields 3-50
 Locale clause, LOAD DATA
 statement 3-29 to 3-32
 locales
 default 3-30
 list of supported C-1
 TMU input files 3-29
 UNLOAD operations 4-3
 use by TMU 1-4
 LOCK, TMU SET command 2-12

- locking
 - by TMU 2-12
 - during TMU backup/restore operations 7-4
 - wait behavior for TMU 2-12
- LTRIM keyword, TMU 3-63
- M**
 - M4DATE keyword, TMU 3-92
 - MAX aggregate operator, TMU 3-59
 - MAXROWS PER SEGMENT parameter
 - and duplicate records 3-45
 - effect on STAR indexes 6-2
 - MAXSPILLSIZE value 2-15
 - messages, TMU, locale of 3-31
 - Metaphor DBS 2 database, input from B-2
 - metavariables, in syntax diagrams xvii
 - MIN aggregate operator, TMU 3-59
 - MODIFY mode
 - ACCEPT/REJECT clause 3-76
 - loading data 3-25
 - MODIFY_ANY task authorization 8-4
 - moving a database 4-14
- N**
 - NLS_LOCALE keyword, TMU 3-29, 3-30
 - NO WAIT on locks, TMU 2-12
 - notation conventions xiv
 - NULL values
 - for input data, example 8-13
 - in external-format data 4-17, 5-8
 - in numeric columns 3-111
 - NULLIF keyword, TMU
 - GENERATE example 5-8
 - LOAD DATA statement 3-58
 - rb_cm example 8-13
 - UNLOAD example 4-17
- O**
 - offline-load operations
 - See also* segments
 - overview 1-4
 - syntax 3-72
 - Optimize clause
 - LOAD DATA statement 3-45 to 3-48
 - REORG statement 6-3
 - OPTIMIZE keyword, TMU 3-46, 6-3
 - Oracle database, input from B-1
 - order
 - input data, TMU 3-10
 - table order for loads 3-9
 - unloaded data 4-6
 - OVERRIDE REFCHECK clause, in REORG statement 6-4
- P**
 - packed decimal datetime
 - inputs 3-101 to 3-103
 - Parallel TMU, *See* PTMU and TMU
 - password
 - TMU command line 2-3
 - TMU control file 2-9
 - pipes
 - as TMU input 2-7
 - for TMU outputs 4-7
 - multiple TMU inputs 2-8
 - TMU GENERATE statement 5-3
 - POSITION keyword, TMU 3-57
 - pseudocolumn, TMU
 - field specification 3-50
 - with ACCEPT/REJECT 3-77
 - with concatenated fields 3-64
 - PTMU
 - effective use 2-22 to 2-24
 - exit status codes 2-4
 - parallel processing
 - parameters 2-17 to 2-18
 - syntax for rb_ptmu 2-3
 - TMU SERIAL MODE parameter 2-19
- R**
 - radix point, TMU
 - overriding locale 3-30
 - specifying 3-85

- rb_cm utility 8-1 to 8-16
 - examples 8-11 to 8-15
 - LOAD control file example 8-12, 8-15
 - necessary authorizations 8-4
 - overview 8-2
 - requirements for running 8-3
 - results, verifying 8-16
 - syntax 8-5
 - TMU control files for use with 8-8
 - UNLOAD control file example 8-12, 8-15
 - RB_CONFIG environment variable, TMU 2-5
 - RB_HOST environment variable, TMU 2-5
 - RB_NLS_LOCALE environment variable
 - TMU 1-4, 3-30
 - See also* NLS_LOCALE keyword
 - RB_PATH environment variable, TMU 2-3, 2-5
 - rb_ptmu file, location 2-2
 - See also* PTMU
 - rb_tmu file, location 2-2
 - See also* TMU
 - RBW_LOADINFO table
 - rb_cm results 8-16
 - retrieving data 3-8, 3-81
 - RBW_LOADINFO_LIMIT
 - parameter 2-21
 - Rdb database, input from B-2
 - REAL keyword, TMU 3-90
 - RECALCULATE RANGES clause, TMU
 - REORG statement 6-3
 - RECORDLEN keyword, TMU 3-24
 - redbrick directory, defined 2-2
 - redbrick user ID, defined 2-2
 - referential integrity
 - maintaining with TMU REORG 6-2
 - with AUTOROWGEN
 - described 3-10 to 3-16
 - syntax 3-34 to 3-41
 - Registry, Windows NT 2-5, 8-5, 8-6
 - REJECT keyword, TMU 3-76
 - REORG operations
 - overview 1-6
 - usage 6-1 to 6-5
 - REORG statement (TMU), syntax 6-3
 - REPLACE mode, loading data 3-25
 - RESET, TEMPSPACE parameters 2-15
 - restated daily totals, with Auto
 - Aggregate A-2
 - restore operations, TMU 7-2 to 7-13
 - See also* backup operations
 - FORCE restore 7-7
 - licensing option 7-2
 - locks during 7-4
 - overview 1-6
 - procedure 7-10
 - RESTORE statement syntax 7-7
 - restoring a segment 7-12
 - sample scenarios 7-10
 - RESTORE statement (TMU), syntax 7-7
 - restoring damaged segment with TMU 7-12
 - restricted datetime masks 3-101
 - RETAIN keyword, TMU 3-52
 - RI_DISCARDFILE keyword, TMU 3-35
 - RTRIM keyword, TMU 3-63
- S**
- scale, for fieldtype 3-83
 - search condition
 - WHERE clause in UNLOAD statement 4-8
 - wildcard characters 4-9
 - Segment clause, LOAD DATA
 - statement 3-72 to 3-74
 - segments
 - converting table to multiple segments 4-14
 - loading data into 3-72
 - offline-load
 - syntax 3-73
 - restoring a segment (TMU) 7-7, 7-12
 - unloading specific segment 4-6
 - selective column updates 3-53 to 3-54
 - selective unload
 - WHERE clause 4-8
 - wildcard character 4-9
 - separated-format records 3-27, 3-108
 - SEQUENCE fields, LOAD DATA
 - statement 3-68 to 3-69
 - serial loader, *See* TMU 2-19

- SET commands, TMU
 - INDEX TEMPSPACE parameters 2-14
 - LOCK 2-12
 - SET DATEFORMAT 2-20
 - TMU BUFFERS 2-13
 - TMU CONVERSION TASKS 2-17
 - TMU INDEX TASKS 2-17
 - TMU SERIAL MODE 2-19
 - usage 2-11
- SET options in copy operation 8-10
- simple fields, LOAD DATA
 - statement 3-57 to 3-61
- SMALLINT fieldtype, TMU 3-89
- space, working space for offline
 - loads 3-73
- standard error, redirecting from TMU 2-7
- standard input, to TMU 2-7
- standard label tapes
 - ANSI 3-106, 3-109
 - IBM 3-106
- standard output, from TMU 4-7
- START RECORD keyword, TMU 3-22
- STOP RECORD keyword, TMU 3-22
- SUBSTR keyword, LOAD DATA
 - statement 3-84
- SUBTRACT aggregate operator,
 - TMU 3-59
- support
 - documentation xx
 - technical xviii
- Support Solutions Warehouse xviii
- SYNCH statement
 - in rb_cm copy operation 8-10
 - syntax 3-104
 - usage 3-104
- syntax
 - BACKUP statement 7-5
 - GENERATE CREATE TABLE
 - statement 5-2
 - GENERATE LOAD DATA
 - statement 5-4
 - rb_cm 8-5
 - rb_ptmu 2-3
 - rb_tm 2-3
 - REORG statement 6-3
 - RESTORE statement 7-7
 - UNLOAD statement 4-5
 - syntax diagrams
 - conventions for xv
 - keywords in xvii
 - metavariables in xvii
 - syntax notation xiv
- T
- Table clause, LOAD DATA
 - statement 3-49 to 3-71
- Table Management Utility, *See* TMU
- table verification utility, *See* tblchk
- tables
 - locking by TMU 2-12
 - multiple segments, converting to 4-14
 - order for load operations 3-9
 - reorganizing 6-1 to 6-5
 - unloading data 4-1 to 4-16
- tab-separated data 3-27
- tapes
 - input data file formats 3-106 to 3-110
 - input data record
 - formats 3-106 to 3-110
 - tape devices 3-106
- TAR file, POSIX limit 4-7
- TAR tapes, with input data 3-106, 3-109
- tblchk 6-4, 7-13
- technical support xviii
- templates
 - GENERATE CREATE TABLE statement,
 - TMU 5-2
 - GENERATE LOAD DATA statement,
 - TMU 5-4
- third-party tools, loading with warehouse
 - data 4-14
- TIME fieldtype, TMU 3-91
- TIMESTAMP fieldtype, TMU 3-91
- timestamp option (-t), TMU 2-3
- TINYINT keyword, TMU 3-89
- TMU_BUFFERS parameter 2-13
- TMU_CONVERSION_TASKS
 - parameter 2-17
- TMU_INDEX_TASKS parameter 2-17
- TMU_SERIAL_MODE parameter 2-19
- TRIM keyword, TMU 3-63
- troubleshooting, general problems xx
- tuning parameters, TMU 2-13

U

- UNDO LOAD keyword, TMU 3-105
- UNLOAD format, TMU input data 3-106
- UNLOAD statement (TMU)
 - rb_cm example 8-12, 8-15
 - rb_cm, use with 8-8
 - syntax 4-5
 - WHERE clause 4-8
- unloading data 4-1 to 4-16
 - See also* GENERATE statements, TMU
 - data conversion for EXTERNAL 4-4
 - external format
 - described 4-3
 - example 4-13
 - procedure 4-12
 - formats, internal and external 4-3
 - generating CREATE TABLE statement 4-6, 4-16
 - generating LOAD DATA statement 4-6
 - internal format
 - described 4-3
 - example 4-11
 - procedure 4-10
 - overview 1-5
 - privileges required 4-2
 - row order 4-2, 4-6
 - selected rows 4-8, 4-15

- UPDATE mode, loading data 3-25
- UPGRADE statement (TMU), syntax 1-7
- upgrading database to new release
 - general 1-7
- USER statement (TMU) 1-4, 2-9 to 2-10
- username, database
 - TMU command line 2-3
 - TMU control file 2-9

W

- WAIT/NO WAIT on locks, TMU 2-12
- warehouse, administrator, defined ix
- wildcard characters
 - LOAD DATA statement 3-77
 - UNLOAD statement 4-9
- WORKING_SPACE keyword, TMU 3-73
- World Wide Web address, for Red Brick Systems xviii

Z

- zoned decimal datetime
 - inputs 3-101 to 3-103
- zoned decimal fieldtypes 3-87



**USA SALES
OFFICES**

1040 Crowne Point Parkway, Suite 250, Atlanta, GA 30338 +1 770 804 2440
2215 York Road, Suite 409, Oak Brook, IL 60521 +1 630 472 8500
1120 Avenue of the Americas, 4th Floor, New York, NY 10036 +1 212 626 6815
5314 Arapaho Road, Dallas, TX 75248 +1 972 702 1750
2010 Corporate Ridge, 7th Floor, McLean, VA 22102 +1 703 883 9310

**UK SALES
OFFICE**

Red Brick Systems UK Ltd., 45 Berkeley Square, Mayfair, London W1A 1EB
United Kingdom +44 171 290 8373

**AUSTRALASIA
HEADQUARTERS**

Red Brick Systems Australasia Pty. Ltd., Level 20, 99 Walker Street,
North Sydney, NSW 2060 Australia +61 02 9911 7744

**JAPAN
HEADQUARTERS**

Red Brick Japan Co. Ltd., Level 16 Shiroyama Hills, 4-3-1 Toranomom,
Minato-ku, Tokyo 105 Japan +81 3 5403 4638