

Oracle7[™] Server Utilities

Release 7.3

February 1996

Part No. A32541-1

ORACLE[®]

Oracle7™ Server Utilities, Release 7.3

Part No. A32541-1

Copyright © Oracle Corporation 1995, 1996.

All rights reserved. Printed in the U.S.A.

Primary Author: Jason Durbin

Contributors: Alan Brumm, Sanford Dreskin, Joyce Fee, Irene Hu, Rita Moran,
Hari Sankar, Raghu Viswanathan

This software was not developed for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It is the customer's responsibility to take all appropriate measures to ensure the safe use of such applications if the programs are used for such purposes.

This software/documentation contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited.

If this software/documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

Restricted Rights Legend Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

If this software/documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with "Restricted Rights", as defined in FAR 52.227-14, Rights in Data - General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error-free.

Oracle, Oracle Parallel Server, SQL*DBA, SQL*Loader, SQL*Net, and SQL*Plus are registered trademarks of Oracle Corporation. Oracle7, Trusted Oracle7, Trusted Oracle, Oracle Server Manager, Oracle Network Manager, Oracle Names, and PL/SQL are trademarks of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.



Preface

This manual describes how to use the following Oracle7 Server utilities for data transfer, maintenance, and database administration.

Export/Import	Export and Import are two complementary programs that constitute a single utility. Export writes data from an Oracle database into transportable files. Import reads data from those files back into an Oracle database. These utilities help you back up data, upgrade to new releases of the Oracle Server, and move data between Oracle databases.
SQL*Loader	The SQL*Loader utility loads data from external files into Oracle database tables. SQL*Loader processes a wide variety of input file formats and gives you control over how records are loaded.
NLS Utilities	The NLS Configuration Utility allows you to configure your boot files so that only the NLS objects that you require will be loaded.
Offline Database Verification (DB_VERIFY)	DB_VERIFY is an external command-line utility that performs a physical data structure integrity check on an offline database. It can be used against backup files and online files (or pieces of files). You use DB_VERIFY primarily when you need to insure that a backup database (or datafile) is valid before it is restored or as a diagnostic aid when you

have encountered data corruption problems. Because DB_VERIFY can be run against an offline database, integrity checks are significantly faster.

This manual describes the basic concepts behind each utility and provides examples to show how the utilities are used.

Some of the information this manual provides must be supplemented for the high-security version of the Oracle7 Server, Trusted Oracle7 Server. Such information is marked with references to the Trusted Oracle7 Server documentation.

Audience

This manual is written for Oracle users who must perform the following tasks:

- archive data, back up an Oracle database, or move data between Oracle databases using the Export/Import utilities
- load data into Oracle tables from operating system files using SQL*Loader

To use this manual, you need a working knowledge of SQL and Oracle7 Server fundamentals, information that is contained in Chapter 1, “Introduction to the Oracle Server”, in the *Oracle7 Server Concepts* manual. In addition, SQL*Loader requires that you know how to use your operating system’s file management facilities.

Note: This manual does not contain installation instructions, since they vary by operating system. Directions for installing the utilities can be found in your operating-system-specific Oracle7 documentation.

How This Book Is Organized

This manual is divided into four parts:

Part I: Export/Import Chapter 1: Export

This chapter describes how to use Export to write data from an Oracle database into transportable files. It discusses guidelines, export modes, interactive and command-line methods, parameter specifications, and incremental exports. It also provides several examples of Export sessions.

Chapter 2: Import

This chapter shows you how to use Import to read data from Export files into an Oracle database. It discusses guidelines, interactive and command-line methods, parameter specifications, and incremental imports. It also provides several examples of Import sessions.

Part II: SQL*Loader Chapter 3: SQL*Loader Concepts

This chapter introduces SQL*Loader and describes its features. It also introduces data loading concepts. It discusses input to SQL*Loader, database preparation, and output from SQL*Loader.

Chapter 4: SQL*Loader Examples

This chapter presents case studies that illustrate some of the features of SQL*Loader. It demonstrates the loading of variable-length data, fixed-format records, a free-format file, multiple physical records as one logical record, multiple tables, and direct file loads.

Chapter 5: SQL*Loader Control File Reference

This chapter describes the data definition language (DDL) used by SQL*Loader to map data to Oracle format. It discusses creating the control file to hold DDL source, using the LOAD DATA statement, specifying data files, specifying tables and columns, and specifying the location of data.

Chapter 6: SQL*Loader Command-Line Reference

This chapter describes the command-line syntax used by SQL*Loader. It discusses the SQLLOAD command, command-line arguments, suppressing SQL*Loader messages, and sizing the bind array.

Chapter 7: SQL*Loader Log File Reference

This chapter describes the information contained in the log file.

Chapter 8: SQL*Loader Conventional and Direct Path Loads

This chapter describes the conventional path load method and the direct path load method— a high performance option that significantly reduces the time required to load large quantities of data.

Part III: NLS Utilities Chapter 9: National Language Support Utilities

This chapter describes three utilities: NLS Data Installation Utility, NLS Configuration Utility, and NLS Calendar Utility.

Part IV: Offline Chapter 10: Offline Database Verification Utility
Verification Utility

This chapter describes how to use the offline database verification utility, DB_VERIFY.

Appendix A: Changes in the Oracle7 Server Utilities

This appendix describes changes (by release number) to the Oracle7 Server utilities.

Appendix B: Reserved Words

This appendix lists words reserved for use by the Oracle7 Server utilities.

Appendix C: Notes for DB2/DXT Users

This appendix describes differences between the data definition language syntax of SQL*Loader and DB2 Load Utility control files. It discusses SQL*Loader extensions to the DB2 Load Utility, the DB2 RESUME option, options included for compatibility, and SQL*Loader restrictions.

Conventions Used in this Manual

This section tells you how to read syntax diagrams, examples, and interface descriptions.

Interface Elements The user-interface descriptions in this manual use the following elements:

Menu Options Operations you can perform by selecting from a menu, submenu, or dialog box are printed in bold.

Menu and
Keyboard
Operations

Operations you can perform from the keyboard are specified in square brackets, with initial capital letters.

Operations are described, rather than keystrokes, since the assigned keys differ based on operating system requirements, and multiple keystrokes can be assigned to the same operation. In addition, the key assignments can be changed at the administrator's discretion.

Key Names

On those occasions when it is necessary to name a particular key, the name is specified in normal text with initial capital letter. For example, "the Return key" or "the Newline key".

Syntax Diagrams

The syntax diagrams in this manual show the complete syntax for the Oracle7 Server utilities control file data definition language (DDL). Syntax diagrams consist of these items:

KEYWORDS

Keywords are words that have special meanings. Keywords are shown in uppercase. When you specify a keyword in a control file, it can be uppercase or lowercase, but the syntax must be exactly as shown in the syntax diagram.

Variables

Keywords often require that a variable be replaced with a text string that specifies a database ID, a table or filename, a hex string etc.

For example, to write an INTO TABLE statement, you must specify the name of the table you want to load, such as EMP, in place of the table variable in the syntax diagram.



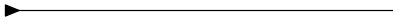

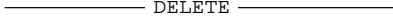

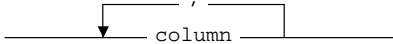
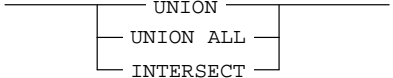
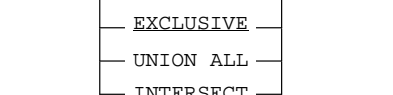
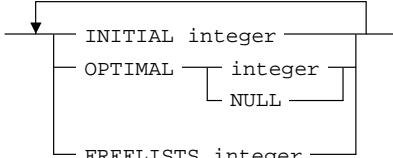
The following list shows common parameters that appear in the syntax diagrams, along with some examples:

Variable	Description	Example
<i>tablename</i>	The substitution value must be the name of a table.	EMP
<i>filename</i>	The substitution value must be the name of a file on your system. Single quotes are needed if the filename contains spaces or special characters.	'\mydir\myfile'
<i>columnname</i>	The substitution value must be the name of a column in a table	SAL
<i>char</i>	The substitution value must be a single character from your computer's character set.	T
<i>'char_str'</i>	The substitution value must be a character literal in single quotes.	'Employee records'
<i>"char_str"</i>	The substitution value must be a character literal in double quotes.	"Employee records"
<i>hex_char</i>	The substitution value must be a hexadecimal character.	F
<i>hex_str</i>	The substitution value must be a hexadecimal string.	4A1F
<i>n length</i>	The substitution value must be an integer.	100
<i>start end</i>	The substitution value must be a column position in a logical or physical record.	6

Table 1 Parameter Descriptions

Parameters Keywords often have optional and/or required parameters which provide additional information that will modify the functionality of the keyword. Parameters are shown in lowercase.

Syntax Diagrams Syntax diagrams use lines and arrows to show syntactic structure. This list shows combinations of lines and arrows and their meanings within railroad diagrams:

	The beginning of a diagram.
	The diagram is continued on the next line.
	The diagram is continued from the previous line.
	The end of a diagram.
	A required item (parameter or keyword). You must use it.
	An optional item. You can use the item or omit it.
	You can optionally repeat the item multiple times. Consecutive items must be separated by a comma.
	Mandatory items. You must use one of these items.
	You can optionally use only one of the items. If there is a default item, it is underlined.
	A list of specific items. Each item can only appear once, unless otherwise specified. The items can appear in any order.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. At the back of this manual is a Reader's Comment Form which we encourage you to use to tell us what you like and dislike about this manual or other Oracle manuals. If the form has been used or you would like to contact us, please contact us at the following address:

Oracle7 Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood Shores, CA 94065
Fax: (415) 506-7200



Contents

PART I

EXPORT/IMPORT

Chapter 1

- Export 1 – 1**
- Export Basics 1 – 2
 - File Handling 1 – 3
 - Access Privileges 1 – 3
 - Displaying the Contents of an Export File 1 – 3
- Warning, Error, and Completion Messages 1 – 4
 - Log File 1 – 4
 - Warning Messages 1 – 4
 - Fatal Error Messages 1 – 4
 - Completion Message 1 – 5
- Export Modes 1 – 5
- Exporting Sequences 1 – 7
 - Sequence Numbers Skipped 1 – 7
- Exporting LONGs 1 – 7
- Trusted Oracle7 Server and Export 1 – 8
- Network Considerations 1 – 8
 - Transporting Export Files Across a Network 1 – 8
 - Exporting/Importing with SQL*Net 1 – 8
- NLS Considerations 1 – 8
 - Character Set Translation 1 – 8
 - Export/Import and Single-Byte Character Sets 1 – 9
 - Export/Import and Multi-Byte Character Sets 1 – 9
- Using Export 1 – 10
 - Before Using Export 1 – 10

Invoking Export	1 – 10
The Parameter File	1 – 11
Table Name Restrictions	1 – 12
Export Parameters	1 – 12
Parameter Interactions	1 – 17
Export's Interactive Mode	1 – 17
Export's Interactive Mode Prompts	1 – 18
Example Export Sessions	1 – 20
Example Export Session in Full Database Mode	1 – 20
Example Export Session in User Mode	1 – 21
Example Export Sessions in Table Mode	1 – 22
Incremental, Cumulative, and Complete Exports	1 – 24
Restrictions	1 – 24
Base Backups	1 – 24
Incremental Exports	1 – 24
Cumulative Exports	1 – 25
Complete Exports	1 – 26
Benefits	1 – 27
A Scenario	1 – 27
Which Data Is Exported?	1 – 28
Command Syntax	1 – 29
Example Incremental Export Session	1 – 30
System Tables	1 – 31
Export Utility: Direct Path Export	1 – 32
Conventional Vs. Direct Export Methods	1 – 32
Preparing the Database for Direct Path Export	1 – 33
Invoking a Direct Path Export	1 – 34
Direct Export Logging Information	1 – 35
Character Set Conversion	1 – 35
Performance Issues	1 – 36
Restrictions	1 – 37
Compatibility & Migration	1 – 37
Using Different Versions of Export	1 – 37
Using a Previous Version of Export	1 – 37
Using a Higher Version Export	1 – 38
Change in Export File Format	1 – 38
Creating Oracle Server Version 6 Export Files from Oracle7	1 – 39
Excluded Objects	1 – 39
Datatype Conversion	1 – 40
Database Link Names Truncated	1 – 40
VARCHAR Errors	1 – 40
LONG Data Errors	1 – 40
System Audit Options Dropped	1 – 41

Import	2 – 1
Import Basics	2 – 2
Table Objects: Order of Import	2 – 2
Storage Parameters	2 – 3
Character Set Translation	2 – 3
Access Privileges	2 – 4
Rollback Segments	2 – 4
Compatibility	2 – 4
Trusted Oracle7 Server	2 – 4
Warning, Error, and Completion Messages	2 – 5
Error Handling	2 – 5
Row Errors	2 – 5
Object Errors	2 – 6
Fatal Errors	2 – 7
Privileges Required to Use Import	2 – 8
Importing Objects into Your Own Schema	2 – 8
Importing Grants	2 – 9
Importing Objects into Other Schemas	2 – 9
Importing System Objects	2 – 9
User Privileges	2 – 9
Import and Stored Procedures, Functions, and Packages	2 – 10
Import and Snapshots	2 – 10
Master Table	2 – 10
Master Table Trigger	2 – 10
Snapshot Log	2 – 11
Snapshots	2 – 11
Character Set Conversion	2 – 12
Import Modes	2 – 13
Using Import	2 – 15
Before Using Import	2 – 15
Invoking Import	2 – 16
Getting Online Help	2 – 16
The Parameter File	2 – 16
Table Name Restrictions	2 – 17
Import Parameters	2 – 18
Import's Interactive Mode	2 – 25
Import's Interactive Mode Prompts	2 – 25
Example Import Sessions	2 – 27
Example Import of Selected Tables for a Specific User	2 – 27
Example Import of Tables Exported by Another User	2 – 28
Example Import of Tables from One User to Another	2 – 29
Importing Incremental, Cumulative and Complete Export Files	2 – 29
Command Syntax	2 – 30

Restoring a Set of Objects	2 – 30
Importing Into Existing Tables	2 – 31
Manually Creating Tables before Importing Data	2 – 31
Disabling Referential Constraints	2 – 31
Manually Ordering the Import	2 – 32
Generating Statistics on Imported Data	2 – 32
Importing LONGs	2 – 33
Network Considerations	2 – 33
Transporting Export Files Across a Network	2 – 33
Exporting/Importing with SQL*Net	2 – 33
Dropping a Tablespace	2 – 34
Reorganizing Tablespaces	2 – 34
Overriding Storage Parameters	2 – 35
Reducing Database Fragmentation	2 – 35
Export/Import Read-Only Tablespaces	2 – 35
NLS Considerations	2 – 36
Export/Import and Single-Byte Character Sets	2 – 36
Export/Import and Multi-Byte Character Sets	2 – 36
Using Oracle Version 6 Export Files	2 – 36
CHAR columns	2 – 36
LONG columns	2 – 36
Syntax of Integrity Constraints	2 – 37
Status of Integrity Constraints	2 – 37
Length of DEFAULT Column Values	2 – 37
Using Oracle Version 5 Export Files	2 – 37

PART II

SQL*LOADER

Chapter 3

SQL*Loader Concepts	3 – 1
SQL*Loader Basics	3 – 2
Direct Path Load vs. Conventional Path Load Method	3 – 3
Conventional Path	3 – 3
Direct Path	3 – 3
Mapping the Data to Oracle Format	3 – 3
The Concept of Mapping Data	3 – 3
The Data Definition Language (DDL)	3 – 4
DDL Definitions	3 – 4
The Control File	3 – 5
The Data	3 – 6
Logging Information	3 – 8

Prerequisites	3 – 9
Privileges Required	3 – 9
Discarded and Rejected Records	3 – 10
The Bad File	3 – 11
SQL*Loader Discards	3 – 11
Data Conversion and Datatype Specification	3 – 12

Chapter 4

SQL*Loader Case Studies	4 – 1
Case Study Files	4 – 2
Tables Used in the Case Studies	4 – 3
Contents of Table EMP	4 – 3
Contents of Table DEPT	4 – 3
References	4 – 3
Notes	4 – 3
Running the Case Study SQL Scripts	4 – 3
Case 1: Loading Variable–Length Data	4 – 4
The Control File	4 – 4
Invoking SQL*Loader	4 – 5
The Log File	4 – 5
Case 2: Loading Fixed–Format Records	4 – 6
The Control File	4 – 6
Datafile	4 – 7
Invoking SQL*Loader	4 – 7
The Log File	4 – 7
Case 3: Loading a Delimited, Free–Format File	4 – 8
The Control File	4 – 9
Invoking SQL*Loader	4 – 10
The Log File	4 – 10
Case 4: Loading Combined Physical Records	4 – 11
The Control File	4 – 12
The Data File	4 – 12
Invoking SQL*Loader	4 – 13
The Log File	4 – 13
The Bad File	4 – 14
Case 5: Loading Data into Multiple Tables	4 – 14
The Control File	4 – 14
The Data File	4 – 15
Invoking SQL*Loader	4 – 16
The Log File	4 – 16
The Loaded Tables	4 – 19

Case 6: Loading using the Direct Path Load Method	4 – 20
The Control File	4 – 20
Invoking SQL*Loader	4 – 21
The Log File	4 – 21
Case 7: Extracting Data from a Formatted Report	4 – 22
The Data File	4 – 22
Insert Trigger	4 – 23
The Control File	4 – 24
Invoking SQL*Loader	4 – 25
The Log File	4 – 25
Dropping the Insert Trigger and the Global-Variable Package	4 – 27

Chapter 5

SQL*Loader Control File Reference	5 – 1
Data Definition Language (DDL) Syntax	5 – 4
High-Level Syntax Diagrams	5 – 6
Expanded Clauses and Their Functionality	5 – 9
Position Specification	5 – 9
Field Condition	5 – 9
Column Name	5 – 10
Datatype Specification	5 – 10
Precision vs. Length	5 – 11
Date Mask	5 – 11
Delimiter Specification	5 – 11
Comments	5 – 11
Specifying Command-Line Parameters in the Control File	5 – 12
OPTIONS	5 – 12
Specifying RECOVERABLE and UNRECOVERABLE	5 – 12
Specifying Filenames and Database Objects	5 – 13
Database Object Names within Double Quotation Marks	5 – 13
SQL String within Double Quotation Marks	5 – 13
Filenames within Single Quotation Marks	5 – 13
Quotation Marks in Quoted Strings	5 – 13
Backslash Escape Character	5 – 14
Using a Backslash in Filenames	5 – 14
Including Data in the Control File with BEGINDATA	5 – 15
Identifying Datafiles	5 – 16
Naming the File	5 – 16
Specifying Multiple Datafiles	5 – 17
Examples of How to Specify a Datafile	5 – 18
Specifying READBUFFERS	5 – 18

Specifying Datafile Format and Buffering	5 – 18
File Processing Example	5 – 18
Specifying the Bad File	5 – 19
Examples of How to Specify a Bad File	5 – 20
Rejected Records	5 – 20
Integrity Constraints	5 – 21
Specifying the Discard File	5 – 21
Using a Control-File Definition	5 – 22
Examples of How to Specify a Discard File	5 – 22
Discarded Records	5 – 23
Limiting the Number of Discards	5 – 23
Handling Different Character Encoding Schemes	5 – 24
Multi-Byte (Asian) Character Sets	5 – 24
Input Character Conversion	5 – 24
Loading into Empty and Non-Empty Tables	5 – 25
How Non-Empty Tables are Affected	5 – 25
INSERT	5 – 26
APPEND	5 – 26
REPLACE	5 – 26
TRUNCATE	5 – 27
Specifying One Method for All Tables	5 – 27
Continuing an Interrupted Load	5 – 27
State of Tables and Indexes	5 – 27
Using the Log File	5 – 27
Dropping Indexes	5 – 28
Continuing Single Table Loads	5 – 28
Continuing Multiple Table Conventional Loads	5 – 28
Continuing Multiple Table Direct Loads	5 – 28
Assembling Logical Records from Physical Records	5 – 29
Examples of How to Specify CONTINUEIF	5 – 32
Loading Logical Records into Tables	5 – 33
Specifying Table Names	5 – 33
Table-Specific Loading Method	5 – 33
Table-Specific OPTIONS keyword	5 – 34
Choosing which Rows to Load	5 – 34
Specifying Default Data Delimiters	5 – 35
Handling Short Records with Missing Data	5 – 35
Index Options	5 – 36
SORTED INDEXES Option	5 – 36
SINGLEROW Option	5 – 36
Specifying Field Conditions	5 – 37
Comparing Fields to BLANKS	5 – 38
Comparing Fields to Literals	5 – 39

Specifying Columns and Fields	5 – 39
Specifying the Datatype of a Data Field	5 – 39
Specifying the Position of a Data Field	5 – 40
Using POSITION with Data that Contains TABs	5 – 41
Using POSITION with Multiple Table Loads	5 – 41
Using Multiple INTO TABLE Statements	5 – 42
Extracting Multiple Logical Records	5 – 43
Distinguishing Different Input Record Formats	5 – 44
Loading Data into Multiple Tables	5 – 45
Summary	5 – 45
Generating Data	5 – 45
Loading Data Without Files	5 – 45
Setting a Column to a Constant Value	5 – 46
Setting a Column to the Datafile Record Number	5 – 46
Setting a Column to the Current Date	5 – 46
Setting a Column to a Unique Sequence Number	5 – 47
Generating Sequence Numbers for Multiple Tables	5 – 48
Specifying Datatypes	5 – 49
Datatype Conversions	5 – 49
Native Datatypes	5 – 50
Character Datatypes	5 – 56
Numeric External Datatypes	5 – 58
Specifying Delimiters	5 – 58
Conflicting Character Datatype Field Lengths	5 – 61
Loading Data Across Different Operating Systems	5 – 62
Trusted Oracle7	5 – 63
Determining the Size of the Bind Array	5 – 63
Minimum Requirements	5 – 63
Performance Implications	5 – 63
Specifying Number of Rows vs. Size of Bind Array	5 – 64
Calculations	5 – 64
Minimizing Memory Requirements for the Bind Array	5 – 67
Multiple INTO TABLE Statements	5 – 67
Generated Data	5 – 67
Setting a Column to Null or Zero	5 – 67
DEFAULTIF Clause	5 – 67
NULLIF Keyword	5 – 68
Null Columns at the End of a Record	5 – 68
Loading All-Blank Fields	5 – 69
Trimming of Blanks and Tabs	5 – 69
Datatypes	5 – 69
Field Length Specifications	5 – 70
Record Formats	5 – 70

	Relative Positioning of Fields	5 – 71
	Leading Whitespace	5 – 71
	Trailing Whitespace	5 – 72
	Enclosed Fields	5 – 72
	Trimming Whitespace: Summary	5 – 73
	Preserving Whitespace	5 – 74
	PRESERVE BLANKS Keyword	5 – 74
	Applying SQL Operators to Fields	5 – 75
	Referencing Fields	5 – 75
	Common Uses	5 – 75
	Combinations of Operators	5 – 76
	Use with Date Mask	5 – 76
	Interpreting Formatted Fields	5 – 76
Chapter 6	SQL*Loader Command-Line Reference	6 – 1
	The SQL*Loader Command Line	6 – 2
	Using Command-Line Keywords	6 – 3
	Specifying Keywords in the Control File	6 – 3
	Command-Line Keywords	6 – 3
Chapter 7	SQL*Loader Log File Reference	7 – 1
	Header Information	7 – 2
	Global Information	7 – 2
	Table Information	7 – 3
	Datafile Information	7 – 4
	Table Load Information	7 – 4
	Summary Statistics	7 – 5
Chapter 8	SQL*Loader Conventional and Direct Path Loads	8 – 1
	Data Loading Methods	8 – 2
	Conventional Path Loads	8 – 2
	Direct Path Loads	8 – 3
	Using Direct Path Load	8 – 7
	Setting Up for Direct Path Loads	8 – 7
	Specifying a Direct Path Load	8 – 7
	Building Indexes	8 – 7
	The SINGLEROW Option	8 – 8
	Indexes Left in Direct Load State	8 – 9
	Data Saves	8 – 9
	Recovery	8 – 10

Loading LONG Data Fields	8 – 11
Maximizing Performance of Direct Path Loads	8 – 12
Allocating I/O Buffers	8 – 13
Pre-allocating Storage for Faster Loading	8 – 13
Pre-sorting Data for Faster Indexing	8 – 13
Infrequent Data Saves	8 – 15
Minimizing Use of the Redo Log	8 – 15
Specifying UNRECOVERABLE	8 – 15
Dropping Indexes	8 – 16
Direct Loads, Integrity Constraints, and Triggers	8 – 16
Integrity Constraints	8 – 17
Database Insert Triggers	8 – 18
Permanently Disabled Triggers & Constraints	8 – 20
Alternative: Partitioned Load	8 – 21
Parallel Data Loading	8 – 21
Restrictions	8 – 21
Initiating Multiple SQL*Loader Sessions	8 – 21
Options Keyword	8 – 22
Specifying Temporary Segments	8 – 23
Enabling Constraints After A Parallel Direct Path Load	8 – 23

PART III

NLS UTILITIES

Chapter 9

National Language Support Utilities	9 – 1
NLS Data Installation Utility	9 – 2
Overview	9 – 2
Syntax	9 – 2
Return Codes	9 – 3
Usage	9 – 3
NLS Data Object Files	9 – 4
NLS Configuration Utility	9 – 6
Overview	9 – 6
Syntax	9 – 7
Menus	9 – 7
NLS Calendar Utility	9 – 9
Overview	9 – 9
Syntax	9 – 9
Usage	9 – 10

PART IV

OFFLINE VERIFICATION UTILITY

Chapter 10

Offline Database Verification Utility	10 – 1
DB_VERIFY	10 – 2
Restrictions	10 – 2
Syntax	10 – 2
Server Manager	10 – 3
Sample DB_VERIFY Output	10 – 3

Appendix A

Summary of Changes	A – 1
Export/Import Changes	A – 2
Release 7.2 Changes	A – 2
Release 7.1 Changes	A – 2
Changes in Export for Release 7.0	A – 2
Changes in Import for Release 7.0	A – 3
Incremental Import and Export	A – 5
SQL*Loader Changes	A – 5
Oracle7 Server Release 7.1	A – 5
Oracle7	A – 6
Release 1.1	A – 7
Release 1.0.26	A – 8
Release 1.0.22	A – 8

Appendix B

Reserved Words	B – 1
SQL*Loader Reserved Words	B – 2

Appendix C

Notes for DB2/DXT Users	C – 1
SQL*Loader Extensions to the DB2 Load Utility	C – 2
Using the DB2 RESUME Option	C – 3
Inclusions for Compatibility	C – 4
LOG Statement	C – 4
WORKDDN Statement	C – 4
SORTDEVT and SORTNUM Statements	C – 4
DISCARD Specification	C – 4
Restrictions	C – 5
FORMAT Statement	C – 5
PART Statement	C – 5
SQL/DS Option	C – 5
DBCS Graphic Strings	C – 5
SQL*Loader Syntax with DB2-compatible Statements	C – 6

Export/Import

Part I describes the Export and Import utilities. These utilities are complementary. Export writes data from an Oracle database into a transportable operating system file. Import reads data from this file back into an Oracle database.

Export and Import are used primarily for the following tasks:

- data archival
- upgrading to new releases of Oracle
- backing up Oracle databases
- moving data between Oracle databases

Export and Import allow you to accomplish the following tasks:

- store Oracle data in operating system files independent of any database
- store definitions of database objects (such as tables, clusters, and indexes) with or without the data
- store inactive or temporary data
- back up only tables whose data has changed since the last export, using either an incremental or a cumulative export
- restore tables that were accidentally dropped, provided they were exported recently
- restore a database by importing from incremental or cumulative exports

- selectively back up parts of your database in a way that requires less storage space than a system backup
- move data from an older to a newer version of Oracle or vice versa
- move data between Oracle databases
- move data between different hardware and operating-system environments
- move data from one owner to another
- move data from one tablespace or schema to another
- save space or reduce fragmentation in your database

Export

This chapter describes how to use the Export utility to write data from an Oracle database into an operating system file in binary format. This file can be stored independently of the database or read into another Oracle database, using the Import utility (described in Chapter 2).

This chapter covers the following topics:

- Export Basics
- Error Handling
- Export Modes
- Interactive and Command-Line Methods
- Export Parameters
- Incremental and Cumulative Exports

It also provides several example export sessions that show how the export utility works in certain situations.

Export Basics

The concept behind Export’s basic function is quite simple: extract the object definitions and table data from an Oracle database and store them in an Oracle–binary format export file that is located typically on tape or on disk. Export files can be used to transfer data between databases that are on machines not connected via a network or as backups in addition to normal backup procedures.

When Export is run against an Oracle database, objects, such as tables, are extracted followed by their related objects, like indexes, comments, and grants, if any, then written to the Export file. See Figure 1 – 1.

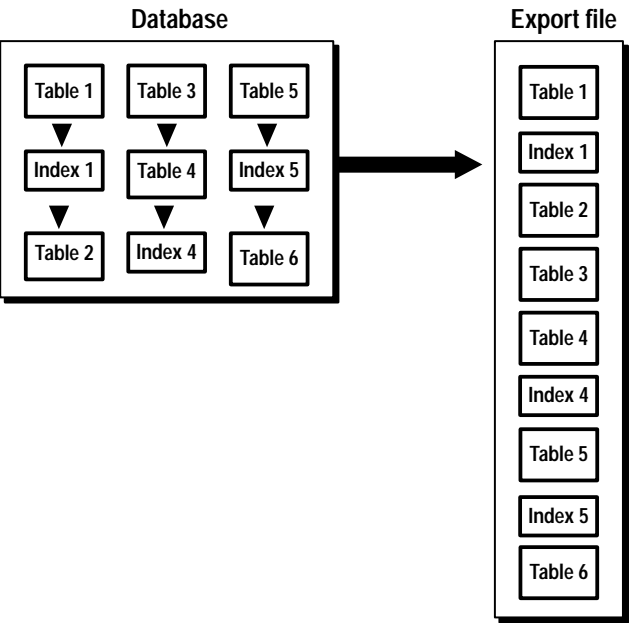


Figure 1 – 1 Exporting a Database

Because Export files are saved in Oracle–binary format, export files cannot be read by utilities other than Import. Similarly, Import can read files written by Export, but cannot read files in other formats. If you need to load data from ASCII fixed–format or delimited files, see Part II of this manual for information about SQL*Loader.

File Handling

Export writes export files using the character set specified for the user session; for example, 7-bit ASCII or IBM Code Page 500 (EBCDIC).

Note: Import automatically maps the data to the character set of its host system or to the character set specified for the user session if it is different from that used in the export file.



OSDoc

Additional Information: For the Trusted Oracle7 Server, export files are labeled by the operating system.

See your platform-specific Trusted Oracle7 Server documentation for details about file naming procedures.

Access Privileges

To use Export you must have the CREATE SESSION privilege on an Oracle database. To export tables owned by another user, you must have the EXP_FULL_DATABASE role enabled. This role is granted to all DBAs.

If you do not have the system privileges contained in the EXP_FULL_DATABASE role, you cannot export objects contained in another user's schema.

For example, you cannot export a table in another user's schema, even if you have created a synonym for it.

Displaying the Contents of an Export File

Since files created by Export are stored in Oracle-binary format, they cannot be read by any Oracle7 Server utility or product other than Import.

You can, however, display the contents of an export file by using the Import SHOW option described in Chapter 2. In addition, export files can be read *only* by Import. You cannot use them to transfer data to non-Oracle systems.

Warning, Error, and Completion Messages

Export attempts to save as much of the database as possible—even when part of it has become corrupted, but errors can occur. This section discusses how Export handles those errors.

Log File

Oracle Corporation recommends that you capture all of Export's messages in a log file either by using the LOG parameter (See page 1 – 15 for the log file specification) or, for those systems that permit it, by redirecting Export's output to a file. This file will contain detailed information about successful loads, and any errors that may occur.



OSDoc

Additional Information: Refer to your Oracle operating system-specific documentation for information on redirecting output.

Warning Messages

Export will not terminate after non-fatal errors.

For example, If an error occurs while exporting a table, Export displays (or logs) an error message, skips to the next table, and continues processing. These non-fatal errors are known as *warnings*.

Export will issue a warning whenever an invalid object is encountered. That is, if a non-existent table was specified as part of a table-mode export, then all other tables would be exported, after which the export would issue a warning and terminate successfully, as shown in the following listing:

```
exp scott/tiger tables=xxx,emp
...
About to export specified tables ...
EXP-00011: SCOTT.XXX does not exist
EXP-00222:
System error message 2
. exporting table          EMP          10 rows
exported
```

```
Export terminated successfully with warnings.
```

Fatal Error Messages

Some errors are *fatal*, and will terminate the Export session. These errors typically occur because of an internal problem or because a resource, such as memory, is not available or has been exhausted.

For example, you see the following message if you attempt to run Export without having created the necessary Export views:

```
EXP-00024: Export views not installed, please notify your DBA
```



Additional Information: Messages are documented in the *Oracle7 Server Messages* manual and in your Oracle operating system-specific documentation.

Completion Message

When Export completes without errors, the message “Export terminated successfully without warnings” is displayed or logged. If one or more non-fatal errors occurred, but Export was able to continue to completion, then the message “Export terminated successfully with warnings” is displayed or logged. If a fatal error occurs, Export terminates immediately with the message “Export terminated unsuccessfully”.

Export Modes

The database objects that are exported depend on the mode you choose. All users have at least two choices of export mode; a user with the EXP_FULL_DATABASE role (a *privileged user*) has three choices:

Table	Exports specified tables in the user’s schema, rather than all tables. A privileged user can qualify the tables by specifying the schema containing them. The default is to export all tables belonging to the user doing the export. Exports in table mode do not include cluster definitions. As a result, the data is imported into unclustered tables. Thus, you can use Export to uncluster tables when there is not enough free space in your database.
User	Exports all objects in a user’s schema (such as tables, data, grants, and indexes). A privileged user exporting in user mode can export all objects in the schemas of a specified set of users.
Full Database	Only users with the EXP_FULL_DATABASE role can export in this mode. All objects in the database are exported, except those in the schema of SYS. (Objects owned by SYS are generated during database creation for internal and administrative use. Because they are not exported, you should not create a user table or any other object under SYS.)

See “Export Parameters” for more information.

Table 1 – 1 shows the objects that are exported in each mode and the order in which they are exported:

Table Mode	User Mode	Full Database Mode
For each table in the TABLES list:	For each user in the Owner's list:	All database objects except for those owned by SYS:
table definitions	snapshots	tablespace definitions
table data	snapshot logs	profiles
table constraints	job queues	user definitions
owner's table grants (1)	refresh groups and children	roles
owner's table indexes (2)	database links	system privilege grants
analyze tables	sequence numbers	role grants
column comments	cluster definitions	default roles
audit	For each table that the user owns:	tablespace quotas
table referential constraints	table definitions	resource costs
table triggers	table data	rollback segment definitions
	table constraints	database links
	owner table grants	sequence numbers
	owner table indexes (3)	all snapshots
	analyze table	all snapshot logs
	column comments	all job queues
	audit	all refresh groups and children
	private synonyms	all cluster definitions
	user views	table definitions
	user stored procedures	table data
	user stored functions	table constraints
	user stored packages	table grants
	analyze cluster	table indexes
	referential constraints	analyze tables
	referential constraints	column comments
	triggers	audit
		referential integrity constraints
		all synonyms
		all views
		all stored procedures,
		all stored packages
		all stored functions

Table Mode	User Mode	Full Database Mode
		all triggers analyze cluster default and system auditing
Notes: 1. Owner's grants for the tables are exported in table mode. 2. Owner's indexes on the specified tables are exported in table mode. 3. Only indexes on the user's tables are exported.		

Table 1 – 1 Exported Objects

Exporting Sequences

If transactions continue to access sequence numbers during an export, sequence numbers can be skipped. This section describes the circumstances under which it can happen. The best way to ensure that sequence numbers are not skipped is to ensure that the sequences are not accessed during the export.

Sequence Numbers Skipped

Sequence numbers can be skipped only when cached sequence numbers are in use. When a cache of sequence numbers has been allocated, they are available for use in the current database. The exported value is the *next* sequence number (after the cached values). Sequence numbers that are cached, but unused, are lost when the sequence is imported.

Exporting LONGs

On import, LONGs require contiguous memory. Therefore, it is not always possible to export LONG columns from one operating system and import them on another system. Even on the same system, memory limitations may make it impossible to import very large LONG columns that were successfully exported on that system.



Warning: LONG columns can be up to 2 gigabytes in length. Because they can be exported in sections, Oracle LONGs are always exportable.

Trusted Oracle7 Server and Export

There are additional steps and considerations when you are exporting data from a Trusted Oracle7 Server. The *Trusted Oracle7 Server Administrator's Guide* contains more guidelines for using Export with the Trusted Oracle7 Server.

Network Considerations

This section describes factors to take into account when using Export and Import across a network.

Transporting Export Files Across a Network

Since the export file is in binary format, when transferring it across a network, be sure to use a protocol that supports binary transfers to prevent corruption of the file. For example, using FTP or a similar file transfer protocol, transmit the file in binary mode. Transmitting export files in character mode causes errors when the file is imported.

Exporting/Importing with SQL*Net

By eliminating the boundaries between different machines and operating systems on a network, SQL*Net provides a distributed processing environment for Oracle7 Server products. SQL*Net lets you export and import over a network. For example, running Export locally, you can write data from a remote Oracle database into a local export file. Running Import locally, you can read data into a remote Oracle database.

To use Export or Import with SQL*Net, include the *@connect_string* when entering the EXP or IMP command. For the exact syntax of this clause, see the user's guide for your SQL*Net protocol. For more information on SQL*Net, see *Understanding SQL*Net*. If you use Oracle Names, see also the *Oracle Names Administrator's Guide*.

NLS Considerations

This section describes behavior of Export and Import with respect to National Language Support (NLS).

Character Set Translation

Export writes export files using the character set specified for the user session; for example, 7-bit ASCII or IBM Code Page 500 (EBCDIC). If necessary, Import translates the data to the character set of its host system. Import converts character data to the user-session character set if that character set is different from the one in the Export file.

The export file identifies the character encoding scheme used for the character data in the file. If that character set is any single-byte character set (for example, EBCDIC or USASCII7), and if the character set used by the target database is also a single-byte character set, then the data is automatically converted to the character encoding scheme specified for the user session during import, as specified by the NLS_LANG parameter. After the data has been converted to the session character set, it is then converted to the database character set.

During the conversion, any characters in the export file that have no equivalent in the target character set are replaced with a default character. (The default character is defined by the target character set.) To guarantee 100% conversion, the target character set should be a superset or equivalent of the source character set.

For multi-byte character sets, conversion is only performed if the length of the character string cannot change as a result of the conversion.

For more information, refer to the National Language Support section of the *Oracle7 Server Reference*.

Export/Import and Single-Byte Character Sets

Some eight-bit characters can be “dropped” when importing an eight-bit character set export file. This occurs if the client machine has a native seven-bit character set, or the NLS_LANG operating system environment parameter is set to a seven-bit character set. “dropped” implies that some eight-bit characters are converted to seven-bit equivalents. Typically, accented European characters lose their accent mark.

This situation occurs because the eight-bit characters in the export file are converted to seven-bit characters via the client application. When sent to the database, the seven-bit characters are converted by the server into eight-bit characters. To avoid this situation, it is necessary to turn off one of these conversions. One possibility is to set NLS_LANG to the character set of the export file data.

When importing an Oracle version 6 export file with a character set different from that of the native operating system or the setting for NLS_LANG, you need to set the CHARSET import parameter to indicate the character set of the export file data. The CHARSET parameter is described on page 2 – 19.

Export/Import and Multi-Byte Character Sets

An export file that is produced with a multi-byte character set (for example, Chinese or Japanese) must be imported on a system that has a character set with the same character length. That is, the export character set and the import character set must have a 1:1 ratio.

Using Export

Before Using Export

To use Export, the script CATEXP.SQL or CATALOG.SQL (which runs CATEXP.SQL) must be run after the database has been created.



Additional Information: The actual names of the script files depend on your operating system. The script file names and the method for running them are described in your Oracle operating system-specific documentation.

CATEXP.SQL or CATALOG.SQL only needs to be run once. Once run, it need not be run before future exports. The following operations are performed to prepare the database for Export.

- create the necessary export views
- assign all necessary privileges to the EXP_FULL_DATABASE role
- assign EXP_FULL_DATABASE to the DBA role

Before running Export, ensure that there is enough disk or tape storage space to write the export file to. If there is not enough space, Export will terminate with a write-failure error. You can use table sizes to estimate the maximum space needed. Table sizes can be found in the USER_SEGMENTS view in the Oracle data dictionary (see the *Oracle7 Server Administrator's Guide* for more information).

Invoking Export

You can invoke Export in three ways:

- Enter the command `EXP username/password PARFILE=filename` PARFILE is a file containing the export parameters you typically use. If you use different parameters for different databases, you can have multiple PARFILES. This is the suggested method.
- Enter the command `EXP username/password` followed by the various parameters you intend to use. Note that the number of parameters cannot exceed the maximum length of a command line on your system.
- Enter only the command `EXP username/password` to begin an interactive session and let Export prompt you for the information it needs. The interactive option does not provide as much functionality as the parameter-driven method. It exists primarily for backward compatibility.

The *username* and *password* can also be specified in the parameter file, although for security reasons, this is not recommended.

If you omit *username/password*, Export will prompt you for it.

Export provides online help. Enter EXP HELP=Y on the command line to see a help screen like the one shown in Figure 1 – 2.

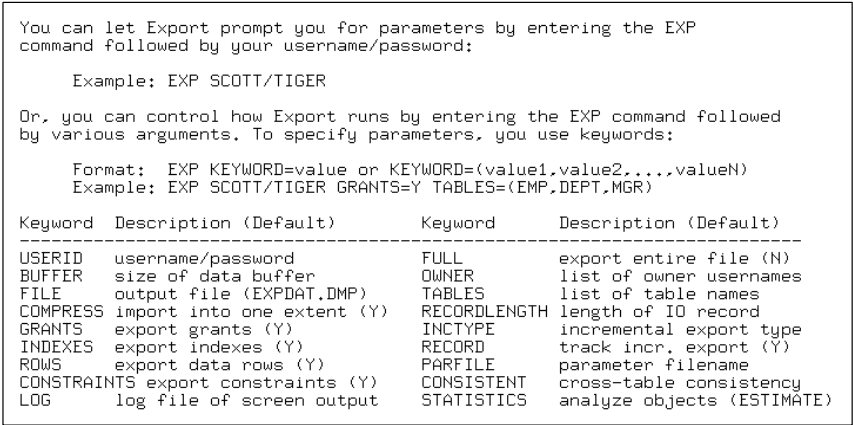


Figure 1 – 2 Export Help Screen

The Parameter File

The parameter file allows you to specify Export parameters in a file where they can easily be modified or reused. Create the parameter file using any flat file text editor. The command line option PARFILE=<filename> tells Export to read the parameters from the specified file rather than from the command line. For example:

```
EXP PARFILE=filename
EXP username/password PARFILE=filename
```

The syntax for parameter file specifications is:

```
KEYWORD=value
or
KEYWORD=(value) or KEYWORD=(value1, value2, ...)
```

The following is an example of a partial parameter file listing:

```
FULL=Y
FILE=DBA.DMP
GRANTS=Y
INDEXES=Y
CONSISTENT=Y
...
```



OSDoc

Additional Information: The maximum size of the parameter file may be limited and operating system file naming conventions will apply. See your Oracle operating system-specific documentation for more information.

Comments in the
Parameter File

Table Name
Restrictions

You can add comments to the parameter file by preceding them with the # sign. All characters to the right of the # sign are ignored.

Table names specified on the command line cannot include a # sign, unless the table name is enclosed in quotation marks. Similarly, in the parameter file, if a table has a # sign in the name, the rest of the line is interpreted as a comment unless the table name is enclosed in quotation marks.



OSDoc

Additional Information: Some operating systems require single vs. double quote marks, or vice versa. See your Oracle operating system-specific documentation.

For example, if the parameter file contains the line:

```
TABLES=(EMP#, DEPT, MYDATA)
```

nothing on the line after EMP# is seen as input by Export. As a result, DEPT and MYDATA are not exported.



Attention: When the name is specified in quotation marks, it is case-sensitive. The name must therefore exactly match the table name stored in the database. By default, database names are stored as uppercase.

Export Parameters

The parameters listed below can be specified in the parameter file. They are described in detail in the remainder of this section.

USERID	RECORDLENGTH
BUFFER	INCTYPE
FILE	RECORD
GRANTS	HELP
INDEXES	LOG
ROWS	CONSISTENT
CONSTRAINTS	STATISTICS
COMPRESS	FEEDBACK
FULL	MLS*
OWNER	MLS_LABEL_FORMAT*
TABLES	

**Trusted Oracle7 Server parameter*

BUFFER Default: operating system dependent

The parameter BUFFER determines the maximum number of rows in an array fetched by Export by specifying the size of the buffer. You can calculate the buffer size as follows:

```
buffer_size = rows_in_array * maximum_row_size
```

buffer_size is the size in bytes of the buffer into which data rows are fetched. If zero is specified, or if rows contain LONG data, only one row at a time is fetched.



OSDoc

Additional Information: See your Oracle operating system-specific documentation to determine the default value for this parameter.

COMPRESS Default: Y

Specifies how Export/Import will manage the initial extent for table data.

The default, COMPRESS=Y, causes Export to flag the table data for consolidation into one initial extent upon Import.

If COMPRESS=N is specified, Export will sum the lengths of all current extents on export and Import will use that value as the size of the initial extent for the imported table data. Note that, when a table has had many deletes, this size may be much larger than the table actually requires.

Note: Although used extents are consolidated into one extent upon import, you must request consolidation when exporting, because your request causes the table definition to be exported differently. Thus, if you request consolidation when exporting, you can import the data in consolidated form only.

CONSISTENT Default: N

Specifies whether Export will use the SET TRANSACTION READ ONLY statement to insure that the export is consistent to a single point in time. Specifying CONSISTENT=Y is important when other applications will be updating the database after an export has started because these transactions will be rolled back, if necessary.

If you specify CONSISTENT=Y, a rollback segment must be retained for the duration of the export to allow backing out of the effects of any uncommitted transactions when a table is exported. Note that, if the volume of updates is large, the rollback segment will itself be large. In addition, the export of each table will be slower, since the rollback segment must be scanned for uncommitted transactions.

Note: CONSISTENT cannot be used with an incremental export.



Suggestion: To minimize the time and space required for such exports, tables that need to remain consistent should be exported separately from those that do not.

For example, export the EMP and DEPT tables together in a consistent export, then export the remainder of the database in a second pass.

Exporting the minimum number of objects that must be guaranteed consistent helps to reduce the chances of encountering a “snapshot too old” error.

This error occurs when rollback space has been used up, and space taken up by committed transactions is reused for new transactions. Reusing space in the rollback segment allows database integrity to be preserved with minimum space requirements, but it imposes a limit on the amount of time that a read-consistent image can be preserved.

If a committed transaction has been overwritten and the information is needed for a read-consistent view of the database, then a “snapshot too old” error results.

To avoid this error, minimize the time taken by a read-consistent export (by restricting the number of objects exported and, if possible, by reducing the database transaction rate). Also, make the rollback segment as large as possible.

CONSTRAINTS Default: Y

A flag to indicate whether to export table constraints.

DIRECT Default: N

Specifying DIRECT=Y causes export to extract data by reading the data directly, bypassing the SQL Command Processing layer (evaluating buffer). This method can be much faster than a conventional path export.

Performance can also be improved by using direct path export with the database in direct read mode. Contention for resources with other users is eliminated because database blocks are read into the private buffer cache, rather than a public buffer cache.

For more information about direct path exports, see page 1 – 32.

FEEDBACK	<p>Default: 0 (zero)</p> <p>Specifies that Export should display a progress meter in the form of a dot for <i>x</i> number of rows exported. For example, were you to specify FEEDBACK=10, Export would display a dot each time 10 rows had been exported. The FEEDBACK value applies to all tables being exported, it cannot be set on a per table basis.</p>
FILE	<p>Default: EXPDAT.DMP</p> <p>The name of the export file. The default extension is .DMP, but you can specify any extension.</p>
FULL	<p>Default: N</p> <p>Specifies whether Export should export the entire database or not. Specify FULL=Y to export in full database mode (you must have the EXP_FULL_DATABASE role enabled to do this).</p>
GRANTS	<p>Default: Y</p> <p>Specifies whether grants should be exported or not.</p>
HELP	<p>Default: N</p> <p>Displays a help message with descriptions of the Export parameters.</p>
INCTYPE	<p>Default: none</p> <p>Specifies the type of incremental export. Options are COMPLETE, CUMULATIVE, and INCREMENTAL. See the section in this chapter called "Incremental, Cumulative and Complete Exports" for a description of these options.</p>
INDEXES	<p>Default: Y</p> <p>Specifies whether indexes should be exported or not.</p>
LOG	<p>Default: none</p> <p>Specifies a file name to receive informational and error messages. For example:</p> <pre>EXP system/manager LOG=export.log</pre> <p>If this parameter is specified, messages are logged in the log file <i>and</i> displayed via the terminal display.</p>

MLS The Export utility can be used on any Trusted Oracle7 database to produce a normal export file. In DBMS Mandatory Access Control (MAC) mode, the MLS parameter can also be specified. With this parameter, the Multi-Level Security (MLS) labels on each row and object in the database are included in the export file. This parameter is further described in the *Trusted Oracle7 Server Administrator's Guide*.

MLS_LABEL_FORMAT This parameter allows you to override the default format specified in the database for human-readable MLS labels. The format you specify will be used in the export file instead of the format defined by the database initialization parameter. This parameter is further described in the *Trusted Oracle7 Server Administrator's Guide*.

OWNER Default: undefined
Specifies a list of usernames whose objects will be exported. Specify OWNER=*userlist* to export in user mode.

RECORD Default: Y
A flag to indicate whether to record an incremental or cumulative export in database tables SYS.INCVID, SYS.INCFIL, and SYS.INCEXP. See page 1 – 31.

RECORDLENGTH Default: operating system dependent
Specifies the length, in bytes, of the file record. The RECORDLENGTH parameter is necessary when you must transfer the export file to another operating system that uses a different default value.



OSDoc

Additional Information: See your Oracle operating system-specific documentation for the system being exported to determine the proper value, or when you want to create a file with a different record size.

ROWS Default: Y
Specifies whether the rows of table data should be exported or not.

STATISTICS Default: ESTIMATE
Specifies the type of database optimizer statistics to generate when the exported data is imported later. Options are ESTIMATE, COMPUTE, and NONE. See Chapter 13 of the *Oracle7 Server Concepts* manual for information about the optimizer.

TABLES	<p>Default: undefined</p> <p>Specifies list of table names to export. Specify TABLES=<i>tablelist</i> to export in table mode.</p> <p>Storage Parameters: Tables are exported using the current storage parameters. Unless ALTER TABLE command has been executed to change them, these are the parameters in effect when the tables were created.</p>
USERID	<p>Default: none</p> <p>Specifies the <i>username/password</i> of the user initiating the export.</p>

Parameter Interactions Certain parameters can conflict with each other. For example, specifying TABLES can conflict with an OWNER specification, the following command will cause Export to terminate with an error:

```
EXP system/manager OWNER=jones TABLES=scott.emp
```

Similarly, OWNER conflicts with FULL=Y.

Although ROWS=N and INCTYPE=INCREMENTAL can both be used, specifying ROWS=N (no data) defeats the purpose of incremental exports, which is to make a backup copy of tables that have changed.

Export's Interactive Mode

Starting Export from the command line with no arguments will initiate Export's interactive mode. Note that the interactive mode does not provide prompts for all of Exports functionality. It is provided only for backward compatibility.

You may not see all the prompts in a given Export session because some prompts depend on your responses to other prompts. Some prompts show a default answer. If the default is acceptable, press [RETURN]. To end your input, enter a period (.) followed by [RETURN].

If you have not specified a *username/password* on the command line, EXPORT first prompts you for this information. Then the prompts described below are displayed.

Export's Interactive Mode Prompts

Enter array fetch buffer size: 4096 >

Default: 4096

The value specified here determines the space available to buffer rows. Export uses the array fetch program interface call to speed performance. The default is usually adequate, but the buffer must be at least as large as the longest row. If you enter zero, only one row at a time is fetched.

Export file: EXPDAT.DMP >

Default: EXPDAT.DMP

Specify the name of the export file (the default is EXPDAT.DMP). If you do not specify an extension, the file extension of the export file will default to .DMP.

E(ntire database), U(sers), or T(ables): U >

Default: U

Specify either (T)able or (U)ser mode. Privileged users can also choose (E)ntire database mode. The options above relate to the Export parameters TABLES, OWNER and FULL respectively.

Export grants (Y/N): Y >

Default: Y

Specify (Y)es to export grants or (N)o. The grants that will be exported depend on whether you are in FULL DATABASE mode or USER mode. In FULL DATABASE mode, all grants on a table are exported. In USER mode, only those granted by the owner of the table are exported.

Export table data (Y/N): Y >

Default: Y

Specify (Y)es, to export data in the tables with the table definitions. Specify (N)o, to export only the table definitions.

Compress extents (Y/N): Y >

Default: Y

Specify (Y)es, to have the data for each exported table's used extents consolidated into one large initial extent when imported. Specifying (N)o, will cause all tables to be created with the storage values in effect when the table was exported.

Note: Although used extents are consolidated into one extent upon import, you must request consolidation when exporting, because your request causes the table definition to be exported differently. Thus, if you request consolidation when exporting, you can import the data in consolidated form only.

Schema to export: (RETURN to quit) >

Default: none

Specify the name of the schema to export. To indicate "no user" (and terminate the current Export session), press [RETURN].

Table to be exported: (RETURN to quit) >

Default: none

Specify the name of the table to export. Entering a null table list causes all tables in the schema to be exported.

If no schema prefix is specified, Export defaults to the exporter's schema or the schema containing the last table exported in the current session.

For example, if BETH is a privileged user exporting in table mode, then Export assumes that all tables are in BETH's schema until another schema is specified. Only a privileged user (someone with BACKUP ANY TABLE privileges) can export tables in another user's schema.

Example Export Sessions

Example Export Session in Full Database Mode

The following example export session shows you how the command line and interactive methods in the full database, user, and table modes might be used.

Only users with the DBA role, the EXP_FULL_DATABASE role, or the BECOME USER privilege can export in full database mode. In this example, an entire database is exported to file DBA.DMP with all GRANTs and all data.

Command Line Method

```
> exp system/manager full=Y file=dba.dmp
```

Interactive Method

```
> exp system/manager
...
Enter array fetch buffer size: 4096 > (RETURN)
Export file: EXPDAT.DMP> dba.dmp
E(ntire database), U(sers), T(ables): U> e
Export grants (Y/N): Y> y
Export table data (Y/N): Y> y
Compress extents (Y/N): Y> y
```

Export Messages

```
About to export the entire database...
. exporting tablespace definitions
. exporting profiles
. exporting user definitions
. exporting roles
. exporting resource costs
. exporting rollback segment definitions
. exporting database links
. exporting sequence numbers
. exporting job queues
. exporting refresh groups and children
. exporting cluster definitions
. about to export SYSTEM's tables ...
. .exporting table          DEF$_CALL          0 rows exported
. .exporting table          DEF$_CALLDEST       0 rows exported
. .exporting table          DEF$_DEFAULTDEST    0 rows exported
. .exporting table          DEF$_ERROR          0 rows exported
. .exporting table          DEF$_SCHEDULE       0 rows exported
. .exporting table          DEF$_TRAN           0 rows exported
. .exporting table          DEF$_TRANDEST       0 rows exported
. about to export SCOTT's tables ...
. . exporting table          BONUS              0 rows exported
. . exporting table          DEPT                5 rows exported
```

```

. . exporting table                      EMP          14 rows exported
. . exporting table                      SALGRADE      5 rows exported
. about to export ADAMS's tables ...
. about to export JONES's tables ...
. about to export CLARK's tables ...
. about to export BLAKE's tables ...
. exporting referential integrity constraints
. exporting posttables actions
. exporting synonyms
. exporting views
. exporting stored procedures
. exporting triggers
. exporting default and system auditing options
Export terminated successfully without warnings.

```

Example Export Session in User Mode

Exports in user mode can back up one database user. For example, aDBA may want to back up the tables of a deleted user for a period of time. User mode is also appropriate for users who want to back up their own data, or who want to move objects from one owner to another.

This example shows user SCOTT exporting all his tables in user mode.

Command Line Method

```
exp scott/tiger file=scott.dmp
```

Interactive Method

```

exp scott/tiger
...
Enter array fetch buffer size: 4096 >      (RETURN)
Export file: EXPDAT.DMP>      scott.dmp
U(sers), or T(ables): U>  u
Export grants (Y/N): Y>  y
Export table data (Y/N): Y>  y
Compress extents (Y/N): Y>  y

```

Export Messages

```

About to export SCOTT's objects ...
. exporting job queues
. exporting refresh groups and children
. exporting database links
. exporting sequence numbers
. exporting cluster definitions
. exporting stored procedures
. about to export SCOTT's tables ...
. exporting table                      BONUS          0 rows exported
. exporting table                      DEPT            7 rows exported
. exporting table                      EMP            22 rows exported

```

```
. exporting table                PROJ          0 rows exported
. exporting table                SALGRADE       5 rows exported
. exporting synonyms
. exporting views
. exporting referential integrity constraints
. exporting triggers
. exporting posttables actions
Export terminated successfully without warnings.
```

Example Export Sessions in Table Mode

In table mode, you can export table data or the table definitions. (If no data is exported, the CREATE TABLE statement is placed in the export file, with grants and indexes, if they are specified.)

A user with the EXP_FULL_DATABASE role can use this mode to export tables from any user's schema by specifying TABLES=(*schema.table*). If *schema* is not specified, it defaults to the previous schema from which an object was exported. If there is not a previous object, it defaults to the exporter's schema. In the following example, *schema* defaults to SYSTEM for table A and to SCOTT for table C:

```
> exp system/manager tables=(a, scott.b, c, mary.d)
```

A user without the EXP_FULL_DATABASE role can export only tables that he or she owns.

Exports in table mode do not include cluster definitions. As a result, the data is imported into unclustered tables. Thus, you can use Export to uncluster tables when there is not enough free space in your database.

Example 1

In this example, a DBA exports specified tables for two users.

Command Line Method

```
> exp system/manager tables=(mort.bar, mary.app) grants=Y indexes=Y
```

Interactive Method

```
> exp system/manager
...
Enter array fetch buffer size: 4096 >      (RETURN)
Export file: EXPDAT.DMP >      (RETURN)
E(ntire database), U(sers), T(ables): U>  t
Export table data (Y/N): Y>      (RETURN)
Compress extents (Y/N): Y> y
```

Export Messages

```
About to export specified tables ...
Table Name:  mort.bar                [interactive session only]
Current user changed to MORT
```

```

. exporting table                BAR    2355 rows exported
Table Name:  mary.app            [interactive session only]
Current user changed to MARY
. exporting table                APP    14947 rows exported
Table Name:  (RETURN)            [interactive session only]
[Export writes file]

```

Example 2

In this example, user LEWIS exports selected tables that he owns.

Command Line Method

```
> exp lewis/newyork file=lew.dmp tables=(credits, debits)
```

Interactive Method

```

exp lewis/newyork
...
Enter array fetch buffer size: 4096 >      (RETURN)
Export file: EXPDAT.DMP>      lew.dmp
U(sers), T(ables): U> T
Export table data (Y/N): Y> y
Compress extents (Y/N): Y> y

```

Export Messages

```

About to export specified tables ...
Table Name:  credits                [interactive session only]
. exporting table                CREDITS  423 rows exported
Table Name:  debits                [interactive session only]
. exporting table                DEBITS   423 rows exported
Table Name:  (RETURN)              [interactive session only]
[Export writes file]

```

Incremental, Cumulative, and Complete Exports

Incremental, cumulative, and complete exports provide time- and space-effective backup strategies. This section shows how to set up and use these export strategies.

Restrictions

You can do incremental, cumulative, and complete exports only in full database mode (FULL=Y). Only users who have the EXP_FULL_DATABASE role can run incremental, cumulative, and complete exports. This role contains the privileges needed to modify the system tables that track incremental exports. Those tables are discussed at the end of this section.

Base Backups

If you choose to use cumulative and incremental exports, it is advised that you periodically perform a complete export to create a *base backup*. Following the complete export, the administrator can take frequent incremental exports and occasional cumulative exports. After a given period of time, the cycle should begin again with another complete export.

Incremental Exports

An *incremental* export backs up only tables that have changed since the last incremental, cumulative, or complete export. An incremental export exports the table's definition and all its data, *not just the changed rows*. Incremental exports are typically done more often than cumulative or complete exports.

Figure 1 – 3 shows an incremental export at time 1, after 3 tables have been modified. Only the modified tables (and associated indexes) are exported.

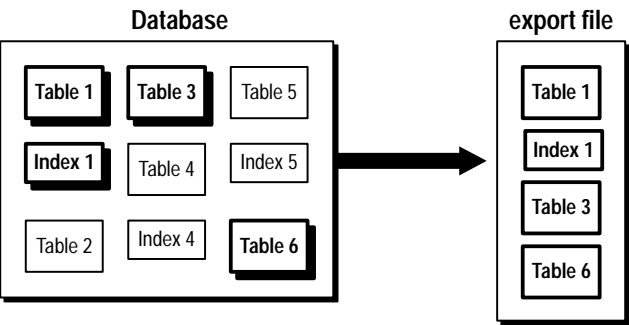


Figure 1 – 3 Incremental Export at Time 1

Figure 1 – 4 shows another incremental export at time 2, after 2 tables have been modified. Table 3 was modified a second time, so it is exported at time 2 as well as at time 1.

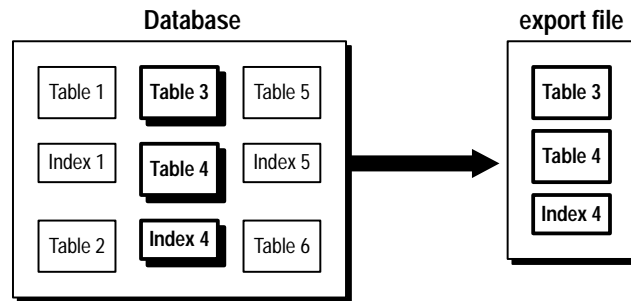


Figure 1 – 4 Incremental Export at Time 2

Note: Incremental exports cannot be specified as read-consistent.

Cumulative Exports A *cumulative* export backs up tables that have changed since the last cumulative or complete export. In essence, a cumulative export compresses a number of incremental exports into a single cumulative export file. It is not necessary to save incremental export files taken before a cumulative export because the cumulative export file replaces them.

Figure 1 – 5 shows a cumulative export at time 1, after 3 tables have been modified. The modified tables (and associated indexes) are exported. This export is equivalent to an incremental export.

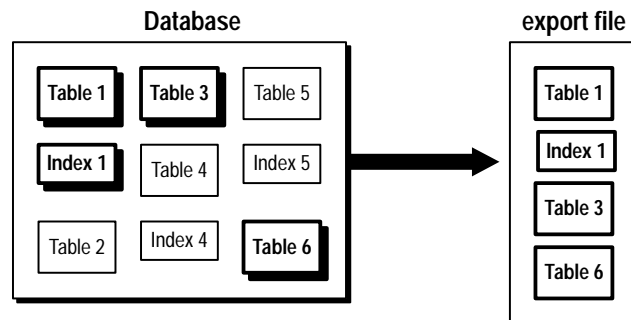


Figure 1 – 5 Cumulative Export at Time 1

Figure 1 – 6 shows a cumulative export at time 2, after 2 tables have been modified. This time, the two tables modified at time 2 are exported, and the tables modified at time 1 are exported as well.

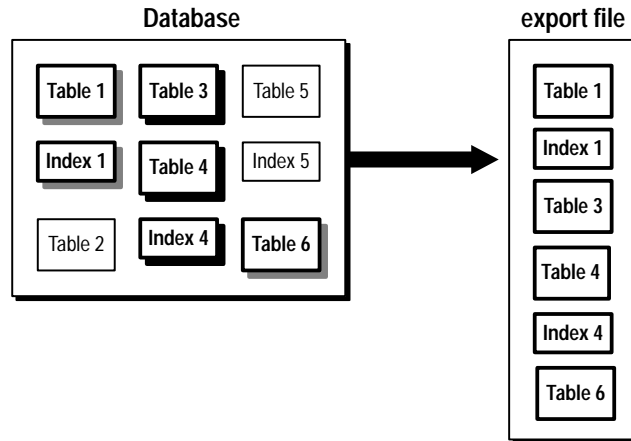


Figure 1 – 6 Cumulative Export at Time 2

This cumulative export file is a combination of the incremental exports from time 1 and time 2, except that table 3 (which was modified at both times) only occurs once in the export file. In this way, cumulative exports save space over multiple incremental exports.

Complete Exports

A *complete* export establishes a base for incremental and cumulative exports. It is equivalent to a full database export, except that it also updates the tables that track incremental and cumulative exports.

Figure 1 – 7 shows a complete export at time 2. With the complete export, all objects in the database are exported regardless of when (or if) they were modified.

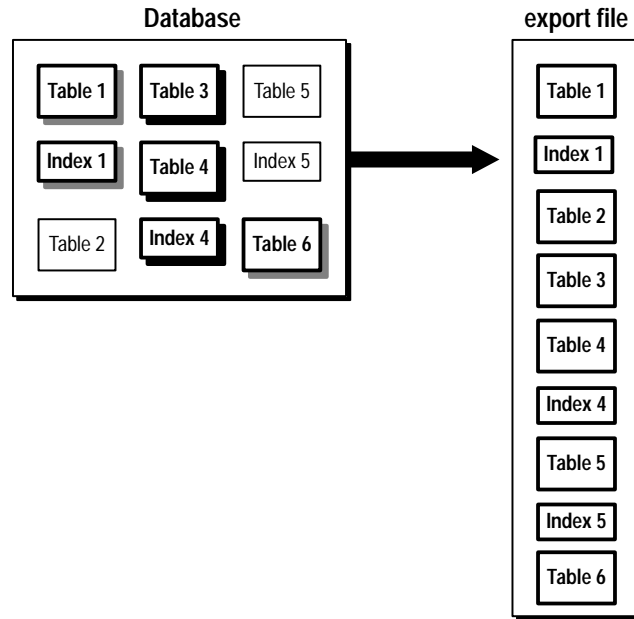


Figure 1 – 7 Complete Export

Benefits

Incremental and cumulative exports help solve the problems faced by administrators who work in environments where many users create their own tables. For example, administrators can restore tables accidentally dropped by users.

The benefits of incremental and cumulative exports include:

- smaller export files
- less time to export

These benefits result because not all tables have changed. So the time and space required for an incremental or cumulative export is shorter than for a full database export.

A Scenario

The following scenario shows how you can use cumulative and incremental exports.

Assume that as manager of a data center, you do the following tasks:

- a complete export (X) every three weeks
- a cumulative export (C) every weekend
- an incremental export (I) every night

Your export schedule follows:

```

DAY: 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
      X  I  I  I  I  I  C  I  I  I  I  I  I  I  C  I  I  I  I  I  X

```

To restore through day 18, you would first import the *system information* from the incremental export taken on day 18. You would then import the *data* from:

- the complete export taken on day 1
- the cumulative export taken on day 7
- the cumulative export taken on day 15
- and three incremental exports taken on days 16, 17, and 18

The incremental exports on days 2 through 6 can be discarded on day 7, when the cumulative export is done, because it incorporates all of them. Similarly, the incremental exports on days 8 through 14 can be discarded after the cumulative export on day 15.

Note: The section “Incremental, Cumulative and Complete Imports” on page 2–29 explains how to do an incremental import.

Which Data Is Exported?

The purpose of an incremental or cumulative export is to identify and export only those database objects (such as clusters, tables, views, and synonyms) that have changed since the last export. Each table is associated with other objects that you can export. Besides the data itself, there are indexes, grants, audits, and comments.

The entire grant structure for tables or views is exported with the underlying base table(s). Indexes are exported with their base table, regardless of who created the index.

Export automatically exports a read-consistent view of a table, even if the table is being updated during the export.

Any modification (UPDATE, INSERT, or DELETE) on a table automatically qualifies that table for export.

Also, if database structures have changed in the following ways, then the underlying base tables and data are exported:

- a table is created
- a table definition is changed by an ALTER TABLE statement
- comments are added or edited
- auditing options are updated
- grants (of any level) are altered
- indexes are added or dropped

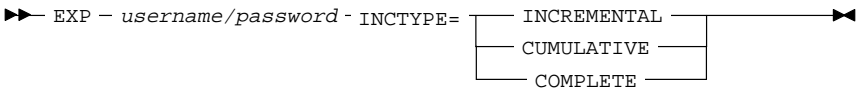
- index storage parameters are changed by an ALTER INDEX statement

In addition, the following data is backed up:

- all system objects (including tablespace definitions, rollback segment definitions, and user privileges, but not including temporary segments)
- information about dropped objects
- clusters, tables, views, and synonyms created since the last export

Command Syntax

The command syntax is as follows:



where:

INCREMENTAL	Exports all database objects that have changed since the last incremental export, as tracked by table SYS.INCEXP, then updates the table with a new ITIME and EXPID.
CUMULATIVE	Exports all database objects that have changed since the last cumulative export, as tracked by SYS.INCEXP, then updates the table with a new CTIME, ITIME, and EXPID.
COMPLETE	Exports all objects and updates the tables SYS.INCEXP and SYS.INCVID. (A FULL=Y export does not update these tables unless you specify the INCTYPE parameter.).

Note: See page 1 – 31 for definitions of ITIME, EXPID and CTIME.

Note: For incremental and cumulative exports, a row is also added to table SYS.INCFIL to identify the export file and the user doing the export. For more information on the system tables that support incremental export, see page 1 – 31. When exporting with the parameter “INCTYPE = COMPLETE”, all the previous entries are removed from SYS.INCFIL and a new row is added specifying an “x” in the column EXPTYPE.

Example Incremental Export Session

The following example shows an incremental export session after the tables SCOTT.EMP and SCOTT.DEPT are modified:

```
> exp system/manager inctype=incremental
...
About to export the entire database ...
. exporting tablespace definitions
. exporting profiles
. exporting user definitions
. exporting roles
. exporting resource costs
. exporting rollback segment definitions
. exporting database links
. exporting sequence numbers
. exporting job queues
. exporting refresh groups and children
. exporting cluster definitions
. exporting stored procedures
. about to export SYSTEM's tables ...
. about to export SCOTT's tables ...
. exporting table                DEPT                7 rows exported
. exporting table                EMP                  22 rows exported
. about to export ADAMS's tables ...
. about to export JONES's tables ...
. about to export CLARK's tables ...
. about to export BLAKE's tables ...
. exporting referential integrity constraints
. exporting triggers
. exporting posttables actions
. exporting synonyms
. exporting views
. exporting default and system auditing options
. exporting information about dropped objects
Export terminated successfully without warnings.
```

System Tables	SYS owns three tables (INCEXP, INCFIL, and INCVID) maintained by Export/Import. None of these tables should be altered in any way.												
SYS.INCEXP	<p>Export maintains a table to track which objects were exported in specific exports. That table, SYS.INCEXP, contains these columns:</p> <table> <tr> <td>OWNER#</td><td>The userid of the schema containing the table.</td></tr> <tr> <td>NAME</td><td>The object name. With OWNER#. The primary key consists of OWNER#, NAME and TYPE.</td></tr> <tr> <td>TYPE</td><td>The type of the object (a code standing for INDEX, TABLE, CLUSTER, VIEW, SYNONYM, SEQUENCE, PROCEDURE, FUNCTION, PACKAGE, TRIGGER, SNAPSHOT, SNAPSHOT LOG, or PACKAGE BODY).</td></tr> <tr> <td>CTIME</td><td>The date and time of the last cumulative export that included this object.</td></tr> <tr> <td>ITIME</td><td>The date and time of the last incremental export that included this object.</td></tr> <tr> <td>EXPID</td><td>The ID of the incremental or cumulative export, also found in table SYS.INCFIL.</td></tr> </table> <p>You can use this information in several ways. For example, you could generate a report from SYS.INCEXP after each export to document the export file.</p>	OWNER#	The userid of the schema containing the table.	NAME	The object name. With OWNER#. The primary key consists of OWNER#, NAME and TYPE.	TYPE	The type of the object (a code standing for INDEX, TABLE, CLUSTER, VIEW, SYNONYM, SEQUENCE, PROCEDURE, FUNCTION, PACKAGE, TRIGGER, SNAPSHOT, SNAPSHOT LOG, or PACKAGE BODY).	CTIME	The date and time of the last cumulative export that included this object.	ITIME	The date and time of the last incremental export that included this object.	EXPID	The ID of the incremental or cumulative export, also found in table SYS.INCFIL.
OWNER#	The userid of the schema containing the table.												
NAME	The object name. With OWNER#. The primary key consists of OWNER#, NAME and TYPE.												
TYPE	The type of the object (a code standing for INDEX, TABLE, CLUSTER, VIEW, SYNONYM, SEQUENCE, PROCEDURE, FUNCTION, PACKAGE, TRIGGER, SNAPSHOT, SNAPSHOT LOG, or PACKAGE BODY).												
CTIME	The date and time of the last cumulative export that included this object.												
ITIME	The date and time of the last incremental export that included this object.												
EXPID	The ID of the incremental or cumulative export, also found in table SYS.INCFIL.												
SYS.INCFIL	<p>The table SYS.INCFIL tracks the incremental and cumulative exports and assigns a unique identifier to each. This table contains the following columns:</p> <table> <tr> <td>EXPID</td><td>The ID of the incremental or cumulative export, also found in table SYS.INCEXP.</td></tr> <tr> <td>EXPTYPE</td><td>The type of export (incremental or cumulative).</td></tr> <tr> <td>EXPFILE</td><td>The name of the export file.</td></tr> <tr> <td>EXPDATE</td><td>The date of the export.</td></tr> <tr> <td>EXPUSER</td><td>The username of the user doing the export.</td></tr> </table>	EXPID	The ID of the incremental or cumulative export, also found in table SYS.INCEXP.	EXPTYPE	The type of export (incremental or cumulative).	EXPFILE	The name of the export file.	EXPDATE	The date of the export.	EXPUSER	The username of the user doing the export.		
EXPID	The ID of the incremental or cumulative export, also found in table SYS.INCEXP.												
EXPTYPE	The type of export (incremental or cumulative).												
EXPFILE	The name of the export file.												
EXPDATE	The date of the export.												
EXPUSER	The username of the user doing the export.												
SYS.INCVID	A third table, SYS.INCVID, contains one column for the EXPID of the last valid export. This information determines the EXPID of the next export.												

Export Utility: Direct Path Export

Oracle Export utility's Direct Path export feature extracts data much faster than a conventional path export. Direct path export achieves this performance gain by reading data directly, bypassing the SQL Command Processing layer.

For added performance, the database can be set to direct read mode thus eliminating contention with other users for database resources because database blocks are read into the Export session's private buffer, rather than a public buffer cache.

Conventional Vs. Direct Export Methods

Export provides two methods for exporting table data:

- conventional path export
- direct path export

Conventional Path Export A conventional path export uses the SQL statement "SELECT * FROM TABLE" to extract data from database tables. Data are read from disk into a buffer cache and rows are transferred to the evaluation buffer. The data, after passing expression evaluation, is transferred over the network to the Export client which then writes the data into the export file.

Direct Path Export A direct path export causes the export data to be read directly, bypassing the evaluation buffer, and saves on data copies whenever possible. It optimizes the execution of a SELECT * FROM TABLE statement.

Used in conjunction with the database in direct read mode (which causes the Export session's private buffer to be used rather than a public buffer), contention with other users for database resources is eliminated and performance improved.

Figure 1 – 8 shows how database table data extraction differs between these two methods.

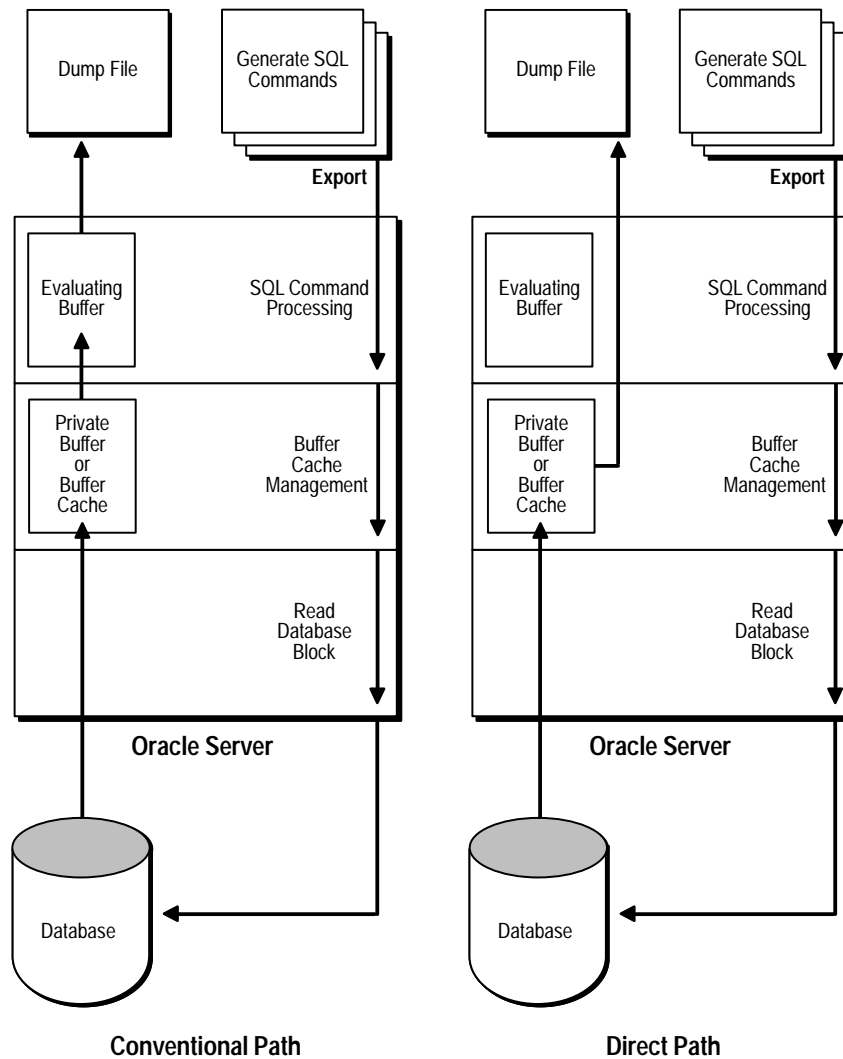


Figure 1 – 8 Database Reads on Direct Path and Conventional Path

In a direct path export, data is read from disk into the buffer cache and rows are transferred *directly* over the network to the Export client. The Evaluating buffer is bypassed. The data is already in the format that Export expects, thus avoiding unnecessary data conversion. The data is transferred over the network to the Export client which then writes the data into the export file.

Preparing the Database for Direct Path Export

Before you can use direct path Export, you must run the upgraded CATEXP.SQL script shipped with release 7.3 after the database has been created.



OSDoc

Additional Information: The actual names of the script files depend on your operating system. The script file names and the method for running them are described in your Oracle operating system-specific documentation.

CATEXP.SQL only needs to be run once. Once run, it need not be run before future exports. This script creates the export views and character set assignments necessary for direct path exports.

Before running Export, ensure that there is enough disk or tape storage space to write the export file to. If there is not enough space, Export will terminate with a write-failure error. You can use table sizes to estimate the maximum space needed.

Table sizes can be found in the USER_SEGMENTS view in the Oracle data dictionary. See the *Oracle7 Server Reference* for more information.

If you do not run CATEXP.SQL before attempting a direct path export, the following errors will occur:

```
EXP-00008: ORACLE error 904 encountered

ORA-00904: invalid column name
EXP-00222: invalid column name
Export terminated successfully with warnings.
```

Invoking a Direct Path Export

You invoke a direct path export using the following syntax:

```
EXPORT DIRECT=Y
```

DIRECT=Y specifies that table data is to be extracted using direct path mode. The default is DIRECT=N, table data is to be extracted using the conventional path.

You can also specify direct path export from a parameter file by using the command-line option PARFILE. For more information about the parameter file, see “The Parameter File” on Page 1 – 11 of this manual.

Note: The Export parameter BUFFER, used to specify the size of the fetch array, applies only to conventional path exports. It has no effect on a direct path export. The parameter RECORDLENGTH can be used to specify the size of the Export I/O buffer.

If you want to export release 7.3 database objects to a previous release, and you are concerned about backward compatibility, you may need to use conventional path export, because the direct path export dump file format is not compatible with releases prior to 7.3.

Following are factors that can affect the size differences between direct path and conventional exports.

- Column data can come in pieces in a direct path export, while it primarily comes in one piece in a conventional path export.

The overhead is 2 extra bytes per extra column piece, which can be significant. This overhead occurs because direct path export reads directly from the buffer cache, where data is not in rows. Conventional path export reads from the SQL interface, where the concept of rows is enforced.

- There can be side effects when the export I/O buffer fills. With direct path export, there is only as much buffer as is returned in a table scan. With conventional path export, data returned from the SQL interface is copied and pads the remaining buffer with zeroes.

Note that when a direct path export completes table scans, it reverts to conventional path export behavior when it must handle other objects or write table-end markers (when in table mode).

The amount of zero padding in direct path exports depends upon the point at which export switches from table scans to writing other objects, as well upon the size of the export I/O buffer (set in the RECORDLENGTH parameter).

Given these factors, it's unlikely that the size of dump files for direct path and conventional exports are never the same.

Direct Export Logging Information

Error, warning, and completion messages are logged as described in “Warning, Error, and Completion Messages” on Page 1 – 4 of this manual.

Character Set Conversion

Direct path export only exports in the database character set. If the export session character set is different from the database character set, a warning will be displayed, and the export will abort. The user must set the session character set to that of the database before retrying the export. However, the import session and target database character set can differ from the source database character set requiring a character set conversion.

Any character set conversion will lengthen the processing time required for an import. Therefore it is advisable to limit the number of character sets conversions to as few as possible.

The ideal scenario is one in which the import session and target database character set are the same as the source database character set requiring no character set conversion.

In the situation where the import session character set and the target character set are the same, but differ from the source database character set, one character set conversion will be required.

If you currently have differing character sets for the source and target databases and/or the import session, and you plan to export/import between these databases regularly, it is advisable to do a one-time export to align these character sets.

Performance Issues

To reduce contention with other users for database resources during a direct path export, you can use database direct read mode. To enable the database direct read mode, enter the following in the INIT.ORA file:

```
compatible = <db_version_number> ,
```

where

db_version_number is 7.1.5 or higher.

Set the RECORDLENGTH Parameter

You may improve performance by increasing the value of the RECORDLENGTH parameter when you invoke a direct path export. Your exact performance gain will vary depending upon the following factors:

- DB_BLOCK_SIZE
- the types of columns in your table
- your I/O layout (Database files should be in a different partition than the export dump file. In fact, in an optimal situation these files will reside on separate disk drives.)

If you leave the RECORDLENGTH parameter undefined, it defaults to your system-dependent value. For more information about your RECORDLENGTH default value, see your operating system-specific documentation.

You can change RECORDLENGTH to any value equal to or greater than your system's BUFSIZE. The highest value is 64k because this value is read into a 2-byte number. Changing the RECORDLENGTH parameter only affects the size of data that accumulates before writing to the disk. It does not affect the operating system file block size. If DB_BLOCK_SIZE is greater than BUFSIZE, it is more advantageous to use DB_BLOCK_SIZE, because each table scan can only return data up to DB_BLOCK_SIZE. If this does not fit in the export I/O buffer, export performs separate writes to the disk.

Consider the following values for RECORDLENGTH:

- multiples of the file system I/O block size
- multiples of DB_BLOCK_SIZE



Additional Information: Other factors affect the use of direct read mode. See the *Oracle7 Server Administrator's Guide* for more information.

Restrictions

The following restrictions apply when executing a direct path Export:

- Direct path export cannot be invoked using Export's interactive mode.
- If the export session's character set is not the same as the database character set, when an export is initiated, a warning will be displayed and the export will abort. The user must set the session character set to that of the database before retrying the export.

Compatibility & Migration

Export files and dump sites generated using direct path export are incompatible with earlier Import versions. For example, an Export file generated using direct path export could not be imported using the release 7.2 Import utility. If backward compatibility is an issue, we recommend that you use Export's conventional path export.

Using Different Versions of Export

This section describes the general behavior and restrictions of running an Export version that is different from the Oracle Server.

Using a Previous Version of Export

In general, any lower version Export utility may be run with a higher version Oracle Server. For example, Export Release 6 can be used with the Oracle7 Server, but note that it creates a Release 6 export file. (This procedure is described in the next section, "Creating Oracle Server Release 6 Export Files from Oracle7".)

Whenever a lower version Export utility runs with a higher version of the Oracle Server, any categories of database objects that did not exist in the lower version are excluded from the export. For example, when running a Release 6 version of Export with the Oracle7 Server, snapshots (which did not exist in Release 6) are excluded from the export. (A complete list of objects excluded in this export is given in the next section.)

Using a Higher Version Export

Attempting to use a higher version of Export with an Oracle Server often produces the following error:

```
EXP-37: Database export views not compatible with Export utility  
EXP-0: Export terminated unsuccessfully
```

The error occurs because views that the higher version of Export expects are not present. To avoid this problem, use the version of the Export utility that matches the Oracle Server.

Change in Export File Format

In Release 7.1, the export file format was changed to accommodate stored procedures, functions, and packages that have comments embedded among the creation-statement keywords. As a result, these code objects (and triggers) exported from release 7.1 and beyond cannot be imported into earlier releases, unless a patch is applied that makes earlier versions of Import and Export use the new export file format. All other objects can be imported normally without the patch.

To export from a 7.0.16 database containing code objects with comments between the creation keywords, for example:

```
CREATE PROCEDURE /* comment */ FOO /* comment */ AS ...
```

or to export from a 7.0.12 or earlier 7.0 database that contains such code objects, or that contain code objects in which the above creation keywords spanned multiple lines; then the export can be made to succeed either by appropriately re-creating the code objects, or by applying a patch that makes earlier versions of Import and Export use the new export file format.

The following table summarizes the situations and solutions:

Exporting From	Objects Affected	Situation	Solution
7.0.12	procedures, functions, packages, package bodies	If the creation keywords for these objects are defined on multiple lines, or if they have comments embedded among the creation keywords, Export fails with a memory violation.	Apply the patch or change the definitions so that the creation keywords are all one line to successfully export these objects.
7.0.16	procedures, functions, packages, package bodies	If these objects have comments embedded among the creation keywords, they cannot be imported from the export file.	Apply the patch or change the definitions to remove the embedded comments, then re-export these objects.
7.1	triggers, procedures, functions, packages, package bodies	These objects cannot be imported into a 7.0 release from a 7.1 export file.	Apply the patch to the earlier release in to read the new export file format or else put the objects into the proper format and use a Release 7.0 export on the 7.1 database.

Table 1 – 2 Exports and Imports Affected by the Change in Export File Format

Creating Oracle Server Version 6 Export Files from Oracle7

It is possible to create a Version 6 export file from Oracle7 by running the Version 6 export executable with the Oracle7 Server. To do so, you must first run CATEXP6.SQL as SYS. This script creates the export views that make the database look to Export like a Version 6 database.

Note: A normal Oracle7 export requires CATEXP.SQL to be run by SYS after CATALOG.SQL to create the necessary views. If needed, both CATEXP.SQL and CATEXP6.SQL can be run, in any order. Once one of these scripts has been run, it does not have to be run again.

Excluded Objects

The Version 6 export utility produces a Version 6 export file by issuing queries against the views created by CATEXP6.SQL. Because this file is Version 6 compatible, it excludes the following Oracle7 objects:

- database triggers
- hash clusters
- job queues
- profiles

- refresh groups
- replication catalog
- roles
- snapshots
- snapshot logs
- stored procedures, functions, and packages
- system auditing options
- tables stored in hash clusters



Suggestion: You can use the UTLEXP6.SQL script to obtain the names of all objects that are excluded from the export file. Review this script for the most up-to-date notes on conversion issues.

Datatype Conversion

The Oracle7 VARCHAR2 datatype is automatically converted to its Version 6 equivalent CHAR. Fixed-length Oracle7 CHAR columns are also converted to variable-length Version 6 CHAR data.

Database Link Names Truncated

Database link names in Oracle7 have the format *name.d1.d2.d3...*, where *name* is the database name, *d1* is a domain specification, and *d2..d3* are optional components in a domain name. In a Version 6 export, link names are truncated, leaving only the *name* component.

VARCHAR Errors

Oracle7 character data (VARCHAR2) can be up to 2,000 characters long, whereas Version 6 CHAR data is limited to 255. The entire column is placed in the export file. The CREATE TABLE statement in the export file includes the actual length of the column, as it was defined in the Oracle7 database.

When Version 6 Import attempts to create the table, the specification of a CHAR column greater than 255 produces an error. One solution is to change the table's datatype to LONG, but be aware of the many restrictions on the Version 6 LONG datatype. See "Selecting a Datatype" in the *Oracle7 Server Application Developer's Guide* for details.

LONG Data Errors

Oracle7 LONG data can be 2 gigabytes. Unlike VARCHAR data, Version 6 Export truncates LONG data when writing to the export file.

In Version 6.0.35.0 and later, Export truncates LONG data to 64K – 3 bytes and produces a warning message. Earlier releases truncate to 64K – 2 bytes with no warning.



Warning: When importing into a Version 6 database, such overly long data produces an error and causes the rest of the table data to be skipped.

System Audit Options Dropped

Oracle7 object audit options on tables and views are automatically exported in their equivalent version 6 form. However, the wide array of Oracle7 privileges makes it impossible to translate them into any Version 6 equivalent. As a result, system audit options are not exported. After Importing into Version 6, issue one or more of the following statements to re-establish system audits:

```
AUDIT DBA [WHENEVER [NOT] SUCCESSFUL]
AUDIT CONNECT [WHENEVER [NOT] SUCCESSFUL]
AUDIT NOT EXISTS
AUDIT ALL [WHENEVER [NOT] SUCCESSFUL]
```


Import

This chapter describes how to use the Import utility. Import reads an Export file into an Oracle database.

Import only reads export files created by Export. For information on how to export a database, see Chapter 1, “Export.” To load data from other operating system files, see the discussion of SQL*Loader in Part II of this manual.

This chapter discusses the following topics:

- Import Basics
- Interactive and Command-Line Methods
- Import Parameters
- Incremental Imports

Several example Import sessions are also provided to help you learn how to use Import.

Import Basics

The basic concept behind Import is very simple. Import extracts the objects from an Export file and puts them into a database. Figure 2 – 1 illustrates the process of importing from an export file:

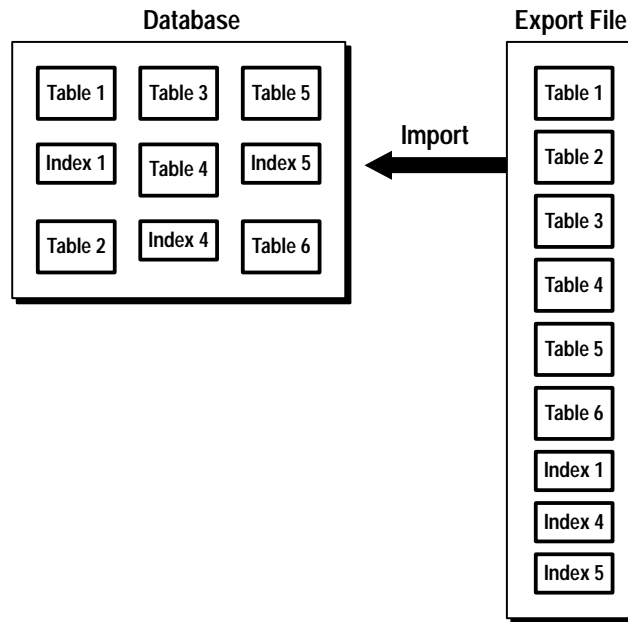


Figure 2 – 1 Importing an Export File

Table Objects: Order of Import Table objects are imported from the export file in the following order:

1. table definitions
2. table data
3. table indexes
4. integrity constraints and triggers

First, new tables are created. Then data is imported. After all data has been imported into all tables, indexes are built. Then triggers are imported, and integrity constraints are enabled on the new tables. This sequence prevents data from being rejected due to the order in which tables are imported. This sequence also prevents redundant triggers from firing twice on the same data (once when it was originally inserted and again during the import).

For example, if the EMP table has a referential integrity constraint on the DEPT table and the EMP table is imported first, then all EMP rows that reference departments that have not yet been imported into DEPT would be rejected provided that the constraints are enabled.

When data is imported into existing tables, however, the order of import can still produce referential integrity failures. In the situation given above, if the EMP table already existed and referential integrity constraints were in force, many rows could be rejected.

A similar situation occurs when a referential integrity constraint on a table references itself. For example, if SCOTT's manager in the EMP table is DRAKE, and DRAKE's row has not yet been loaded, then SCOTT's row will fail—even though it would be valid at the end of the import.



Suggestion: For the reasons mentioned above, it is a good idea to disable referential constraints when importing into an existing table. You can then re-enable the constraints after the import is completed.

Storage Parameters

By default, a table is imported into its original tablespace using the original storage parameters.

If the tablespace no longer exists, or the user does not have sufficient quota in the tablespace, the system uses the default tablespace for that user. If the user does not have sufficient quota in the default tablespace, the user's tables are not imported. (See “Reorganizing Tablespaces” on page 2 – 34 to see how you can use this to your advantage.)

The Parameter OPTIMAL

The storage parameter OPTIMAL for rollback segments is not preserved during export and import.

The COMPRESS Option

If you specified COMPRESS=Y at export time, then the storage parameters for large tables are adjusted to consolidate all data imported for a table into its initial extent. To preserve the original size of an initial extent, you must specify at export time that extents *not* be consolidated. See page 1 – 13 for a description of the COMPRESS parameter.

Character Set Translation

Export writes export files using the character set specified for the user session, for example, 7-bit ASCII or IBM Code Page 500 (EBCDIC). If necessary, Import automatically translates the data to the character set of its host system. Import converts character data to the user-session character set if that character set is different from the one in the Export file. See also page 1 – 8 for a description of how Export handles character set issues.

Access Privileges	<p>To use Import, you need the CREATE SESSION privilege to log on to the Oracle7 Server. This privilege belongs to the CONNECT role established during database creation.</p> <p>You can do an import even if you did not create the export file. However, if the export file is a full database export (created by someone using the EXP_FULL_DATABASE role), then it will only be possible to import that file if you have the IMP_FULL_DATABASE role as well.</p>
Read-Only Tablespaces	<p>You cannot import into a read-only tablespace, unless you first declare it to be a read-write tablespace. When the import completes, you can return the tablespace to its read-only status.</p>
Rollback Segments	<p>When you initialize a database, Oracle creates a single system rollback segment (named SYSTEM). Oracle uses this rollback segment only for transactions that manipulate objects in the SYSTEM tablespace. This restriction does not apply if you intend to import only into the SYSTEM tablespace. However, if you want to import into a different tablespace, you must create a new rollback segment. For details on creating rollback segments, see Chapter 10 “Managing Rollback Segments” of the <i>Oracle7 Server Administrator’s Guide</i>.</p>
Compatibility	<p>Import can read export files created by Export Version 5.1.22 and later.</p>
Trusted Oracle7 Server	<p>There are additional steps and considerations when you are importing to a Trusted Oracle7 Server database. The <i>Trusted Oracle7 Server Administrator’s Guide</i> contains more guidelines for using Import with Trusted Oracle7 Server.</p>

Warning, Error, and Completion Messages

By default, all error messages are displayed. If a log file has been specified, error messages are displayed and written to the log file. A log file should always be generated during import. (On those systems that permit I/O redirection, Import's output can be redirected to a file.)



OSDoc

Additional Information: For information on the LOG specification, see page 2 – 23. Also see your operating system-specific Oracle7 Server documentation for information on redirecting output.

When an import completes without errors, the message “Import terminated successfully without warnings” is issued. If one or more non-fatal errors occurred, but Import was able to continue to completion, then the message “Import terminated successfully with warnings” occurs. If a fatal error occurs, then Import ends immediately with the message “Import terminated unsuccessfully”.

Additional Information: Specific messages are documented in the *Oracle7 Server Messages* manual and in your operating system-specific Oracle7 Server documentation.

Messages that are specific to Trusted Oracle7 Server are documented in the *Trusted Oracle7 Server Administrator's Guide*.

Error Handling

Row Errors

Row errors occur when a row is rejected for some reason, such as a violation of integrity constraints or invalid data. If a row error occurs, Import displays a warning message but continues processing the rest of the table.

Failed Integrity Constraints

A row error is generated if a row violates one of the integrity constraints in force on your system, including:

- not null constraints
- uniqueness constraints
- primary key (not null and unique) constraints
- referential integrity constraints
- check constraints

See Chapter 5, “Maintaining Data Integrity”, in the *Oracle7 Server Application Developer’s Guide* and Chapter 7, “Data Integrity”, in the *Oracle7 Server Concepts* manual for more information on integrity constraints.

Invalid Data

Row errors also occur when importing data into an existing table with a slightly shorter character–data column. The error is caused by data that is too long to fit into a new table’s columns, by invalid data types, and by any other INSERT error.

Long Raw Data Errors

Some LONG RAW data columns may be too large to fit into Import’s data buffer due to memory limitations. On Export, Oracle allows selecting such columns in sections. However, on Import, a contiguous region of memory is required. As a result, some columns that are successfully exported may not be importable until a sufficiently large contiguous region of memory is available. Such columns can produce row errors.

Setting Buffer Size for LONG Data

Tables with LONG data usually require large insert buffer sizes. If you are trying to import LONG data and Import fails with the message:

```
IMP-00020 column (size num) in export file too large for column
        buffer (size num)
```

then you can gradually increase the insert buffer size (for example, by 10,000 bytes at a time) up to 66,000 or greater.

Solving LONG Data Errors

If LONG or LONG RAW data on your system can become sufficiently large to create an import problem, then you need to make alternative plans for backup or file transfer. One suggestion is to put such columns in a separate table so that all other data can be exported and imported normally.

Object Errors

Object errors can occur for many reasons, some of which are described below. When an object error does occur, import of the current object is discontinued. Import then attempts to continue with the next object in the export file. If COMMIT=N has been specified, a rollback occurs before Import continues. Otherwise, a commit is issued. (See the description of COMMIT on page 2 – 19 for more information.)

Object Already Exists

If an object to be imported already exists in the database, then an object creation error occurs. What happens next depends on the setting of the IGNORE parameter.

If IGNORE=N (the default), the error is reported, and Import continues with the next object. The current object is not replaced. For tables, this behavior means that rows contained in the export file are not imported.

If IGNORE=Y, object creation errors are not reported. Although the object is still not replaced, if the object is a table, rows are imported into it. Note that only *object creation errors* will be ignored, all other errors (operating system, database, SQL, etc.) *will* be reported and processing may stop.



Warning: Specifying IGNORE=Y can cause duplicate rows to be entered into a table unless one or more columns of the table are specified with the UNIQUE integrity constraint. This could occur, for example, if Import were run twice.

Sequences

Before importing an export file into an existing database, sequences should be dropped. A sequence that is not dropped before the import will not be set to the value captured in the export dump file since Import does not drop and re-create a sequence that already exists. If the sequence already exists, then the export file's "create sequence" statement fails and the sequence is not imported.

Resource Errors

Resource limitations can cause objects to be skipped. When importing tables, for example, resource errors can occur as a result of internal problems, or when a resource such as memory has been exhausted.

If a resource error occurs while importing a row, Import stops processing the current table and skips to the next table. If you have specified COMMIT=Y, Import will commit the partial import of the current table. If not, a rollback of the current table will occur before Import continues. (See the description of COMMIT on page 2 – 19 for information about the COMMIT parameter.)

Fatal Errors

When a fatal error occurs, Import terminates. For example, entering an invalid *username/password* combination or attempting to run Export or Import without having prepared the database by running the scripts CATEXP.SQL or CATALOG.SQL will cause a fatal error and Import/Export will terminate.

Privileges Required to Use Import

Importing Objects into Your Own Schema The table below lists the privileges required to import objects into your own schema. All of these privileges initially belong to the RESOURCE role.

Object	Privileges	Privilege Type
clusters	CREATE CLUSTER	system
	And: tablespace quota, or UNLIMITED TABLESPACE	system
database links	CREATE DATABASE LINK	system
	CREATE SESSION on remote db	system
database triggers indexes	CREATE TRIGGER	system
	CREATE INDEX	system
	And: tablespace quota, or UNLIMITED TABLESPACE	system
integrity constraints	ALTER TABLE	object
packages	CREATE PROCEDURE	system
private synonyms	CREATE SYNONYM	system
sequences	CREATE SEQUENCE	system
snapshots	CREATE SNAPSHOT	system
stored functions	CREATE PROCEDURE	system
stored procedures	CREATE PROCEDURE	system
table data	INSERT TABLE	object
table definitions	CREATE TABLE	system
	And: tablespace quota, or UNLIMITED TABLESPACE	system
views	CREATE VIEW	system
	And: SELECT on the base table, or	object
	SELECT ANY TABLE	system

Table 2 – 1 Privileges Required to Import Objects into Your Own Schema

Notes:

- Table definitions include comments and audit options.
- If using the Trusted Oracle7 Server, you must have write access to all labels for which you are importing data. See the *Trusted Oracle7 Server Administrator’s Guide* for more information.

Importing Grants

To be able to import the privileges that a user has granted to others, the user initiating the import must either own the objects or have object privileges with the WITH GRANT OPTION. The following table shows the required conditions for the authorizations to be valid on the target system:

Grant	Conditions
object privileges	Object must exist in the user's schema, <i>or</i> user must have the object privileges with the WITH GRANT OPTION.
system privileges	User must have system privileges as well as the WITH ADMIN OPTION.

Table 2 – 2 Privileges Required to Import Grants

Importing Objects into Other Schemas

To import objects into another user's schema, you must have the IMP_FULL_DATABASE role enabled.

Importing System Objects

To import system objects from a full database export file, the IMP_FULL_DATABASE role must be enabled. The import parameter FULL specifies that system objects are included in the import.

The system objects are:

- profiles
- public database links
- public synonyms
- roles
- rollback segment definitions
- system audit options
- system privileges
- tablespace definitions
- tablespace quotas
- user definitions

Additional Information: To perform a full database import in Trusted Oracle7 Server, your operating system label must be equivalent to DBHIGH. See the *Trusted Oracle7 Server Administrator's Guide* for more information.

User Privileges

When user definitions are imported into an Oracle database, they are created with the CREATE USER command. So, when importing from export files created by previous versions of Export, users are *not* granted CREATE SESSION privileges automatically.

Import and Stored Procedures, Functions, and Packages

When a local stored procedure, function, or package is imported, it retains its original timestamp. If the timestamp of the imported version differs from the timestamp of the version currently in the database, it is marked for recompilation. If the timestamp matches, it is not marked and it will not be recompiled.

Recompilation of a local procedure, function, or package does not occur until it is invoked or until the `COMPILE ALL` command is used to recompile all marked procedures, functions, and packages. For more information, see the *Oracle7 Server Application Developer's Guide* for more information about the `COMPILE ALL` command.

If a local procedure that has not been recompiled is invoked by a remote package or procedure it is recompiled then. However, the timestamp of the local procedure then changes, causing an error in the remote procedure that called it.

Remote procedures are not recompiled at import time; while this functionality prevents unnecessary recompilation, it does mean that you will have to ensure that remote procedures that have been imported are recompiled.

Import and Snapshots

There are four interrelated objects in a snapshot system: the master table, optional snapshot log, the master table trigger that updates the snapshot log, and the snapshot itself. All of these objects are exported normally. The master table trigger, if it exists, is exported as part of the master table. The tables (master table, snapshot log table definition, and snapshot tables) can be exported independently of one another. This section discusses how fast refreshes are affected when these objects are imported.

Master Table

The imported data is recorded in the snapshot log if the master table already exists for the database being imported to and it has a snapshot log.

Master Table Trigger

The snapshot log update trigger, if it exists, is exported as part of the master table. Similarly, it is automatically imported from the export file. Because all triggers are imported after all tables, the update trigger is imported after the master table and snapshot log.

Snapshot Log

When a snapshot log is exported, only the table definition is placed in the export file, because the master table ROWIDs stored in the snapshot log would have no meaning upon import. Therefore, an imported snapshot log will be empty. As a result, each snapshot's first attempt to do a fast refresh will fail, generating an error indicating that a complete refresh is required.

To avoid the refresh error, do a complete refresh after importing a snapshot log. After doing a complete refresh, subsequent fast refreshes will work properly.

Snapshots

A snapshot that has been restored from a backup file has “gone back in time” to a previous state. On import, the time of the last refresh is imported as part of the snapshot table definition. The function that calculates the next refresh time is also imported.

Each refresh leaves a signature. A fast refresh uses the log entries that date from the time of that signature to bring the snapshot up to date. When the fast refresh is complete, the signature is deleted and a new signature is created. Any log entries that are not needed to refresh other snapshots are also deleted (all log entries with times before the earliest remaining signature).

Importing a Backup

When restoring a dropped snapshot from a backup, there are two potential problems:

- If a snapshot is refreshed at time A, exported at time B, and refreshed again at time C, then the re-imported version has the last refresh time recorded as time A. Log entries needed for a fast refresh may no longer exist. If the log entries do exist (because they are needed for another snapshot that has yet to be refreshed), then they are used, and the fast refresh completes successfully. Otherwise, the fast refresh fails, generating an error that says a complete refresh is required.
- Whether the fast refresh succeeds or fails, the signature left at time C remains. All log entries after time C will remain in the log, waiting for a refresh that never occurs. This situation causes the log to grow continuously — it does not shrink, even when all snapshots have been refreshed. Subsequent fast refreshes fail, generating an error that says a complete refresh is required. This problem can be solved by purging the snapshot log.



Suggestion: Both of these problems can be avoided by exporting backup copies of snapshots immediately after doing a fast refresh. If out of date backups must be imported, do a complete refresh at the earliest opportunity.

Importing into an Existing Snapshot

If snapshot rows have been lost, and the snapshot is restored from a backup without dropping it; other problems may result. If the snapshot is exported at time A and a refresh occurs at time B, the snapshot's signature reflects time B, but data is imported as of time A.

Subsequent fast refreshes update from time B, not from time A. So changes logged in the master table between time A and B are not reflected in the snapshot.

This situation can only occur if IGNORE=Y is specified for the import. (Otherwise, the snapshot rows are not imported.) As a result, the situation is undetectable — the IGNORE setting prevents an object creating error from being issued.



Suggestion: To avoid such problems, export backup copies of snapshots immediately after doing a fast refresh. If out of date backups must be imported, do a complete refresh at the earliest opportunity.

Second Copy of a Snapshot

If a snapshot is imported into two databases or into two different schemas, then there are two copies of the snapshot, but only one signature.



Warning: The first snapshot to do a fast refresh recognizes the signature and gets the expected results. When a fast refresh is performed for the second snapshot the signature will be missing, causing an error because a complete refresh is required. After the complete refresh is performed, the second snapshot will have its own signature and subsequent fast refreshes will work properly.

Character Set Conversion

The export file identifies the character encoding scheme used for the character data in the file. If that character set is any single-byte character set (for example, EBCDIC or USASCII7), and if the character set used by the target database is also a single-byte character set; then the data is automatically converted to the character encoding scheme specified for the user session during import, as specified by the NLS_LANG parameter. After the data has been converted to the session character set, it is then converted to the database character set.

During the conversion, any characters in the export file that have no equivalent in the target character set are replaced with a default character. (The default character is defined by the target character set.)

To guarantee 100% conversion, the target character set should be a superset or equivalent of the source character set.

For multi-byte character sets, conversion is only performed if the length of the character string cannot change as a result of the conversion.

For more information, refer to the National Language Support section of the *Oracle7 Server Reference*.

Import Modes

The objects that are imported depend on the Import mode you choose for import and the mode chosen during export. All users have two choices of import mode. A user with the IMP_FULL_DATABASE role (a *privileged user*) has three choices:

Table	This mode allows you to import specified tables in your schema, rather than all your tables. A privileged user can qualify the tables by specifying the schema that contains them. The default is to import all tables in the schema of the user doing the import.
User	This mode allows you to import all objects that belong to you (such as tables, data, grants, and indexes). A privileged user importing in user mode can import all objects in the schemas of a specified set of users.
Full Database	Only users with the IMP_FULL_DATABASE role can import in this mode. All objects in the export file are imported.

To select table, user, or full database mode, specify TABLES=*tablelist*, FROMUSER=*userlist*, or FULL=Y, respectively. A user with the IMP_FULL_DATABASE role must specify one of these options, or else an error results. If a user without the IMP_FULL_DATABASE role fails to specify one of these options, then a user-level import is performed.

Table 2 – 3 shows objects that are imported in each mode and the order in which they are imported.

Table Mode	User Mode	Full Database Mode
For each table in the TABLES list:	For each user in the Owner's list:	All database objects except for those owned by SYS:
table definitions (1)	snapshots	tablespace definitions
table data	snapshot logs	profiles
table constraints	job queues	user definitions
owner's table grants (2)	refresh groups and children	roles
owner's table indexes (3)	database links	system privilege grants
analyze tables	sequence numbers	role grants
column comments	cluster definitions	default roles
audit	For each table that the user owns:	tablespace quotas
table referential constraints	table definitions (1)	resource costs
table triggers	table data	rollback segment definitions
	table constraints	database links
	table grants	sequence numbers
	table indexes (4)	all snapshots
	analyze table	all snapshot logs
	column comments	all job queues
	audit	all refresh groups and children
	private synonyms	all cluster definitions
	user views	table definitions (1)
	user stored procedures, packages, and functions	table data
	analyze cluster	table constraints
	referential constraints	table grants
	triggers	table indexes
	postable actions	analyze table
		column comments
		audit
		referential integrity constraints
		postable actions
		all synonyms
		all views

Table Mode	User Mode	Full Database Mode
		all stored procedures, packages, and functions all triggers analyze cluster default and system auditing
Notes: 1. Table definitions include comments and audit options. 2. Owner's grants for the tables are exported in table mode. 3. Owner's indexes on the specified tables are exported in table mode. 4. Only indexes on the user's tables are exported.		

Table 2 – 3 Exported Objects

Using Import

Before Using Import

To use Import, the script CATEXP.SQL must be run. After creating the database, do one of the following:

- run CATEXP.SQL
- run CATALOG.SQL (which runs CATEXP.SQL)



Additional Information: The actual names of the script files operating system dependent. The script file names and the method for running them are described in your Oracle operating system–specific documentation.

CATEXP.SQL only needs to be run once. Once run, it does not need to be run again before future imports. CATEXP.SQL performs the following operations to prepare the database for Import:

- assigns all necessary privileges to the IMP_FULL_DATABASE role
- assigns IMP_FULL_DATABASE to the DBA role

Note: You can import any export file into a Trusted Oracle7 database. For example, if you create an export file from an OS MAC database, you can import that file into a DBMS MAC database.

Invoking Import

You can invoke Import in three ways:

- Enter the command `IMP username/password PARFILE=filename`. `PARFILE` is a file containing the Import parameters you typically use. If you use different parameters for different databases, you can have multiple `PARFILES`. This is the preferred method.
- Enter the command `IMP username/password` followed by various parameters you intend to use. Note that the number of parameters cannot exceed the maximum length of a command line on your operating system.
- Enter the command `IMP username/password` to begin an interactive session, and let Import prompt you for the information it needs. The interactive option does not provide as much functionality as the parameter-driven method. It exists only for backward compatibility.

The username and password can also be specified in the parameter file, although, for security reasons, it is not recommended that you do so.

If you omit `username/password`, Import will prompt for it.

Getting Online Help

Import provides online help. Enter `IMP HELP=Y` on the command line to see a help screen like the one shown in Figure 2 – 2:

```
You can let Import prompt you for parameters by entering the IMP
command followed by your username/password:

Example: IMP SCOTT/TIGER

Or, you can control how Import runs by entering the IMP command followed
by various arguments. To specify parameters, you use keywords:

Format: IMP KEYWORD=value or KEYWORD=(value1,value2,...,valueN)
Example: IMP SCOTT/TIGER IGNORE=Y TABLES=(EMP,DEPT) FULL=N
```

Keyword	Description (Default)	Keyword	Description (Default)
USERID	username/password	FULL	import entire file (N)
BUFFER	size of data buffer	FROMUSER	list of owner usernames
FILE	output file (EXPDAT.DMP)	TOUSER	list of usernames
SHOW	just list file contents (N)	TABLES	list of table names
IGNORE	ignore create errors (N)	RECORDLENGTH	length of IO record
GRANTS	import grants (Y)	INCTYPE	incremental import type
INDEXES	import indexes (Y)	COMMIT	commit array insert (N)
ROWS	import data rows (Y)	PARFILE	parameter filename
LOG	log file of screen output		
DESTROY	overwrite tablespace data file (N)		
INDEXFILE	write table/index info to specified file		
CHARSET	character set of export file (NLS_LANG)		

Figure 2 – 2 Import Help Screen

The Parameter File

The parameter file allows you to specify Import parameters in a file where they can be easily modified or reused. Create a parameter file using any flat file text editor. The command line option `PARFILE=<filename>` tells Import to read the parameters from the specified file rather than from the command line.

For example:

```
IMP PARFILE=filename  
IMP username/password PARFILE=filename
```

The syntax for parameter file specifications is:

```
KEYWORD=value
```

or

```
KEYWORD=(value)
```

or

```
KEYWORD=(value1, value2, ...)
```

The following is an example of a partial parameter file listing:

```
FULL=Y  
FILE=DBA.DMP  
GRANTS=Y  
INDEXES=Y
```



OSDoc

Additional Information: The maximum size of the parameter file may be limited and operating system file naming conventions will apply. See your Oracle operating system-specific documentation for more information.

Comments

You can add comments to the parameter file by preceding them with the # sign. All characters to the right of the # sign are ignored.

Table Name Restrictions

Table names specified on the command line cannot include a # sign, unless the table name is enclosed in quotation marks.

Similarly, in a parameter file, if a table has a # sign in the name, the rest of the line is interpreted as a comment unless the table name is enclosed in quotation marks.



OSDoc

Additional Information: Some operating systems require single vs. double quotes. See your Oracle operating system-specific documentation.

For example, if a parameter file contains the line

```
TABLES=(EMP#, DEPT, MYDATA)
```

then nothing after EMP# is seen as input by Import. As a result, DEPT and MYDATA are not imported.

The following modification fixes the problem:

```
TABLES=( "EMP#" , DEPT, MYDATA)
```



Attention: When the name is specified in quotation marks, it is case-sensitive. The name must then exactly match the table name stored in the database. By default, database names are stored as uppercase.

Import Parameters

The following parameters can be specified in the parameter file. They are described in detail in the remainder of this section.

USERID	RECORDLENGTH
BUFFER	INCTYPE
FILE	COMMIT
SHOW	HELP
IGNORE	LOG
GRANTS	DESTROY
INDEXES	INDEXFILE
ROWS	CHARSET
FULL	FEEDBACK
FROMUSER	MLS*
TOUSER	MLS_LISTLABELS*
TABLES	MLS_MAPFILE*

**Trusted Oracle7 Server parameter.*

BUFFER Default: operating system dependent

The BUFFER (buffer size) parameter determines the number of rows in the array inserted by Import. The following formula gives a rough approximation of the buffer size that inserts a given array of rows:

*buffer_size = rows_in_array * maximum_row_size*

The size in bytes of the buffer through which data rows are transferred.



OSDoc

Additional Information: See your operating Oracle system-specific documentation to determine the default value for this parameter.

CHARSET	<p>Default: none</p> <p><i>Release 6 export files only</i></p> <p>Specifies the NLS character set used in the export file.</p> <p>Use of this parameter is not recommended. It is only provided for compatibility with previous versions. Eventually, it will no longer be supported.</p> <p>If you are using Oracle7 Server's Export, the character set is specified within the export file, and conversion to the current database's character set is automatic. Specification of this parameter serves only as a check to ensure that the export file's character set matches the expected value. If not, an error results.</p> <p>Valid values for CHARSET are contained in the V\$NLS_PARAMETERS view. Consult the <i>Oracle7 Server Reference</i> for details.</p>
COMMIT	<p>Default: N</p> <p>Specifies whether Import should commit after each array insert. By default, Import commits after loading each object and Import will perform a rollback when an error occurs before continuing with the next object.</p> <p>Specifying COMMIT=Y prevents rollback segments from growing inordinately large and improves the performance of large imports. Enabling this parameter is advisable if the table has a uniqueness constraint. If the import is restarted, any rows that have already been imported will be rejected with a non-fatal error. Note that, if there is no uniqueness constraint, enabling this parameter could produce duplicate rows when re-importing the data.</p>
DESTROY	<p>Default: N</p> <p>Specifies whether the existing data files making up the database should be reused. That is, the DESTROY option specifies that IMPORT should include the reuse option in the datafile clause of the CREATE TABLESPACE command.</p> <p>The export file contains the datafile names used in each tablespace. Attempting to create a second database on the same machine (for testing or other purposes) has the undesirable effect of overwriting the original database's data files when creating the tablespace. With this parameter set to N (the default), an error occurs if the data files already exist when the tablespace is created.</p>

To eliminate this error when importing into a secondary database, pre-create the tablespace and specify its data files. (Specifying IGNORE=Y suppresses the object creation error that the tablespace already exists.)

To bypass the error when importing into the original database, specify IGNORE=Y to add to the existing data files without replacing them. To reuse the original database's data files after eliminating their contents, specify DESTROY=Y.

Note that, if you have pre-created your tablespace, you should specify DESTROY=N or your pre-created tablespace will be lost.

FEEDBACK Default: 0 (zero)

If set to other than zero, specifies that Import should display a progress meter in the form of a dot for x number of rows imported. For example, were you to specify FEEDBACK=10, import would display a dot each time 10 rows had been imported. The FEEDBACK value applies to all tables being imported, it cannot be set on a per table basis.

FILE Default: EXPDAT.DMP

The name of the Export file to import.

FROMUSER Default: none

A list of schemas containing objects to import. The default is a user mode import. That is, all objects for the current user are imported. (If the TABLES parameter is also specified, then a table mode import is performed.)

When importing in user mode, all other objects in the export file are ignored. The effect is the same as if the export file had been created in user mode (or table mode). See page 1 – 6 for the list of objects that are imported in user mode and table mode.

For example, the following command treats the export file as though it were simply a user mode export of SCOTT's objects:

```
IMP system/manager FROMUSER=scott
```

If user SCOTT does not exist in the current database, then his objects are imported into the importer's schema — in this case, the system manager's. Otherwise, the objects are installed in SCOTT's schema. If a list of schemas is given, each schema can be specified only once. Duplicate schema names are ignored.

Note: Specifying `FROMUSER=SYSTEM` does **not** import system objects. It imports only those objects that belong to user `SYSTEM`.

To import system objects (for example, user definitions and tablespaces), you must import from a full export file specifying `FULL=Y`.

FULL	<p>Default: N</p> <p>Specifies whether to import the entire export file or not. This parameter can be specified only by a user with the <code>IMP_FULL_DATABASE</code> role enabled.</p>
GRANTS	<p>Default: Y</p> <p>Specifies whether to import grants or not.</p>
HELP	<p>Default: N</p> <p>Displays a description of import parameters.</p>
IGNORE	<p>Default: N</p> <p>Specifies how object creation errors should be handled. Specifying <code>IGNORE=Y</code> causes Import to overlook object creation errors when attempting to create database objects. For objects other than tables, if you specify <code>IGNORE=Y</code>, Import continues without reporting the error. If you accept the default <code>IGNORE=N</code>, Import logs and/or displays the object creation error before continuing.</p> <p>For tables, <code>IGNORE=Y</code> causes rows to be imported into existing tables. No message is given. <code>IGNORE=N</code> causes an error to be reported, and the table is skipped if it already exists.</p> <p>Note that only <i>object creation errors</i> will be ignored, operating system, database, SQL, etc. errors <i>will not</i> be ignored and may cause processing to stop.</p> <p>In situations where multiple refreshes from a single export file are done with <code>IGNORE=Y</code>, certain objects can be created multiple times (although they will have unique system-defined names). This can be prevented for certain objects, for example, constraints, by doing an export in table mode with the <code>CONSTRAINTS=NO</code> option. Note that, if a full export is done with the <code>CONSTRAINTS</code> option set to <code>NO</code>, no constraints for any tables will be exported. See page 1 – 12 for information about which objects you can prevent from being exported.</p>



Warning: When importing into existing tables, if no column in the table is uniquely indexed, then rows could be duplicated if they were already present in the table. (This warning applies to non-incremental imports only. Incremental imports replace the table from the last complete export and then rebuild it to its last backup state from a series of cumulative and incremental exports.)

INCTYPE Default: undefined

Specifies the type of incremental import. Valid values are SYSTEM and RESTORE. See the section called “Importing Incremental, Cumulative and Complete Export Files” later in this chapter for more information.

INDEXES Default: Y

Specifies whether to import indexes or not.

INDEXFILE Default: none

Specifies a file to receive index-creation commands.

When this parameter is specified, index-creation commands are extracted and written to the specified file, rather than used to create indexes in the database. Tables and other database objects are not imported.

The file can then be edited (for example, to change storage parameters) and used as a SQL script to create the indexes. This is the most efficient way to create indexes for new tables—even if the index file is not edited. To make it easier to identify the indexes defined in the file, the export file’s CREATE TABLE statements and CREATE CLUSTER statements are included as comments.


Note: As of release 7.1, the commented CREATE TABLE statement in the indexfile no longer includes primary/unique key clauses.

Perform the following steps to make use of this feature:

1. Import using the INDEXFILE command to create a file of index-creation commands.
2. Edit the file making certain to add a valid password to the CONNECT string.
3. Rerun Import, specifying INDEXES=N.

This step imports the database objects while preventing Import from using the index definitions stored in the export file.

4. Run the file of index-creation commands to create the index.

LOG	<p>Default: none</p> <p>Specifies a file to receive informational and error messages. If specified, all information written to the terminal display is also written to the log.</p>
MLS	<p>Used when importing data into a secure database. Specifies that the export file contains Multi-Level Security (MLS) labels. For more information on this parameter, see the <i>Trusted Oracle7 Server Administrator's Guide</i>.</p>
MLS_LISTLABELS	<p>When specified, lists the MLS labels in the export file. For more information on this parameter, see the <i>Trusted Oracle7 Server Administrator's Guide</i>.</p>
MLS_MAPFILE	<p>When specified, maps the MLS labels in the export file to the labels used in the target database for the import. For more information, see the <i>Trusted Oracle7 Server Administrator's Guide</i>.</p>
RECORDLENGTH	<p>Default: operating system dependent</p> <p>Specifies the length in bytes of the file record. The RECORDLENGTH parameter is required when you import to another operating system that uses a different default value.</p> <div>  <p>Additional Information: See your Oracle operating system-specific documentation to determine the default value for this parameter.</p> </div>
ROWS	<p>Default: Y</p> <p>Specifies whether to import the rows of table data or not.</p>
SHOW	<p>Default: N</p> <p>When specified, the contents of the export file will be listed to the display and not imported. SHOW is not typically used with other parameters.</p>

TABLES Default: none

Specifies a list of table names to import. Use an asterisk (*) to indicate all tables. When specified, this parameter initiates a table mode import, which restricts the import to tables and their associated objects, as listed on page 1 – 6. The number of tables that can be specified at the same time is dependent on command line limits and buffer settings. Although you can qualify table names with schema names (as in SCOTT.EMP) when exporting, you *cannot* do so when importing. In the following example, the TABLES parameter is specified incorrectly:

```
IMP system/manager TABLES=(jones.accts, scott.emp,scott.dept)
```

The valid specification to import these tables should be:

```
IMP system/manager FROMUSER=(scott,jones) TABLES=(accts,emp,dept)
```

If user SCOTT does not exist in the current database, then his tables are imported into the importer's schema — in the above example, the system/manager's. Otherwise, the tables and associated objects are installed in SCOTT's schema.

TOUSER Default: none

Specifies a list of usernames to whose schemas data will be imported. The IMP_FULL_DATABASE role is required to use this parameter.

To import to a different schema than the one that originally contained the object, specify TOUSER. For example:

```
IMP system/manager FROMUSER=scott TOUSER=joe TABLES=emp
```

If multiple schemas are specified, then the schema names are paired. For example,

```
IMP system/manager FROMUSER=(scott,fred) TOUSER=(joe, ted)
```

imports SCOTT's objects into JOE's schema, and FRED's objects into TED's schema.

USERID Default: undefined

The *username/password* of the user performing the import.

Import's Interactive Mode

Starting Import from the command line with no arguments will initiate Import's interactive mode. Note that the interactive mode does not provide prompts for all of Import's functionality. It is provided only for backward compatibility.

You may not see all the prompts in a given Import session because some prompts depend on your responses to other prompts. Some prompts show a default answer; if the default is acceptable, press [RETURN]. To end your input, enter a period (.) followed by [RETURN].

If you have not specified a *username/password* on the command line, Import first prompts you for this information. Then the following prompts are displayed:

Import's Interactive Mode Prompts

```
Import file: EXPDAT.DMP >
```

Default: EXPDAT.DMP

Specify the name of the export file to be imported. You do not have to be the Oracle user who exported the file. However, you do need to have current access to the file.

```
Enter insert buffer size (minimum is 4096): 10240 >
```

Default: 10240

Specify the buffer size for your tables. You may need to increase the buffer size for tables that have LONG columns. The buffer must be at least as large as the longest row.

```
List contents of import file only (Y/N): N >
```

Default: N

When specified, allows you to see what is in the export file (for example, a particular table exported to the export file). Equivalent to the command-line parameter SHOW.

If you specify Y, Import will not import the rows but will prompt you whether to display the entire export file or only portions of it. Import can also display the SQL statements contained in the export file in the order they would be executed on import.

Ignore create errors due to object existence (Y/N): N >

Default: N

Import will generate an error if an object to be imported already exists. This prompt allows you to specify how you want the errors to be handled.

If you accept the default (N) Import will issue an error message each time it encounters a table that already exists and will not import that table's data.

Specifying Y has two effects on the import process. If Import encounters non-table objects, it skips that object and does not issue an error message. If Import encounters a pre-existing table, it imports the rows into the table.

For example, you may want to import data into a pre-existing table — perhaps because you want to use unique storage parameters, or because you have pre-created the table in a cluster. You can do so by specifying Y at this prompt. The rows of data will be imported into the pre-existing table.

Import grants (Y/N): Y >

Default: Y

By default, any grants that were exported will be imported. If the export was done in user mode, only first-level grants (those granted by the owner) are in the export file. If the export was done in full database mode, all grants are in the export file, including lower-level grants (those granted by users given a privilege with the WITH GRANT option). Specifying N will prevent any grants from being imported.

Import table data (Y/N): Y >

Default: Y

By default, Import will import the data rows. However, specifying N, causes Import to execute such SQL statements as CREATE TABLE or CREATE VIEW, but does not insert any data rows. Indexes and grants *are* created.

Import entire export file (Y/N): Y >

Default: Y

By default, Import will import the entire export file. By specifying N at this prompt, you can choose specific tables for import. If you choose to

import specific tables, you will need to specify the tables at the next prompt.

Username:

Enter table names. Null list means all tables for user

Enter table name or . if done:

If you specified N at the previous prompt, Import will prompt you for a schema name and the tables names you want to import for that schema. Entering a null table list causes all tables in the schema to be imported. Only one schema at a time can be specified in interactive mode.

Example Import Sessions

This section gives some examples of import sessions that show you how to use the command-line and interactive methods. The examples illustrate three scenarios:

- tables imported by an administrator into the same schema from which they were exported.
- tables imported by a user from another schema into the user's own schema.
- tables imported into a different schema by an administrator.

Example Import of Selected Tables for a Specific User

In this example, using a full database export file, an administrator imports selected tables from a specific schema. With the command-line method, you specify FROMUSER=(*schema*) and TABLES=(*tablename*). If *schema* is not specified, it defaults to the schema from which the previous table was imported.

If there is not a previous table, it defaults to the importer's schema. In the following example, *schema* defaults to SCOTT for table DEPT.

Command Line Method

```
imp system/manager file=dba.dmp fromuser=scott tables="(emp,dept)"
```

Interactive Method

```
imp system/manager
...
Import file: expdat.dmp >
Enter insert buffer size (minimum is 4096) 30720>
```

```
Export file created by EXPORT:V07.01.03
```

```
List contents of import file only (yes/no): no >
Ignore create error due to object existence (yes/no): yes >
Import grants (yes/no): yes >
Import table data (yes/no): yes >
Import entire export file (yes/no): yes > no
```

```
Username: scott
Enter table names. Null list means all tables for user
Enter table name or . if done: dept
Enter table name or . if done: emp
Enter table name or . if done: .
```

Import Messages

```
Export file created by EXPORT:V07.01.03
```

```
Warning: the objects were exported by SCOTT, not by you
```

```
. importing SCOTT's objects into SYSTEM
. . importing table "DEPT"                      7 rows imported
. . importing table "EMP"                      22 rows imported
Import terminated successfully without warnings.
```

Example Import of Tables Exported by Another User

This example illustrates importing selected tables from a file exported by another Oracle user.

Command Line Method

```
imp adams/ez4me file=scott.dmp tables="(emp,dept)"
```

Interactive Method

```
imp adams/ez4me
...
Import file: expdat.dmp > scott.dmp
Enter insert buffer size (minimum is 4096) 30720>
Export file created by EXPORT:V07.01.03
```

```
Warning: the objects were exported by SCOTT, not by you
```

```
List contents of import file only (yes/no): no >
Ignore create error due to object existence (yes/no): yes >
Import grants (yes/no): yes > n
Import table data (yes/no): yes >
Import entire export file (yes/no): yes > n
Username: scott
Enter table names. Null list means all tables for user
Enter table name or . if done: emp
Enter table name or . if done: dept
Enter table name or . if done: .
```

Import Messages

Export file created by EXPORT:V07.01.03

Warning: the objects were exported by SCOTT, not by you

```
. importing SCOTT's objects into ADAMS
. . importing table "EMP"                22 rows imported
. . importing table "DEPT"              7 rows imported
Import terminated successfully without warnings.
```

Example Import of Tables from One User to Another

In this example, a DBA imports all tables belonging to one user into another user's account. The command-line interface is required for this case, because the TOUSER parameter *cannot* be specified interactively.

Command Line Method

```
imp system/manager file=scott.dmp fromuser=scott touser=rosemary
tables=(*)                [use an asterisk to specify all tables]
```

Import Messages

Export created by Oracle version EXPORT:V7.2

```
.. importing table "BONUS"                32 rows imported
.. importing table "SALGRADE"            10 rows imported
```

Importing Incremental, Cumulative and Complete Export Files

Since an incremental export extracts only tables that have changed since the last incremental, cumulative, or complete export, an import from an incremental export file imports the table's definition and all its data, *not just the changed rows*. Such exports are typically done more often than cumulative or complete exports.

Since imports from incremental export files are dependent on the method used to export the data, you should also read the section on page 1 – 24 called "Incremental, Cumulative and Complete Exports".

It is important to note that, since importing an incremental export file imports new versions of existing objects, existing objects are dropped before new ones are imported. This behavior differs from a normal import. During a normal import, objects are not dropped and an error is generated if the object already exists.

Note: Imports from incremental export files can only be applied to an entire database. There is no user-mode or table-mode for such an import. Any user with the BECOME_USER system privilege can do an import from an

incremental export file. This privilege is contained in the IMP_FULL_DATABASE role.

Command Syntax

```
➤ IMP - username/password - INCTYPE= SYSTEM  
RESTORE ➤
```

where:

SYSTEM	Imports the most recent version of system objects (except those owned by SYS) using the most recent incremental export file. A SYSTEM import does not import user data or objects.
RESTORE	Imports all user database objects and data that have changed since the last export using export files in chronological order.

Restoring a Set of Objects

The order in which incremental, cumulative and complete exports are done is important. A set of objects cannot be restored until a complete export/import has been run on a database. Once that has been done, the process of restoring objects would follow the steps listed below.

Note: To restore a set of objects, you must first import the *most recent* incremental export file to import the system objects (i.e. specify INCTYPE=SYSTEM for the export). Then you must import the export files in chronological order, based on their export time (i.e. specify INCTYPE=RESTORE for the import).

1. Import the most recent incremental export file (specify INCTYPE=SYSTEM for the export) or cumulative export file, if no incremental exports have been taken.
2. Import the most recent complete export file.
3. Import all cumulative export files after the last complete export.
4. Import all incremental export files after the last cumulative export.

For example, if you have

- one complete export called X1
- two cumulative exports called C1 and C2
- three incremental exports called I1, I2, and I3

then you should import in the following order:

```
IMP system/manager INCTYPE=SYSTEM FULL=Y FILE=I3  
IMP system/manager INCTYPE=RESTORE FULL=Y FILE=X1  
IMP system/manager INCTYPE=RESTORE FULL=Y FILE=C1
```

```
IMP system/manager INCTYPE=RESTORE FULL=Y FILE=C2
IMP system/manager INCTYPE=RESTORE FULL=Y FILE=I1
IMP system/manager INCTYPE=RESTORE FULL=Y FILE=I2
IMP system/manager INCTYPE=RESTORE FULL=Y FILE=I3
```

Notes:

- You import the last incremental export file twice; once at the beginning to import the most recent version of the system objects, and once at the end to apply the most recent changes made to the user data and objects.
- When restoring tables with this method, you should drop the tables from the database (if they exist) before starting the import sequence. This step prevents you from importing duplicate rows.

Importing Into Existing Tables

Manually Creating Tables before Importing Data

When you choose to create tables manually before importing data into them from an export file, you should either use the same table definition as was previously used or a compatible format. For example, while you can increase the width of columns and change their order, you cannot add NOT NULL columns or change the datatype of a column to an incompatible datatype (LONG to NUMBER, for example).

Disabling Referential Constraints

In the normal import order, constraints are imported only after all tables are imported. This sequence prevents the errors that could occur if a referential integrity constraint existed for data that has not yet been imported.

These errors can still occur when data is loaded into existing tables, however. For example, if table EMP has a referential integrity constraint on the MGR column that verifies the manager number exists in EMP, then a perfectly legitimate employee row might fail the referential integrity constraint if the manager's row has not yet been imported. When such an error occurs, Import generates an error message, bypasses the failed row, and continues importing other rows in the table. You can disable constraints manually to avoid this.

Similarly, a referential check from table AEMP into table BDEPT using DEPTNO would cause rows to fail, because the tables are loaded in alphabetic order, and rows from AEMP would be loaded before the corresponding rows in BDEPT.

To prevent errors like these, it is a good idea to disable referential integrity constraints when importing data into existing tables.

Manually Ordering the Import

When the constraints are re-enabled after importing, the entire table is checked, which may take a long time for a large table. If the time required for that check is too long, it may be beneficial to order the import manually.

To do so, do several imports from an export file instead of one. First, import tables that are the targets of referential checks, before importing the tables that reference them. Provided tables do not reference each other in circular fashion, and provided a table does not reference itself, this option works.

Generating Statistics on Imported Data

An export parameter controls the generation of database optimizer statistics during import. To get statistics, specify one of the following parameters on export:

```
STATISTICS=COMPUTE  
STATISTICS=ESTIMATE
```

When one of these options is specified during the export, all database objects that have had ANALYZE applied to them are exported with the commands necessary to generate the appropriate statistics (estimated or computed) on import.

Note: Generation of statistics is limited to those objects that already had them before export. So statistics are not suddenly generated on every object in the database.

If your installation generally uses either estimated or computed statistics, then it is a good idea to include one of these commands whenever you use Export. The cost during export is negligible — statistics are not recorded in the export file, only a command to generate them. See Chapter 13 of the *Oracle7 Server Concepts* manual for more information about the optimizer.

By issuing this command during export, you ensure that the appropriate statistics are gathered when the data is imported. If your export file was created without this command, or if you have changed your method of collecting statistics, use Import's INDEXFILE option to generate a list of imported objects. Then edit that list to produce a series of ANALYZE commands on them. (For more information, see the INDEXFILE parameter on page 2 – 22).

Importing LONGs



Warning: LONG columns can be up to 2 gigabytes in length. Because they can be exported in sections, Oracle LONGs are always exportable. On import, however, LONGs require contiguous memory. Therefore, it is not always possible to export LONG columns from one operating system and import them on another system. Even on the same system, memory limitations may make it impossible to import very large LONG columns that were successfully exported on that system.

Network Considerations

This section describes factors to take into account when using Export and Import across a network.

Transporting Export Files Across a Network

When transferring an export file across a network, be sure to transmit the file using a protocol that preserves the integrity of the file. For example, using FTP or a similar file transfer protocol, transmit the file in *binary* mode. Transmitting export files in character mode causes errors when the file is imported.

Exporting/Importing with SQL*Net

By eliminating the boundaries between different machines and operating systems on a network, SQL*Net provides a distributed processing environment for Oracle7 Server products. SQL*Net lets you export and import over a network.

For example, running Export locally, you can write data from a remote Oracle database into a local export file. Running Import locally, you can read data into a remote Oracle database.

To use Export or Import with SQL*Net, include the `@ connect_string` clause when entering the EXP or IMP command. For the exact syntax of this clause, see the user's guide for your SQL*Net protocol. For more information on SQL*Net, see *Understanding SQL*Net*. If you are using Oracle Names see the *Oracle Names Administrator's Guide*.

Dropping a Tablespace

You can drop a tablespace by doing a full database export, then creating a zero-block tablespace with the same name (before logging off) as the tablespace you want to drop. During import, the relevant CREATE TABLESPACE command will fail and prevent the creation of the unwanted tablespace. All objects from that tablespace will be imported into their owner's default tablespace.

Reorganizing Tablespaces

If a user's quotas allow it, the user's tables are imported into the same tablespace from which they were exported. However, if the tablespace no longer exists or the user does not have the necessary quota, the system uses the default tablespace for that user.

If the user is unable to access the default tablespace, then the tables cannot be imported. This scenario can be used to move user's tables from one tablespace to another.

For example, you need to move JOE's tables from tablespace A to tablespace B after a full database export. Follow these steps:

1. If JOE has the UNLIMITED TABLESPACE privilege, revoke it. Set JOE's quota on tablespace A to zero. Also revoke all roles that might have such privileges or quotas.

Note: Role revokes do not cascade. Therefore, users who were granted other roles by JOE will be unaffected.

2. Export JOE's tables.
3. Drop JOE's tables from the tablespace.
4. Give JOE a quota on tablespace B and make it the default tablespace.
5. Create JOE's tables in tablespace B.
6. Import JOE's tables. (By default, Import puts JOE's tables in tablespace B.)

Note: An index on the table is created in the same tablespace as the table itself, unless it already exists.

Overriding Storage Parameters

Export files include table storage parameters, so you may want to pre-create large tables with the different storage parameters before importing the data. If so, you must either specify:

```
IGNORE=Y
```

at the command line or in the parameter file or, if using Import's interactive mode, respond "Yes" to the prompt:

```
Ignore create errors due to object existence (Y/N) : Y >
```

Note: The storage parameter OPTIMAL is not exported for rollback segments and therefore cannot be imported at any time.

Reducing Database Fragmentation

A database with many non-contiguous, small blocks of free space is said to be fragmented. A fragmented database should be reorganized to make space available in contiguous, larger blocks. You can reduce fragmentation by performing a full database export/import as follows:

1. Do a full database export (FULL=Y) to back up the entire database.
2. USE the MONITOR command in SQL*DBA or Server Manager to check for active database users. Do not shut down Oracle until all users are logged off.
3. Recreate the database using the CREATE DATABASE command.
4. Do a full database import (FULL=Y) to restore the entire database.

See the *Oracle7 Server Administrator's Guide* for more information about creating databases.

Export/Import Read-Only Tablespaces

Read-only tablespaces can be exported. On import, if the tablespace does not already exist in the target database, then the tablespace is created as a read/write tablespace. If you want read-only functionality, you must manually make the tablespace read-only after the import.

If the tablespace already exists in the target database and is already read-only, then you must make it read/write before the import.

NLS Considerations

Export/Import and Single-Byte Character Sets

This section describes the behavior of Export and Import with respect to National Language Support (NLS).

Some 8-bit characters can be lost (i.e. converted to 7-bit equivalents) when importing an 8-bit character set export file. This occurs if the machine on which the import occurs has a native 7-bit character set, or the NLS_LANG operating system environment variable is set to a 7-bit character set. Most often, this is seen when accented characters lose the accent mark.

To avoid this unwanted conversion, you can set the NLS_LANG operating system environment variable to be that of the export file character set.

When importing an Oracle Version 6 export file with a character set different from that of the native operating system, you must set the CHARSET import parameter to specify the character set of the export file.

Export/Import and Multi-Byte Character Sets

An export file that is produced on a system with a multi-byte character set (e.g. Chinese or Japanese) must be imported on a similar system.

Using Oracle Version 6 Export Files

This section describes the guidelines and restrictions that apply when importing data from an Oracle Version 6 database into the Oracle7 Server. Additional information may be found in the *Oracle7 Server Migration* manual.

CHAR columns

Oracle Version 6 CHAR columns are automatically converted into the Oracle VARCHAR2 datatype.

If you pre-create tables from prepared scripts, then the CHAR columns are created as Oracle fixed-width columns instead of the desired VARCHAR2 (variable-width) columns. If you modify the scripts to create VARCHAR2 columns, however, the the columns will be created.

LONG columns

Memory limitations may make it impossible to Import some LONG columns when the values contained in those columns are extremely long. Although Export can output LONG data in sections, Import requires contiguous memory.

Syntax of Integrity Constraints

The SQL syntax for integrity constraints in Oracle Version 6 is different from the Oracle7 Server syntax. Import automatically adjusts the declaration so that integrity constraints are properly imported into the Oracle7 Server, *unless* Version 6–compatibility mode is in effect.

Status of Integrity Constraints

Oracle Version 6 integrity constraints are imported with the status recorded in the export file. In Version 6, all constraints other than NOT NULL are disabled, and that status is recorded in the export file. NOT NULL constraints imported into Oracle are either ENABLED or DISABLED, depending on the status recorded in the export file.

Length of DEFAULT Column Values

A table with a default column value that is longer than the maximum size of that column generates the following error on import to Oracle7:

```
ORA-1401: inserted value too large for column
```

Oracle Version 6 did not check the columns in a CREATE TABLE statement to be sure they were long enough to hold their DEFAULT values so these tables could be imported into a Version 6 database. The Oracle7 Server does make this check, however. As a result, tables that could be imported into a Version 6 database may not import into Oracle7.

If the DEFAULT is a value returned by a function, then the column must be large enough to hold the maximum value that can be returned by that function. Otherwise, the CREATE TABLE statement recorded in the export file produces an error on import.

Note: The maximum value of the USER function increased in the Oracle7 Server, so columns with a default of USER may not be long enough. To determine the maximum size that the USER function will return, execute the following SQL command:

```
DESCRIBE user_sys_privs
```

The length shown for the USERNAME column is the maximum length returned by the USER function.

Using Oracle Version 5 Export Files

In Version 6 compatibility mode, Oracle Import reads export files created by Oracle 5.1.22 and later. The only way to do this is by pre-creating the tables in Version 6 compatibility mode from SQL*Plus. See the *Oracle7 Server Migration* manual for information on pre-creating tables.

SQL*Loader

Part II explains how to use SQL*Loader, a utility for loading data from external files into Oracle database tables. SQL*Loader processes a wide variety of input file formats and gives you control over how records are loaded into Oracle tables.

If you have never used a data loading product, begin by reading the introduction in Chapter 3 and then examine the case studies provided in Chapter 4. These two chapters provide a good introduction to data loading concepts.

If you are comfortable with data loading concepts, you may want to start with the examples in Chapter 4 and then consult Chapters 5 and 6 for detailed reference information on SQL*Loader.

The following topics are covered:

- overview of SQL*Loader functionality (Chapter 3)
- case studies of SQL*Loader in action (Chapter 4)
- control file reference (Chapter 5)
- command-line reference (Chapter 6)
- log file reference (Chapter 7)
- using the direct path for high-performance loads (Chapter 8)

SQL*Loader Concepts

This chapter explains the basic concepts of loading data into an Oracle database with SQL*Loader. This chapter covers the following topics:

- SQL*Loader Basics
- Direct Path and Conventional Path Load Methods
- Mapping the Data to Oracle Format
- Logging Information
- Prerequisites
- Discarded and Rejected Records

SQL*Loader Basics

SQL*Loader moves data from external files into tables in an Oracle database. It has many features of the DB2 Load Utility from IBM. It also has several other features that add power and flexibility.

SQL*Loader loads data in a variety of formats, performs filtering (selectively loading records based upon the data values), and loads multiple tables simultaneously.

During execution, SQL*Loader produces a detailed *log file* with statistics about the load. It may also produce a *bad file* (containing records rejected because of incorrect data) and a *discard file* (containing records that did not meet the specified selection criteria).

SQL*Loader can:

- load data from multiple datafiles of different file types
- handle fixed-format, delimited-format, and variable-length records
- manipulate data fields with SQL functions before inserting the data into database columns
- support a wide range of datatypes, including DATE, BINARY, PACKED DECIMAL, and ZONED DECIMAL
- load multiple tables during the same run, loading selected rows into each table
- combine multiple physical records into a single logical record
- treat a single physical record as multiple logical records
- generate unique, sequential key values in specified columns
- use your operating system's file or record management system to access datafiles
- load data from disk or tape
- provide thorough error reporting capabilities, so you can easily adjust and load all records
- use high-performance "direct" loads to load data directly into database files without Oracle processing. This feature is discussed in Chapter 8, "SQL*Loader Conventional and Direct Path Loads".

Direct Path Load vs. Conventional Path Load Method

SQL*Loader can use one of two methods to load data: conventional path (which uses the bind array) and direct path (which stores data directly into the database).

Conventional Path

During conventional path loads, multiple data records are read in and placed in a bind array. When the bind array is full (or there is no more data left to read), it is passed to Oracle for insertion. Conventional path uses the Oracle SQL interface with the array option.

For more information on conventional path loads, see the section “Data Loading Methods” on page 8 – 2. For information on the bind array, see page 5 – 63.

Direct Path

A direct path load creates data blocks that are already in Oracle database block format. These database blocks are then written directly to the database bypassing most RDBMS processing.

This path is much faster than the conventional load, but entails several restrictions. For more information on the direct path, see the section “Data Loading Methods” on page 8 – 2.

Mapping the Data to Oracle Format

Data to be loaded into the Oracle database must exist in files on disk or on tape. These datafiles require mapping (translation to Oracle format) to be loaded by SQL*Loader. You specify how SQL*Loader interprets the data via data definitions contained in control files. The control file also is the repository for certain file management information.

The Concept of Mapping Data

SQL*Loader must be told where and in what format the data to be loaded is and how to map the data to Oracle format. Definitions in the control file will include:

- specifications for loading logical records into tables
- field condition specifications
- column and field specifications
- data field position specifications
- datatype specifications
- bind array size specifications
- specifications for setting columns to null or zero
- specifications for loading all-blank fields

- specifications for trimming blanks and tabs
- specifications to preserve whitespace
- specifications for applying SQL operators to fields

To define the specifications listed above you will use the SQL*Loader Data Definition Language (DDL).

The Data Definition Language (DDL)

The SQL*Loader data definition language (DDL) is used to specify exactly how SQL*Loader should interpret the data you are loading into the Oracle database. DDL is used to create DDL definitions which are the map that SQL*Loader uses to translate the loaded data into Oracle format.

The syntax and semantics of DDL is explained in detail in Chapter 5.

SQL*Loader DDL is upwardly compatible with the DB2 Load Utility from IBM. If you have a control file for the DB2 Load Utility, you can also use it with SQL*Loader. See Appendix C, “Notes for DB2/DXT Users,” for differences in syntax.

DDL Definitions

DDL definitions can serve several purposes. Certain definitions specify data location or format. Other DDL definitions specify how SQL*Loader should map specific objects in the loaded data to comparable objects in an Oracle database. Other definitions deal with column definitions, datatype mapping and field specifications. DDL definitions are stored in control files which are read by SQL*Loader on startup.

A single DDL definition is composed of one or more keywords and the arguments and options that modify that keyword’s functionality. An example of a control file containing several definitions defining how SQL*Loader should interpret a simple datafile might look like this:

```
LOAD DATA
INFILE 'example.dat'
INTO TABLE emp
(empno      POSITION(01:04)  INTEGER EXTERNAL,
ename      POSITION(06:15)  CHAR,
job        POSITION(17:25)  CHAR,
mgr        POSITION(27:30)  INTEGER EXTERNAL,
sal        POSITION(32:39)  DECIMAL EXTERNAL,
comm       POSITION(41:48)  DECIMAL EXTERNAL,
...
```

The keywords are LOAD DATA, INFILE, INTO TABLE, POSITION, etc. (shown in all capital letters).

The Control File

Control files contain DDL definitions. You can create a control file using your system text editor.

How you store control files depends on how your operating system organizes data. For example, in UNIX environments, control files are stored in files; in MVS environments, they can be stored as members in a partitioned dataset. They must be located where SQL*Loader has access to them.

Some DDL definitions are mandatory, such as where to find the data and how it corresponds to the database tables. However, many options are also available to describe and manipulate the file data. For example, the instructions can include directions on how to format or filter the data, or to generate unique ID numbers.

A control file can also contain the data itself as well as the DDL definitions, as shown in Case 1 on page 4 – 4, or in separate files, as shown in Case 2 on page 4 – 6. Detailed information on creating control files using DDL definitions is found on page 5 – 4.

Control File Guidelines

- The file is written in free format. That is, statements can continue from line to line with new lines beginning at any word.
- The file is written in uppercase or lowercase. Case is not significant except in strings specified with single or double quotation marks.
- Comments can be included by prefixing them with two hyphens, as in the following example:

```
--This is a comment
```

- The double hyphens can appear anywhere on a line; SQL*Loader ignores anything from the double hyphens to the end of line. Comments should not appear in the datafile or the data portion of the control file. If a double dash appears in this area, it is treated as data.
- SQL*Loader reserved words must be enclosed in quotation marks if you want to use them for table or column names. (See Appendix B for a complete list of SQL*Loader reserved words.)

The Data

Binary versus Character
Format Data

SQL*Loader can load data stored in various formats.

SQL*Loader can load numeric data in binary or character format. Character format is sometimes referred to as numeric external format.

Binary data is one example of native datatypes — datatypes that are implemented differently on different operating systems. For more information on these and other native datatypes, see “Native Datatypes” on page 5 – 50. SQL*Loader cannot handle binary data in variable record format. See also the section called “Loading Data Across Different Operating Systems” on page 5 – 62.

Data in *character format* can be included in both fixed-format and variable-format files. For more information on the character datatypes, see “Character Datatypes” on page 5 – 56.

Fixed versus Variable
Format

Data records may be in fixed or variable format. In *fixed format*, the data is contained in records that all have the same (fixed) format. That is, the records have a fixed length, and the data fields in those records have fixed length, type, and position, as shown in Figure 3 – 1.

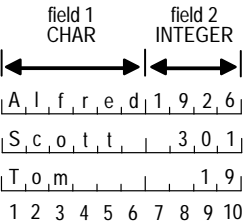


Figure 3 – 1 Fixed Format Records

In this example, columns 1 to 6 contain a character variable while columns 7 to 10 contain an integer for all the records. The fields are the same size in each record, regardless of the length of the data. The fields are fixed length, rather than variable length. In consequence, the record size is also fixed at 10 characters for each of the records.

In *variable format* (sometimes called *stream format*), each record is only as long as necessary to contain the data. Figure 3 – 2 shows variable length records containing one varying-length character fields and one fixed length integer field.

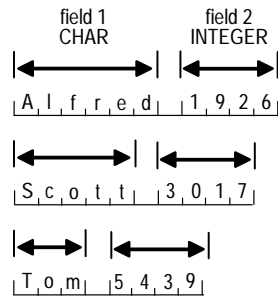


Figure 3 – 2 Variable Format Records

In addition, the type of data in each record may vary. One record may contain a character string, the next may contain seven integers, the third may contain three decimals and a float, and so on. Operating systems use a record terminator character (such as newline) to mark where variable records end.

Data Fields

Data in records is divided into *fields*. Fields can be specified with specific positions and lengths, or their position and length can vary based on *delimiters*.

There are two types of delimited fields: terminated and enclosed. *Terminated fields* are followed by a specified character (called a *termination delimiter*), such as the commas in the following example:

1, 1, 2, 3, 5, 8, 13

Enclosed fields are both preceded and followed by specified characters (called *enclosure delimiters*), such as the quotation marks in the following example:

"BUNKY"

Case 2 on page 4 – 6 shows fixed-length records. Case 1 on page 4 – 4 shows delimited fields. For more details on delimited data, see "Specifying Delimiters" on page 5 – 58.

Logical versus Physical Records

A final distinction concerns the difference between logical and physical records. A record or line in a file (either of fixed length or terminated) is referred to as a *physical record*. An operating system-dependent file/record management system, such as DEC's Record Management System (RMS) or IBM's Sequential Access Method (SAM) returns physical records.

Logical records, on the other hand, correspond to a row in a database table. Sometimes the logical and physical records are equivalent. Such is the case when only a few short columns are being loaded. However, sometimes several physical records must be combined to make one logical record. For example, you could have a file containing 24 10-character columns in a format of 80-character, fixed-length records. In this case, three physical records would constitute a single logical record.

SQL*Loader allows you to compose logical records from multiple physical records using *continuation fields*. Physical records are combined into a single, logical record when some condition of the continuation field is true. You can specify that a logical record should be composed of multiple, physical records in the following ways:

- A fixed number of physical records are concatenated to form a logical record (no continuation field is used).
- Physical records are appended if the continuation field contains a specified string (or another test, such as “not equal”, succeeds when applied to the continuation field).
- Physical records are appended if they contain a specified character as their last non-blank character.

Case 4 on page 4 – 11 uses continuation fields to form one logical record from multiple physical records.

Logging Information

When SQL*Loader begins execution, it creates a *log file*. If it cannot create a log file, execution terminates. The log file contains a detailed summary of the load, including a description of any errors that occurred during the load. For details on the information contained in the log file, see page 7 – 1. All of the case studies in Chapter 4 also contain sample log files.

Prerequisites

To load data using the conventional load method, the tables to receive the data must already exist in the database. There are no special requirements for these tables. The tables may be clustered or indexed, or they may actually be a view for which you have insert privileges. Tables may already contain data, or they may be empty.

Privileges Required

The following privileges are required for a conventional load:

- You must have INSERT privileges on the table to be loaded.
- You must have DELETE privilege on the table to be loaded, when using the REPLACE option to empty out the table's old data before loading the new data in its place.

In addition to the above privileges, you must have write access to all labels you are loading data into a Trusted Oracle7 Server database. See the *Trusted Oracle7 Server Administrator's Guide*.

Discarded and Rejected Records

Records that are read from the input file might not be inserted into the database. Figure 3 – 3 shows the stages at which records may be *rejected* or *discarded*.

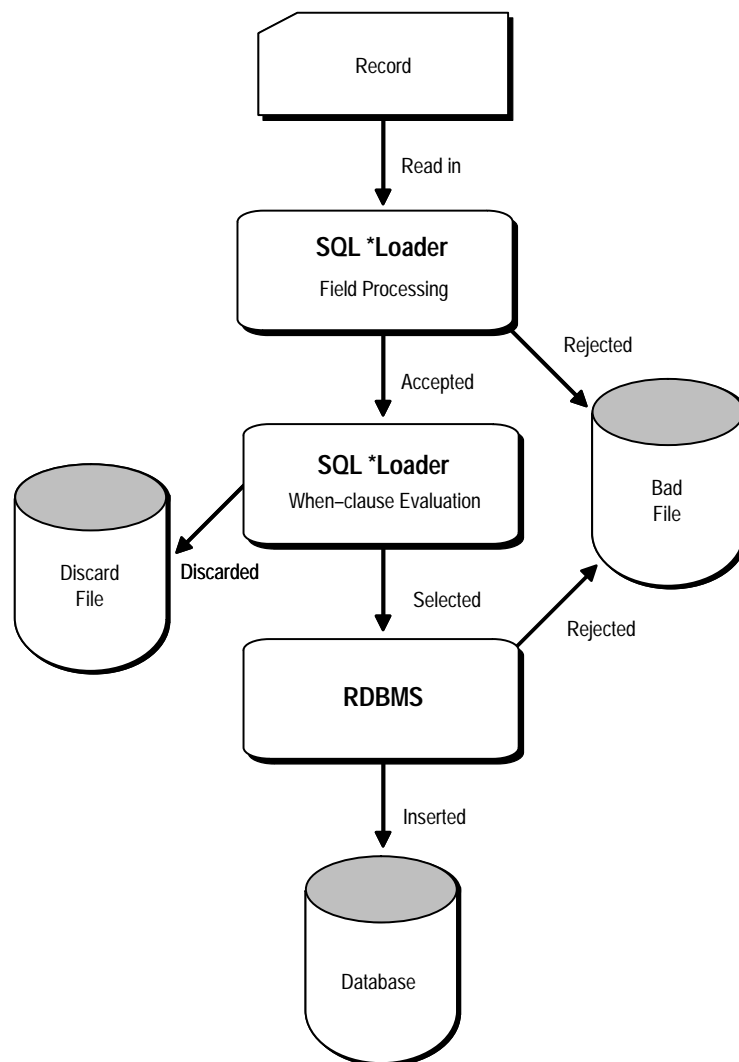


Figure 3 – 3 Record Filtering

The Bad File

The *bad file* contains records that are rejected, either by SQL*Loader or by Oracle. Some of the possible reasons for rejection are discussed in the next sections.

SQL*Loader Rejects

Records are rejected by SQL*Loader when the input format is invalid. For example, if the second enclosure delimiter is missing, or if a delimited field exceeds its maximum length, SQL*Loader rejects the record. Rejected records are placed in the *bad file*. For details on how to specify the bad file, see “Specifying the Bad File” on page 5 – 19.

Oracle Rejects

After a record is accepted for processing by SQL*Loader, a row is sent to Oracle for insertion. If Oracle determines that the row is valid, then the row is inserted into the database. If not, the record is rejected, and SQL*Loader puts it in the bad file. The row may be rejected, for example, because a key is not unique, because a required field is null, or because the field contains invalid data for the Oracle datatype.

The bad file is written in the same format as the datafile. So the rejected data can be loaded with the existing control file, after any necessary corrections are made.

Case 4 on page 4 – 11 is an example of the use of a bad file.

SQL*Loader Discards

As SQL*Loader executes, it may create a file called the *discard file*. This file is created only when it is needed, and only if you have specified that a discard file should be enabled. The discard file contains records that were filtered out of the load because they did not match any of the record-selection criteria specified in the control file.

The discard file therefore contains records that were not inserted into any table, up to a specifiable maximum. If a record's data is written to any table, it is not written to the discard file.

The discard file is written in the same format as the datafile. The discard data can be loaded with the existing control file, after any necessary editing or correcting.

Case 4 on page 4 – 11 shows how the discard file is used. For more details, see “Specifying the Discard File” on page 5 – 21.

Data Conversion and Datatype Specification

Figure 3 – 4 shows the stages in which *fields* in the datafile are converted into *columns* in the database during a conventional path load. The top of the diagram shows a data record containing one or more fields. The bottom shows the database column in which the data winds up. It is important to understand the intervening steps when using SQL*Loader.

Figure 3 – 4 depicts the “division of labor” between SQL*Loader and the Oracle7 Server. The field specifications tell SQL*Loader how to interpret the format of the datafile. The Oracle7 Server then converts that data and inserts it into the database columns, using the column datatypes as a guide.

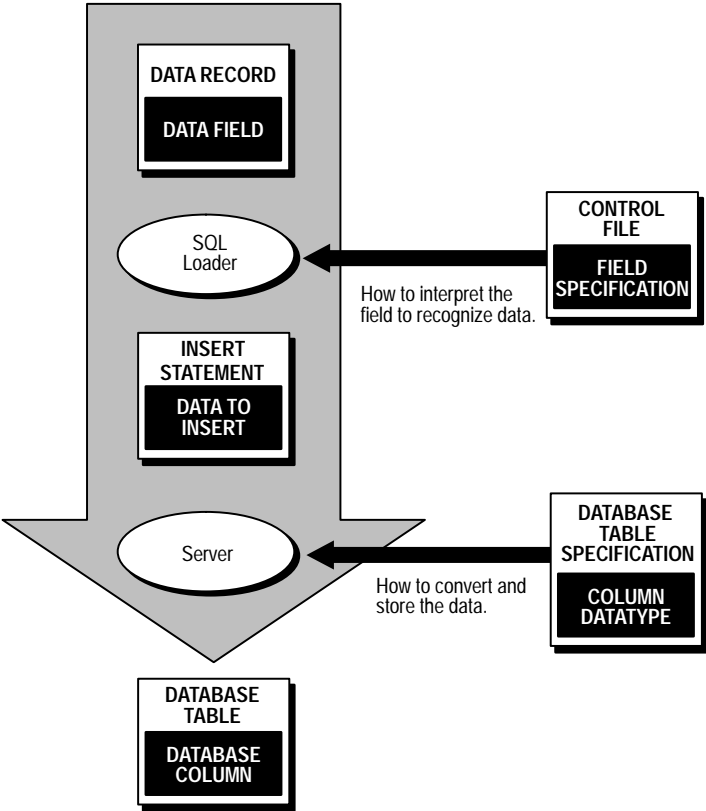


Figure 3 – 4 Field to Column Translation

Keep in mind the distinction between a *field* (in a datafile) and a *column* (in the database). It is also important to remember that the *field datatypes* defined in a SQL*Loader control file are *not* the same as the *column datatypes*.

SQL*Loader uses the field specifications in the control file to recognize data and creates a SQL insert statement using that data. The insert statement is then passed to the Oracle7 Server to be stored in the table. The Oracle7 Server uses the datatype of the column to convert the data into its final, stored form.

In actuality, there are two conversion steps:

1. SQL*Loader identifies a field in the datafile, interprets the data, and passes it to the Oracle7 Server.
2. The Oracle7 Server accepts the data and stores it in the database.

In Figure 3 – 5, two CHAR fields are defined for a data record. The field specifications are contained in the control file. Note that the control file CHAR specification is not the same as the database CHAR specification. A data field defined as CHAR in the control file merely tells SQL*Loader how to create the row insert. The data could then be inserted into a CHAR, VARCHAR2, or even a NUMBER column in the database, with the Oracle7 Server handling any necessary conversions.

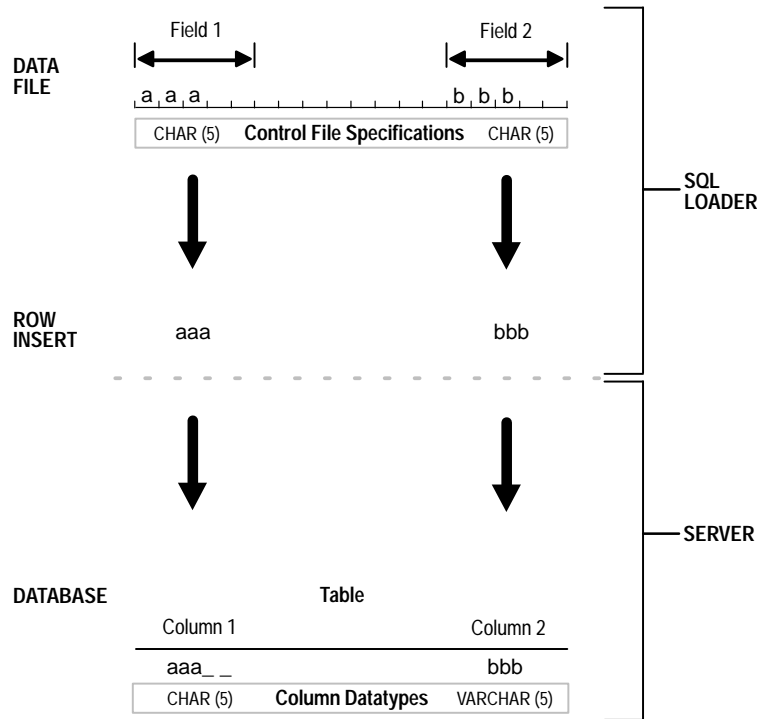


Figure 3 – 5 Example of Field Conversion

By default, SQL*Loader removes trailing spaces from CHAR data before passing it to the database. So, in Figure 3 – 5, both field A and field B are passed to the database as three-column fields. When the data is inserted into the table, however, there is a difference.

Column A is defined in the database as a fixed-length CHAR column of length 5. So the data (aaa) is left justified in that column, which remains five characters wide. The extra space on the right is padded with blanks. Column B, however, is defined as a varying length field with a *maximum* length of five characters. The data for that column (bbb) is left-justified as well, but the length remains three characters.

The *name* of the field tells SQL*Loader what column to insert the data into. Because the first data field has been specified with the name “A” in the control file, SQL*Loader knows to insert the data into column A of the target database table.

It will be useful to keep the following points in mind:

- The name of the data field corresponds to the name of the table column into which the data is loaded.
- The datatype of the field tells SQL*Loader how to read the data in the datafile. It is *not* the same as the column datatype.
- Data is converted from the datatype specified in the control file to the datatype of the column in the database.

SQL*Loader Case Studies

The case studies in this chapter illustrate some of the features of SQL*Loader. These case studies start simply and progress in complexity.

The cases are:

Case 1: Loads stream format records in which the fields are delimited by commas and may be enclosed by quotation marks. The data is found at the end of the control file.

Case 2: Loads a datafile with fixed-length, fixed-format records

Case 3: Loads data from stream format records with delimited fields and sequence numbers. The data is found at the end of the control file.

Case 4: Combines multiple physical records into one logical record corresponding to one database row

Case 5: Loads data into multiple tables in one run

Case 6: Loads data using the direct path load method

Case 7: Extracts data from a formatted report

Case Study Files

The distribution media for SQL*Loader contains files for each case:

- control files (for example, ULCASE1.CTL)
- data files (for example, ULCASE2.DAT)
- setup files (for example, ULCASE3.SQL)

If the sample data for the case study is contained in the control file, then there will be no .DAT file for that case. If there are no special setup steps for a case study, there may be no .SQL file for that case. Starting (setup) and ending (cleanup) scripts are denoted by an S or E after the case number. The table below lists the files associated with each case:

CASE	.CTL	.DAT	.SQL
1	x		x
2	x	x	
3	x		x
4	x	x	x
5	x	x	x
6	x	x	x
7	x	x	x S, E

Table 4 – 1 Case Studies and their Related Files



Additional Information: The actual names of the case study files are operating system–dependent. See your Oracle operating system–specific documentation for the exact names.



Tables Used in the Case Studies

The case studies are based upon the standard Oracle demonstration database tables EMP and DEPT owned by SCOTT/TIGER. (In some of the case studies, additional columns have been added.)

Contents of Table EMP

empno	NUMBER(4) NOT NULL,
ename	VARCHAR2(10),
job	VARCHAR2(9),
mgr	NUMBER(4),
hiredate	DATE,
sal	NUMBER(7,2),
comm	NUMBER(7,2),
deptno	NUMBER(2)

Contents of Table DEPT

deptno	NUMBER(2) NOT NULL,
dname	VARCHAR2(14),
loc	VARCHAR2(13)



References

The summary at the beginning of each case study contains page number references, directing you to the sections of this guide that discuss the SQL*Loader feature being demonstrated in more detail.



Notes

In the control file fragment and log file listing shown for each case study, the numbers that appear to the left are not actually in the file; they are keyed to the numbered notes following the listing. Do not use these numbers when you write your control files.



Running the Case Study SQL Scripts

You should run the SQL scripts ULCASE1.SQL and ULCASE3.SQL through ULCASE7.SQL to prepare and populate the tables. Note that there is no ULCASE2.SQL as Case 2 is handled by ULCASE1.SQL.

Case 1: Loading Variable–Length Data

Case 1 demonstrates

- A simple control file identifying one table and three columns to be loaded. See page 5 – 16.
- Including data to be loaded from the control file itself, so there is no separate datafile. See page 5 – 15.
- Loading data in stream format, with both types of delimited fields — terminated and enclosed. See page 5 – 70.

The Control File

The control file is ULCASE1.CTL:

```
1) LOAD DATA
2) INFILE *
3) INTO TABLE dept
4) FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
5) (deptno, dname, loc)
6) BEGINDATA
   12,RESEARCH,"SARATOGA"
   10,"ACCOUNTING",CLEVELAND
   11,"ART",SALEM
   13,FINANCE,"BOSTON"
   21,"SALES",PHILA.
   22,"SALES",ROCHESTER
   42,"INT'L","SAN FRAN"
```


Notes:

- 1) The LOAD DATA statement is required at the beginning of the control file.
- 2) INFILE * specifies that the data is found in the control file and not in an external file.
- 3) The INTO TABLE statement is required to identify the table to be loaded (DEPT) into. By default, SQL*Loader requires the table to be empty before it inserts any records.
- 4) FIELDS TERMINATED BY specifies that the data is terminated by commas, but may also be enclosed by quotation marks. Datatypes for all fields default to CHAR.
- 5) Specifies that the names of columns to load are enclosed in parentheses.
- 6) BEGINDATA specifies the beginning of the data.

Invoking SQL*Loader To run this example, invoke SQL*Loader with the command:

```
sqlldr userid=scott/tiger control=ulcase1.ctl log=ulcase1.log
```

SQL*Loader loads the DEPT table and creates the log file.


OSDoc

Additional Information: The command “sqlldr” is a UNIX-specific invocation. To invoke SQL*Loader on your operating system, refer to your Oracle operating system-specific documentation.

The Log File

The following shows a portion of the log file:

```
Control File:      ULCASE1.CTL
Data File:        ULCASE1.DAT
  Bad File:       ULCASE1.BAD
  Discard File:    none specified
(Allow all discards)

Number to load:    ALL
Number to skip:    0
Errors allowed:    50
Bind array:        64 rows, maximum of 65336 bytes
Continuation:      none specified
Path used:         Conventional
```

Table DEPT, loaded from every logical record.
Insert option in effect for this table: INSERT

	Column Name	Position	Len	Term	Encl	Datatype
	-----	-----	---	----	-----	-----
1)	DEPTNO	FIRST	*	,	O(")	CHARACTER
	DNAME	NEXT	*	,	O(")	CHARACTER
2)	LOC	NEXT	*	WHT	O(")	CHARACTER

```
Table DEPT:
  7 Rows successfully loaded.
  0 Rows not loaded due to data errors.
  0 Rows not loaded because all WHEN clauses were failed.
  0 Rows not loaded because all fields were null.
```

```
Space allocated for bind array:  49920 bytes(64 rows)
Space allocated for memory besides bind array: 76000 bytes
```

```
Total logical records skipped:      0
Total logical records read:          7
Total logical records rejected:      0
Total logical records discarded:     0
```

Notes:

- 1) Position and length for each field are determined for each record, based on delimiters in the input file.
- 2) WHT signifies that field LOC is terminated by WHITESPACE. The notation O(") signifies optional enclosure by quotation marks.

Case 2: Loading Fixed-Format Records

Case 2 demonstrates

- A separate datafile. See page 5 – 16.
- Fixed-format data. See page 5 – 70.
- Data conversions. See page 5 – 49.

In this case, the field positions and datatypes are specified explicitly.

The Control File

The control file is ULCASE2.CTL.

```
1)  LOAD DATA
2)  INFILE 'ulcase2.dat'
3)  INTO TABLE emp
4)  (empno          POSITION(01:04)    INTEGER EXTERNAL,
    ename           POSITION(06:15)    CHAR,
    job             POSITION(17:25)    CHAR,
    mgr             POSITION(27:30)    INTEGER EXTERNAL,
    sal             POSITION(32:39)    DECIMAL EXTERNAL,
    comm            POSITION(41:48)    DECIMAL EXTERNAL,
5)  deptno          POSITION(50:51)    INTEGER EXTERNAL)
```

Notes:

- 1) The LOAD DATA statement is required at the beginning of the control file.
- 2) The name of the file containing data follows the keyword INFILE.
- 3) The INTO TABLE statement is required to identify the table to be loaded into.
- 4) Lines 4 and 5 identify a column name and the location of the data in the datafile to be loaded into that column. EMPNO, ENAME, JOB, and so on are names of columns in table EMP. The datatypes (INTEGER EXTERNAL, CHAR, DECIMAL EXTERNAL) identify the datatype of data fields in the file, not of corresponding columns in the EMP table.
- 5) Note that the set of column specifications is enclosed in parentheses.

Datafile

Below are a few sample data lines from the file ULCASE2.DAT. Blank fields are set to null automatically.

7782	CLARK	MANAGER	7839	2572.50		10
7839	KING	PRESIDENT		5500.00		10
7934	MILLER	CLERK	7782	920.00		10
7566	JONES	MANAGER	7839	3123.75		20
7499	ALLEN	SALESMAN	7698	1600.00	300.00	30
7654	MARTIN	SALESMAN	7698	1312.50	1400.00	30

Invoking SQL*Loader

Invoke SQL*Loader with a command such as:

```
sqlldr userid=scott/tiger control=ulcase2.ctl log=ulcase2.log
```

The EMP records loaded in this example contain department numbers. Unless the DEPT table is loaded first, referential integrity checking rejects these records (if referential integrity constraints are enabled for the EMP table).



OSDoc

Additional Information: The command “sqlldr” is a UNIX-specific invocation. To invoke SQL*Loader on your operating system, refer to your Oracle operating system-specific documentation.

The Log File

The following shows a portion of the log file:

```
Control File:      ULCASE2.CTL
Data File:         ULCASE2.DAT
  Bad File:        ULCASE2.BAD
  Discard File:     none specified
(Allow all discards)

Number to load:    ALL
Number to skip:    0
Errors allowed:    50
Bind array:        64 rows, maximum of 65336 bytes
Continuation:      none specified
Path used:         Conventional
```


Table EMP, loaded from every logical record.
Insert option in effect for this table: INSERT

Column Name	Position	Len	Term	Encl	Datatype
-----	-----	----	----	----	-----
EMPNO	1:4	4			CHARACTER
ENAME	6:15	10			CHARACTER
JOB	17:25	9			CHARACTER
MGR	27:30	4			CHARACTER
SAL	32:39	8			CHARACTER
COMM	41:48	8			CHARACTER
DEPTNO	50:51	2			CHARACTER

Table EMP:
7 Rows successfully loaded.
0 Rows not loaded due to data errors.
0 Rows not loaded because all WHEN clauses were failed.
0 Rows not loaded because all fields were null.

Space allocated for bind array 4352 bytes(64 rows)
Space allocated for memory besides bind array: 37051 bytes

Total logical records skipped: 0
Total logical records read: 7
Total logical records rejected: 0
Total logical records discarded: 0

Case 3: Loading a Delimited, Free-Format File

Case 3 demonstrates

- Loading data in (enclosed and terminated) stream format. See page 5 – 70.
- Loading dates. See page 5 – 56.
- Using the SEQUENCE function to generate unique keys for loaded data. See page 5 – 47.
- Using APPEND to indicate that the table need not be empty before inserting new records. See page 5 – 25.
- Using comments in the control file set off by double dashes. See page 5 – 11.
- Overriding general specifications with declarations for individual fields. See page 5 – 34.

The Control File

This control file loads the same table as Case 2, but it loads three additional columns (HIREDATE, PROJNO, LOADSEQ). The demonstration table EMP does not have columns PROJNO and LOADSEQ. So if you want to test this control file, add these columns to the EMP table with the command:

```
ALTER TABLE EMP ADD (PROJNO NUMBER, LOADSEQ NUMBER)
```

The data is in a different format than in Case 2. Some data is enclosed in quotation marks, some is set off by commas, and the values for DEPTNO and PROJNO are separated by a colon.

```
1)  -- Variable-length, delimited and enclosed data format
LOAD DATA
2)  INFILE *
3)  APPEND
    INTO TABLE emp
4)  FIELDS TERMINATED BY "," OPTIONALLY ENCLOSED BY '"'
    (empno, ename, job, mgr,
5)  hiredate DATE(20) "DD-Month-YYYY",
    sal, comm, deptno CHAR TERMINATED BY ': ',
    projno,
6)  loadseq SEQUENCE(MAX,1))
7)  BEGINDATA
8)  7782, "Clark", "Manager", 7839, 09-June-1981, 2572.50,, 10:101
    7839, "King", "President", , 17-November-1981,5500.00,,10:102
    7934, "Miller", "Clerk", 7782, 23-January-1982, 920.00,, 10:102
    7566, "Jones", "Manager", 7839, 02-April-1981, 3123.75,, 20:101
    7499, "Allen", "Salesman", 7698, 20-February-1981, 1600.00,
    (same line continued) 300.00, 30:103
    7654, "Martin", "Salesman", 7698, 28-September-1981, 1312.50,
    (same line continued) 1400.00, 3:103
    7658, "Chan", "Analyst", 7566, 03-May-1982, 3450,, 20:101
```

Notes:

- 1) Comments may appear anywhere in the command lines of the file, but they should not appear in data. They are preceded with a double dash that may appear anywhere on a line.
- 2) INFILE * specifies that the data is found at the end of the control file.
- 3) Specifies that the data can be loaded even if the table already contains rows. That is, the table need not be empty.
- 4) The default terminator for the data fields is a comma, and some fields may be enclosed by double quotation marks ("").
- 5) The data to be loaded into column HIREDATE appears in the format DD-Month-YYYY. The length of the date field is dependent on the mask specified.

6) The SEQUENCE function generates a unique value in the column LOADSEQ. This function finds the current maximum value in column LOADSEQ and adds the increment (1) to it to obtain the value for LOADSEQ for each row inserted.

7) BEGINDATA specifies the end of the control information and the beginning of the data.

8) Although each physical record equals one logical record, the fields vary in length so that some records are longer than others. Note also that several rows have null values for COMM.

Invoking SQL*Loader

Invoke SQL*Loader with a command such as:

```
sqlldr userid=scott/tiger control=ulcase3.ctl log=ulcase3.log
```



OSDoc

Additional Information: The command “sqlldr” is a UNIX-specific invocation. To invoke SQL*Loader on your operating system, see your Oracle operating system-specific documentation.

The Log File

The following shows a portion of the log file:

```
Control File:      ULCASE3.CTL
Data File:        YLCASE3.DAT
  Bad File:       ULCASE3.BAD
  Discard File:    none specified
(Allow all discards)

Number to load:    ALL
Number to skip:    0
Errors allowed:    50
Bind array:        64 rows, maximum of 65336 bytes
Continuation:      none specified
Path used:         Conventional
```

Table EMP, loaded from every logical record.
 Insert option in effect for this table: APPEND

Column Name	Position	Len	Term	Encl	Datatype
EMPNO	FIRST	*	,	O("	CHARACTER
ENAME	NEXT	*	,	O("	CHARACTER
JOB	NEXT	*	,	O("	CHARACTER
MGR	NEXT	*	,	O("	CHARACTER
HIREDATE	NEXT	20	,	O("	DATE DD-Month-YYYY
SAL	NEXT	*	,	O("	CHARACTER
COMM	NEXT	*	,	O("	CHARACTER
DEPTNO	NEXT	*	:	O("	CHARACTER
PROJNO	NEXT	*	,	O("	CHARACTER

LOADSEQ SEQUENCE (MAX, 1)

Table EMP:
 7 Rows successfully loaded.
 0 Rows not loaded due to data errors.
 0 Rows not loaded because all WHEN clauses were failed.
 0 Rows not loaded because all fields were null.

Space allocated for bind array: 63810 bytes(30 rows)
 Space allocated for memory besides bind array: 94391 bytes

Total logical records skipped: 0
 Total logical records read: 7
 Total logical records rejected: 0
 Total logical records discarded: 0

Case 4: Loading Combined Physical Records

Case 4 demonstrates

- Combining multiple physical records to form one logical record with CONTINUEIF. See page 5 – 29.
- Inserting negative numbers
- Indicating with REPLACE that the table should be emptied before the new data is inserted. See page 5 – 26.
- Specifying a discard file in the control file using DISCARDFILE. See page 5 – 22.
- Specifying a maximum number of discards using DISCARDMAX. See page 5 – 23.

- Rejecting records due to duplicate values in a unique index or due to invalid data values. See page 5 – 20.

The Control File

The control file is ULCASE4.CTL:

```
LOAD DATA
INFILE 'ulcase4.dat'
1) DISCARDFILE 'ulcase4.dsc'
2) DISCARDMAX 999
3) REPLACE
4) CONTINUEIF THIS (1) = '*'
INTO TABLE emp
(empno      POSITION(1:4)      INTEGER EXTERNAL,
ename      POSITION(6:15)     CHAR,
job         POSITION(17:25)    CHAR,
mgr         POSITION(27:30)    INTEGER EXTERNAL,
sal         POSITION(32:39)    DECIMAL EXTERNAL,
comm        POSITION(41:48)    DECIMAL EXTERNAL,
deptno      POSITION(50:51)    INTEGER EXTERNAL,
hiredate    POSITION(52:60)    INTEGER EXTERNAL)
```

Notes:

- 1) DISCARDFILE specifies a discard file named ULCASE4.DSC.
- 2) DISCARDMAX specifies a maximum of 999 discards allowed before terminating the run (for all practical purposes, this allows all discards).
- 3) REPLACE specifies that if there is data in the table being loaded, then SQL*Loader should delete that data before loading new data.
- 4) CONTINUEIF THIS specifies that if an asterisk is found in column 1 of the current record, then the next physical record after that record should be appended to it to form the logical record. Note that column 1 in each physical record should then contain either an asterisk or a non-data value.

The Data File

The datafile for this case, ULCASE4.DAT, is listed below. Note the asterisks in the first position and, though not visible, a new line indicator is in position 20 (following “MA”, “PR”, and so on). Note that CLARK’s commission is –10, and SQL*Loader loads the value converting it to a negative number.

*7782	CLARK	MANAGER	7839	2572.50	-10	2512-NOV-85
*7839	KING	PRESIDENT		5500.00		2505-APR-83
*7934	MILLER	CLERK	7782	920.00		2508-MAY-80
*7566	JONES	MANAGER	7839	3123.75		2517-JUL-85
*7499	ALLEN	SALESMAN	7698	1600.00	300.00	25 3-JUN-84
*7654	MARTIN	SALESMAN	7698	1312.50	1400.00	2521-DEC-85
*7658	CHAN	ANALYST	7566	3450.00		2516-FEB-84
*	CHEN	ANALYST	7566	3450.00		2516-FEB-84

Rejected Records

The last two records are rejected, given two assumptions. If there is a unique index created on column EMPNO, then the record for CHIN will be rejected because his EMPNO is identical to CHAN's. If EMPNO is defined as NOT NULL, then CHEN's record will be rejected because it has no value for EMPNO.

Invoking SQL*Loader

Invoke SQL*Loader with a command such as:

```
sqlldr userid=scott/tiger control=ulcase4.ctl log=ulcase4.log
```



OSDoc

Additional Information: The command “sqlldr” is a UNIX-specific invocation. To invoke SQL*Loader on your operating system, see your operating Oracle system-specific documentation.

The Log File

The following is a portion of the log file:

```
Control File:      ULCASE4.CTL
Data File:        ULCASE4.DAT
  Bad File:       ULCASE4.BAD
  Discard File:   ULCASE4.DSC
(Allow 999 discards)
```

```
Number to load:    ALL
Number to skip:    0
Errors allowed:    50
Bind array:        64 rows, maximum of 65336 bytes
  Continuation:    1:1 = 0X2a(character '**'),
                  in current physical record
Path used:         Conventional
```

```
Table EMP, loaded from every logical record.
Insert option in effect for this table: REPLACE
```

Column Name	Position	Len	Term	Encl	Datatype
-----	-----	---	----	-----	-----
EMPNO	1:4	4			CHARACTER
ENAME	6:15	10			CHARACTER
JOB	17:25	9			CHARACTER
MGR	27:30	4			CHARACTER
SAL	32:39	8			CHARACTER
COMM	41:48	8			CHARACTER
DEPTNO	50:51	2			CHARACTER
HIREDATE	52:60	9			CHARACTER

```
Record 8: Rejected - Error on table EMP,          --EMPNO null
ORA-01400: mandatory (NOT NULL) column is missing or NULL during
          insert
```

```

Record 9: Rejected - Error on table EMP.          --EMPNO not unique
ORA-00001: unique constraint (SCOTT.EMPIX) violated
Table EMP:
  7 Rows successfully loaded.
  2 Rows not loaded due to data errors.
  0 Rows not loaded because all WHEN clauses were failed.
  0 Rows not loaded because all fields were null.

Space allocated for bind array:          5120 bytes(64 rows)
Space allocated for memory besides bind array: 40195 bytes

Total logical records skipped:           0
Total logical records read:              9
Total logical records rejected:          2
Total logical records discarded:         0

```

The Bad File

The bad file, shown below, lists records 8 and 9 for the reasons stated earlier. (The discard file is not created.)

```

*      CHEN      ANALYST
      7566      3450.00      2516-FEB-84
*      CHIN      ANALYST
      7566      3450.00      2516-FEB-84

```

Case 5: Loading Data into Multiple Tables

Case 5 demonstrates

- Loading multiple tables. See page 5 – 45.
- Using SQL*Loader to break down repeating groups in a flat file and load the data into normalized tables — one file record may generate multiple database rows
- Deriving multiple logical records from each physical record. See page 5 – 43.
- Using a WHEN clause. See page 5 – 34.
- Loading the same field (EMPNO) into multiple tables.

The Control File

The control file is ULCASE5.CTL.

```

-- Loads EMP records from first 23 characters
-- Creates and loads PROJ records for each PROJNO listed
-- for each employee

LOAD DATA
INFILE 'ulcase5.dat'

```

```

        BADFILE 'ulcase5.bad'
        DISCARDFILE 'ulcase5.dsc'
1) REPLACE

2) INTO TABLE emp
   (empno    POSITION(1:4)      INTEGER EXTERNAL,
    ename     POSITION(6:15)    CHAR,
    deptno    POSITION(17:18)   CHAR,
    mgr       POSITION(20:23)   INTEGER EXTERNAL)

2) INTO TABLE proj
   -- PROJ has two columns, both not null: EMPNO and PROJNO
3) WHEN projno != ' '
   (empno    POSITION(1:4)      INTEGER EXTERNAL,
3) projno    POSITION(25:27)    INTEGER EXTERNAL) -- 1st proj
3) INTO TABLE proj
4) WHEN projno != ' '
   (empno    POSITION(1:4)      INTEGER EXTERNAL,
4) projno    POSITION(29:31)    INTEGER EXTERNAL) -- 2nd proj

2) INTO TABLE proj
5) WHEN projno != ' '
   (empno    POSITION(1:4)      INTEGER EXTERNAL,
5) projno    POSITION(33:35)    INTEGER EXTERNAL) -- 3rd proj

```

Notes:

- 1) REPLACE specifies that if there is data in the tables to be loaded (EMP and PROJ), SQL*loader should delete the data before loading new rows.
- 2) Multiple INTO clauses load two tables, EMP and PROJ. The same set of records is processed three times, using different combinations of columns each time to load table PROJ.
- 3) WHEN loads only rows with non-blank project numbers. When PROJNO is defined as columns 25...27, rows are inserted into PROJ only if there is a value in those columns.
- 4) When PROJNO is defined as columns 29...31, rows are inserted into PROJ only if there is a value in those columns.
- 5) When PROJNO is defined as columns 33...35, rows are inserted into PROJ only if there is a value in those columns.

The Data File

The following is datafile for Case 5:

```

1234 BAKER      10 9999 101 102 103
1234 JOKER      10 9999 777 888 999
2664 YOUNG      20 2893 425 abc 102
5321 OTOOLE     10 9999 321 55 40

```



```

2134 FARMER      20 4555 236 456
2414 LITTLE     20 5634 236 456  40
6542 LEE        10 4532 102 321  14
2849 EDDS       xx 4555      294  40
4532 PERKINS    10 9999   40
1244 HUNT       11 3452 665 133 456
123  DOOLITTLE  12 9940      132
1453 MACDONALD  25 5532      200

```

Invoking SQL*Loader

Invoke SQL*Loader with a command such as:

```
sqlldr userid=scott/tiger control=ulcase5.ctl log=ulcase5.log
```



OSDoc

Additional Information: The command “sqlldr” is a UNIX-specific invocation. To invoke SQL*Loader on your operating system, see your Oracle operating system-specific documentation.

The Log File

The following is a portion of the log file:

```

Control File:      ULCASE5.CTL
Data File:         ULCASE5.DAT
  Bad File:        ULCASE5.BAD
  Discard File:    ULCASE5.DSC
(Allow all discards)

Number to load:    ALL
Number to skip:    0
Errors allowed:    50
Bind array:        64 rows, maximum of 65336 bytes
Continuation:      none specified
Path used:         Conventional

```

Table EMP, loaded from every logical record.
 Insert option in effect for this table: REPLACE

Column Name	Position	Len	Term	Encl	Datatype
-----	-----	---	----	----	-----
EMPNO	1:4	4			CHARACTER
ENAME	6:15	10			CHARACTER
DEPTNO	17:18	2			CHARACTER
MGR	20:23	4			CHARACTER

Table PROJ, loaded when PROJNO != 0x202020(character ' ')
 Insert option in effect for this table: REPLACE

Column Name	Position	Len	Term	Encl	Datatype
-----	-----	---	----	----	-----
EMPNO	1:4	4			CHARACTER
PROJNO	25:27	3			CHARACTER

Table PROJ, loaded when PROJNO != 0x202020(character ' ')
 Insert option in effect for this table: REPLACE

Column Name	Position	Len	Term	Encl	Datatype
-----	-----	---	----	----	-----
EMPNO	1:4	4			CHARACTER
PROJNO	29:31	3			CHARACTER

Table PROJ, loaded when PROJNO != 0x202020(character ' ')
 Insert option in effect for this table: REPLACE

Column Name	Position	Len	Term	Encl	Datatype
-----	-----	---	----	----	-----
EMPNO	1:4	4			CHARACTER
PROJNO	33:35	3			CHARACTER

- 1) Record 2: Rejected - Error on table EMP, column DEPTNO.
- 1) ORA-00001: unique constraint (SCOTT.EMPPIX) violated
- 1) ORA-01722: invalid number

- 1) Record 8: Rejected - Error on table EMP, column DEPTNO.
- 1) ORA-01722: invalid number

- 1) Record 3: Rejected - Error on table PROJ, column PROJNO.
- 1) ORA-01722: invalid number

Table EMP:

- 2) 9 Rows successfully loaded.
- 2) 3 Rows not loaded due to data errors.
- 2) 0 Rows not loaded because all WHEN clauses were failed.
- 2) 0 Rows not loaded because all fields were null.

Table PROJ:

- 3) 7 Rows successfully loaded.
- 3) 2 Rows not loaded due to data errors.
- 3) 3 Rows not loaded because all WHEN clauses were failed.
- 3) 0 Rows not loaded because all fields were null.

Table PROJ:

- 4) 7 Rows successfully loaded.
- 4) 3 Rows not loaded due to data errors.
- 4) 2 Rows not loaded because all WHEN clauses were failed.
- 4) 0 Rows not loaded because all fields were null.

Table PROJ:

- 5) 6 Rows successfully loaded.
- 5) 3 Rows not loaded due to data errors.
- 5) 3 Rows not loaded because all WHEN clauses were failed.
- 5) 0 Rows not loaded because all fields were null.

Space allocated for bind array: 5120 bytes (64 rows)

Space allocated for memory besides bind array: 46763 bytes

Total logical records skipped: 0
Total logical records read: 12
Total logical records rejected: 3
Total logical records discarded: 0

Notes:

- 1) Errors are not encountered in the same order as the physical records due to buffering (array batch). The bad file and discard file contain records in the same order as they appear in the log file.
- 2) Of the 12 logical records for input, three rows were rejected (rows for JOKER, YOUNG, and EDDS). No data was loaded for any of the rejected records.
- 3) Nine records met the WHEN clause criteria, and two (JOKER and YOUNG) were rejected due to data errors.
- 4) Ten records met the WHEN clause criteria, and three (JOKER, YOUNG, and EDDS) were rejected due to data errors.
- 5) Nine records met the WHEN clause criteria, and three (JOKER, YOUNG, and EDDS) were rejected due to data errors.

The Loaded Tables

These are results of this execution of SQL*Loader:

```
SQL> SELECT empno, ename, mgr, deptno FROM emp;
```

EMPNO	ENAME	MGR	DEPTNO
-----	-----	-----	-----
1234	BAKER	9999	10
5321	OTOOLE	9999	10
2134	FARMER	4555	20
2414	LITTLE	5634	20
6542	LEE	4532	10
4532	PERKINS	9999	10
1244	HUNT	3452	11
123	DOOLITTLE	9940	12
1453	MACDONALD	5532	25

```
SQL> SELECT * from PROJ order by EMPNO;
```

EMPNO	PROJNO
-----	-----
123	132
1234	101
1234	103
1234	102
1244	665
1244	456
1244	133
1453	200
2134	236
2134	456
2414	236
2414	456
2414	40
4532	40
5321	321
5321	40
5321	55
6542	102
6542	14
6542	321

Case 6: Loading using the Direct Path Load Method

This case study loads the EMP table using the direct path load method and concurrently builds all indexes. It illustrates the following functions:

- Use of the direct path load method to load and index data. See Chapter 8.
- How to specify the indexes for which the data is pre-sorted. See page 8 – 13.
- Loading all-blank numeric fields as null. See page 5 – 69.
- The NULLIF clause. See page 5 – 68.

Note: Specify the name of the table into which you want to load data; otherwise, you will see LDR-927. Specifying DIRECT=TRUE as a command-line parameter is not an option when loading into a synonym for a table.

In this example, field positions and datatypes are specified explicitly.

The Control File

The control file is ULCASE6.CTL.

```
LOAD DATA
INFILE 'ulcase6.dat'
INSERT
INTO TABLE emp
1) SORTED INDEXES (empix)
2) (empno POSITION(01:04) INTEGER EXTERNAL NULLIF empno=BLANKS,
    ename POSITION(06:15) CHAR,
    job POSITION(17:25) CHAR,
    mgr POSITION(27:30) INTEGER EXTERNAL NULLIF mgr=BLANKS,
    sal POSITION(32:39) DECIMAL EXTERNAL NULLIF sal=BLANKS,
    comm POSITION(41:48) DECIMAL EXTERNAL NULLIF comm=BLANKS,
    deptno POSITION(50:51) INTEGER EXTERNAL NULLIF deptno=BLANKS)
```

Notes:

- 1) The SORTED INDEXES clause identifies the indexes on which the data is sorted. This clause indicates that the datafile is sorted on the columns in the EMPIX index. This clause allows SQL*Loader to optimize index creation by eliminating the sort phase for this data when using the direct path load method.
- 2) The NULLIF...BLANKS clause specifies that the column should be loaded as NULL if the field in the datafile consists of all blanks. For more information, refer to “Loading All-Blank Fields” on page 5 – 69.

Invoking SQL*Loader Invoke SQL*Loader with a command such as:

```
sqlldr scott/tiger ulcase6.ctl log=ulcase6.log direct=true
```



Additional Information: The command “sqlldr” is a UNIX-specific invocation. To invoke SQL*Loader on your operating system, see your Oracle operating system-specific documentation.

The Log File

The following is a portion of the log file:

```
Control File:      ULCASE6.CTL
Data File:        ULCASE6.DAT
  Bad File:       ULCASE6.BAD
  Discard File:    none specified
(Allow all discards)
```

```
Number to load: ALL
Number to skip: 0
Errors allowed: 50
Continuation:     none specified
Path used:        Direct
```

Table EMP, loaded from every logical record.
Insert option in effect for this table: REPLACE

Column Name	Position	Len	Term	Encl	Datatype
-----	-----	-----	----	----	-----
EMPNO	1:4	4			CHARACTER
ENAME	6:15	10			CHARACTER
JOB	17:25	9			CHARACTER
MGR	27:30	4			CHARACTER
SAL	32:39	8			CHARACTER
COMM	41:48	8			CHARACTER
DEPTNO	50:51	2			CHARACTER

Column EMPNO is NULL if EMPNO = BLANKS
Column MGR is NULL if MGR = BLANKS
Column SAL is NULL if SAL = BLANKS
Column COMM is NULL if COMM = BLANKS
Column DEPTNO is NULL if DEPTNO = BLANKS

The following index(es) on table EMP were processed:
Index EMPIDX was loaded.

```
Table EMP:
7 Rows successfully loaded.
0 Rows not loaded due to data errors.
0 Rows not loaded because all WHEN clauses were failed.
0 Rows not loaded because all fields were null.

Bind array size not used in direct path.
Space allocated for memory besides bind array:      164342 bytes

Total logical records skipped:      0
Total logical records read:          7
Total logical records rejected:      0
Total logical records discarded:     0
```

Case 7: Extracting Data from a Formatted Report

In this case study, SQL*Loader’s string processing functions extract data from a formatted report. It illustrates the following functions:

- Using SQL*Loader with an INSERT trigger (see Chapter 7, “Using Database Triggers”, in the *Oracle7 Server Application Developer’s Guide*)
- Use of the SQL string to manipulate data. See page 5 – 75.
- Different initial and trailing delimiters. See page 5 – 60.
- Use of SYSDATE. See page 5 – 46.
- Use of the TRAILING NULLCOLS clause. See page 5 – 36.
- Ambiguous field length warnings. See pages 5 – 55 and 5 – 61.

Note: This example creates a trigger that uses the last value of unspecified fields.

The Data File

The following listing of the report shows the data to be loaded:

Today’s Newly Hired Employees							
Dept	Job	Manager	MgrNo	Emp Name	EmpNo	Salary	(Comm)
20	Salesman	Blake	7698	Shepard	8061	\$1,600.00	(3%)
				Falstaff	8066	\$1,250.00	(5%)
				Major	8064	\$1,250.00	(14%)
30	Clerk	Scott	7788	Conrad	8062	\$1,100.00	
		Ford	7369	DeSilva	8063	\$800.00	
	Manager	King	7839	Provo	8065	\$2,975.00	

Insert Trigger

In this case, a BEFORE INSERT trigger is required to fill in department number, job name, and manager's number when these fields are not present on a data line. When values are present, they should be saved in a global variable. When values are not present, the global variables are used.

The INSERT trigger and the package defining the global variables is:

```
CREATE OR REPLACE PACKAGE uldemo7 AS    -- Global Package Variables
    last_deptno    NUMBER(2);
    last_job       VARCHAR2(9);
    last_mgr       NUMBER(4);
END uldemo7;

/

CREATE OR REPLACE TRIGGER uldemo7_emp_insert
    BEFORE INSERT ON emp
    FOR EACH ROW
BEGIN
    IF :new.deptno IS NOT NULL THEN
        uldemo7.last_deptno := :new.deptno; -- save value for later
    ELSE
        :new.deptno := uldemo7.last_deptno; -- use last valid value
    END IF;
    IF :new.job IS NOT NULL THEN
        uldemo7.last_job := :new.job;
    ELSE
        :new.job := uldemo7.last_job;
    END IF;
    IF :new.mgr IS NOT NULL THEN
        uldemo7.last_mgr := :new.mgr;
    ELSE
        :new.mgr := uldemo7.last_mgr;
    END IF;
END;

/
```

Note: The phrase FOR EACH ROW is important. If it was not specified, the INSERT trigger would only fire once for each array of inserts because SQL*Loader uses the array interface.

The Control File

The control file is ULCASE7.CTL.

```
LOAD DATA
INFILE 'ULCASE7.DAT'
APPEND
INTO TABLE emp
1)   WHEN (57) = ' .'
2)   TRAILING NULLCOLS
3)   (hiredate SYSDATE,
4)     deptno POSITION(1:2)  INTEGER EXTERNAL(3)
5)     NULLIF deptno=BLANKS,
        job    POSITION(7:14) CHAR   TERMINATED BY WHITESPACE
6)     NULLIF job=BLANKS   "UPPER(:job)",
7)     mgr    POSITION(28:31) INTEGER EXTERNAL
        TERMINATED BY WHITESPACE, NULLIF mgr=BLANKS,
        ename  POSITION(34:41) CHAR
        TERMINATED BY WHITESPACE "UPPER(:ename)",
        empno  POSITION(45)  INTEGER EXTERNAL
        TERMINATED BY WHITESPACE,
        sal    POSITION(51) CHAR   TERMINATED BY WHITESPACE
8)     "TO_NUMBER(:sal, '$99,999.99')",
9)     comm    INTEGER EXTERNAL  ENCLOSED BY '(' AND '%'
        ":comm * 100"
)
```

Notes:

- 1) The decimal point in column 57 (the salary field) identifies a line with data on it. All other lines in the report are discarded.
- 2) The TRAILING NULLCOLS clause causes SQL*Loader to treat any fields that are missing at the end of a record as null. Because the commission field is not present for every record, this clause says to load a null commission instead of rejecting the record when only six fields are found instead of the expected seven.
- 3) Employee's hire date is filled in using the current system date.
- 4) This specification generates a warning message because the specified length does not agree with the length determined by the field's position. The specified length (3) is used.
- 5) Because the report only shows department number, job, and manager when the value changes, these fields may be blank. This control file causes them to be loaded as null, and an RDBMS insert trigger fills in the last valid value.
- 6) The SQL string changes the job name to uppercase letters.

7) It is necessary to specify starting position here. If the job field and the manager field were both blank, then the job field's TERMINATED BY BLANKS clause would cause SQL*Loader to scan forward to the employee name field. Without the POSITION clause, the employee name field would be mistakenly interpreted as the manager field.

8) Here, the SQL string translates the field from a formatted character string into a number. The numeric value takes less space and can be printed with a variety of formatting options.

9) In this case, different initial and trailing delimiters pick the numeric value out of a formatted field. The SQL string then converts the value to its stored form.

Invoking SQL*Loader Invoke SQL*Loader with a command such as:

```
sqlldr scott/tiger ulcase7.ctl ulcase7.log
```

The Log File The following is a portion of the log file:

```
1) SQL*Loader-307: Warning: conflicting lengths 2 and 3 specified
   for column EMP.DEPTNO.
```

```
Control File:   ulcase7.ctl
Data File:      ulcase7.dat
  Bad File:     ulcase7.bad
  Discard File: none specified
```

```
(Allow all discards)
```

```
Number to load: ALL
Number to skip: 0
Errors allowed: 50
Bind array:     64 rows, maximum of 65536 bytes
Continuation:   none specified
Path used:      Conventional
```

```
Table EMP, loaded when 57:57 = 0X2e(character '.')
Insert option in effect for this table: APPEND
TRAILING NULLCOLS option in effect
```

Column Name	Position	Len	Term	Encl	Datatype
DEPTNO	1:2	3			CHARACTER
JOB	7:14	8	WHT		CHARACTER
MGR	28:31	4	WHT		CHARACTER
ENAME	34:41	8	WHT		CHARACTER
EMPNO	NEXT	*	WHT		CHARACTER
SAL	51	*	WHT		CHARACTER
COMM	NEXT	*	(CHARACTER
					%
HIREDATE	SYSDATE				

Column DEPTNO is NULL if DEPTNO = BLANKS

Column JOB is NULL if JOB = BLANKS

Column JOB had SQL string

"UPPER(:job)"

applied to it.

Column MGR is NULL if MGR = BLANKS

Column ENAME had SQL string

"UPPER(:ename)"

applied to it.

Column SAL had SQL string

"TO_NUMBER(:sal,'\$99,999.99')"

applied to it.

Column COMM had SQL string

":comm * 100"

applied to it.

- 2) Record 1: Discarded - failed all WHEN clauses.
- Record 2: Discarded - failed all WHEN clauses.
- Record 3: Discarded - failed all WHEN clauses.
- Record 4: Discarded - failed all WHEN clauses.
- Record 5: Discarded - failed all WHEN clauses.
- Record 6: Discarded - failed all WHEN clauses.
- Record 10: Discarded - failed all WHEN clauses.

Table EMP:

6 Rows successfully loaded.

0 Rows not loaded due to data errors.

- 2) 7 Rows not loaded because all WHEN clauses were failed.
- 0 Rows not loaded because all fields were null.

Space allocated for bind array: 52480 bytes(64 rows)

Space allocated for memory besides bind array: 108185 bytes

Total logical records skipped: 0

Total logical records read: 13

Total logical records rejected: 0

- 2) Total logical records discarded: 7

Notes:

- 1) A warning is generated by the difference between the specified length and the length derived from the position specification.
- 2) The 6 header lines at the top of the report are rejected, as is the blank separator line in the middle.

**Dropping the Insert
Trigger and the
Global-Variable
Package**

After running the example, use ULCASE7E.SQL to drop the insert trigger and global-variable package.

CHAPTER

5

SQL*Loader Control File Reference

This chapter describes the SQL*Loader data definition language (DDL) used to map data to Oracle format.

The information in this chapter falls into the following main categories:

- general syntactical information
- managing files
- managing data

The sections that belong to each category follow:

General Syntactical Information:

- control file description and guidelines (page 3 – 5)
- data definition language syntax (page 5 – 4)
- adding comments (page 5 – 11)
- specifying command-line parameters (page 5 – 12)
- specifying RECOVERABLE and UNRECOVERABLE (page 5 – 12)
- specifying filenames and database objects (page 5 – 13)

Managing Files:

- including data in the control file (page 5 – 15)
- identifying datafiles (page 5 – 16)
- Specifying READBUFFERS (page 5 – 18)
- specifying datafile format and buffering (page 5 – 18)
- specifying the bad file (page 5 – 19)
- rejected records (page 5 – 20)
- specifying the discard file (page 5 – 21)
- discarded records (page 5 – 23)
- handling different character encoding schemes (page 5 – 24)
- loading data for different countries (page 5 – 24)
- loading into non-empty database tables (page 5 – 25)
- continuing interrupted loads (page 5 – 27)
- assembling logical records from physical records (page 5 – 29)

Managing Data:

- loading logical records into tables (page 5 – 33)
- specifying field conditions (page 5 – 37)
- specifying columns and fields (page 5 – 39)
- specifying the position of a data field (page 5 – 40)
- using multiple INTO TABLE clauses (page 5 – 42)
- generating data (page 5 – 45)
- loading without files (page 5 – 45)
- specifying datatypes (page 5 – 49)
- loading data across different operating systems (page 5 – 62)
- determining bind array size (page 5 – 63)
- setting a column to null or zero (page 5 – 67)
- loading all-blank fields (page 5 – 69)
- trimming of blanks and tabs (page 5 – 69)
- preserving whitespace (page 5 – 74)
- applying SQL operators to fields (page 5 – 75)

Data Definition Language (DDL) Syntax

Syntax Notation

Brackets []	<p>Brackets enclose optional items. For example:</p> <pre>[INTERNAL]</pre> <p>If you include the optional item, omit the brackets.</p>
Vertical Bar	<p>A vertical bar separates alternative items within brackets or braces. For example:</p> <pre>[ASC DESC]</pre> <p>Type one of the items, but omit the vertical bar and brackets.</p>
Braces { }	<p>Braces enclose two or more alternative mandatory items. For example:</p> <pre>DEFINE { procedure_name function_name }</pre> <p>Type one of the items, but omit the braces and vertical bar.</p>
Ellipsis ...	<p>An ellipsis represents an arbitrary number of one or more similar items. For example:</p> <pre>CONCAT (column1, column2, ... columnN)</pre> <p>where $N \geq 3$.</p> <p>Type similar items and delimiters as required, but omit the ellipsis.</p>

Symbols

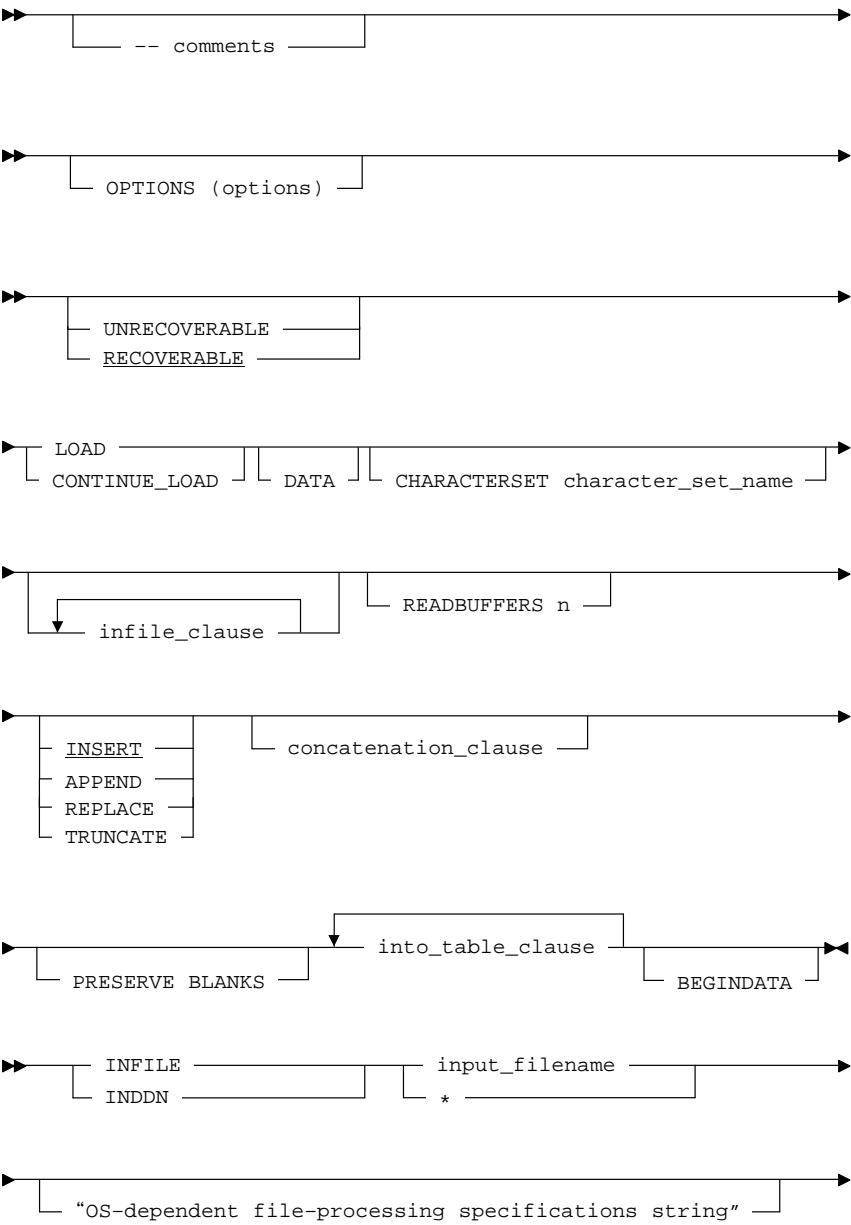
Type the following symbols precisely as they appear in the syntax notation:

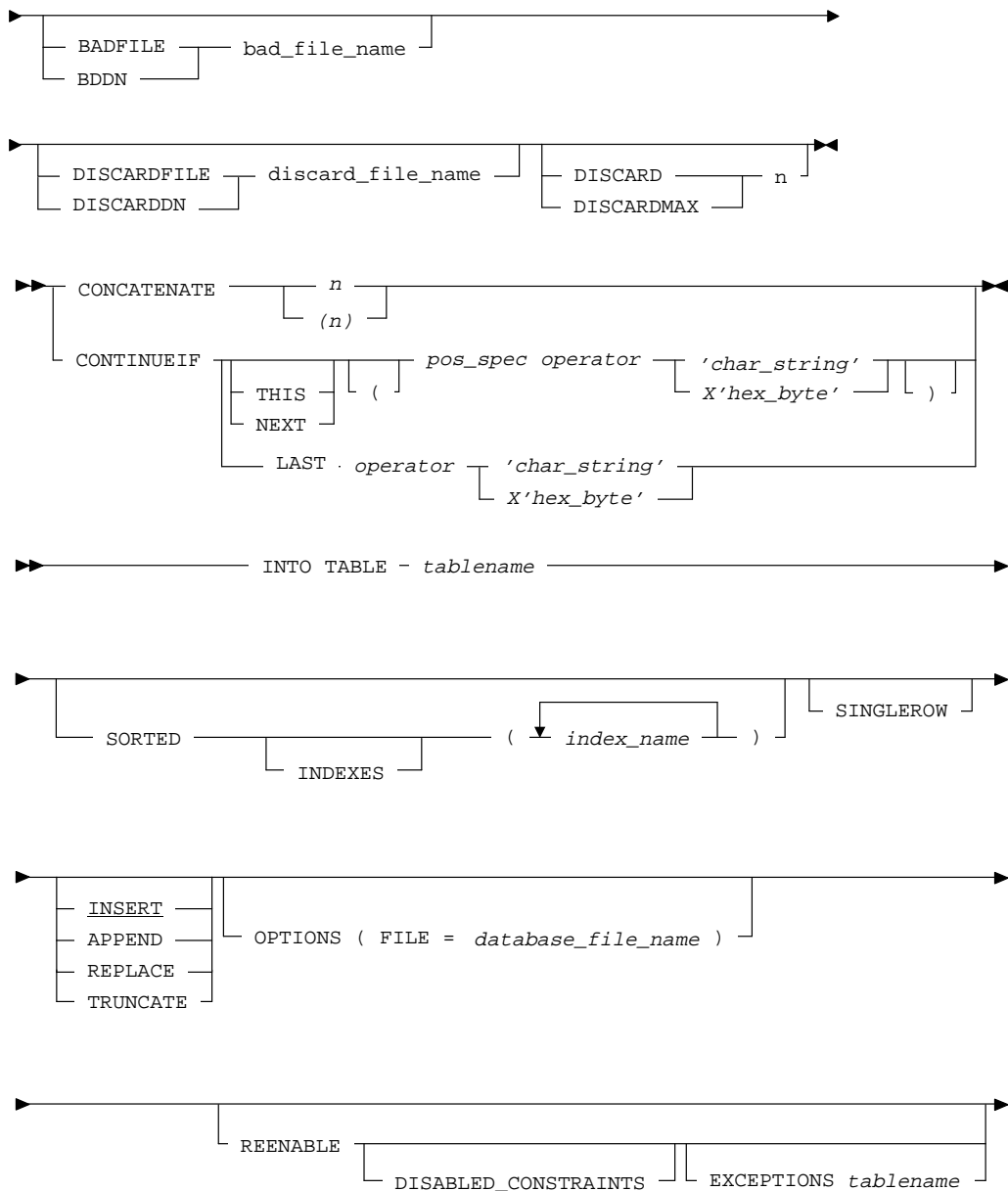
.	Period
,	Comma
–	Hyphen
*	Asterisk
;	Semicolon
:	Colon
=	Equal sign
\	Backslash
'	Single quote
”	Double quote
()	Parentheses
<	Less than
>	Greater than

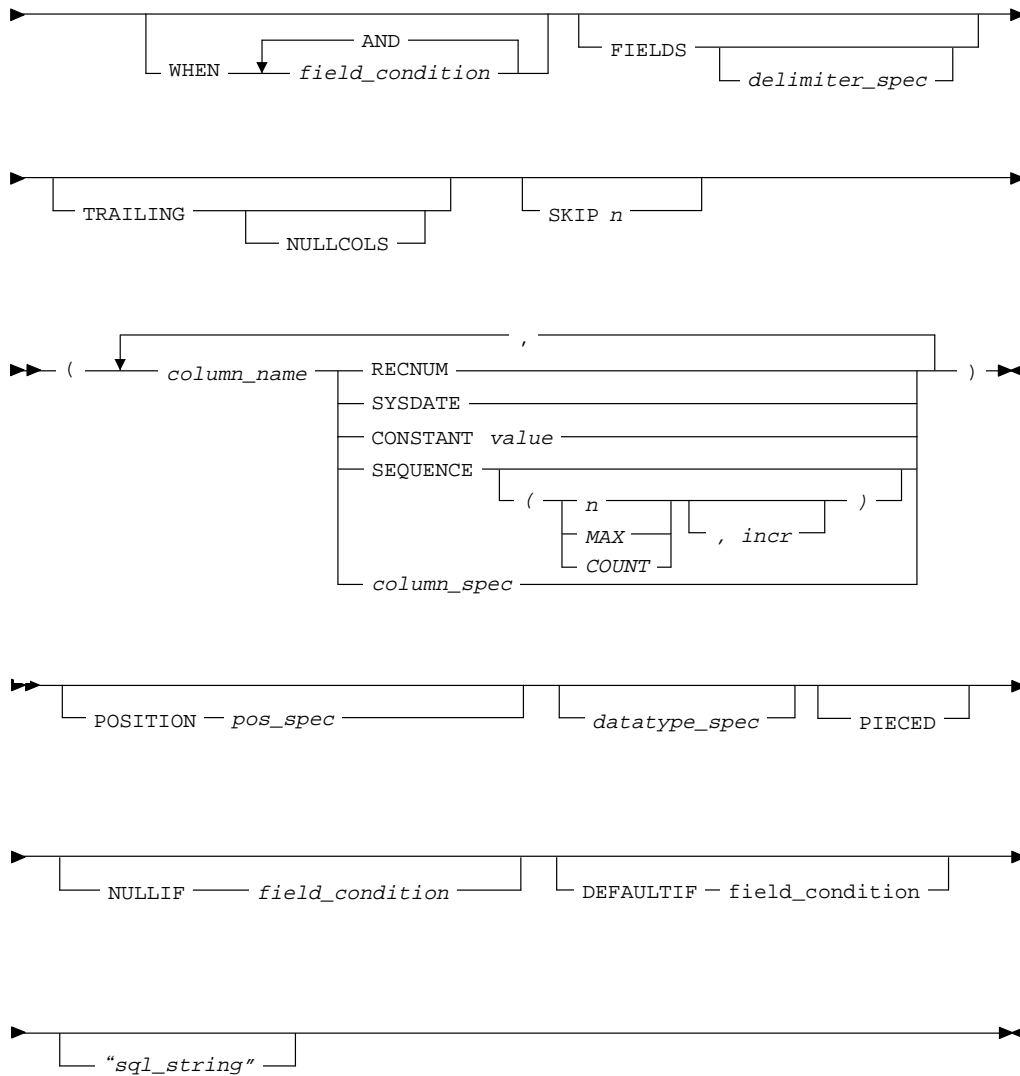
For details of the notation used in the syntax diagrams in this Reference, see the *PL/SQL User's Guide and Reference* or the preface in the *Oracle7 Server SQL Reference*.

High-Level Syntax
Diagrams

The following diagrams of DDL definitions are shown with certain clauses collapsed (e.g. position_spec, into_table clause, etc.)The statements are expanded and explained in more detail in later sections.





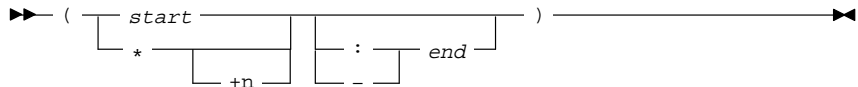


Expanded Clauses and Their Functionality

Position Specification

pos_spec

A position specification (*pos_spec*) gives the starting location for a field and, optionally, the ending location as well. A *pos_spec* is specified as follows:

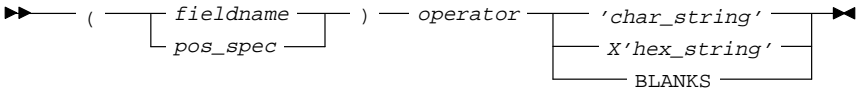


The position must be surrounded by parentheses. The starting location may be specified as a column number, as * (next column), or *+n (next column plus an offset). The *start* and *end* locations may be separated with either a colon (:) or a dash (-).

Field Condition

field_condition

A field condition compares a named field or an area of the record to some value. When the condition evaluates to true, the specified function is performed. For example, a true condition might cause the NULLIF function to insert a NULL data value, or cause DEFAULTIF to insert a default value. The *field_condition* is specified as follows:



The *char_string* and *hex_string* can be enclosed in either single quotation marks or double quotation marks. The *hex_string* is a string of hexadecimal digits, where each pair of digits corresponds to one byte in the field. The BLANKS keyword allows you to test a field to see if it consists entirely of blanks. It is necessary when you are loading delimited data and you cannot predict the length of the field, or when using a multi-byte character set that has multiple blanks.

There must not be any spaces between the operator and the operands on either side of it. Thus,

(1)='x'

is legal, while

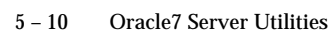
(1) = 'x'

generates an error.

column_name

The *datatype_spec* tells SQL*Loader how to interpret the field in the input record. The syntax is as follows:

`datatype_spec`



Precision vs. Length
precision
length

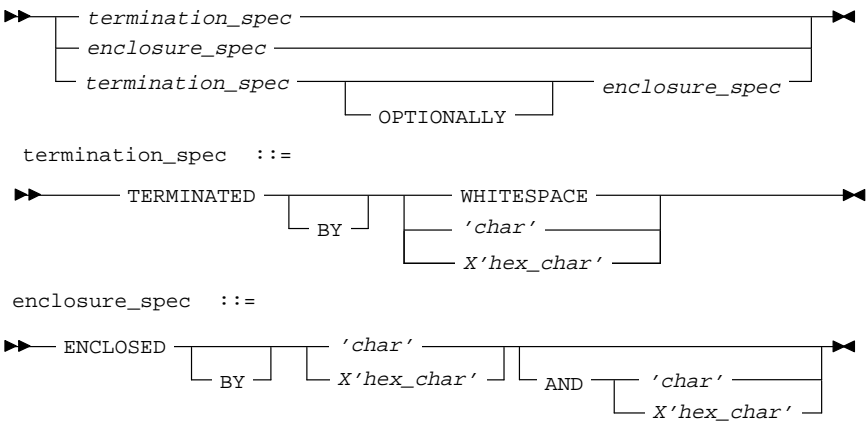
The precision of a numeric field is the number of digits it contains. The length of a numeric field is the number of byte positions on the record. The byte length of a ZONED decimal field is the same as its precision. However, the byte length of a (packed) DECIMAL field is $(p+1)/2$, rounded up, where p is the number's precision, because packed numbers contain two digits (or digit and sign) per byte.

Date Mask

The date mask specifies the format of the date value. For more information, see the DATE datatype on page 5 – 56.

Delimiter Specification
delimiter_spec

The *delimiter_spec* can specify a termination delimiter, enclosure delimiters, or a combination of the two, as shown below:



For more information, see “Specifying Delimiters” on page 5 – 58.

Comments

Comments can appear anywhere in the command section of the file, but they should not appear in the data. Precede comments with a double hyphen, which may appear anywhere on a line. For example,

```
--This is a comment
```

All text to the right of the double hyphen is ignored, until the end of the line. Case 3 on page 4 – 8 contains an example in a control file.

Specifying Command-Line Parameters in the Control File

The **OPTIONS** statement is useful when you typically invoke a control file with the same set of options, or when the number of arguments makes the command line very long. The **OPTION** statement precedes the **LOAD DATA** statement.

OPTIONS

This keyword allows you to specify runtime arguments in the control file, rather than on the command line. The following arguments can be specified with the **OPTIONS** keyword. They are described in Chapter 6, “SQL Loader Command-Line Reference”.

```
SKIP = n
LOAD = n
ERRORS = n
ROWS = n
BINDSIZE = n
SILENT = {FEEDBACK | ERRORS | DISCARDS | ALL}
DIRECT = {TRUE | FALSE}
PARALLEL = {TRUE | FALSE}
```

For example:

```
OPTIONS (BINDSIZE=100000, SILENT=(ERRORS, FEEDBACK) )
```

Values specified on the command line override values specified in the **OPTIONS** statement of the control file. The **OPTIONS** keyword file establishes default values that are easily changed from the command line.

Specifying RECOVERABLE and UNRECOVERABLE

The following options apply to *direct path loads*:

RECOVERABLE	Loaded data is logged in the redo log. This option is the default for direct path loads. All conventional loads are recoverable.
UNRECOVERABLE	This option can be specified for a direct path load only. Loaded data is not logged, which improves load performance. (Other changes to the database <i>are</i> logged.) For details, see “Specifying UNRECOVERABLE” on page 8 – 15. This option cannot be specified with a conventional load.

Specifying Filenames and Database Objects

Database Object Names within Double Quotation Marks

This section explains how to use quotation marks for specifying database objects and filenames in the load control file. It also shows how the escape character is used in quoted strings.

SQL*Loader follows the SQL standard for specifying object names (for example, table and column names): SQL and SQL*Loader reserved words must be specified within double quotation marks. The reserved words most likely to be column names are:

COUNT	DATA	DATE	FORMAT
OPTIONS	PART	POSITION	

So if you had an inventory system with columns named PART, COUNT, and DATA, you would specify these column names within double quotation marks in your SQL*Loader control file. For example:

```
INTO TABLE inventory
(partnum    INTEGER,
 "PART"    CHAR(15),
 "COUNT"  INTEGER,
 "DATA"    VARCHAR2(30))
```

See Appendix B, “Reserved Words”, for a complete list of reserved words.

You use double quotation marks if the object name contains special characters other than those recognized by SQL (\$, #, _), or if the name is case sensitive.

SQL String within Double Quotation Marks

You also specify the SQL string within double quotation marks. The SQL string applies SQL operators to data fields. It is described on page 5 – 75.

Filenames within Single Quotation Marks

On many operating systems, attempting to specify a complete file pathname produces an error, due to the use of special characters other than \$, #, or _. Usually, putting the pathname within single quotation marks avoids the error. Filenames that use the backslash character, \, may require special treatment, as described in the section, “Using a Backslash in Filenames”, on page 5 – 14.

For example:

```
INFILE 'mydata.dat'
BADFILE 'mydata.bad'
```

Quotation Marks in Quoted Strings

SQL*Loader uses strings within double quotation marks and strings within single quotation marks in the control file. Each type of string can appear within the other.

Backslash Escape Character

In DDL syntax *only*, you can place a double quotation mark inside a string delimited by double quotation marks by preceding it with the escape character, \, whenever the escape character is allowed in the string. (The following section tells when the escape character is allowed.) The same holds true for putting a single quotation mark into a string delimited by single quotation marks. For example, a double quotation mark is included in the following string which points to the homedir\data\norm\myfile datafile by preceding it with \:

```
INFILE 'homedir\data\"norm\mydata'
```

To put the escape character itself into a string, enter it twice, like this: \\
For example:

"so\"far"	or	'so\"far'	is parsed as	so\"far
"'so\\far'"	or	'\\'so\\far\\''	is parsed as	'so\\far'
"so\\\\far"	or	'so\\\\far'	is parsed as	so\\far

Note: A double quote in the initial position cannot be escaped, therefore you should avoid creating strings with an initial quote.

Using a Backslash in Filenames

This section is of interest only to users of PCs and other systems that use backslash characters in file specifications. For all other systems, a backslash is always treated as an escape character, as described in the preceding section.

Non-Portable Strings

There are two kinds of character strings in a SQL*Loader control file that are not portable between operating systems: filename strings and file processing options strings. When converting to a different operating system, these strings must generally be rewritten. They are the *non-portable strings*. All other strings in a SQL*Loader control file are portable between operating systems.

Escaping the Backslash

If your operating system uses the backslash character to separate directories in a pathname *and* if the version of Oracle running on your operating system implements the backslash escape character for filenames and other non-portable strings, then you need to specify double backslashes in your pathnames and use single quotation marks.



Additional Information: To find out if your version of Oracle implements the backslash escape character for filenames, see your Oracle operating system-specific documentation.

For example, to load a file named “topdir\mydir\mydata”, you must specify:

```
INFILE 'topdir\\mydir\\mydata'
```

Escape Character Sometimes Disallowed

The version of Oracle running on your operating system may not implement the escape character for non-portable strings. When the escape character is disallowed, a backslash is treated as a normal character, rather than as an escape character (although it is still usable in all other strings). Then pathnames such as:

```
INFILE 'topdir\mydir\myfile'
```

can be specified normally. Double backslashes are not needed.

Because the backslash is not recognized as an escape character, strings within single quotation marks cannot be embedded inside another string delimited by single quotation marks. This rule also holds for double quotation marks: A string within double quotation marks cannot be embedded inside another string delimited by double quotation marks.

Determining If the Escape Character is Allowed



As previously mentioned, you can learn if the backslash is used as an escape character in non-portable strings by checking your operating-system-specific Oracle7 documentation. Another way is to specify “test\me” in the file processing options string. Then check the log file. If the log file shows the file processing options string as

```
"test\me"
```

then the backslash is *not* used as an escape character, and double backslashes are not required for file specifications.

However, if the log file shows the file processing options string as:

```
"testme"
```

then the backslash is treated as an escape character, and double backslashes are needed.

Including Data in the Control File with BEGINDATA

If your data is to be contained in the control file, it is placed at the end of the control specifications. You must place the BEGINDATA keyword before the first data record to separate the data from your data definitions. The syntax is:

```
BEGINDATA
```

This keyword is used with the INFILE keyword, described in the next section. Case 1 on page 4 – 4 contains an example.

If you omit BEGINDATA, SQL*Loader tries to interpret your data as control information, and you receive an error message. If the data is in a separate file, reaching the end of the control file signals that control information is complete, and BEGINDATA should not be used.

There should not be any spaces or other characters on the same line after the BEGINDATA clause. Otherwise, the line containing BEGINDATA is interpreted as the first line of data.

Do not put comments after BEGINDATA—they are also interpreted as data.

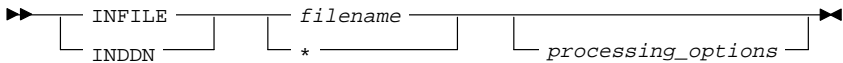
Identifying Datafiles

To specify the datafile fully, use a filename keyword, optionally followed by a file-processing options string. You may specify multiple files by using multiple INFILE keywords. You can also specify the datafile from the command line, using the DATA parameter described on page 6 – 4.

Naming the File

To specify the file containing the data to be loaded, use the INFILE or INDDN keyword, followed by the filename and optional processing options string. A filename specified on the command line overrides the first INFILE or INDDN keyword in the control file. If no filename is specified, the filename defaults to the control filename with an extension or file type of DAT.

If the control file also contains the data to be loaded, specify a filename of “*”. This specification works with the BEGINDATA keyword, described on page 5 – 15.



where:

INFILE or
INDDN

Either keyword may be used.

filename

Name of the file containing the data to be loaded.
May be any valid filename for your operating
system.

All filenames containing spaces or punctuation marks should be enclosed in single quotation marks. For more details, see “Specifying Filenames and Database Objects” on page 5 – 13.

- *** An asterisk (*) replaces a filename to specify that the data is in the control file. If multiple datafiles are specified, this specification must be first.
- processing_options*** This is the file-processing options string. It indicates datafile format. It also optimizes datafile reads. See “Specifying Datafile Format and Buffering” on page 5 – 18.

Specifying Multiple Datafiles

To load data from multiple datafiles in one run of SQL*Loader, use an INFILE statement for each datafile. Datafiles do not need the same file format, although the layout of the records must be identical. For example, two files could be specified with completely different file processing options strings, and a third could consist of data in the control file.

For each datafile, you can also specify a discard file and a bad file. These files should be declared after each datafile name. The following portion of a control file specifies four files:

```
INFILE mydat1.dat BADFILE mydat1.bad DISCARDFILE mydat1.dis
INFILE mydat2.dat
INFILE mydat3.dat DISCARDFILE mydat3.dis
INFILE mydat4.dat DISCARDMAX 10
```

For the first datafile (MYDAT1.DAT), both a bad file and discard file are explicitly named. So both files are created, if needed.

For the second datafile (MYDAT2.DAT), neither a bad file nor a discard file is specified. So only the bad file is created, if it is needed. If created, the bad file has a default filename and extension. The discard file is *not* created, even if rows are discarded.

For the third file (MYDAT3.DAT), the default bad file is created, if needed. A discard file with the given name is also created, if it is needed.

For the fourth file (MYDAT4.DAT), the default bad file is created, if needed. Because the DISCARDMAX option is used, SQL*Loader assumes that a discard file is wanted and creates it with the default name (MYDAT4.DSC), if it is needed.

Note: It is not possible to join physical records from separate datafiles into one logical record.

Examples of How to Specify a Datafile

In the first example, you specify that the data is contained in the control file itself:

```
INFILE *
```

In the next example, you specify that the data is contained in a file named WHIRL with the default file extension or file type of DAT:

```
INFILE WHIRL
```

The following example specifies the full path to a file:

```
INFILE 'c:/topdir/subdir/datafile.dat'
```


Note: Filenames that include spaces or punctuation marks should be enclosed in single quotation marks. For more details on filename specification, see “Specifying Filenames and Database Objects” on page 5 – 13.

Specifying READBUFFERS

The READBUFFERS keyword control memory usage. This clause can be specified for direct path loads only. For more information, see page 8 – 12.

Specifying Datafile Format and Buffering

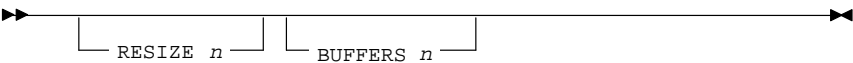
You specify an operating-system-dependent *file processing options string* to control file processing. You use this string to specify file format and buffering.



Additional Information: For details on the syntax of the file processing options string, see your Oracle operating system-specific documentation.

File Processing Example

For example, suppose that your operating system has the following option-string grammar:



where RECSIZE is the size of a fixed-length record, and BUFFERS is the number of buffers to use for asynchronous I/O.



OSDoc

Note: This example is operating system–specific and may not work on your operating system. For details on the syntax of the file processing options string on your system, see your Oracle operating system–specific documentation.

To declare a file named MYDATA.DAT as a file that contains 80–byte records and tell SQL*Loader to use eight I/O buffers with this syntax, you would use the following clause:

```
INFILE 'mydata.dat' "RECSIZE 80 BUFFERS 8"
```

Note: This example uses the recommended convention of single quotation marks for filenames and double quotation marks for everything else. See “Specifying Filenames and Database Objects” on page 5 – 13 for more details.

Specifying the Bad File

When SQL*Loader executes, it may create a file called a *bad file* or *reject file* where it places records that were rejected because of formatting errors or because they caused Oracle errors. The bad file is created according to the following rules:

- A bad file is created only if one or more records are rejected.
- If no records are rejected, then a bad file is not created.
- If the bad file is created, it overwrites an existing file with the same name.
- If a bad file is not created, then an existing file with the same name remains intact.



Suggestion: If a file exists with the same name as the bad file that SQL*Loader may create, delete or rename it before running SQL*Loader.



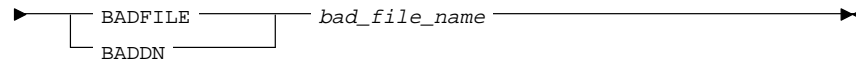
OSDoc

Additional Information: On some systems a new version of the file is created if a file with the same name already exists. See your Oracle operating system–specific documentation to find out if this is the case on your system.

To specify the name of this file, use the BADFILE or BADDN keyword, followed by the filename. If you do not specify a name for the bad file, the name defaults to the name of the datafile with an extension or file type of BAD. You can also specify the bad file from the command line with the BAD parameter described on page 6 – 3.

A filename specified on the command line is associated with the first INFILE or INDDN clause in the control file, overriding any bad file that may have been specified as part of that clause.

The bad file is created in the same record and file format as the datafile so that the data can be reloaded after corrections. The syntax is



where:

BADFILE or BADDN Either keyword may be used.

bad_file_name Any valid file specification, naming a file to receive rejected records.

Note: Filenames that include spaces or punctuation marks should be enclosed in single quotation marks. For more details on filename specification, see “Specifying Filenames and Database Objects” on page 5 – 13.

Examples of How to Specify a Bad File

In the following example, you specify a bad file with filename UGH and default file extension or file type of BAD:

```
BADFILE UGH
```

In the next examples, you specify a bad file with filename BAD0001 and file extension or file type of REJ:

```
BADDN BAD0001.REJ
BADDN '/REJECT_DIR/BAD0001.REJ'
```

Rejected Records

A record is rejected if it meets either of the following conditions:

- Upon insertion the record causes an Oracle error (such as invalid data for a given datatype).
- SQL*Loader cannot determine if the data is acceptable. That is, it cannot determine if the record meets WHEN-clause criteria, as in the case of a field that is missing its final delimiter.

If the data can be evaluated according to the WHEN-clause criteria (even with unbalanced delimiters) then it is either inserted or rejected.

If a record is rejected on insert, then no part of that record is inserted into any table. For example, if data in a record is to be inserted into multiple tables, and most of the inserts succeed, but one insert fails; then all the inserts from that record are rolled back. The record is then written to the bad file, where it can be corrected and reloaded. Previous inserts from records without errors are not affected.

The log file indicates the Oracle error for each rejected record. Case 4 on page 4 – 11 has an example of rejected records.

Integrity Constraints

All integrity constraints are honored for conventional path loads. On the direct path, some constraints are unenforceable. See Chapter 8 for more details.

Specifying the Discard File

As SQL*Loader executes, it may create a *discard file* for records that do not meet any of the loading criteria. The records contained in this file are called *discarded records*. Discarded records do not satisfy any of the WHEN clauses specified in the control file. These records are different from rejected records. Discarded records do not necessarily have any bad data. No insert is attempted on a discarded record.

The discard file is created according to the following rules:

- A discard file is only created if a discard filename is specified.
- A discard file is only created if one or more records fail to satisfy any of the WHEN clauses specified in the control file, even if it has been specified.
- If the discard file is created, it overwrites an existing file with the same name.
- If no records are discarded, then a discard file is not created.
- If a discard file is not created, then an existing file with the same name remains intact.



Suggestion: If a file exists with the same name as the discard file that SQL*Loader may create, delete or rename it before running SQL*Loader.

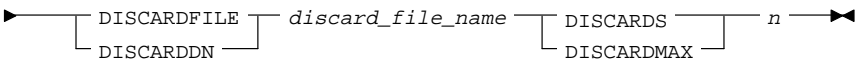
To create a discard file, use any of the following options:

<i>In a Control File</i>	<i>On the Command Line</i>
<code>DISCARDFILE filename</code>	<code>DISCARD</code>
<code>DISCARDN filename</code>	<code>DISCARDMAX</code>
<code>DISCARDS</code>	
<code>DISCARDMAX</code>	

Note that you can request the discard file directly with a parameter specifying its name, or indirectly by specifying the maximum number of discards.

Using a Control-File Definition

To specify the name of the file, use the DISCARDFILE or DISCARDN keyword, followed by the filename.



where:

DISCARDFILE or DISCARDN Either keyword may be used.

discard_file_name Any valid filename, specifying a file to receive discard records.

Note: Filenames that include spaces or punctuation marks should be enclosed in single quotation marks. For more details on filename specification, see “Specifying Filenames and Database Objects” on page 5 – 13.

The default filename is the name of the datafile, and the default file extension or file type is DSC. A discard filename specified on the command line overrides one specified in the control file. If a discard file with that name already exists, it is either overwritten or a new version is created, depending on your operating system.

The discard file is created with the same record and file format as the datafile. So it can easily be used for subsequent loads with the existing control file, after changing the WHEN clauses or editing the data.

Examples of How to Specify a Discard File

In the first example, you specify a discard file with filename CIRCULAR and default file extension or file type of DSC:

```
DISCARDFILE CIRCULAR
```

In this example, you specify a file extension or file type of MAY:

```
DISCARDN NOTAPPL.MAY
```

In the next example, you specify a full path to filename FORGET.ME:

```
DISCARDFILE '/DISCARD_DIR/FORGET.ME'
```

Discarded Records

If there is no INTO TABLE keyword specified for a record, the record is discarded. This situation occurs when every INTO TABLE keyword in the SQL*Loader control file has a WHEN clause; and either the record fails to match any of them or all fields are null.

No records are discarded if an INTO TABLE keyword is specified without a WHEN clause. An attempt is made to insert every record into such a table. So records may be rejected, but none are discarded.

Case 4 on page 4 – 11 has an example of a discard file.

Limiting the Number of Discards

You may limit the number of records to be discarded for each datafile with the clause:



The diagram shows a horizontal line with arrowheads at both ends. A bracket is drawn below the line, spanning a portion of its length. Inside the bracket, the word "DISCARD" is written above the word "DISCARDMAX". To the right of the bracket, the letter "n" is written. This represents the clause DISCARD n DISCARDMAX.

where *n* must be an integer. When the discard limit is reached, processing of that datafile terminates and continues with the next datafile, if one exists.

You can specify a different number of discards for each datafile. Alternatively, if the number of discards is only specified once, then the maximum number of discards is the same for all files.

If you specify a maximum number of discards, but no discard filename; SQL*Loader creates a discard file with the default filename and file extension or file type. Case 4 on page 4 – 11 has an example.

Using a Command-Line Parameter

You can specify the discard file from the command line, with the DISCARD parameter or the DISCARDMAX parameter described on page 6 – 4.

A filename specified on the command line goes with the first INFILE or INDDN clause in the control file, overriding any bad file that may have been specified as part of that clause.

Handling Different Character Encoding Schemes

This section describes the features that allow SQL*Loader to operate with different character encoding schemes (called character sets, or code pages). SQL*Loader uses Oracle's NLS (National Language Support) features to handle the different single-byte and multi-byte character encoding schemes used on different computers and in different countries.

Multi-Byte (Asian) Character Sets

Multi-byte character sets support Asian languages. Data can be loaded in multi-byte format, and database objects (fields, tables, and so on) can be specified with multi-byte characters. In the control file, comments and object names may also use multi-byte characters.

Input Character Conversion

SQL*Loader also has the capacity to convert data from the datafile character set to the database character set, when they are different. When using the conventional path, data is converted into the session character set specified by the NLS_LANG parameter for that session. Then the data is loaded using SQL INSERT statements. The session character set is the character set supported by your terminal.

During a direct path load, data converts directly into the database character set. As a consequence, the direct path load method allows data in a character set that is not supported by your terminal to be loaded.

When data conversion occurs, it is essential that the target character set contains a representation of all characters that exist in the data. Otherwise, characters that have no equivalent in the target character set are converted to a default character, with consequent loss of data. When using the direct path, load method the database character set should be a superset of, or equivalent to, the datafile character sets. Similarly, when using the conventional path, the session character set should be a superset of, or equivalent to, the datafile character sets.

The character set used in each input file is specified with the CHARACTERSET keyword.

CHARACTERSET Keyword

The CHARACTERSET definition tells SQL*Loader what character set is used in each datafile. Different datafiles can be specified with different character sets. Only one character set can be specified for each datafile.

Using the CHARACTERSET keyword causes character data to be automatically converted when it is loaded into Oracle. Only CHAR, DATE, and numeric EXTERNAL fields are affected. If the CHARACTERSET keyword is not specified, then no conversion occurs.

The syntax for this option is:

`CHARACTERSET character_set_spec`

where *character_set_spec* is the acronym used by Oracle to refer to your particular encoding scheme.

Additional Information: For more information on supported character sets, code pages, and the NLS_LANG parameter, see the National Language Support section of the *Oracle7 Server Reference*.

Control File Characterset

The SQL*Loader control file itself is assumed to be in the character set specified for your session by the NLS_LANG parameter. However, delimiters and comparison clause values must be specified to match the character set in use in the datafile. To ensure that the specifications are correct, it may be preferable to specify hexadecimal strings, rather than character string values.

Any data included after the BEGINDATA statement is also assumed to be in the character set specified for your session by the NLS_LANG parameter. Data that uses a different character set must be in a separate file.

Loading into Empty and Non-Empty Tables

You can specify one of the following methods for loading tables:



This section describes those methods.

How Non-Empty Tables are Affected

This section corresponds to the DB2 keyword RESUME; users of DB2 should also refer to the description of RESUME in Appendix C. If the tables you are loading already contain data, you have four choices for how SQL*Loader proceeds:

- | | |
|--------|--|
| INSERT | Returns an error and terminates the load if the table contains data. This option only loads data into empty tables. INSERT is the default. |
| APPEND | Extends the table by adding new rows. |

REPLACE	Deletes the rows in the table and loads the new data in its place.
TRUNCATE	TRUNCATEs the table and loads the new data in place of the old.



Warning: When the REPLACE or TRUNCATE keyword is specified, the entire *table* is replaced, not individual rows. After the rows are successfully deleted, a commit is issued. You cannot recover the data that was in the table before the load, unless it was saved with Export or a comparable utility.

The remainder of this section provides additional detail on these options.

INSERT

This is the default method. It requires the table to be empty before loading. SQL*Loader terminates with an error if the table contains rows. Case 1 on page 4 – 4 has an example.

APPEND

If data already exists in the table, SQL*Loader appends the new rows to it. If data doesn't already exist, the new rows are simply loaded. Case 3 on page 4 – 8 has an example.

REPLACE

All rows in the table are deleted and the new data is loaded. The table must be in your schema, or you must have DELETE privilege on the table. Case 4 on page 4 – 11 has an example.

The row deletes cause any delete triggers defined on the table to fire. If DELETE CASCADE has been specified for the table, then the cascaded deletes are carried out, as well. For more information on cascaded deletes, see the “Data Integrity” chapter of the *Oracle7 Server Concepts* manual.

Updating Existing Rows

The REPLACE method is a *table* replacement, not a replacement of individual rows. SQL*Loader does not update existing records, even if they have null columns. To update existing rows, use the following procedure:

1. Load your data into a temporary table.
2. Use the SQL language UPDATE statement with correlated subqueries.
3. Drop the temporary table.

For more information, see the “UPDATE” statement in the *Oracle7 Server SQL Reference*.

TRUNCATE

With this method, SQL*Loader uses the SQL TRUNCATE command to achieve the best possible performance. For the TRUNCATE command to operate, the table's referential integrity constraints must first be disabled. If they have not been disabled, SQL*Loader returns an error.

Once the integrity constraints have been disabled, DELETE CASCADE is no longer defined for the table. If the DELETE CASCADE functionality is needed, then the contents of the table must be manually deleted before the load begins.

The table must be in your schema, or you must have the DELETE ANY TABLE privilege.

Specifying One Method for All Tables

You specify one table-loading method that applies to all tables by placing the keyword before any INTO TABLE clauses. This choice applies to any table that does not have its own method. You can specify a table-loading method for a single table by including the keyword in the INTO TABLE clause, as described in “Loading Logical Records into Tables” on page 5 – 33.

Continuing an Interrupted Load

If SQL*Loader runs out of space for data rows or index entries, the load is discontinued. (For example, the table might reach its maximum number of extents.) Discontinued loads can be continued after more space is made available.

State of Tables and Indexes

When a load is discontinued, any data already loaded remains in the tables, and the tables are left in a valid state. If the conventional path is used, all indexes are left in a valid state.

If the direct path load method is used, any indexes that run out of space are left in direct load state. They must be dropped before the load can continue. Other indexes are valid provided no other errors occurred. (See “Indexes Left in Direct Load State” on page 8 – 9 for other reasons why an index might be left in direct load state.)

Using the Log File

SQL*Loader's log file tells you the state of the tables and indexes and the number of logical records already read from the input datafile. Use this information to resume the load where it left off.

Dropping Indexes

Before continuing a direct path load, inspect the SQL*Loader log file to make sure that no indexes are in direct load state. Any indexes that are left in direct load state must be dropped before continuing the load. The indexes can then be re-created either before continuing or after the load completes.

Continuing Single Table Loads

To continue a discontinued direct or conventional path load involving only one table, specify the number of logical records to skip with the command-line parameter SKIP. If the SQL*Loader log file says that 345 records were previously read, then the command to continue would look like this:

```
SQLLDR USERID=scott/tiger CONTROL=FAST1.CTL DIRECT=TRUE SKIP=345
```

Continuing Multiple Table Conventional Loads

It is not possible for multiple tables in a conventional path load to become unsynchronized. So a multiple table conventional path load can also be continued with the command-line parameter SKIP. Use the same procedure that you would use for single-table loads, as described in the preceding paragraph.

Continuing Multiple Table Direct Loads

If SQL*Loader cannot finish a multiple-table direct path load, the number of logical records processed could be different for each table. If so, the tables are not synchronized and continuing the load is slightly more complex.

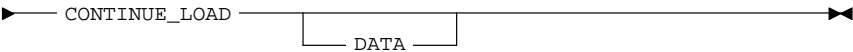
To continue a discontinued direct path load involving multiple tables, inspect the SQL*Loader log file to find out how many records were loaded into each table. If the numbers are the same, you can use the previously described simple continuation.

CONTINUE_LOAD

If the numbers are different, use the CONTINUE_LOAD keyword and specify SKIP at the table level, instead of at the load level. These statements exist to handle unsynchronized interrupted loads. Instead of specifying:

```
LOAD DATA...
```

at the start of the control file, specify:



SKIP

Then, for each INTO TABLE clause, specify the number of logical records to skip for that table using the SKIP keyword:

```
...  
INTO TABLE emp  
SKIP 2345  
...  
INTO TABLE dept  
SKIP 514  
...
```

Combining SKIP and CONTINUE_LOAD

The CONTINUE_LOAD keyword is only needed after a direct load failure because multiple table loads cannot become unsynchronized when using the conventional path.

If you specify CONTINUE_LOAD, you cannot use the command-line parameter SKIP. You must use the table-level SKIP clause. If you specify LOAD, you can optionally use the command-line parameter SKIP, but you cannot use the table-level SKIP clause.

Assembling Logical Records from Physical Records

You can create one logical record from multiple physical records using one of the following two clauses, depending on your data:

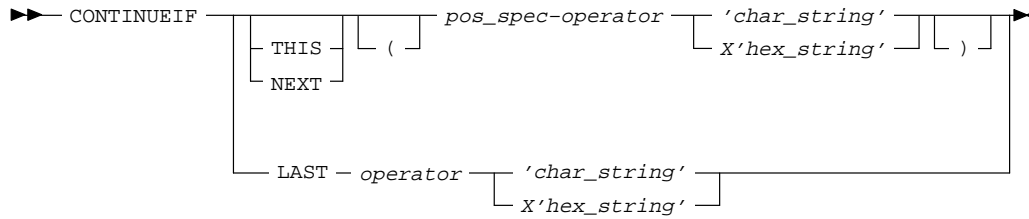
```
CONCATENATE  
CONTINUEIF
```

CONCATENATE is appropriate in the simplest case, when SQL*Loader should always add the same number of physical records to form one logical record. The syntax is:

```
CONCATENATE n
```

where *n* indicates the number of physical records to combine.

If the number of physical records to be continued varies, then CONTINUEIF must be used. The keyword CONTINUEIF is followed by a condition that is evaluated for each physical record, as it is read. For example, two records might be combined if there were a pound sign (#) in character position 80 of the first record. If any other character were there, the second record would not be added to the first. The full syntax for CONTINUEIF adds even more flexibility:



where:

THIS

If the condition is true in this record, then the next physical record is read and concatenated to the current physical record, continuing until the condition is false. If the condition is false in the current record, then the current physical record is the last physical record of the current logical record. THIS is the default.

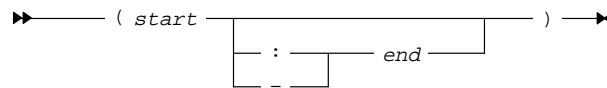
NEXT

If the condition is true in the next record, then the next physical record is concatenated to the current record, continuing until the condition is false.

If the condition is false in the next record, then the current physical record is the last physical record of the current logical record.

pos_spec

Indicates starting and ending column numbers in the physical record, as shown below:



Column numbers start with 1. Either a hyphen or a colon is acceptable (*start-end* or *start:end*).

If you omit *end*, the length of the continuation field is the length of the byte string or character string. If you use *end*, and the length of the resulting continuation field is not the same as that of the byte string or character string, the shorter one is padded. Character strings are padded with blanks, hexadecimal strings with zeros.

LAST	This test is similar to THIS, but the test is always against the last non-blank character. If the last non-blank character in this physical record meets the test, then the next physical record is read and concatenated to the current physical record, continuing until the condition is false. If the condition is false in the current record, then the current physical record is the last physical record of the current logical record.
operator	<p>The supported operators are <i>equal</i> and <i>not equal</i>:</p> <p>= != <=></p> <p>For the <i>equal</i> operator, the field and comparison string must match exactly for the condition to be true. For the <i>not equal</i> operator, they may differ in any character.</p>
char_string	A string of characters to be compared to the continuation field defined by <i>start</i> and <i>end</i> , according to the operator. The string must be enclosed in double or single quotation marks. The comparison is made character by character, blank padding on the right if necessary.
X'hex_string'	A string of bytes in hexadecimal format, used in the same way as the character string above. X'1FB033' would represent the three bytes with values 1F, B0 and 33 (hex).

Note: The positions in the CONTINUEIF clause refer to positions in each physical record. This is the only time you refer to character positions in physical records. All other references are to logical records.

For CONTINUEIF THIS and CONTINUEIF NEXT, the continuation field is removed from all physical records before the logical record is assembled. This allows data values to span the records with no extra characters (continuation characters) in the middle. Two examples showing CONTINUEIF THIS and CONTINUEIF NEXT follow:

CONTINUEIF THIS	CONTINUEIF NEXT
(1:2) = '%%'	(1:2) = '%%'

Assume physical data records 12 characters long and that a period means a space:

%aaaaaaaaa...	..aaaaaaaaa...
%bbbbbbbbb...	%bbbbbbbbb...
..cccccccc...	%cccccccc...
%dddddddddd..	..dddddddddd..
%eeeeeeeeeee..	%eeeeeeeeeee..
..ffffffffff..	%ffffffffff..

The logical records would be the same in each case:

```
aaaaaaaaa...bbbbbbbbb...cccccccc...  
dddddddddd..eeeeeeeeeee..ffffffffff..
```

Notes:

- CONTINUEIF LAST differs from CONTINUEIF THIS and CONTINUEIF NEXT. With CONTINUEIF LAST the continuation character is *not* removed from the physical record. Instead, this character is included when the logical record is assembled.
- Trailing blanks in the physical records *are* part of the logical records.

Examples of How to Specify CONTINUEIF

In the first example, you specify that if the current physical record (record1) has an asterisk in column 1. Then the next physical record (record2) should be appended to it. If record2 also has an asterisk in column 1, then record3 is appended also.

If record2 does not have an asterisk in column 1, then it is still appended to record1, but record3 begins a new logical record.

```
CONTINUEIF THIS (1) = "*"
```

In the next example, you specify that if the current physical record (record1) has a comma in the last non-blank data column. Then the next physical record (record2) should be appended to it. If a record does not have a comma in the last column, it is the last physical record of the current logical record.

```
CONTINUEIF LAST = ", "
```

In the last example, you specify that if the next physical record (record2) has a "10" in columns 7 and 8. Then it should be appended to the preceding physical record (record1). If a record does not have a "10" in columns 7 and 8, then it begins a new logical record.

```
CONTINUEIF NEXT (7:8) = '10'
```

Case 4 on page 4 – 11 shows the CONTINUEIF clause in use.

Loading Logical Records into Tables

This section describes the way in which you specify:

- which tables you want to load
- which records you want to load into them
- default characteristics for the columns in those records

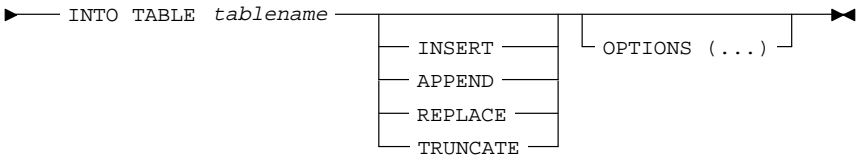
Specifying Table Names

The INTO TABLE keyword of the LOAD DATA statement allows you to identify tables, fields, and datatypes. It defines the relationship between records in the datafile and tables in the database. The specification of fields and datatypes is described in later sections.

INTO TABLE

Among its many functions, the INTO TABLE keyword allows you to specify the table into which you load data. To load multiple tables, you include one INTO TABLE clause for each table you wish to load.

To begin an INTO TABLE clause, use the keywords INTO TABLE, followed by the name of the Oracle table that is to receive the data.



The table must already exist. The table name should be enclosed in double quotation marks if it is the same as any SQL or SQL*Loader keyword, if it contains any special characters, or if it is case sensitive.

```
INTO TABLE SCOTT."COMMENT"
INTO TABLE SCOTT."comment"
INTO TABLE SCOTT."-COMMENT"
```

The user running SQL*Loader should have INSERT privileges on the table. Otherwise, the table name should be prefixed by the username of the owner as follows:

```
INTO TABLE SOPHIA.EMP
```

Table-Specific Loading Method

The INTO TABLE clause may include a table-specific loading method (INSERT, APPEND, REPLACE, or TRUNCATE) that applies only to that table. Specifying one of these methods within the INTO TABLE clause overrides the global table-loading method. The global table-loading method is INSERT, by default, unless a different method was specified before any INTO TABLE clauses. For more information on these options, see “Loading into Empty and Non-Empty Tables” on page 5 – 25.

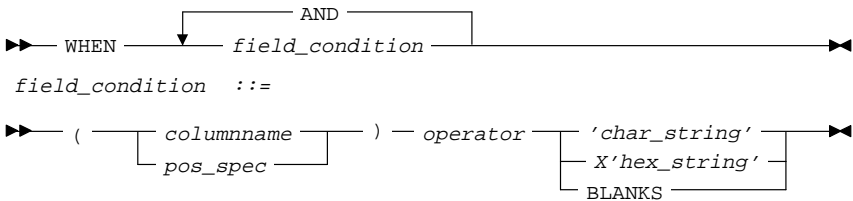
**Table-Specific
OPTIONS keyword**

The OPTIONS keyword can be specified for individual tables in a parallel load. (It is only valid for a parallel load.) For more information, see “Parallel Data Loading” beginning on page 8 – 21.

**Choosing which Rows
to Load**

You can choose to load or discard a logical record by using the WHEN clause to test a condition in the record.

The WHEN clause appears after the table name and is followed by one or more field conditions.



For example, the following clause indicates that any record with the value “q” in the fifth column position should be loaded:

```
WHEN (5) = 'q'
```

A WHEN clause can contain several comparisons provided each is preceded by AND. Parentheses are optional, but should be used for clarity with multiple comparisons joined by AND. For example

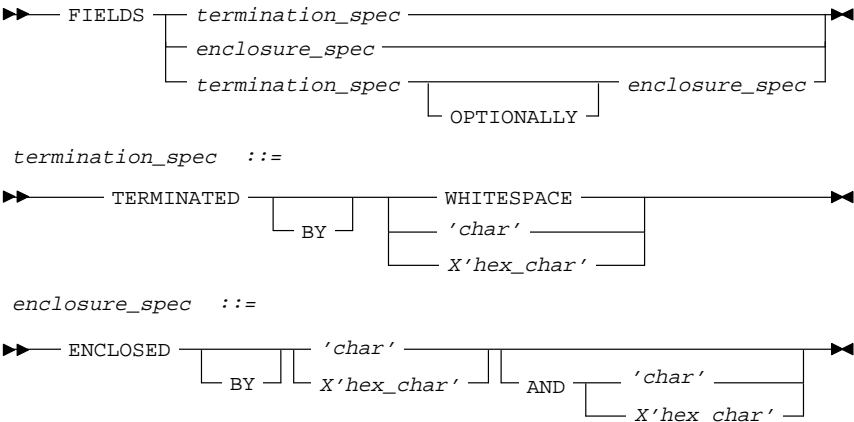
```
WHEN (DEPTNO = '10') AND (JOB = 'SALES')
```

To evaluate the WHEN clause, SQL*Loader first determines the values of all the fields in the record. Then the WHEN clause is evaluated. A row is inserted into the table only if the WHEN clause is true.

Field conditions are discussed in detail on page 5 – 37. Case 5 on page 4 – 14 shows the WHEN clause in use.

Specifying Default Data Delimiters

If all data fields are terminated similarly in the datafile, you can use the FIELDS clause to indicate the default delimiters. The syntax is:



You can override the delimiter for any given column by specifying it after the column name. Case 3 on page 4 – 8 contains an example. See “Specifying Delimiters” on page 5 – 58 for more information on delimiter specification.

Handling Short Records with Missing Data

When the control file definition specifies more fields for a record than are present in the record, SQL*Loader must determine whether the remaining (specified) columns should be considered null or whether an error should be generated.

If the control file definition explicitly states that a field’s starting position is beyond the end of the logical record, then SQL*Loader always defines the field as null. If a field is defined with a relative position (such as DNAME and LOC in the example below), and the record ends before the field is found; then SQL*Loader could either treat the field as null or generate an error. SQL*Loader uses the presence or absence of the TRAILING NULLCOLS clause to determine the course of action.

TRAILING NULLCOLS

TRAILING NULLCOLS tells SQL*Loader to treat any relatively positioned columns that are not present in the record as null columns.

For example, if the following data

```
10 Accounting
```

is read with the following control file

```

INTO TABLE dept
  TRAILING NULLCOLS
( deptno CHAR TERMINATED BY " ",
  dname  CHAR TERMINATED BY WHITESPACE,
  loc    CHAR TERMINATED BY WHITESPACE
)
```

and the record ends after DNAME. The remaining LOC field is set to null. Without the TRAILING NULLCOLS clause, an error would be generated due to missing data.

Case 7 on page 4 – 22 provides an example of using TRAILING NULLCOLS.

Index Options

This section describes the SQL*Loader options that control how index entries are created.

SORTED INDEXES Option

The SORTED INDEXES option applies to direct path loads. It tells SQL*Loader that the incoming data has already been sorted on the specified indexes, allowing SQL*Loader to optimize performance. Syntax for this feature is given on page 5 – 7. Further details are on page 8 – 14.

SINGLEROW Option

The SINGLEROW option is intended for use during a direct path load with APPEND on systems with limited memory, or when loading a small number of rows into a large table. This option inserts each index entry directly into the index, one row at a time.

By default, SQL*Loader does not use SINGLEROW when APPENDING rows to a table. Instead, index entries are put into a separate, temporary storage area and merged with the original index at the end of the load. This method achieves better performance and produces an optimal index, but it requires extra storage space. During the merge, the original index, the new index, and the space for new entries all simultaneously occupy storage space.

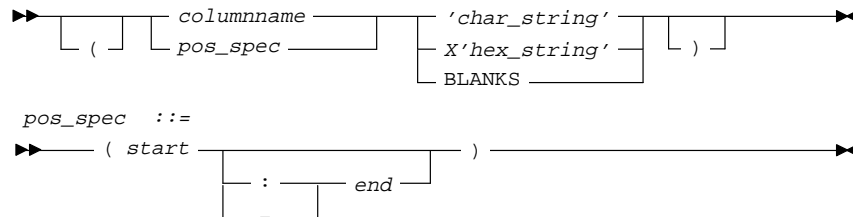
With the SINGLEROW option, storage space is not required for new index entries or for a new index. The resulting index may not be as optimal as a freshly sorted one, but it takes less space to produce. It also takes more time, since additional UNDO information is generated for each index insert. This option is suggested for use when:

- available storage is limited, or
- the number of rows to be loaded is small compared to the size of the table (a ratio of 1:20, or less, is recommended).

Specifying Field Conditions

A field condition is a statement about a field in a logical record that evaluates as true or false. It is used in the NULLIF and DEFAULTIF clauses, as well as in the WHEN clause.

A field condition is similar to the condition in the CONTINUEIF clause, with two important differences. First, positions in the field condition refer to the logical record, not to the physical record. Second, you may specify either a position in the logical record or the name of a field that is being loaded.



where:

<i>start</i>	Specifies the starting position of the comparison field in the logical record.
<i>end</i>	Specifies the ending position of the comparison field in the logical record. Either <i>start:end</i> or <i>start-end</i> is acceptable. If you omit end, the length of the field is determined by the length of the comparison string. If the lengths are different, the shorter field is padded: character strings are padded with blanks, hexadecimal strings are padded with zeroes.

<i>column_name</i>	The name of a column in the database table. If <i>column_name</i> is used instead of <i>start:end</i> , then the specification for that column defines the comparison field. <i>Column_name</i> must match exactly the name of the column in the table's database definition. Use quotation marks around the column name if it is a SQL or SQL*Loader keyword, contains special characters, or is case sensitive. For more information, see “Specifying Filenames and Database Objects” on page 5 – 13.
<i>operator</i>	A comparison operator for either <i>equal</i> or <i>not equal</i> : = != <> <>
<i>'char string'</i>	A string of characters enclosed within single or double quotation marks that is compared to the comparison field. If the comparison is true, then this row is inserted into the table.
<i>X'hex string'</i>	A byte string in hexadecimal format used in the same way as the character string above.
BLANKS	A keyword denoting an arbitrary number of blanks, described next.

Comparing Fields to BLANKS

The BLANKS keyword makes it possible to determine easily if a field of unknown length is blank.

For example, use the following clause to load a blank field as null:

```
column_name ... NULLIF column_name=BLANKS
```

The BLANKS keyword only recognizes blanks, not tabs. It can be used in place of a literal string in any field comparison. The condition is TRUE whenever the column is entirely blank.

The BLANKS keyword also works for fixed-length fields. Using it is the same as specifying an appropriately-sized literal string of blanks. For example, the following specifications are equivalent:

```
fixed_field CHAR(2) NULLIF (fixed_field)=BLANKS
fixed_field CHAR(2) NULLIF (fixed_field)=" "
```

Note: There can be more than one “blank” in a multi-byte character set. It is a good idea to use the BLANKS keyword with these character sets instead of specifying a string of blank characters. The character string will match only a specific sequence of blank characters, while the BLANKS keyword will match combinations of different blank characters. For more information on multi-byte character sets, see page 5 – 24.

Comparing Fields to Literals

When a data field is compared with a shorter literal string, the literal string is padded for the comparison. Character strings are padded with blanks. For example

```
NULLIF (1:4)="_"
```

compares the data in position 1:4 with 4 blanks. If position 1:4 contains 4 blanks, then the clause evaluates as true.

Hexadecimal strings are padded with hexadecimal zeroes. The clause

```
NULLIF (1:4)=X'FF'
```

compares position 1:4 to hex 'FF000000'.

Specifying Columns and Fields

You may load any number of a table's columns. Columns defined in the database, but not specified in the control file, are assigned null values (this is the proper way to insert null values).

A *column specification* is the name of the column, followed by a specification for the value to be put in that column. The list of columns is enclosed by parentheses and separated with commas as follows:

```
( columnspec, columnspec, ... )
```

Each column name must correspond to a column of the table named in the INTO TABLE clause. A column name must be enclosed in quotation marks if it is a SQL or SQL*Loader reserved word, contains special characters, or is case sensitive.

If the value is to be generated by SQL*Loader, the specification includes the keyword RECNUM, the SEQUENCE function, or the keyword CONSTANT. See “Generating Data” on page 5 – 45.

If the column's value is read from the datafile, the data field that contains the column's value is specified. In this case, the column specification includes a *column name* that identifies a column in the database table, and a *field specification* that describes a field in a data record. The field specification includes position, datatype, null restrictions, and defaults.

Specifying the Datatype of a Data Field

A field's datatype specification tells SQL*Loader how to interpret the data in the field. For example, a datatype of INTEGER specifies binary data, while INTEGER EXTERNAL specifies character data that represents a number. A CHAR field, however, can contain any character data.

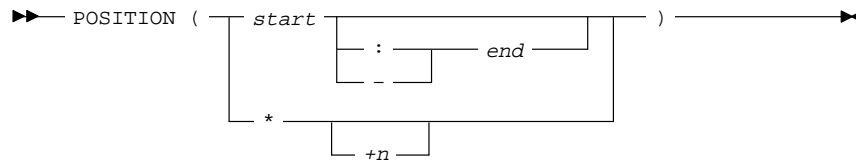
You may only specify one datatype for each field. If you omit the datatype, a type of CHAR is assumed.

“Specifying Datatypes” on page 5 – 49 describes how SQL*Loader datatypes are converted into the Oracle datatypes and gives detailed information on each of SQL*Loader’s datatypes.

Before specifying the datatype, however, the field’s position must be specified. That is the subject of the next section.

Specifying the Position of a Data Field

To load data from the datafile SQL*Loader must know a field’s location and its length. To specify a field’s position in the logical record, use the POSITION keyword in the column specification. The position may either be stated explicitly or relative to the preceding field. Arguments to POSITION must be enclosed in parentheses.



where:

- | | |
|--------------|---|
| <i>start</i> | The starting column of the data field in the logical record. The first character position in a logical record is 1. |
| <i>end</i> | Indicates the ending position of the data field in the logical record. Either <i>start:end</i> or <i>start-end</i> is acceptable. If <i>end</i> is omitted, the length of the field is derived from the datatype in the datafile. (See the sections on each datatype. Note that CHAR data specified without a <i>start</i> and <i>end</i> is assumed to be length 1.) If it is impossible to derive a length from the datatype, an error message results. |
| * | Indicates that the data field follows immediately after the previous field. If * is used for the first data field in the control file, the field is assumed to be at the beginning of the logical record. When * is used for position, the length of the field is derived from the datatype. |

+n An offset, specified as **+n**, may be used with ***** to offset this field from the previous one. *n* characters are skipped before reading the value for this field.

You may omit POSITION entirely. If you do, the position specification for the data field is the same as if POSITION(*) had been used.

For example

```
ENAME POSITION (1:20) CHAR
EMPNO POSITION (22-26) INTEGER EXTERNAL
ALLOW POSITION (*+2) INTEGER EXTERNAL TERMINATED BY "/"
```

Column ENAME is character data in positions 1 through 20, followed by column EMPNO, which is presumably numeric data in columns 22 through 26. Column ALLOW is offset from the end of EMPNO by +2. So it starts in column 28 and continues until a slash is encountered.

Using POSITION with Data that Contains TABs

When you are determining field positions, be alert for TABs in the datafile. The following situation is highly likely when using SQL*Loader's advanced SQL string capabilities to load data from a formatted report:

- You look at a printed copy of the report, carefully measuring all of the character positions, and create your control file.
- The load then fails with multiple "invalid number" and "missing field" errors.

These kinds of errors occur when the data contains TABs. When printed, each TAB expands to consume several columns on the paper. In the datafile, however, each TAB is still only one character. As a result, when SQL*Loader reads the datafile, the POSITION specifications are wrong.

To fix the problem, inspect the datafile for tabs and adjust the POSITION specifications, or else use delimited fields.

The use of delimiters to specify relative positioning of fields is discussed in detail beginning on page 5 – 58. Especially note how the delimiter WHITESPACE can be used.

Using POSITION with Multiple Table Loads

In a multiple table load, you specify multiple INTO TABLE clauses. When you specify POSITION(*) for the first column of the first table, the position is calculated relative to the beginning of the logical record. When you specify POSITION(*) for the first column of subsequent tables, the position is calculated relative to the last column of the last table loaded.

Thus, when a subsequent INTO TABLE clause begins, the position is *not* set to the beginning of the logical record automatically. This allows multiple INTO TABLE clauses to process different parts of the same physical record. For an example, see the second example in the section “Extracting Multiple Logical Records” beginning on page 5 – 43.

A logical record may contain data for one of two tables, but not both. In this case, you *would* reset POSITION. Instead of omitting the position specification or using POSITION(*+*n*) for the first field in the INTO TABLE clause, use POSITION(1) or POSITION(*n*).

Some examples follow:

```
SITEID  POSITION (*) SMALLINT
SITELOC POSITION (*) INTEGER
```

If these were the first two column specifications, SITEID would begin in column1, and SITELOC would begin in the column immediately following.

```
ENAME  POSITION (1:20)  CHAR
EMPNO  POSITION (22-26) INTEGER EXTERNAL
ALLOW  POSITION (*+2)   INTEGER EXTERNAL TERMINATED BY "/"
```

Column ENAME is character data in positions 1 through 20, followed by column EMPNO which is presumably numeric data in columns 22 through 26. Column ALLOW is offset from the end of EMPNO by +2, so it starts in column 28 and continues until a slash is encountered.

Using Multiple INTO TABLE Statements

Multiple INTO TABLE statements allow you to:

- load data into different tables
- extract multiple logical records from a single input record
- distinguish different input record formats

In the first case, it is common for the INTO TABLE statements to refer to the same table. This section illustrates the different ways to use multiple INTO TABLE statements and shows you how to use the POSITION keyword.

Note: A key point when using multiple INTO TABLE statements is that *field scanning continues from where it left off* when a new INTO TABLE statement is processed. The remainder of this section details important ways to make use of that behavior. It also describes alternative ways using fixed field locations or the POSITION keyword.

Extracting Multiple Logical Records

Some data storage and transfer media have fixed-length physical records. When the data records are short, more than one can be stored in a single, physical record to use the storage space efficiently.

In this example, SQL*Loader treats a single physical record in the input file as two logical records and uses two INTO TABLE clauses to load the data into the EMP table. For example, if the data looks like

```
1119 Smith      1120 Snyder
1121 Spellini   1130 Thompson
```

then the following control file extracts the logical records:

```
INTO TABLE emp
  (empno POSITION(1:4)  INTEGER EXTERNAL,
   ename POSITION(6:15) CHAR)

INTO TABLE emp
  (empno POSITION(17:20) INTEGER EXTERNAL,
   ename POSITION(21:30) CHAR)
```

Relative Positioning

The same record could be loaded with a different specification. The following control file uses relative positioning instead of fixed positioning. It specifies that each field is delimited by a single blank (" "), or with an undetermined number of blanks and tabs (WHITESPACE):

```
INTO TABLE emp
  (empno INTEGER EXTERNAL TERMINATED BY " ",
   ename CHAR              TERMINATED BY WHITESPACE)

INTO TABLE emp
  (empno INTEGER EXTERNAL TERMINATED BY " ",
   ename CHAR              TERMINATED BY WHITESPACE)
```

The important point in this example is that the second EMPNO field is found immediately after the first ENAME, although it is in a separate INTO TABLE clause. Field scanning does not start over from the beginning of the record for a new INTO TABLE clause. Instead, scanning continues where it left off.

To force record scanning to start in a specific location, you use the POSITION keyword. That mechanism is described next.

Distinguishing Different Input Record Formats

A single datafile might contain records in a variety of formats. Consider the following data, in which EMP and DEPT records are intermixed:

```
1 50 Manufacturing -- DEPT record
2 1119 Smith 50 -- EMP record
2 1120 Snyder 50
1 60 Shipping
2 1121 Stevens 60
```

A record ID field distinguishes between the two formats. Department records have a “1” in the first column, while employee records have a “2”. The following control file uses exact positioning to load this data:

```
INTO TABLE dept
  WHEN recid = 1
    (recid POSITION(1:1) INTEGER EXTERNAL,
     deptno POSITION(3:4) INTEGER EXTERNAL,
     ename POSITION(8:21) CHAR)

INTO TABLE emp
  WHEN recid <> 1
    (recid POSITION(1:1) INTEGER EXTERNAL,
     empno POSITION(3:6) INTEGER EXTERNAL,
     ename POSITION(8:17) CHAR,
     deptno POSITION(19:20) INTEGER EXTERNAL)
```

Relative Positioning

Again, the records in the previous example could also be loaded as delimited data. In this case, however, it is necessary to use the POSITION keyword. The following control file could be used:

```
INTO TABLE dept
  WHEN recid = 1
    (recid INTEGER EXTERNAL TERMINATED BY WHITESPACE,
     deptno INTEGER EXTERNAL TERMINATED BY WHITESPACE,
     dname CHAR TERMINATED BY WHITESPACE)

INTO TABLE emp
  WHEN recid <> 1
    (recid POSITION(1) INTEGER EXTERNAL TERMINATED BY ' ',
     empno INTEGER EXTERNAL TERMINATED BY ' ',
     ename CHAR TERMINATED BY WHITESPACE,
     deptno INTEGER EXTERNAL TERMINATED BY ' ')
```

The POSITION keyword in the second INTO TABLE clause is necessary to load this data correctly. This keyword causes field scanning to start over at column 1 when checking for data that matches the second format. Without it, SQL*Loader would look for the RECID field after DNAME.

Loading Data into Multiple Tables

By using the POSITION clause with multiple INTO TABLE clauses, data from a single record can be loaded into multiple normalized tables. Case 5 on page 4 – 14 illustrates this concept.

Summary

Multiple INTO TABLE clauses allow you to extract multiple logical records from a single input record and recognize different record formats in the same file.

For delimited data, proper use of the POSITION keyword is essential for achieving the expected results.

When the POSITION keyword is *not* used, multiple INTO TABLE clauses process different parts of the same (delimited data) input record, allowing multiple tables to be loaded from one record. When the POSITION keyword *is* used, multiple INTO TABLE clauses can process the same record in different ways, allowing multiple formats to be recognized in one input file.

Generating Data

The functions described in this section provide the means for SQL*Loader to generate the data stored in the database row, rather than reading it from a datafile. The following functions are described:

- CONSTANT
- RECNUM
- SYSDATE
- SEQUENCE

Loading Data Without Files

It is possible to use SQL*Loader to generate data by specifying only sequences, record numbers, system dates, and constants as field specifications.

SQL*Loader inserts as many rows as are specified by the LOAD keyword. The LOAD keyword is required in this situation. The SKIP keyword is not permitted.

SQL*Loader is optimized for this case. Whenever SQL*Loader detects that *only* generated specifications are used, it ignores any specified datafile — no read I/O is performed.

In addition, no memory is required for a bind array. If there are any WHEN clauses in the control file, SQL*Loader assumes that data evaluation is necessary, and input records are read.

Setting a Column to a Constant Value

CONSTANT

This is the simplest form of generated data. It does not vary during the load, and it does not vary between loads.

To set a column to a constant value, use the keyword CONSTANT followed by a value:

```
CONSTANT value
```

CONSTANT data is interpreted by SQL*Loader as character input. It is converted, as necessary, to the database column type.

You may enclose the value within quotation marks, and must do so if it contains white space or reserved words. Be sure to specify a legal value for the target column. If the value is bad, every row is rejected.

Numeric values larger than $2^{32} - 1$ (4,294,967,295) must be enclosed in quotes.

Note: Do not use the CONSTANT keyword to set a column to null. To set a column to null, do not specify that column at all. Oracle automatically sets that column to null when loading the row. The combination of CONSTANT and a value is a complete column specification.

Setting a Column to the Datafile Record Number

RECNUM

Use the RECNUM keyword after a column name to set that column to the number of the logical record from which that row was loaded. Records are counted sequentially from the beginning of the first datafile, starting with record 1. RECNUM is incremented as each logical record is assembled. Thus it increments for records that are discarded, skipped, rejected, or loaded. If you use the option SKIP=10, then the first record loaded has a RECNUM of 11.

The combination of column name and the RECNUM keyword is a complete column specification.

```
column_name RECNUM
```

Setting a Column to the Current Date

SYSDATE

A column specified with SYSDATE gets the current system date, as defined by the SQL language SYSDATE function. See the section “DATE Datatype” in *Oracle7 Server SQL Reference*.

The combination of column name and the SYSDATE keyword is a complete column specification.

```
column_name SYSDATE
```

The database column must be of type CHAR or DATE. If the column is of type CHAR, then the date is loaded in the form 'dd-mon-yy.' After the load, it can be accessed only in that form. If the system date is loaded into a DATE column, then it can be accessed in a variety of forms that include the time and the date.

A new system date/time is used for each array of records inserted in a conventional path load and for each block of records loaded during a direct path load.

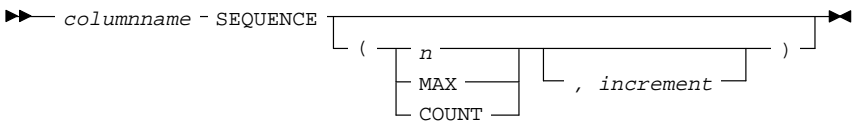
Setting a Column to a Unique Sequence Number

The SEQUENCE keyword ensures a unique value for a particular column. SEQUENCE increments for each record that is loaded or rejected. It does not increment for records that are discarded or skipped.

SEQUENCE takes two optional arguments. The first argument is the starting value. The second is the increment. If the start point is a positive integer *n*, the first row inserted has a value of *n* for that column. The values of successive rows are increased by the increment. However, both the starting value and the increment default to 1.

SEQUENCE

The combination of column name and the SEQUENCE function is a complete column specification.



where:

SEQUENCE	Use the SEQUENCE keyword to specify the value for a column.
<i>n</i>	The sequence starts with the integer value <i>n</i> . The value must be positive or zero. Default value is 1.
COUNT	The sequence starts with the number of rows already in the table, plus the increment.
MAX	The sequence starts with the current maximum value for the column, plus the increment.
<i>increment</i>	The sequence is incremented by this amount for each successive row. The default increment is 1. The increment must be positive.

If a row is rejected (that is, it has a format error or causes an Oracle error), the generated sequence numbers are not reshuffled to mask this. If four rows are assigned sequence numbers 10, 12, 14, and 16 in a particular column, and the row with 12 is rejected; the three rows inserted are numbered 10, 14, and 16, not 10, 12, 14. This allows the sequence of inserts to be preserved despite data errors. When you correct the rejected data and reinsert it, you can manually set the columns to agree with the sequence.

Case 3 on page 4 – 8 provides an example of using SEQUENCE.

Generating Sequence Numbers for Multiple Tables

Because a unique sequence number is generated for each logical input record, rather than for each table insert, the same sequence number can be used when inserting data into multiple tables. This is frequently useful behavior. Case 3 on page 4 – 8 illustrates this situation.

Sometimes, you might want to generate different sequence numbers for each INTO TABLE clause. For example, your data format might define three logical records in every input record. In that case, you can use three INTO TABLE clauses, each of which inserts a different part of the record into the same table.

To generate sequence numbers for these records, you must generate unique numbers for each of the three inserts. There is a simple technique to do so. Use the number of table-inserts per record as the sequence increment and start the sequence numbers for each insert with successive numbers.

Example

Suppose you want to load the following department names into the DEPT table. Each input record contains three department names, and you want to generate the department numbers automatically.

Accounting	Personnel	Manufacturing
Shipping	Purchasing	Maintenance
...		

You could use the following control file to generate unique department numbers:

```
INTO TABLE dept
(deptno  sequence(1, 3),
  dname   position(1:14) char)

INTO TABLE dept
(deptno  sequence(2, 3),
  dname   position(16:29) char)
```

```

INTO TABLE dept
(deptno  sequence(3, 3),
dname    position(31:44) char)

```

The first INTO TABLE clause generates department number 1, the second number 2, and the third number 3. They all use 3 as the sequence increment (the number of department names in each record). This control file loads Accounting as department number 1, Personnel as 2, and Manufacturing as 3. The sequence numbers are then incremented for the next record, so Shipping loads as 4, Purchasing as 5, and so on.

Specifying Datatypes

This section describes SQL*Loader's datatypes and explains how they are converted to Oracle datatypes.

Datatype Conversions

The datatype specifications in the control file tell SQL*Loader how to interpret the information in the datafile. The server defines the datatypes for the columns in the database. The link between these two is the *column name* specified in the control file.

SQL*Loader extracts data from a field in the input file, guided by the datatype specification in the control file. SQL*Loader then sends the field to the server to be stored in the appropriate column (as part of an array of row inserts). The server does any necessary data conversion to store the data in the proper internal format. The "Data Conversion and Datatype Specification" section beginning on page 3 – 12 contains diagrams that illustrate these points.

The datatype of the data in the file does not necessarily have to be the same as the datatype of the column in the Oracle table. Oracle automatically performs conversions, but you need to ensure that the conversion makes sense and does not generate errors. For instance, when a datafile field with datatype CHAR is loaded into a database column with datatype NUMBER, you must make sure that the contents of the character field represent a valid number.


Note: SQL*Loader does *not* contain datatype specifications for Oracle internal datatypes such as NUMBER or VARCHAR2. SQL*Loader's datatypes describe data that can be produced with text editors (*character* datatypes) and with standard programming languages (*native* datatypes). However, although SQL*Loader does not recognize datatypes like NUMBER and VARCHAR2, any data that Oracle is capable of converting may be loaded into these or other database columns.

Native Datatypes

Some datatypes consist entirely of binary data or contain binary data in their implementation. See page 3 – 6 for a discussion of binary vs. character data. These non-character datatypes are the *native* datatypes:

INTEGER	ZONED
SMALLINT	VARCHAR
FLOAT	GRAPHIC
DOUBLE	GRAPHIC EXTERNAL
BYTEINT	VARGRAPHIC
(packed) DECIMAL	RAW
MLSLABEL (Trusted Oracle)	

Since these datatypes contain binary data, most of them do not readily transport across operating systems. (See “Loading Data Across Different Operating Systems” on page 5 – 62.) RAW data and GRAPHIC data is the exceptions. SQL*Loader does not attempt to interpret these datatypes, but simply stores them “as is”.


OSDoc

Additional Information: Native datatypes cannot be specified with delimiters. The size of the native datatypes INTEGER, SMALLINT, FLOAT, and DOUBLE are determined by the host operating system. Their size is fixed — it cannot be overridden in the control file. (Refer to your Oracle operating system-specific documentation for more information.) The sizes of the other native datatypes may be specified in the control file.

INTEGER


The data is a fullword binary integer. If you specify *start:end* in the POSITION clause, *end* is ignored. The length of the field is the length of a fullword integer on your system. (Datatype LONG INT in C.) This length cannot be overridden in the control file.

INTEGER

SMALLINT

The data is a half-word binary integer. If you specify *start:end* in the POSITION clause, *end* is ignored. The length of the field is a half-word integer is on your system.

SMALLINT


OSDoc

Additional Information: This is the SHORT INT datatype in the C programming language. One way to determine its length is to make a small control file with no data and look at the resulting log file. This length cannot be overridden in the control file. See your Oracle operating system-specific documentation for details.

FLOAT The data is a single-precision, floating-point, binary number. If you specify *end* in the POSITION clause, it is ignored. The length of the field is the length of a single-precision, floating-point binary number on your system. (Datatype FLOAT in C.) This length cannot be overridden in the control file.

FLOAT

DOUBLE The data is a double-precision, floating-point binary number. If you specify *end* in the POSITION clause, it is ignored. The length of the field is the length of a double-precision, floating-point binary number on your system. (Datatype DOUBLE or LONG FLOAT in C.) This length cannot be overridden in the control file.

DOUBLE

BYTEINT The decimal value of the binary representation of the byte is loaded. For example, the input character x"1C" is loaded as 28. The length of a BYTEINT field is always 1 byte. If POSITION(*start:end*) is specified, *end* is ignored.

The syntax for this datatype is

BYTEINT

An example is

```
(column1 position(1) BYTEINT,
 column2 BYTEINT,
 ...
)
```

ZONED ZONED data is in zoned decimal format: a string of decimal digits, one per byte, with the sign included in the last byte. (In COBOL, this is a SIGN TRAILING field.) The length of this field is equal to the precision (number of digits) that you specify.

The syntax for this datatype is:

► ZONED (— *precision* — , *scale*) ◄

where *precision* is the number of digits in the number, and scale (if given) is the number of digits to the right of the (implied) decimal point. For example:

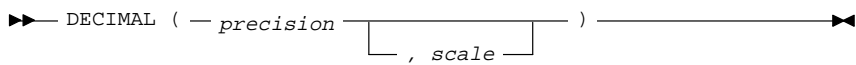
```
sal    POSITION(32)    ZONED(8),
```

specifies an 8-digit integer starting at position 32.

DECIMAL

DECIMAL data is in packed decimal format: two digits per byte, except for the last byte which contains a digit and sign. DECIMAL fields allow the specification of an implied decimal point, so fractional values can be represented.

The syntax for the this datatype is:



where:

- precision* Is the number of digits in the value. The character length of the field, as computed from digits, is $(\text{digits} + 2) / 2$, rounded down.
- scale* Is the scaling factor, or number of digits to the right of the decimal point. Default is zero (indicating an integer). May be greater than the number of digits but may not be negative.

For example,

```
sal DECIMAL (7,2)
```

would load a number equivalent to +12345.67. In the data record, this field would take up 4 bytes, as shown in Figure 5 – 1. (The byte length of a DECIMAL field is equivalent to $(N+1)/2$, rounded up, where N is the number of digits in the value, and one is added for the sign.)

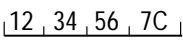
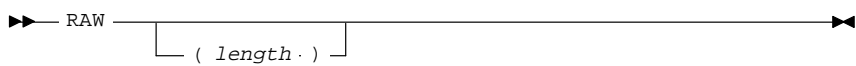


Figure 5 – 1 Packed Decimal Representation of +12345.67

RAW

The data is raw, binary data loaded “as is”. It does not undergo character set conversion. If loaded into a RAW database column, it is not converted by Oracle. If it is loaded into a CHAR column, Oracle converts it to hexadecimal. It cannot be loaded into a DATE or number column.

The syntax for this datatype is



The length of this field is the number of bytes specified in the control file. This length is limited only by the length of the target column in the database and by memory resources.

GRAPHIC

The data is a string of double-byte characters (DBCS). Oracle does not support DBCS, however SQL*Loader reads DBCS as single bytes. Like RAW data, GRAPHIC fields are stored without modification in whichever column you specify.

The syntax for this datatype is

►► GRAPHIC _____ ◄◄
 └ (*graphic_char_length*) ┘

For both GRAPHIC and GRAPHIC EXTERNAL, if you specify POSITION(*start:end*) you give the exact location of the field in the logical record. If you specify the length after the GRAPHIC (EXTERNAL) keyword, however, then you give the number of double-byte graphic characters. That value is multiplied by 2 to find the length of the field in bytes. If the number of graphic characters is specified, then any length derived from POSITION is ignored.

GRAPHIC EXTERNAL

If the DBCS field is surrounded by shift-in and shift-out characters, use GRAPHIC EXTERNAL. This is identical to GRAPHIC, except that the first and last characters (the shift-in and shift-out) are not loaded.

The syntax for this datatype is:

►► GRAPHIC EXTERNAL _____ ◄◄
 └ (*graphic_char_length*) ┘

where:

GRAPHIC Data is double-byte characters.

EXTERNAL First and last characters are ignored.

graphic_char_length Length in DBCS (see GRAPHIC, above)

For example, let [] represent shift-in and shift-out characters, and let # represent any double-byte character.

To describe #####, use "POSITION(1:4) GRAPHIC" or "POSITION(1) GRAPHIC(2)".

To describe [#####], use "POSITION(1:6) GRAPHIC EXTERNAL" or "POSITION(1) GRAPHIC EXTERNAL(2)".

VARGRAPHIC

The data is a varying-length, double-byte character string. It consists of a *length subfield* followed by a string of double-byte characters (DBCS).

Additional Information: The size of the length subfield is the size of the SQL*Loader SMALLINT datatype on your system (C type SHORT INT). See SMALLINT on page 5 – 50 for more information.

The length of the current field is given in the first two bytes. This length is a count of graphic (double-byte) characters. So it is multiplied by two to determine the number of bytes to read.

The syntax for this datatype is

```

▶▶ VARGRAPHIC _____ ▶▶
      ( maximum_length )

```

A maximum length specified after the VARGRAPHIC keyword does *not* include the size of the length subfield. The maximum length specifies the number of graphic (double byte) characters. So it is also multiplied by two to determine the maximum length of the field in bytes.

The default maximum field length is 4K graphic characters, or 8 Kb (2 * 4K). It is a good idea to specify a maximum length for such fields whenever possible, to minimize memory requirements. See “Determining the Size of the Bind Array” on page 5 – 63 for more details.

The POSITION clause, if used, gives the location of the length subfield, not of the first graphic character. If you specify POSITION(*start:end*), the end location determines a maximum length for the field. Both *start* and *end* identify single-character (byte) positions in the file. *Start* is subtracted from (*end* + 1) to give the length of the field in bytes. If a maximum length is specified, it overrides any maximum length calculated from POSITION.

If a VARGRAPHIC field is truncated by the end of the logical record before its full length is read, a warning is issued. Because a VARCHAR field’s length is embedded in every occurrence of the input data for that field, it is assumed to be accurate.

VARGRAPHIC data cannot be delimited.

VARCHAR

A VARCHAR field is a varying-length character string. It is considered a native datatype, rather than a character datatype because it includes binary data (a length). It consists of a *length subfield* followed by a character string of the given length.

Additional Information: The size of the length subfield is the size of the SQL*Loader SMALLINT datatype on your system (C type SHORT INT). See SMALLINT on page 5 – 50 for more information.

The syntax for this datatype is:

```

▶▶ VARCHAR _____ ▶▶
      ( maximum_length )

```

A maximum length specified in the control file does *not* include the size of the length subfield. If you specify the optional maximum length after the VARCHAR keyword, then a buffer of that size is allocated for these fields.

The default buffer size is 4 Kb. Specifying the smallest maximum length that is needed to load your data can minimize SQL*Loader's memory requirements, especially if you have many VARCHAR fields. See "Determining the Size of the Bind Array" on page 5 – 63 for more details.

The POSITION clause, if used, gives the location of the length subfield, not of the first text character. If you specify POSITION(*start:end*), the end location determines a maximum length for the field. *Start* is subtracted from (*end* + 1) to give the length of the field in bytes. If a maximum length is specified, it overrides any length calculated from POSITION.

If a VARCHAR field is truncated by the end of the logical record before its full length is read, a warning is issued. Because a VARCHAR field's length is embedded in every occurrence of the input data for that field, it is assumed to be accurate.

VARCHAR data cannot be delimited.

Conflicting Native Datatype Field Lengths

There are several ways to specify a length for a field. If multiple lengths are specified and they conflict, then one of the lengths takes precedence. A warning is issued when a conflict exists. The following rules determine which field length is used:

1. The size of INTEGER, SMALLINT, FLOAT, and DOUBLE data is fixed. It is not possible to specify a length for these datatypes in the control file. If starting and ending positions are specified, the end position is ignored — only the start position is used.
2. If the length specified (or precision) of a DECIMAL, ZONED, GRAPHIC, GRAPHIC EXTERNAL, or RAW field conflicts with the size calculated from a POSITION(*start:end*) specification, then the specified length (or precision) is used.
3. If the maximum size specified for a VARCHAR or VARGRAPHIC field conflicts with the size calculated from a POSITION(*start:end*) specification, then the specified maximum is used.

For example, if the native datatype INTEGER is 4 bytes long and the following field specification is given:

```
column1 POSITION(1:6) INTEGER
```

then a warning is issued, and the proper length (4) is used. In this case, the log file shows the actual length used under the heading “Len” in the column table:

Column Name	Position	Len	Term	Encl	Datatype
-----	-----	-----	-----	-----	-----
COLUMN1	1:6	4			INTEGER

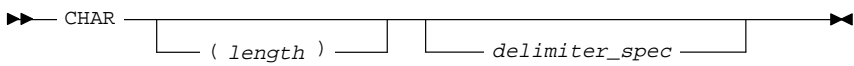
Character Datatypes

The character datatypes are CHAR, DATE, and the numeric EXTERNAL datatypes. These fields can be delimited and can have lengths (or maximum lengths) specified in the control file.


Additional Information: In addition, the MLSLABEL character datatype exists in Trusted Oracle. See the *Trusted Oracle7 Server Administrator’s Guide* for more information on this datatype.

CHAR

The data field contains character data. The length is optional and is taken from the POSITION specification if it is not present here. If present, this length overrides the length in the POSITION specification. If no length is given, CHAR data is assumed to have a length of 1. The syntax is:

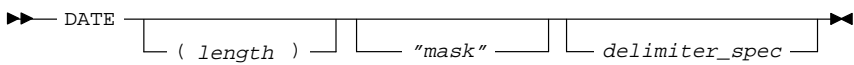


A field of datatype CHAR may also be variable-length delimited or enclosed. See “Specifying Delimiters” on page 5 – 58.

 **Attention:** If the column in the database table is defined as LONG, you must explicitly specify a maximum length (maximum for a LONG is 2 gigabytes) either with a length specifier on the CHAR keyword or with the POSITION keyword. This guarantees that a large enough buffer is allocated for the value and is necessary even if the data is delimited or enclosed.

DATE

The data field contains character data that should be converted to an Oracle date using the specified date mask. The syntax is:



For example:

```

LOAD DATA
INTO TABLE DATES (COL_A POSITION (1:15) DATE "DD-Mon-YYYY")
BEGINDATA
1-Jan-1991
1-Apr-1991 28-Feb-1991

```



Attention: Whitespace is ignored and dates are parsed from left to right unless delimiters are present.

The length specification is optional, unless a varying-length date mask is specified. In the example above, the date mask specifies a fixed-length date format of 11 characters. SQL*Loader counts 11 characters in the mask, and therefore expects a maximum of 11 characters in the field, so the specification works properly. But, with a specification such as

```
DATE "Month dd, YYYY"
```

the date mask is 14 characters, while the maximum length of a field such as

```
September 30, 1991
```

is 18 characters. In this case, a length must be specified. Similarly, a length is required for any Julian dates (date mask "J")—a field length is required any time the length of the date string could exceed the length of the mask (that is, the count of characters in the mask).

If an explicit length is not specified, it can be derived from the POSITION clause. It is a good idea to specify the length whenever you use a mask, unless you are absolutely sure that the length of the data is less than, or equal to, the length of the mask.

An explicit length specification, if present, overrides the length in the POSITION clause. Either of these overrides the length derived from the mask. The mask may be any valid Oracle date mask. If you omit the mask, the default Oracle date mask of "dd-mon-yy" is used. The length must be enclosed in parentheses and the mask in quotation marks. Case 3 on page 4 – 8 has an example of the DATE datatype.

A field of datatype DATE may also be specified with delimiters. For more information, see "Specifying Delimiters" on page 5 – 58.

A date field that consists entirely of whitespace produces an error unless NULLIF BLANKS is specified. For more information, see "Loading All-Blank Fields" on page 5 – 69.

MLSLABEL

This is a Trusted Oracle7 datatype that stores the binary format of an operating system label. For more information see the *Trusted Oracle7 Server Administrator's Guide*.

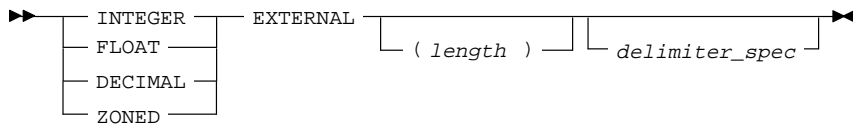
Numeric External Datatypes

The *numeric external* datatypes are the numeric datatypes (INTEGER, FLOAT, DECIMAL, and ZONED) specified with the EXTERNAL keyword with optional length and delimiter specifications. These datatypes are the human-readable, character form of numeric data.

The data is a number in character form (not binary representation). As such, these datatypes are identical to CHAR and are treated identically, with one exception: the use of DEFAULTIF. If you want the default to be null, use CHAR. If you want it to be zero, use EXTERNAL.

Numeric EXTERNAL may be specified with lengths and delimiters, just like CHAR data. Length is optional, but if specified, overrides POSITION.

The syntax for this datatype is:



FLOAT EXTERNAL Data Values

FLOAT EXTERNAL data can be given in either scientific or regular notation. Both "5.33" and "533E-2" are valid representations of the same value.

Specifying Delimiters

The boundaries of CHAR, DATE, MLSLABEL, or numeric EXTERNAL fields may also be marked by specific delimiter characters contained in the input data record. You indicate how the field is delimited by using a delimiter specification after specifying the datatype.

Delimited data can be TERMINATED or ENCLOSED.

TERMINATED Fields

TERMINATED fields are read from the starting position of the field up to, but not including, the first occurrence of the delimiter character. If the terminator delimiter is found in the first column position, the field is null.

TERMINATED BY WHITESPACE

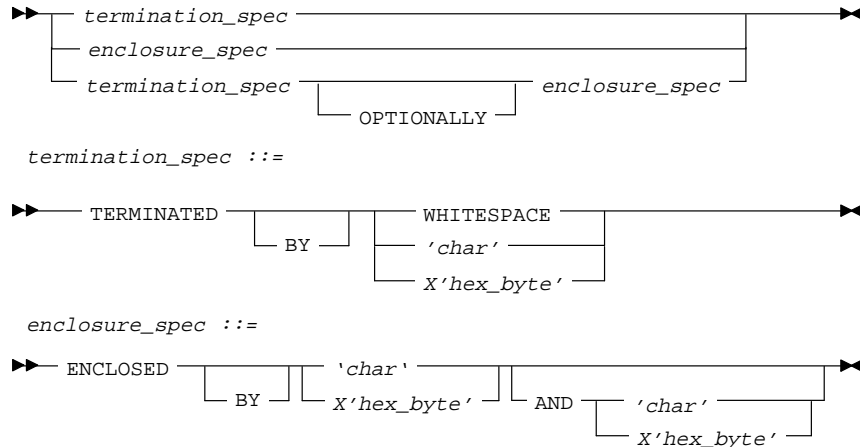
If TERMINATED BY WHITESPACE is specified, data is read until the first occurrence of a whitespace character (space, tab, newline). Then the current position is advanced until no more adjacent whitespace characters are found. This allows field values to be delimited by varying amounts of whitespace.

Enclosed Fields

Enclosed fields are read by skipping whitespace until a non-whitespace character is encountered. If that character is the delimiter, then data is read up to the second delimiter. Any other character causes an error.

If two delimiter characters are encountered next to each other, a single occurrence of the delimiter character is used in the data value. For example, 'DON""T' is stored as DON'T. However, if the field consists of just two delimiter characters, its value is null. You may specify a TERMINATED BY clause, an ENCLOSED BY clause, or both. If both are used, the TERMINATED BY clause must come first.

The syntax for delimiter specifications is:



where:

TERMINATED	Data is read until first occurrence of a delimiter.
BY	This is an optional keyword for readability.
WHITESPACE	Delimiter is any whitespace character, including linefeed, formfeed, or carriage return. (Only used with TERMINATED, not with ENCLOSED.)
OPTIONALLY	Data may be enclosed by the indicated character. If SQL*Loader finds a first occurrence of the character, it reads the data value until it finds the second occurrence. If the data is not enclosed, the data is read as a terminated field. If optional enclosure is specified, there must be a TERMINATED BY clause—either locally in the field definition, or globally, in the FIELDS clause.
ENCLOSED	The data value is found between two delimiters.
char	Delimiter is the single character <i>char</i> .
X'hex-byte'	Delimiter is the single character that has the value specified by <i>hex-byte</i> in the character encoding

scheme, such as X'1F' (equivalent to 31 decimal).
"X" must be uppercase.

Note: Due to National Language Support (NLS) requirements, hex 00 cannot be used as a separator. All other hex values are supported.

AND

This keyword specifies a *trailing* enclosure delimiter, which may be different from the *initial* enclosure delimiter. If the AND clause is not present, then the initial and trailing enclosure delimiters are the same.

Here are some examples, with samples of the data they describe:

```
TERMINATED BY ','          a data string,
ENCLOSED BY '""'          "a data string"
TERMINATED BY ',' ENCLOSED BY '""' "a data string",
ENCLOSED BY "(" AND ")"'   (a data string)
```

Delimiter Marks in the Data

Sometimes the same punctuation mark that is a delimiter also needs to be included in the data. To make that possible, two adjacent delimiter characters are interpreted as a single occurrence of the character, and this character is included in the data. For example, this data:

```
(The delimiters are left paren's, (, and right paren's, ).)
```

with this field specification:

```
ENCLOSED BY "(" AND ")"
```

puts the following string into the database:

```
The delimiters are left paren's, (, and right paren's, ).
```

For this reason, problems can arise when adjacent fields use the same delimiters. For example, the following specification:

```
field1 TERMINATED BY "/"
field2 ENCLOSED BY "/"
```

the following data will be interpreted properly:

```
This is the first string/          /This is the second string/
```

But if field1 and field2 were adjacent, then the results would be incorrect, because

```
This is the first string//This is the second string/
```

would be interpreted as a single character string with a "/" in the middle of it, and that string would belong to field1.

Maximum Length of Delimited Data	The default maximum length of delimited data is 255 bytes. So delimited fields can require significant amounts of storage for the bind array. It is a good idea to specify the smallest possible maximum value. See “Determining the Size of the Bind Array” on page 5 – 63.
Loading Trailing Blanks with Delimiters	Trailing blanks can only be loaded with delimited datatypes. If a data field is nine characters long and contains the value DANIELbbb, where bbb is three blanks, it is loaded into Oracle as “DANIEL” if declared as CHAR(9). If you want the trailing blanks, you could declare it as CHAR(9) TERMINATED BY ‘.’, and add a colon to the datafile so that the field is DANIELbbb:. This field is loaded as “DANIEL ”, with the trailing blanks. For more discussion on whitespace in fields, see “Trimming of Blanks and Tabs” on page 5 – 69.
Conflicting Character Datatype Field Lengths	A control file can specify multiple lengths for the character–data fields CHAR, DATE, MLSLABEL, and numeric EXTERNAL. If conflicting lengths are specified, one of the lengths takes precedence. A warning is also issued when a conflict exists. This section explains which length is used.
Predetermined Size Fields	<p>If you specify a starting position and ending position for one of these fields, then the length of the field is determined by these specifications. If you specify a length as part of the datatype and do not give an ending position, the field has the given length. If starting position, ending position, and length are all specified, and the lengths differ; then the length given as part of the datatype specification is used for the length of the field.</p> <p>For example, if</p> <pre>position(1:10) char(15)</pre> <p>is specified, then the length of the field is 15.</p>
Delimited Fields	<p>If a delimited field is specified with a length, or if a length can be calculated from the starting and ending position, then that length is the <i>maximum</i> length of the field. The actual length can vary up to that maximum, based on the presence of the delimiter. If a starting and ending position are both specified for the field and if a field length is specified in addition, then the specified length value overrides the length calculated from the starting and ending position.</p> <p>If the expected delimiter is absent and no maximum length has been specified, then the end of record terminates the field. If TRAILING NULLCOLS is specified, remaining fields are null. If either the delimiter or the end of record produce a field that is longer than the specified maximum, SQL*Loader generates an error.</p>

Date Field Masks

The length of a date field depends on the mask, if a mask is specified. The mask provides a format pattern, telling SQL*Loader how to interpret the data in the record. For example, if the mask is specified as:

`"Month dd, yyyy"`

then "May 3, 1991" would occupy 11 character positions in the record, while "January 31, 1992" would occupy 16.

If starting and ending positions *are* specified, however, then the length calculated from the position specification overrides a length derived from the mask. A specified length such as "DATE (12)" overrides either of those. If the date field is also specified with terminating or enclosing delimiters, then the length specified in the control file is interpreted as a maximum length for the field.

MLSLABEL Field Masks

These are Trusted Oracle7 masks. For more information, see the *Trusted Oracle7 Server Administrator's Guide*.

Loading Data Across Different Operating Systems

When a datafile is created on one operating system that is to be loaded under a different operating system, the data must be written in a form that the target system can read. For example, if the source system has a native, floating-point representation that uses 16 bytes, and the target system's floating-point numbers are 12 bytes; then there is no way for the target system to directly read data generated on the source system. One solution is to load data across a SQL*Net link, taking advantage of the automatic conversion of datatypes. This is the recommended approach, whenever feasible.

In general, the problems of inter-operating system loads occur with the *native* datatypes. Sometimes, it is possible to get around them by padding a field with zeros to lengthen it, or reading only part of the field to shorten it. (For example, when an 8-byte integer is to be read on a system that uses 6-byte integers, or vice versa.) Frequently, however, problems of incompatible byte-ordering, or incompatible implementations of the datatypes, make even this approach unworkable.

Without a SQL*Net link, it is a good idea to use only the CHAR, DATE, and NUMERIC EXTERNAL datatypes. Datafiles written in this manner are longer than those written with native datatypes. They take more time to load, but they transport most readily across operating systems. However, where incompatible byte-ordering is an issue, special filters may still be required to reorder the data.

Determining the Size of the Bind Array

The determination of bind array size pertains to SQL*Loader's conventional path option. It does not apply to the direct path load method. Because a direct path load formats database blocks directly, rather than using Oracle's SQL interface, it does not use a bind array.

SQL*Loader uses the SQL array-interface option to transfer data to the RDBMS. Multiple rows are read at one time and stored in the *bind array*. When SQL*Loader sends Oracle an INSERT command, the entire array is inserted at one time. After the rows in the bind array are inserted, a COMMIT is issued.

Minimum Requirements

The bind array has to be large enough to contain a single row. If the maximum row length exceeds the size of the bind array, as specified by the BINDSIZE parameter, SQL*Loader generates an error. Otherwise, the bind array contains as many rows as can fit within it, up to the limit set by the value of the ROWS parameter. The BINDSIZE parameter is described on page 6 – 3, the ROWS parameter on page 6 – 6.

Although the entire bind array need not be in contiguous memory, the buffer for each field in the bind array must occupy contiguous memory. If the operating system cannot supply enough contiguous memory to store a field, SQL*Loader generates an error.

Performance Implications

To minimize the number of calls to Oracle and maximize performance, large bind arrays are preferable. In general, you gain large improvements in performance with each increase in the bind array size up to 100 rows. Increasing the bind array size above 100 rows generally delivers more modest improvements in performance. So the size (in bytes) of 100 rows is typically a good value to use. The remainder of this section details the method for determining that size.

In general, any reasonably large size will permit SQL*Loader to operate effectively. It is not usually necessary to perform the detailed calculations described in this section. This section should be read when maximum performance is desired, or when an explanation of memory usage is needed.

Specifying Number of Rows vs. Size of Bind Array

When you specify a bind array size using the command-line parameter BINDSIZE (see page 6 – 3) or the OPTIONS clause in the control file (see page 5 – 12), you impose an upper limit on the bind array. The bind array never exceeds that maximum.

As part of its initialization, SQL*Loader determines the space required to load a single row. If that size is too large to fit within the specified maximum, the load terminates with an error.

SQL*Loader then multiplies that size by the number of rows for the load, whether that value was specified with the command-line parameter ROWS (see page 6 – 3) or the OPTIONS clause in the control file (see page 5 – 12). If that size fits within the bind array maximum, the load continues—SQL*Loader does not try to expand the number of rows to reach the maximum bind array size. That is, if the number of rows and the maximum bind array size are both specified, SQL*Loader always uses the smaller value for the bind array.

If the maximum bind array size is too small to accommodate the initial number of rows, SQL*Loader uses a smaller number of rows that fits within the maximum.

Calculations

The bind array's size is equivalent to the number of rows it contains times the maximum length of each row. The maximum length of a row is equal to the sum of the maximum field lengths, plus overhead.

`bind array size = (number of rows) * (maximum row length)`

where:

`(maximum row length) = SUM(fixed field lengths) +
SUM(maximum varying field lengths) +
SUM(overhead for varying length fields)`

Many fields do not vary in size. These *fixed-length fields* are the same for each loaded row. For those fields, the maximum length of the field is the field size, in bytes, as described in “Specifying Datatypes” on page 5 – 49. There is no overhead for these fields.

The fields that *can* vary in size from row to row are

VARCHAR	VARGRAPHIC
CHAR	DATE
numeric	EXTERNAL

The maximum length of these datatypes is described in “Specifying Datatypes” on page 5 – 49. The maximum lengths describe the number of bytes, or character positions, that the fields can occupy in the input data record. That length also describes the amount of storage that each field occupies in the bind array, but the bind array includes additional overhead for fields that can vary in size.

When the character datatypes (CHAR, DATE, and numeric EXTERNAL) are specified with delimiters, any lengths specified for these fields are maximum lengths. When specified without delimiters, the size in the record is fixed, but the size of the inserted field may still vary, due to whitespace trimming. So internally, these datatypes are always treated as varying-length fields—even when they are fixed-length fields.

A length indicator is included for each of these fields in the bind array. The space reserved for the field in the bind array is large enough to hold the longest possible value of the field. The length indicator gives the actual length of the field for each row.

In summary:

```
bind array size =
    (number of rows) * ( SUM(fixed field lengths)
                        + SUM(maximum varying field lengths)
                        + ( (number of varying length fields)
                          * (size of length-indicator) )
                      )
```

Determining the Size of the Length Indicator

On most systems, the size of the length indicator is two bytes. On a few systems, it is three bytes. To determine its size, use the following control file:

```
OPTIONS (ROWS=1)
LOAD DATA
INFILE *
APPEND
INTO TABLE DEPT
(deptno POSITION(1:1) CHAR)
BEGINDATA
a
```

This control file “loads” a one-character field using a one-row bind array. No data is actually loaded, due to the numeric conversion error that occurs when “a” is loaded as a number. The bind array size shown in the log file, minus one (the length of the character field) is the value of the length indicator.

Note: A similar technique can determine bind array size without doing any calculations. Run your control file without any data and with ROWS=1 to determine the memory requirements for a single row of data. Multiply by the number of rows you want in the bind array to get the bind array size.

Calculating the Size of
Field Buffers

The following tables summarize the memory requirements for each datatype. “L” is the length specified in the control file. “P” is precision. “S” is the size of the length indicator. For more information on these values, see “Specifying Datatypes” starting on page 5 – 49.

Datatype	Size
INTEGER	OS-dependent
SMALLINT	
FLOAT	
DOUBLE	

Table 5 – 1 Invariant fields

Datatype	Default Size	Specified Size
(packed) DECIMAL	None	(P+1)/2, rounded up
ZONED	None	P
RAW	None	L
CHAR (no delimiters)	1	L+S
DATE (no delimiters)	None	
numeric EXTERNAL (no delimiters)	None	
MLSLABEL	None	

Table 5 – 2 Non-graphic fields

Datatype	Default Size	Length Specified with POSITION	Length Specified with DATATYPE
GRAPHIC	None	L	2*L
GRAPHIC EXTERNAL	None	L – 2	2*(L–2)
VARGRAPHIC	4Kb*2	L+S	(2*L)+S

Table 5 – 3 Graphic Fields

Datatype	Default Size	Maximum Length Specified (L)
VARCHAR	4Kb	L+S
CHAR (delimited) DATE (delimited) numeric EXTERNAL (delimited) MLSLABEL (delimited)	255	L+S

Table 5 – 4 Variable-length fields

Minimizing Memory Requirements for the Bind Array

Pay particular attention to the default sizes allocated for VARCHAR, VARCHAR2, and the delimited forms of CHAR, DATE, and numeric EXTERNAL fields. They can consume enormous amounts of memory—especially when multiplied by the number of rows in the bind array. It is best to specify the smallest possible maximum length for these fields. For example:

```
CHAR(10) TERMINATED BY ','
```

uses $(10 + 2) * 64 = 768$ bytes in the bind array, assuming that the length indicator is two bytes long. However:

```
CHAR TERMINATED BY ','
```

uses $(255 + 2) * 64 = 16,448$ bytes, because the default maximum size for a delimited field is 255. This can make a considerable difference in the number of rows that fit into the bind array.

Multiple INTO TABLE Statements

When calculating a bind array size for a control file that has multiple INTO TABLE statements, calculate as if the INTO TABLE statements were not present. Imagine all of the fields listed in the control file as one, long data structure — that is, the format of a single row in the bind array.

If the same field in the data record is mentioned in multiple INTO TABLE clauses, it requires additional space in the bind array each time it is mentioned. So, it is especially important to minimize the buffer allocations for fields like these.

Generated Data

Generated data is produced by the SQL*Loader functions CONSTANT, RECNUM, SYSDATE, and SEQUENCE. Such generated data does not require any space in the bind array.

Setting a Column to Null or Zero

If you want all inserted values for a given column to be null, omit the column's specifications entirely. To set a column's values *conditionally* to null based on a test of some condition in the logical record, use the NULLIF clause, described in this section. To set a numeric column to zero instead of NULL, use the DEFAULTIF clause, described next.

DEFAULTIF Clause

Using DEFAULTIF on numeric data sets the column to zero when the specified field condition is true. Using DEFAULTIF on character data (CHAR, DATE, or numeric EXTERNAL) data sets the column to null. See “Specifying Field Conditions” on page 5 – 37 for details on the conditional tests.


```
DEFAULTIF field_condition
```

A column may have both a NULLIF clause and a DEFAULTIF clause, although this often would be redundant.

Note: The same effects can be achieved with the SQL string and the DECODE function. See “Applying SQL Operators to Fields” on page 5 – 75.

NULLIF Keyword

Use the NULLIF keyword after the datatype and optional delimiter specification, followed by a condition. The condition has the same format as that specified for a WHEN clause. The column’s value is set to null if the condition is true. Otherwise, the value remains unchanged.

```
NULLIF field_condition
```

The NULLIF clause may refer to the column that contains it, as in the following example:

```
COLUMN1 POSITION(11:17) CHAR NULLIF (COLUMN1 = "unknown")
```

This specification may be useful if you want certain data values to be replaced by nulls. The value for a column is first determined from the datafile. It is then set to null just before the insert takes place. Case 6 on page 4 – 20 includes more examples of the NULLIF clause.

Note: The same effect can be achieved with the SQL string and the NVL function. See “Applying SQL Operators to Fields” on page 5 – 75.

Null Columns at the End of a Record

When the control file specifies more fields for a record than are present in the record, SQL*Loader must determine whether the remaining (specified) columns should be considered null or whether an error should be generated. The TRAILING NULLCOLS clause, described on page 5 – 36, tells SQL*Loader how to proceed in this case.

Loading All-Blank Fields

Totally blank fields for numeric or DATE fields cause the record to be rejected. To load one of these fields as null, use the NULLIF clause with the BLANKS keyword, as described in the section “Comparing Fields to BLANKS” on page 5 – 38. Case 6 on page 4 – 20 shows how to load all-blank fields as null with the NULLIF clause.

If an all-blank CHAR field is surrounded by enclosure delimiters, then the blanks within the enclosures are loaded. Otherwise, the field is loaded as null. More details on whitespace trimming in character fields are presented in the following section.

Trimming of Blanks and Tabs

Blanks and tabs constitute *whitespace*. Depending on how the field is specified, whitespace at the start of a field (*leading whitespace*) and at the end of a field (*trailing whitespace*) may, or may not be, included when the field is inserted into the database. This section describes the way character data fields are recognized, and how they are loaded. In particular, it describes the conditions under which whitespace is trimmed from fields.

Note: Specifying PRESERVE BLANKS changes this behavior. See “Preserving Whitespace” on page 5 – 74 for more information.

Datatypes

The information in this section applies only to fields specified with one of the *character-data* datatypes:

- CHAR datatype
- DATE datatype
- numeric EXTERNAL datatypes:
 - INTEGER EXTERNAL
 - FLOAT EXTERNAL
 - (packed) DECIMAL EXTERNAL
 - ZONED (decimal) EXTERNAL

VARCHAR Fields

Although VARCHAR fields also contain character data, these fields are never trimmed. A VARCHAR field includes all whitespace that is part of the field in the datafile.

Field Length Specifications

There are two ways to specify field length. If a field has a constant length that is defined in the control file, then it has a *predetermined size*. If a field's length is not known in advance, but depends on indicators in the record, then the field is *delimited*.

Record Formats

When all of the fields in a record have known positions and lengths, then the record is in *fixed format*, as shown in Case 2 on page 4 – 6. If the size or position of any fields in the record vary from record to record, then the record has a *variable format*, as shown in Case 1 on page 4 – 4. When the record size also varies, then the file is in *stream format*. Variable format fields are specified with delimiters. Fixed format fields are specified with predetermined sizes and fixed positions.

Predetermined Size Fields

Fields that have a predetermined size are specified with a starting position and ending position, or with a length, as in the following examples:

```
loc POSITION(19:31)
loc CHAR(14)
```

In the second case, even though the field's exact position is not specified, the field's length is predetermined.

Delimited Fields

Delimiters are characters that demarcate field boundaries. *Enclosure* delimiters surround a field, like the quotes in:

```
"__aa__"
```

where "__" represents blanks or tabs. *Termination* delimiters signal the end of a field, like the comma in:

```
__aa__,
```

Delimiters are specified with the control clauses TERMINATED BY and ENCLOSED BY, as shown in the following examples:

```
loc POSITION(19) TERMINATED BY ", "
loc POSITION(19) ENCLOSED BY ' '
loc TERMINATED BY "." OPTIONALLY ENCLOSED BY ' | '
```

Combining Delimiters with Predetermined Size

If predetermined size is specified for a delimited field, and the delimiter is not found within the boundaries indicated by the size specification; then an error is generated. For example, if you specify:

```
loc POSITION(19:31) CHAR TERMINATED BY ", "
```

and no comma is found between positions 19 and 31 of the input record, then the record is rejected. If a comma is found, then it delimits the field.

Relative Positioning of Fields

When a starting position is not specified for a field, it begins immediately after the end of the previous field. Figure 5 – 2 illustrates this situation when the previous field has a predetermined size.

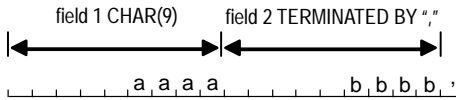


Figure 5 – 2 Relative positioning after a fixed field

If the previous field is terminated by a delimiter, then the next field begins immediately after the delimiter, as shown in Figure 5 – 3.

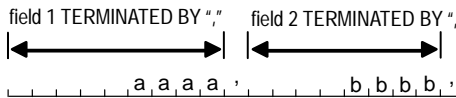


Figure 5 – 3 Relative positioning after a delimited field

When a field is specified both with enclosure delimiters and a termination delimiter, then the next field starts after the termination delimiter, as shown in Figure 5 – 4. If a non-whitespace character is found after the enclosure delimiter, but before the terminator, then SQL*Loader generates an error.

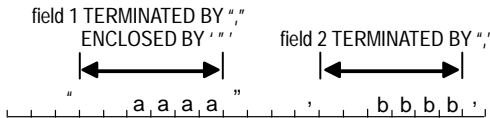


Figure 5 – 4 Relative positioning after enclosure delimiters

Leading Whitespace

In Figure 5 – 4, both fields are stored with leading whitespace. Fields do *not* include leading whitespace in the following cases:

- when the previous field is terminated by whitespace, and no starting position is specified for the current field
- when optional enclosure delimiters are specified for the field, and the enclosure delimiters are *not* present

These cases are illustrated in the following sections.

Previous Field Terminated by Whitespace If the previous field is TERMINATED BY WHITESPACE, then all the whitespace after the field acts as the delimiter. The next field starts at the next non-whitespace character. Figure 5 – 5 illustrates this case.

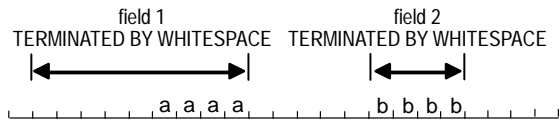


Figure 5 – 5 Fields terminated by whitespace

This situation occurs when the previous field is explicitly specified with the TERMINATED BY WHITESPACE clause, as shown in the example. It also occurs when you use the global FIELDS TERMINATED BY WHITESPACE clause.

Optional Enclosure Delimiters Leading whitespace is also removed from a field when optional enclosure delimiters are specified but not present.

Whenever optional enclosure delimiters are specified, SQL*Loader scans forward, looking for the first delimiter. If none is found, then the first non-whitespace character signals the start of the field. SQL*Loader skips over whitespace, eliminating it from the field. This situation is shown in Figure 5 – 6.

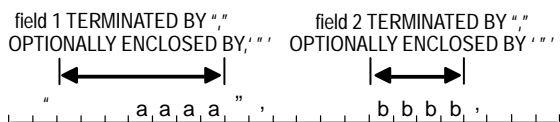


Figure 5 – 6 Fields terminated by optional enclosing delimiters

Unlike the case when the previous field is TERMINATED BY WHITESPACE, this specification removes leading whitespace even when a starting position is specified for the current field.

Note: If enclosure delimiters are present, leading whitespace after the initial enclosure delimiter is kept, but whitespace before this delimiter is discarded. See the first quote in FIELD1, Figure 5 – 6.

Trailing Whitespace Trailing whitespace is only trimmed from character-data fields that have a predetermined size. It is always trimmed from those fields.

Enclosed Fields If a field is enclosed, or terminated and enclosed, like the first field shown in Figure 5 – 6, then any whitespace outside the enclosure delimiters is not part of the field. Any whitespace between the enclosure delimiters belongs to the field, whether it is leading or trailing whitespace.

Trimming Whitespace: Table 5 – 5 summarizes when and how whitespace is removed from input data fields when PRESERVE BLANKS is not specified. See the following section, “Preserving Whitespace”, for details on how to prevent trimming.

Summary

Specification	Data	Result	Leading Whitespace Present ⁽¹⁾	Trailing Whitespace Present ⁽¹⁾
Predetermined Size	__aa__	__aa	Y	N
Terminated	__aa__,	__aa__	Y	Y ⁽²⁾
Enclosed	"__aa__"	__aa__	Y	Y
Terminated and Enclosed	"__aa__",	__aa__	Y	Y
Optional Enclosure (present)	"__aa__",	__aa__	Y	Y
Optional Enclosure (absent)	__aa__,	aa__	N	Y
Previous Field Terminated by Whitespace	__aa__	aa ⁽³⁾	N	(3)
<p>⁽¹⁾ When an allow-blank field is trimmed, its value is null.</p> <p>⁽²⁾ Except for fields that are TERMINATED BY WHITESPACE</p> <p>⁽³⁾ Presence of trailing whitespace depends on the current field's specification, as shown by the other entries in the table.</p>				

Table 5 – 5 Trim Table

Preserving Whitespace

To prevent whitespace trimming in all CHAR, DATE, and NUMERIC EXTERNAL fields, you specify PRESERVE BLANKS in the control file. Whitespace trimming is described in the previous section, “Trimming of Blanks and Tabs”.

PRESERVE BLANKS **Keyword**

PRESERVE BLANKS retains leading whitespace when optional enclosure delimiters are not present. It also leaves trailing whitespace intact when fields are specified with a predetermined size. This keyword preserves tabs and blanks.

For example, if the field

`__aa__`,

(where underscores represent blanks) is loaded with the following control clause:

```
TERMINATED BY ' ','' OPTIONALLY ENCLOSED BY ' "'
```

then both the leading whitespace and the trailing whitespace are retained if PRESERVE BLANKS is specified. Otherwise, the leading whitespace is trimmed.

Note: The word BLANKS is not optional. Both words must be specified.

Terminated by Whitespace When the previous field is terminated by whitespace, then PRESERVE BLANKS does not preserve the space at the beginning of the next field, unless that field is specified with a POSITION clause that includes some of the whitespace. Otherwise, SQL*Loader scans past all whitespace at the end of the previous field until it finds a non-blank, non-tab character.

Applying SQL Operators to Fields

A wide variety of SQL operators may be applied to field data with the SQL string. This string may contain any combination of SQL expressions that are recognized by Oracle as valid for the VALUES clause of an INSERT statement. In general, any SQL function that returns a single value may be used. See the section “Expressions” in Chapter 3, “Operators, Functions, Expressions, Conditions”, in the *Oracle7 Server SQL Reference*.

The SQL string must be enclosed in double quotation marks. It appears after any other specifications for a given column. It is evaluated after any NULLIF or DEFAULTIF clauses, but before a DATE mask. It may not be used on RECNUM, SEQUENCE, CONSTANT, or SYSDATE fields. If the RDBMS does not recognize the string, the load terminates in error. If the string is recognized, but causes a database error, the row that caused the error is rejected.

Referencing Fields

To refer to fields in the record, precede the field name with a colon (:). Field values from the current record are substituted. The following examples illustrate references to the current field:

```
field1 POSITION(1:6) CHAR "LOWER(:field1)"
field1 CHAR TERMINATED BY ','
      NULLIF ((1) = 'a') DEFAULTIF ((1)= 'b')
      "RTRIM(:field1)"
field1 CHAR(7) "TRANSLATE(:field1, ':field1', ':1')"
```

In the last example, only the *:field1* that is *not* in single quotes is interpreted as a column name. For more information on the use of quotes inside quoted strings, see “Specifying Filenames and Database Objects” on page 5 – 13.

Other fields in the same record can also be referenced, as in the following example:

```
field1 POSITION(1:4) INTEGER EXTERNAL
      "decode(:field2, '22', '34', :field1)"
```

Common Uses

Loading external data with an implied decimal point:

```
field1 POSITION(1:9) DECIMAL EXTERNAL(8) ":field1/1000"
```

Truncating fields that could be too long:

```
field1 CHAR TERMINATED BY "," "SUBSTR(:field1, 1, 10)"
```


Combinations of Operators

Multiple operators can also be combined, as in the following examples:

```
field1 POSITION(*+3) INTEGER EXTERNAL
      "TRUNC(RPAD(:field1,6,'0'), -2)"
field1 POSITION(1:8) INTEGER EXTERNAL
      "TRANSLATE(RTRIM(:field1),'N/A', '0')"
field1 CHARACTER(10)
      "NVL( LTRIM(RTRIM(:field1)), 'unknown' )"

```

Use with Date Mask

When used with a date mask, the date mask is evaluated after the SQL string. A field specified as:

```
field1 DATE 'dd-mon-yy' "RTRIM(:field1)"

```

would be inserted as:

```
TO_DATE(RTRIM(<field1_value>), 'dd-mon-yyyy')
```

Interpreting Formatted Fields

It is possible to use the TO_CHAR operator to store formatted dates and numbers. For example:

```
field1 ... "TO_CHAR(:field1, '$09999.99')"
```

could store numeric input data in formatted form, where *field1* is a character column in the database. This field would be stored with the formatting characters (dollar sign, period, and so on) already in place.

You have even more flexibility, however, if you store such values as numeric quantities or dates. You can then apply arithmetic functions to the values in the database, and still select formatted values for your reports.

The SQL string is used in Case 7 (page 4 – 22) to load data from a formatted report.

SQL*Loader Command-Line Reference

This chapter shows you how to run SQL*Loader with command-line keywords. The following subjects are discussed:

- Invoking SQL*Loader (see page 6 – 2)
- Command-Line Keywords and Their Valid Arguments (see page 6 – 3)

The SQL*Loader Command Line

You can invoke SQL*Loader from the command line followed by certain keywords.



OSDoc

Additional Information: The command to invoke SQL*Loader is operating system–dependent. The following examples use the UNIX–based name, “sqlldr”. See your Oracle operating system–specific documentation for the correct command for your system.

If you invoke SQL*Loader with no keywords, SQL*Loader displays a help screen with the available keywords and default values. The following example shows default values that are the same on all operating systems.

```
sqlldr
```

```
...
```

Valid Keywords:

```
userid - Oracle username/password
control - Control file name
  log - Log file name
  bad - Bad file name
  data - Data file name
discard - Discard file name
discardmax - Number of discards to allow
              (Default all)
  skip - Number of logical records to skip
              (Default 0)
  load - Number of logical records to load
              (Default all)
errors - Number of errors to allow
              (Default 50)
  rows - Number of rows in conventional path bind array
          or between direct path data saves
              (Default: Conventional Path 64, Direct path all)
bindsize - Size of conventional path bind array in bytes
              (System-dependent default)
silent - Suppress messages during run
              (header, feedback, errors, discards)
direct - Use direct path
              (Default FALSE)
parfile - Parameter file: name of file that contains
          parameter specifications
parallel - Perform parallel load
              (Default FALSE)
  file - File to allocate extents from
```

Using Command-Line Keywords

Keywords are optionally separated by commas. They are entered in any order. Keywords are followed by valid arguments.

For example, :

```
SQLLDR CONTROL=foo.ctl, LOG=bar.log, BAD=baz.bad, DATA=etc.dat
      USERID=scott/tiger, ERRORS=999, LOAD=2000, DISCARD=toss.dis,
      DISCARDMAX=5
```

Specifying Keywords in the Control File

If the command line's length exceeds the size of the maximum command line on your system, you can put some of the command-line keywords in the control file, using the control file keyword **OPTIONS**. See page 5 – 12.

They can also be specified in a separate file specified by the keyword **PARFILE** (see page 6 – 5). These alternative methods are useful for keyword entries that seldom change. Keywords specified in this manner can still be overridden from the command line.

Command-Line Keywords

This section describes each of SQL*Loader's command-line keywords.

Keyword	Identifies
BAD	Bad File

BAD specifies the name the bad file created by SQL*Loader to store records that cause errors during insert or that are improperly formatted. If a filename is not specified, the name of the control file is used by default with the extension .BAD. This file has the same format as the input datafile, so it can be loaded by the same control file after appropriate updates or corrections are made.

A bad file filename specified on the command line becomes the bad file associated with the first INFILE statement in the control file. If the bad file filename was also specified in the control file, the command-line value overrides it.

BINDSIZE	Maximum sizes
----------	---------------

BINDSIZE specifies the maximum size of the bind array in bytes. The size of the bind array given by BINDSIZE overrides the default size (which is system dependent) and any size determined by ROWS. The bind array is discussed on page 5 – 63.

Keyword	Identifies
CONTROL	Control File
CONTROL specifies the control file that describes how to load data. If a file extension or file type is not specified, it defaults to CTL. If omitted, you are prompted for it.	
DATA	Datafile
DATA specifies the data file containing the data to be loaded. If a filename is not specified, the name of the control file is used by default. If a file extension or file type is not specified, it defaults to DAT.	
DIRECT	Data path
DIRECT specifies the load method to use, conventional path or direct path. TRUE specifies a direct path load. FALSE specifies a conventional path load. The defaults is FALSE. Load methods are explained in Chapter 8.	
DISCARD	Discard file
DISCARD specifies an optional discard file which will be created by SQL*Loader to store records that are neither rejected, nor inserted, into a table. If a filename is not specified, the name of the control file is used by default with the default extension .DSC. This file has the same format as the input datafile. So it can be loaded by the same control file after appropriate updates or corrections are made. A discard file filename specified on the command line becomes the discard file associated with the first INFILE statement in the control file. If the discard file filename was also specified in the control file, the command-line value overrides it.	
DISCARDMAX	Discards to allow
DISCARDMAX specifies the number of discard records that will terminate the load. The default value is all discards are allowed. To stop on the first discarded record, specify one (1).	
ERRORS	Errors to allow
ERRORS specifies the number of insert errors that will terminate the load. The default is 50. To stop on the first error, specify one (1). To specify that all errors be allowed, use a very high number.	

Keyword	Identifies
FILE	File to Load Into

FILE specifies the database file to allocate extents from. It is used only for parallel loads. By varying the value of the FILE parameter for different SQL*Loader processes, data can be loaded onto a system with minimal disk contention. For more information, see “Parallel Data Loading” on page 8 – 21.

LOAD	Records to load
------	-----------------

LOAD specifies the maximum number of logical records to load (after skipping the specified number of records). By default all records are loaded. No error occurs if fewer than the maximum number of records are found.

LOG	Log File
-----	----------

LOG specifies the log file which SQL*Loader will create to store logging information about the loading process. If a filename is not specified, the name of the control file is used by default with the default extension of .LOG.

PARFILE	Parameter File
---------	----------------

PARFILE specifies the name of a file that contains commonly-used command-line parameters. For example, the command line could read:

```
SQLLDR PARFILE=example.par
```

and the parameter file could have the following contents:

```
userid=scott/tiger
control=example.ctl
errors=9999
log=example.log
```

Note: Although it is not usually important, on some systems it may be necessary to have no spaces around the equal sign (“=”) in the parameter specifications.

PARALLEL	Parallel Load
----------	---------------

PARALLEL specifies whether direct loads can operate in multiple concurrent sessions to load data into the same table. For more information on PARALLEL loads, see “Parallel Data Loading” on page 8 – 21.

Keyword	Identifies
ROWS	Rows per commit

Conventional path loads only: ROWS specifies the number of rows in the bind array. The default is 64. (The bind array is discussed on page 5 – 63.)

Direct path, loads only: ROWS identifies the number of rows you want to read from the data file before a data save. The default is to save data once at the end of the load. For more information, see “Data Saves” on page 8 – 9.

Because the direct load is optimized for performance, it uses buffers that are the same size and format as the system’s I/O blocks. Only full buffers are written to the database, so the value of ROWS is approximate.

SILENT	Feedback mode
--------	---------------

When SQL*Loader begins, a *header message* like this appears on the screen and is placed in the log file:

```
SQL*Loader:   Production on Wed Feb 24 15:07:23...
Copyright (c) Oracle Corporation...
```

As SQL*Loader executes, you also see *feedback messages* on the screen, like this:

```
Commit point reached - logical record count 20
```

SQL*Loader may also display *data error messages* like these:

```
Record 4: Rejected - Error on table EMP
ORA-00001: unique constraint <name> violated
```

You can suppress these messages by specifying SILENT with an argument. For example, you can suppress the header and feedback messages that normally appear on the screen with the following command–line argument:

```
SILENT=(HEADER, FEEDBACK)
```

Use the appropriate keyword to suppress:

- | | |
|----------|---|
| HEADER | Suppresses the SQL*Loader header messages that normally appear on the screen. Header messages still appear in the log file. |
| FEEDBACK | Suppresses the “commit point reached” feedback messages that normally appear on the screen. |

ERRORS	Suppresses the data error messages in the log file that occur when a record generates an Oracle error that causes it to be written to the bad file. A count of rejected records still appears.
DISCARDS	Suppresses the messages in the log file for each record written to the discard file.
ALL	All of the above.

Keyword	Identifies
SKIP	Records to skip

SKIP specifies the number of logical records from the beginning of the file that should not be loaded. By default, no records are skipped.

This parameter continues loads that have been interrupted for some reason. It is used for all conventional loads, for single-table direct loads, and for multiple-table direct loads when the same number of records were loaded into each table. It is not used for multiple table direct loads when a different number of records were loaded into each table. See “Continuing Multiple Table Direct Loads” on page 5 – 28 for more information.

USERID	Userid
--------	--------

USERID is used to provide your Oracle *username/password*. If omitted, you are prompted for it. If just a slash is used, USERID defaults to your operating system logon. A SQL*Net database specification string can be used for a conventional path load into a remote database. For more information on SQL*Net, see your SQL*Net documentation.

SQL*Loader Log File Reference

When SQL*Loader begins execution, it creates a log file. The log file contains a detailed summary of the load.

Most of the log file entries will be records of successful SQL*Loader execution. However, errors can also cause log file entries. For example, errors found during parsing of the control file will appear in the log file.

This chapter describes the following log file entries:

- Header Entries
- Global Entries
- Table Entries
- Data File Entries
- Table Load Entries
- Summary Statistics

Header Information

The Header Section contains the following entries:

- date of the run
- software version number

For example:

```
SQL*Loader: Version 7.1.3.0.0 - Production on Mon Nov 26...  
Copyright (c) Oracle Corporation...
```

Global Information

The Global Information Section contains the following entries:

- names of all input/output files
- echo of command-line arguments
- continuation character specification

If the data is in the control file, then the data file is shown as “*”.

For example:

```
Control File:   LOAD.CTL  
Data File:     LOAD.DAT  
  Bad File:    LOAD.BAD  
  Discard File: LOAD.DSC
```

```
(Allow all discards)
```

```
Number to load: ALL  
Number to skip: 0  
Errors allowed: 50  
Bind array:    64 rows, maximum of 65536 bytes  
Continuation:  1:1 = '*', in current physical record  
Path used:     Conventional
```

Table Information

The Table Information Section provides the following entries for each table loaded:

- table name
- load conditions, if any. That is, whether all record were loaded or only those meeting WHEN–clause criteria.
- INSERT, APPEND, or REPLACE specification
- the following column information:
 - if found in data file, the position, length, datatype, and delimiter
 - if specified, RECNUM, SEQUENCE, or CONSTANT
 - if specified, DEFAULTIF, or NULLIF

For example:

Table EMP, loaded from every logical record.
Insert option in effect for this table: REPLACE

Column Name	Position	Len	Term	Encl	Datatype
-----	-----	---	---	---	-----
EMPNO	1:4	4			CHARACTER
ENAME	6:15	10			CHARACTER
JOB	17:25	9			CHARACTER
MGR	27:30	4			CHARACTER
SAL	32:39	8			CHARACTER
COMM	41:48	8			CHARACTER
DEPTNO	50:51	2			CHARACTER

Column EMPNO is NULL if EMPNO = BLANKS
Column MGR is NULL if MGR = BLANKS
Column SAL is NULL if SAL = BLANKS
Column COMM is NULL if COMM = BLANKS
Column DEPTNO is NULL if DEPTNO = BLANKS

Datafile Information

The Datafile Information Section appears only for datafiles with data errors, and provides the following entries:

- SQL*Loader/Oracle data records errors
- records discarded

For example:

```
Record 2: Rejected - Error on table EMP.  
ORA-00001: unique constraint <name> violated  
Record 8: Rejected - Error on table EMP, column DEPTNO.  
ORA-01722: invalid number  
Record 3: Rejected - Error on table PROJ, column PROJNO.  
ORA-01722: invalid number
```

Table Load Information

The Table Load Information Section provides the following entries for each table that was loaded:

- number of rows loaded
- number of rows that qualified for loading but were rejected due to data errors
- number of rows that were discarded because they met no WHEN-clause tests
- number of rows whose relevant fields were all null

For example:

```
The following indexes on table EMP were processed:  
Index EMPIDX was left in Direct Load State due to  
ORA-01452: cannot CREATE UNIQUE INDEX; duplicate keys found
```

```
Table EMP:  
  7 Rows successfully loaded.  
  2 Rows not loaded due to data errors.  
  0 Rows not loaded because all WHEN clauses were failed.  
  0 Rows not loaded because all fields were null.
```

Summary Statistics

The Summary Statistics Section displays the following data:

- amount of space used:
 - for bind array (what was actually used, based on what was specified by BINDSIZE)
 - for other overhead (always required, independent of BINDSIZE)
- cumulative load statistics. That is, for all data files, the number of records that were:
 - skipped
 - read
 - rejected
 - discarded
- beginning/ending time of run
- total elapsed time
- total CPU time (includes all file I/O but may not include background Oracle CPU time)

For example:

```
Space allocated for bind array:          65336 bytes (64 rows)
Space allocated for memory less bind array: 6470 bytes
```

```
Total logical records skipped:          0
Total logical records read:              7
Total logical records rejected:          0
Total logical records discarded:         0
```

```
Run began on Mon Nov 26 10:46:53 1990
Run ended on Mon Nov 26 10:47:17 1990
```

```
Elapsed time was:      00:00:15.62
CPU time was:          00:00:07.76
```


SQL*Loader Conventional and Direct Path Loads

This chapter describes SQL*Loader's conventional and direct path load methods. The following topics are covered:

- Overview of Data Loading Methods
- Conventional Path Load Method
- Direct Path Load Method
- Maximizing Performance of Direct Path Loads

For an example of loading with using the direct path load method, see Case 6 on page 4 – 20. The other cases use the conventional path load method.

Note: You can use the direct path load method with Trusted Oracle7 Server just as you can with the standard Oracle7 Server.

Data Loading Methods

SQL*Loader provides two methods for loading data:

- conventional path load
- direct path load

Direct path loads can be significantly faster than conventional path loads. Direct path loads achieve this performance gain by eliminating much of the Oracle database overhead by writing directly to the database files. The direct load, therefore, does not compete with other users for database resources so it can usually load data at nearly disk speed. Certain considerations, inherent to this method of access to database files, such as security and backup implications, are discussed in this chapter.

Conventional Path Loads

Conventional path loads (the default) use the SQL command INSERT and a bind array buffer to load data into database tables. This method is used by all Oracle tools and applications.

When SQL*Loader performs a conventional path load, it competes equally with all other processes for buffer resources. This can slow the load significantly. Extra overhead is added as SQL commands are generated, passed to Oracle, and processed.

Oracle looks for partially filled blocks and attempts to fill them on each insert. Although appropriate during normal use, this can slow bulk loads dramatically.

When to Use a Conventional Path Load

Because the direct path is many times faster than the conventional path, it is highly desirable to use the direct path. But there are times when the conventional path is preferred. You should use the conventional path in the following situations:

- When accessing an indexed table concurrently with the load, or when applying inserts or updates to a non-indexed table concurrently with the load.

To use the direct path (excepting parallel loads), SQL*Loader must have exclusive write access to the table and exclusive read-write access to any indexes.

- When loading data with SQL*Net.

You cannot load data through the direct path with SQL*Net; unless both systems belong to the same family of computers, and both are using the same character set. Even then, load performance can be significantly impaired by network overhead.

- When loading data into a clustered table.

Clustered tables cannot be loaded through the direct path.

- When loading a relatively small number of rows into a large indexed table.

On the direct path, the existing index is copied when it is merged with the new index keys. If the existing index is very large and the number of new keys is very small, then the index copy time can offset the time saved by loading the data with the direct path.

- When loading a relatively small number of rows into a large table with referential and column-check integrity constraints.

Because these constraints cannot be applied to rows loaded on the direct path, they are disabled for the duration of the load. Then they are applied to the whole table when the load completes. The costs could outweigh the savings for a very large table and a small number of new rows.

- When you want to apply SQL functions to data fields.

SQL functions are not available on the direct path. For more information on the SQL functions, see “Applying SQL Operators to Fields” on page 5 – 75.

Direct Path Loads

Direct path loads are optimized for maximum data loading capability. Like the conventional path method, SQL*Loader’s direct path method provides full support for media recovery.

Instead of filling a bind array buffer and passing it to Oracle with a SQL INSERT command, the direct path option creates data blocks that are already in Oracle database block format. These database blocks are then written directly to the database.

Internally, multiple buffers are used for the formatted data. While one buffer is being filled, multiple buffers are being written if asynchronous I/O is available on the host platform. This parallelism increases load performance.

Figure 8 – 1 shows how conventional and direct path loads perform database writes.

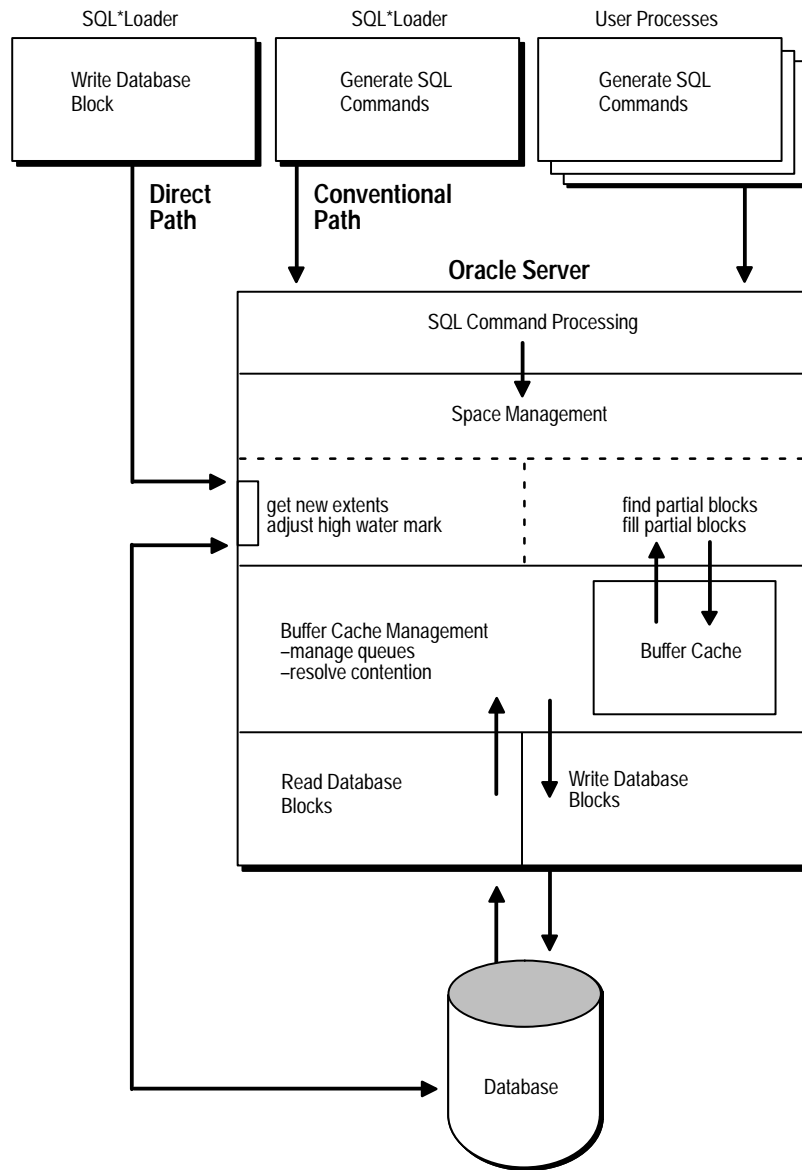


Figure 8 – 1 Database Writes on Direct Path and Conventional Path

Although direct path loads minimize the necessity of database processing, a few, fast calls to Oracle are made at the beginning and end of the load. Tables are locked and the locks are released at the end. Also, during the load, space management routines are used to get new extents when needed and to adjust the *high-water mark*. The high-water mark is described in “Data Saves” on page 8 – 9.

Oracle calls are also used to sort the data and build the index.

SQL calls are *not* performed anytime during the load.

Advantages of Direct Path Loads

The direct path method is faster than the conventional path for the following reasons:

- Partial blocks are not used, so no reads are needed to find them and fewer writes are performed.
- SQL INSERT commands are not generated by SQL*Loader, and therefore, processing load on the Oracle database is reduced.
- The bind-array buffer is not used — formatted database blocks are written directly.
- The direct path method calls on Oracle to lock tables and indexes at the start of the load and releases them when the load is finished. The conventional path calls Oracle once for each array of rows to process a SQL INSERT statement.
- Unlike conventional path loads, direct path loads use asynchronous I/O, if available, to perform these operations in parallel:
 - reading from input files
 - writing to database files
- Processes using the direct path perform their own write I/O, instead of using Oracle's buffer cache in contention with other Oracle users. Therefore, the direct path does not contend for free buffers in the buffer cache.
- The direct path's pre-sorting option allows you to use high-performance sort routines that are native to your system or installation.
- When the table to be loaded is empty, the pre-sorting option eliminates the sort and merge phases of index-building — the index is simply filled in as data arrives.
- Protection against instance failure does not require redo log file entries during direct path loads. Therefore, if Oracle is operating in NOARCHIVELOG mode, no time is required to log the load. See "Instance Recovery with the Direct Path" on page 8 – 10.

When to Use a Direct Path Load You should use a direct path load in the following situations:

- You have a large amount of data to load quickly. A direct path can quickly load and index large amounts of data. It can also load data into either an empty or non-empty table.
- You want to load data in PARALLEL for maximum performance. See page 8 – 21.
- You want to load data in a character set that cannot be supported in your current session, or when the conventional conversion to the database character set would cause errors.

Conditions for Using Direct PATH LOADS

In addition to the general load conditions described on page 3 – 9, the following conditions must be satisfied to use the direct path load method:

- Tables are not clustered.
- Tables to be loaded do not have any active transactions pending.

To check for this condition, use the SQL*DBA command MONITOR TABLE to find the object ID for the table(s) you want to load. Then use the command MONITOR LOCK to see if there are any locks on the table.

- SQL strings are not used in the control file.
- If the table(s) is indexed, there are no current SELECT statements on the table(s).

It may be necessary to log off users that have SELECT statements on the table(s). You can use the The SQL*DBA command MONITOR TABLE to see which Oracle users are accessing the table.

Integrity Constraints

All integrity constraints are enforced during direct path loads, although not necessarily at the same time. All constraints that can be checked without referring to other rows or tables, such as the NOT NULL constraint, are enforced during the load. Records that fail these constraints are rejected.

Integrity constraints that depend on other rows or tables, such as referential constraints, are disabled before the direct path load and must be re-enabled afterwards. If REENABLE is specified, SQL*Loader can re-enable them automatically at the end of the load. When the constraints are re-enabled, the entire table is checked. Any rows that fail this check are reported in the specified error log. See the section in this chapter called “Direct Loads, Integrity Constraints, and Triggers”.

Field Defaults on the Direct Path	DEFAULT column specifications defined in the database are not available when loading on the direct path. Fields for which default values are desired must be specified with the DEFAULTIF clause, described on page 5 – 67. If a DEFAULTIF clause is not specified, and the field is NULL, then a NULL value is inserted into the database.
Loading into Synonyms	You can load data into a synonym for a table during a the direct path load, but the synonym must point directly to a table. It cannot be a synonym for a view or a synonym for another synonym.
Exact Version Requirement	A SQL*Loader direct load can only be done for a database of the same version. For example, you cannot do a SQL*Loader Version 7.1.2 direct path load to load into a Oracle Version 7.1.3 database.

Using Direct Path Load

This section explains you how to use SQL*Loader's direct path load.

Setting Up for Direct Path Loads

To prepare the database for direct path loads, you must run the setup script, CATLDR.SQL to create the necessary views. You need only run this script once for each database you plan to do direct loads to. This script can be run during database installation if you know then that you will be doing direct loads.

Specifying a Direct Path Load

To start SQL*Loader in direct load mode, the parameter DIRECT must be set to TRUE on the command line or in the parameter file, if used, in the format:

```
DIRECT=TRUE
```

See Case 6 on page 4 – 20 for an example.

Building Indexes

During a direct path load, performance is improved by using temporary storage. After the data is loaded into the table, the new keys are copied to a temporary segment and sorted. The old index and the new keys are then merged to create the new index. The old index, temporary segment, and new index all require storage until the merge is complete. Then the old index and temporary segment are removed.

Note that, during a conventional path load, every time a row is inserted the index is updated. This method does not require temporary storage space, but it does add processing time.

The SINGLEROW Option Performance on systems with limited memory can also be improved by using the SINGLEROW option. For more information see page 5 – 36.

Note: If, during a direct load, you have specified that the data is to be pre-sorted and the existing index is empty, a temporary segment is not required, and no merge occurs—the keys are put directly into the index. See “Maximizing Performance of Direct Path Loads” on page 8 – 12 for more information.

When multiple indexes are built, the temporary segments corresponding to each index exist simultaneously, in addition to the old indexes. The new keys are then merged with the old indexes, one index at a time. As each new index is created, the old index and the corresponding temporary segment are removed.

Index Storage Requirements

The formula for calculating the amount of space needed for storing the index itself can be found in Chapter 8 “Managing Database Files” of the *Oracle7 Server Administrator’s Guide*. Remember that two indexes exist until the load is complete: the old index and the new index.

Temporary Segment Storage Requirements

The amount of temporary segment space needed for storing the new index keys (in bytes) can be estimated using the following formula:

$1.3 * key_storage$

where:

$key_storage = (number_of_rows) * (10 + sum_of_column_sizes + number_of_columns)$

The columns included in this formula are the columns in the index. There is one length byte per column, and 10 bytes per row are used for a ROWID and additional overhead.

The constant 1.3 reflects the average amount of extra space needed for sorting. This value is appropriate for most randomly ordered data. If the data arrives in exactly opposite order, twice the key-storage space is required for sorting, and the value of this constant would be 2.0. That is the worst case.

If the data is fully sorted, only enough space to store the index entries is required, and the value of this constant reduces to 1.0. See “Pre-sorting Data for Faster Indexing” on page 8 – 13 for more information.

Indexes Left in Direct Load State

SQL*Loader may leave indexes in *direct load state* if a direct path load does not complete successfully.

Any SQL statement that tries to use an index that is in direct load state returns an error. The following conditions cause the direct path option to leave an index in direct load state:

- SQL*Loader runs out of space for the index.
- The data is not in the order specified by the SORTED INDEXES clause.
- There is an instance failure while building the index.
- There are duplicate keys in a unique index.

To determine if an index is in direct load state, you can execute a simple query:

```
SELECT INDEX_NAME, STATUS
FROM USER_INDEXES
WHERE TABLE_NAME = 'tablename';
```

If you are not the owner of the table, then search ALL_INDEXES or DBA_INDEXES instead of USER_INDEXES.

Data Saves

You can use *data saves* to protect against loss of data due to instance or media failure. All data loaded up to the last data save is protected against instance failure. To continue the load after an instance failure, determine how many rows from the input file were processed before the failure, then use the SKIP option to skip those processed rows. If there were any indexes on the table, drop them before continuing the load, then recreate them after the load. See “Recovery” on page 8 – 10 for more information on media and instance failure.

Note: Indexes are not protected by a data save, because SQL*Loader usually does not build indexes until after data loading completes. (The only time indexes are built during the load is when pre-sorted data is loaded into an empty table — but these indexes are also unprotected.)

Using the ROWS Parameter

The parameter ROWS determines when data saves occur during a direct path load. The value you specify for ROWS is the number of rows you want SQL*Loader to read from the input file before saving inserts in the database.

The number of rows you specify for a data save is an approximate number. Direct loads always act on full data buffers that match the format of Oracle database blocks. So, the actual number of data rows

saved is rounded up to a multiple of the number of rows in a database block.

SQL*Loader always reads the number of rows needed to fill a database block. Discarded and rejected records are then removed, and the remaining records are inserted into the database. So the actual number of rows inserted before a save is the value you specify, rounded up to the number of rows in a database block, minus the number of discarded and rejected records.

Data Save Versus Commit In a conventional load, ROWS is the number of rows to read before a commit. A direct load data save is similar to a conventional load commit, but it is not identical. The similarities are:

- Data save will make the rows visible to other users
- Rows cannot be rolled back after a data save

The major difference is that the indexes will be unusable (in DIRECT load state) until the load completes.

Recovery

SQL *Loader provides full support for data recovery when using the direct path option. There are two main types of recovery:

Media Recovery *Media recovery* is recovering from the loss of a database file. You must operate in ARCHIVELOG mode to recover after a file has been lost.

Instance Recovery *Instance recovery* is recovering from a system failure in which in-memory data was changed (but not written to disk) before the failure occurred. Oracle can always recover from instance failures, even if redo log files are not archived.

See the *Oracle7 Server Administrator's Guide* for more information about recovery.

Instance Recovery and Direct Path Loads

Because SQL*Loader writes directly to the database files, all rows inserted up to the last data save will automatically be present in the database files if the instance is restarted. Changes do not need to be recorded in the redo log file to make instance recovery possible.

If an instance failure occurs, the indexes being built may be left in direct load state. Drop and re-create any affected indexes before using the table or continuing the load. See "Indexes Left in Direct Load State" on page 8 – 9 for more information on how to determine if an index has been left in direct load state.

Media Recovery and Direct Path Loads

If redo log file archiving is enabled (you are operating in ARCHIVELOG mode), SQL*Loader logs loaded data when using the direct path, making media recovery possible. If redo log archiving is not enabled (you are operating in NOARCHIVELOG mode), then media recovery is not possible.

To recover a database file that was lost while it was being loaded, use the same method that you use to recover data loaded with the conventional path:

1. Restore the most recent backup of the affected database file.
2. Recover the tablespace using the RECOVER command. (See the *Oracle7 Server Administrator's Guide* for more information on the RECOVER command.)

Loading LONG Data Fields

Data that is longer than SQL*Loader's maximum buffer size can be loaded on the direct path with either the PIECED option or by specifying the number of READBUFFERS. This section describes those two options.

Loading Data as PIECED

The data can be loaded in sections with the pieced option if it is the last column of the logical record. The syntax for this specification is given on page 5 – 8.

Declaring a column as PIECED informs the direct path loader that the field may be processed in pieces, one buffer at once.

The following restrictions apply when declaring a column as PIECED:

- This option is only valid on the direct path.
- Only one field per table may be PIECED.
- The PIECED field must be the last field in the logical record.
- The PIECED field may not be used in any WHEN, NULLIF, or DEFAULTIF clauses.
- The PIECED field's region in the logical record must not overlap with any other field's region.
- The PIECED corresponding database column may not be part of the index.
- It may not be possible to load a rejected record from the bad file if it contains a PIECED field.

For example, a PIECED file could span 3 records. SQL*Loader loads the piece from the first record and then reuses the buffer for the second buffer. After loading the second piece, the buffer is reused for the third record. If an error is then discovered, only the third record is placed in the bad file because the first two records no longer exist in the buffer. As a result, the record in the bad file would not be valid.

Using the READBUFFERS Keyword

For data that is not divided into separate sections, or not in the last column, READBUFFERS can be specified. With READBUFFERS a buffer transfer area can be allocated that is large enough to hold the entire logical record at one time.

READBUFFERS specifies the number of buffers to use during a direct path load. (A LONG can span multiple buffers.) The default value is four buffers. If the number of read buffers is too small, the following error results:

```
ORA-02374 ... No more slots for read buffer queue
```

Note: Do not specify a value for READBUFFERS unless it becomes necessary, as indicated by ORA-2374. Values of READBUFFERS that are larger than necessary do not enhance performance. Instead, higher values unnecessarily increase system overhead.

Maximizing Performance of Direct Path Loads

You can control the time and temporary storage used during direct path loads.

To minimize time:

- Allocate I/O Buffers.
- Pre-allocate storage space.
- Pre-sort the data.
- Perform infrequent data saves.
- Disable archiving of redo log files.

To minimize space:

- When sorting data before the load, sort data on the index that requires the most temporary storage space.
- Drop indexes and recreate them after the load.

Allocating I/O Buffers

When doing a direct path load, it is advisable to specify a large number of buffers, if your operating system provides for that capacity. Buffers are allocated with the I/O processing options string, described on page 5 – 18.

On some systems, as many as 200 buffers are needed to keep the CPU busy. Otherwise, the CPU spends most of its time idling, waiting for I/O to complete. If you can measure CPU utilization, you will achieve maximum performance of direct loads when the number of buffers allow the CPU to operate at 95% to 98% utilization.



OSDoc

Additional Information: The procedure for allocating additional I/O buffers depends on your operating system. See your Oracle operating system-specific documentation for more information.

Pre-allocating Storage for Faster Loading

SQL*Loader automatically adds extents to the table if necessary, but this process takes time. For faster loads into a new table, allocate the required extents when the table is created.

To calculate the space required by a table, see Chapter 8 “Managing Database Files” in the *Oracle7 Server Administrator’s Guide*. Then use the INITIAL or MINEXTENTS clause in the SQL command CREATE TABLE to allocate the required space.

Pre-sorting Data for Faster Indexing

You can improve the performance of direct path loads by pre-sorting your data on indexed columns. Pre-sorting minimizes temporary storage requirements during the load. Pre-sorting also allows you to take advantage of high-performance sorting routines that are optimized for your operating system or application.

If the data is pre-sorted and the existing index is not empty, then pre-sorting minimizes the amount of temporary segment space needed for the new keys. The sort routine appends each new key to the key list. Instead of requiring extra space for sorting, only space for the keys is needed. To calculate the amount of storage needed, use a sort factor of 1.0 instead of 1.3. For more information on estimating storage requirements, see “Temporary Segment Storage Requirements” on page 8 – 8.

If pre-sorting is specified and the existing index is empty, then maximum efficiency is achieved. The sort routines are completely bypassed, with the merge phase of index creation. The new keys are simply inserted into the index. Instead of having a temporary segment and new index existing simultaneously with the empty, old index, only the new index exists. So, temporary storage is not required, and time is saved.

SORTED INDEXES
Statement

The SORTED INDEXES statement identifies the indexes on which the data is presorted. This statement is allowed only for direct path loads. See Chapter 5, “SQL*Loader Control File Reference,” for the syntax. See Case 6 on page 4 – 20 for an illustration.

Generally, you specify only one index in the SORTED INDEXES statement because data that is sorted for one index is not usually in the right order for another index. When the data is in the same order for multiple indexes, however, all of the indexes can be specified at once.

All indexes listed in the SORTED INDEXES statement must be created before you start the direct path load.

Unsorted Data

If you specify an index in the SORTED INDEXES statement, and the data is not sorted for that index, then the index is left in *direct load state* at the end of the load. The data is present, but any attempt to use the index results in an error. Any index which is left in direct load state must be dropped and re-created after the load.

Multiple Column Indexes

If you specify a multiple-column index in the SORTED INDEXES statement, the data should be sorted so that it is ordered first on the first column in the index, next on the second column in the index, and so on.

For example, if the first column of the index is city, and the second column is last name; then the data should be ordered by name within each city, as in the following list:

Albuquerque	Adams
Albuquerque	Hartstein
Albuquerque	Klein
...	...
Boston	Andrews
Boston	Bobrowski
Boston	Heigham
...	...

Choosing the Best Sort
Order

For the best overall performance of direct path loads, you should presort the data based on the index that requires the most temporary segment space. For example, if the primary key is one numeric column, and the secondary key consists of three text columns, then you can minimize both sort time and storage requirements by pre-sorting on the secondary key.

To determine the index that requires the most storage space, use the following procedure:

1. For each index, add up the widths of all columns in that index.
2. For a single-table load, pick the index with the largest overall width.
3. For each table in a multiple table load, identify the index with the largest, overall width for each table. If the same number of rows are to be loaded into each table, then again pick the index with the largest overall width. Usually, the same number of rows are loaded into each table.
4. If a different number of rows are to be loaded into the indexed tables in a multiple table load, then multiply the width of each index identified in step 3 by the number of rows that are to be loaded into that index. Multiply the number of rows to be loaded into each index by the width of that index and pick the index with the largest result.

Infrequent Data Saves

Frequent data saves resulting from a small ROWS value adversely affect the performance of a direct path load. Because direct path loads can be many times faster than conventional loads, the value of ROWS should be considerably higher for a direct load than it would be for a conventional load.

During a data save, loading stops until all of SQL*Loader's buffers are successfully written. You should select the largest value for ROWS that is consistent with safety. It is a good idea to determine the average time to load a row by loading a few thousand rows. Then you can use that value to select a good value for ROWS.

For example, if you can load 20,000 rows per minute, and you do not want to repeat more than 10 minutes of work after an interruption, then set ROWS to be 200,000 (20,000 rows/minute * 10 minutes).

Minimizing Use of the Redo Log

One way to speed a direct load dramatically is to minimize use of the redo log. There are two ways to do this. You can disable archiving, or you can specify that the load is UNRECOVERABLE. This section discusses both methods.

Specifying UNRECOVERABLE

Use UNRECOVERABLE to save time and space in the redo log file. An UNRECOVERABLE load does not record loaded data in the redo log file.

Therefore, media recovery is disabled for the loaded table, although database changes by other users may continue to be logged.

Note: Because the data load is not logged, you may want to make a backup of the data after loading.

If media recovery becomes necessary on data that was loaded with the UNRECOVERABLE phrase, the data blocks that were loaded are marked as logically corrupted.

To recover the data, drop and re-create the data. It is a good idea to do backups immediately after the load to preserve the otherwise unrecoverable data.

By default, a direct path load is RECOVERABLE. See “Data Definition Language Syntax” on page 5 – 4 for information on RECOVERABLE and UNRECOVERABLE.

Dropping Indexes

For both the conventional path and the direct path, SQL*Loader builds all existing indexes for a table. The only way to avoid building an index is to drop it before the load and re-create it afterwards.

Dropping and re-creating indexes is one way to save temporary storage while using the direct load facility. This action minimizes the amount of space required during the load, for the following reasons:

- You can build multiple indexes one at a time, saving the temporary segment space that would otherwise be needed for each index.
- Only one index segment exists when an index is built, instead of the three segments that temporarily exist when the new keys are merged into the old index to make the new index.

This approach is quite reasonable when the number of rows to be loaded is large compared to the size of the table. But if relatively few rows are added to a large table, then the time required to re-sort the indexes may be excessive. In such cases, it is usually better to make use of the conventional path.

Direct Loads, Integrity Constraints, and Triggers

With the conventional path, arrays of data are inserted with standard SQL statements — integrity constraints and insert triggers are automatically applied. But when loading data on the direct path, some integrity constraints and all database triggers are disabled. This section discusses the implications of using direct path loads with respect to these features.

Integrity Constraints During a direct path load, some integrity constraints are automatically disabled. Others are not. For a description of the constraints, see Chapter 5 “Maintaining Data Integrity” of the *Oracle7 Server Application Developer’s Guide*.

Enabled Constraints The constraints that remain in force are:

- not null
- unique
- primary keys

Not Null constraints are checked at insertion time. Any row that violates this constraint is rejected. *Unique* constraints are verified when indexes are rebuilt at the end of the load. The index will be left in direct load state if a violation is detected. (Direct load state is explained on page 8 – 9.) A *primary key* constraint is merely a unique–constraint on a not–null column.

Disabled Constraints The following constraints are disabled:

- check constraints
- referential constraints (foreign keys)

Reenable Constraints When the load completes, the integrity constraints will be re–enabled automatically if the REENABLE clause is specified. The syntax for this clause is as follows:



The optional keyword `DISABLED_CONSTRAINTS` is provided for readability. If the `EXCEPTIONS` clause is included, the table must already exist and, you must be able to insert into it. This table contains the ROWIDs of all rows that violated one of the integrity constraints. It also contains the name of the constraint that was violated. See the *Oracle7 Server SQL Reference* for instructions on how to create an exceptions table.

If the `REENABLE` clause is not used, then the constraints must be re–enabled manually. All rows in the table are verified then. If Oracle finds any errors in the new data, error messages are produced. The names of violated constraints and the ROWIDs of the bad data are placed in an exceptions table, if one is specified. See `ENABLE` in the *Oracle7 Server SQL Reference*.

The `SQL*Loader` log file describes the constraints that were disabled, the ones that were re–enabled and what error, if any, prevented

re-enabling of each constraint. It also contains the name of the exceptions table specified for each loaded table.



Warning: As long as bad data remains in the table, the integrity constraint cannot be successfully re-enabled.



Suggestion: Because referential integrity must be reverified for the entire table, performance may be improved by using the conventional path, instead of the direct path, when a small number of rows are to be loaded into a very large table.

Database Insert Triggers

Table insert triggers are also disabled when a direct path load begins. After the rows are loaded and indexes rebuilt, any triggers that were disabled are automatically re-enabled. The log file lists all triggers that were disabled for the load. There should not be any errors re-enabling triggers.

Unlike integrity constraints, insert triggers are not reapplied to the whole table when they are enabled. As a result, insert triggers do *not* fire for any rows loaded on the direct path. When using the direct path, the application must ensure that any behavior associated with insert triggers is carried out for the new rows.

Replacing Insert Triggers with Integrity Constraints

Applications commonly use insert triggers to implement integrity constraints. Most of these application insert triggers are simple enough that they can be replaced with Oracle's automatic integrity constraints.

When Automatic Constraints Cannot Be Used

Sometimes an insert trigger cannot be replaced with Oracle's automatic integrity constraints. For example, if an integrity check is implemented with a table lookup in an insert trigger, then automatic check constraints cannot be used, because the automatic constraints can only reference constants and columns in the current row. This section describes two methods for duplicating the effects of such a trigger.

Preparation

Before either method can be used, the table must be prepared. Use the following general guidelines to prepare the table:

1. Before the load, add a one-character column to the table that marks rows as “old data” or “new data”.
2. Let the value of null for this column signify “old data”, because null columns do not take up space.
3. When loading, flag all loaded rows as “new data” with SQL*Loader’s CONSTANT clause.

After following this procedure, all newly loaded rows are identified, making it possible to operate on the new data without affecting the old rows.

Using An Update Trigger

Generally, you can use a database update trigger to duplicate the effects of an insert trigger. This method is the simplest. It can be used whenever the insert trigger does not raise any exceptions.

1. Create an update trigger that duplicates the effects of the insert trigger.

Copy the trigger. Change all occurrences of “*new.column_name*” to “*old.column_name*”.

2. Replace the current update trigger, if it exists, with the new one
3. Update the table, changing the “new data” flag to null, thereby firing the update trigger
4. Restore the original update trigger, if there was one

Note: Depending on the behavior of the trigger, it may be necessary to have exclusive update access to the table during this operation, so that other users do not inadvertently apply the trigger to rows they modify.

Duplicating the Effects of Exception Conditions

If the insert trigger can raise an exception, then more work is required to duplicate its effects. Raising an exception would prevent the row from being inserted into the table. To duplicate that effect with an update trigger, it is necessary to mark the loaded row for deletion.

The “new data” column cannot be used for a delete flag, because an update trigger cannot modify the column(s) that caused it to fire. So another column must be added to the table. This column marks the row for deletion. A null value means the row is valid. Whenever the insert trigger would raise an exception, the update trigger can mark the row as invalid by setting a flag in the additional column.

Summary: When an insert trigger can raise an exception condition, its effects can be duplicated by an update trigger, provided:

- two columns (which are usually null) are added to the table
- the table can be updated exclusively (if necessary)

Using a Stored Procedure

The following procedure always works, but it is more complex to implement. It can be used when the insert trigger raises exceptions. It does not require a second additional column; and, because it does not replace the update trigger, and it can be used without exclusive access to the table.

1. Create a stored procedure that duplicates the effects of the insert trigger. Follow the general outline given below. (For implementation details, see the *PL/SQL User's Guide and Reference* for more information about cursor management.)
 - declare a cursor for the table, selecting all the new rows
 - open it and fetch rows, one at a time, in a processing loop
 - perform the operations contained in the insert trigger
 - if the operations succeed, change the “new data” flag to null
 - if the operations fail, change the “new data” flag to “bad data”
2. Execute the stored procedure using an administration tool such as Server Manager.
3. After running the procedure, check the table for any rows marked “bad data”.
4. Update or remove the bad rows.
5. Re-enable the insert trigger.

Permanently Disabled Triggers & Constraints

SQL*Loader needs to acquire several locks on the table to be loaded to disable triggers and constraints. If a competing process is enabling triggers or constraints at the same time that SQL*Loader is trying to disable them for that table, then SQL*Loader may not be able to acquire exclusive access to the table.

SQL*Loader attempts to handle this situation as gracefully as possible. It attempts to re-enable disabled triggers and constraints before exiting. However, the same table-locking problem that made it impossible for SQL*Loader to continue may also have made it impossible for SQL*Loader to finish enabling triggers and constraints. In such cases, triggers and constraints will remain permanently disabled until they are manually enabled.

Although such a situation is unlikely, it is possible. The best way to prevent it is to make sure that no applications are running that could enable triggers or constraints for the table, while the direct load is in progress.

If a direct load is aborted due to failure to acquire the proper locks, carefully check the log. It will show every trigger and constraint that was disabled, and each attempt to re-enable them. Any triggers or constraints that were not re-enabled by SQL*Loader should be manually enabled with the ENABLE clause described in the *Oracle7 Server SQL Reference*.

Alternative: Partitioned Load

If triggers or integrity constraints pose a problem, but you want faster loading, you should consider a *partitioned* load. A partitioned load works on a multiple-CPU system. Divide the data set into separate partitions, and then load each part through different CPUs with the conventional path. The resulting load is faster than a single-CPU conventional load, although possibly not as fast as a direct load. But triggers fire, and integrity constraints are applied to the loaded rows.

Parallel Data Loading

SQL*Loader now permits multiple, concurrent sessions to perform a direct path load into the same table. Multiple SQL*Loader sessions improve the performance of a direct path load given the available resources on your system.

Restrictions

To load a table in parallel, the table must not be indexed.

You can only use parallel load to append rows. REPLACE, TRUNCATE, and INSERT should not be used. If you must truncate a table before a parallel load, you must do it manually.

Initiating Multiple SQL*Loader Sessions

Each SQL*Loader session takes a different source file as input. In all sessions executing a direct load on the same table, you must set PARALLEL to TRUE. The syntax is:

```
▶▶ PARALLEL = [ FALSE ]
                [ TRUE ]
```

PARALLEL can be specified on the command line or in a parameter file. It can also be specified in the control file with the OPTIONS clause.

For example, to invoke three SQL*Loader direct path load sessions on the same table, you would execute the following commands at the operating system prompt:

```
SQLLOAD USERID=SCOTT/TIGER CONTROL=LOAD1.CTL DIRECT=TRUE PARALLEL=TRUE
SQLLOAD USERID=SCOTT/TIGER CONTROL=LOAD2.CTL DIRECT=TRUE PARALLEL=TRUE
SQLLOAD USERID=SCOTT/TIGER CONTROL=LOAD3.CTL DIRECT=TRUE PARALLEL=TRUE
```

The previous commands must be executed in separate sessions, or if permitted on your operating system, as separate background jobs. Note the use of multiple control files. This allows you to be flexible in specifying the files to use for the direct path load (see the example of one of the control files below).

Note: Indexes are not created during a parallel load. Any indexes must be created manually after the load completes. You can use the parallel index creation feature to speed the creation of large indexes after a parallel load.

When you perform a PARALLEL load, SQL*Loader creates temporary segments for each concurrent session and then merges the segments upon completion. The segment created from the merge is then added to the existing table in the database above the table's high water mark. The last extent used for each loader session is trimmed of any free space before being combined with the other extents of the SQL*Loader session.

Options Keyword

It is recommended that each concurrent session use files located on different disks to allow for the maximum I/O throughput. You can specify the filename of any valid datafile in the table's tablespace with the FILE keyword of the OPTIONS clause. The following example illustrates a portion of one of the control files used for the SQL*Loader sessions in the previous example:

```
LOAD DATA
INFILE 'load1.dat'
INSERT INTO TABLE emp
OPTIONS(FILE='/dat/data1.dat')
(empno POSITION(01:04) INTEGER EXTERNAL NULLIF empno=BLANKS
...
```

Note: The FILE keyword is specified in a table-specific OPTIONS clause. That is, the OPTIONS clause is placed *within* an INTO TABLE statement, rather than before.

Specifying Temporary Segments

You can specify the database file from which the temporary segments are allocated with the FILE keyword in the OPTIONS clause for each table in the control file. You can also specify the FILE parameter on the command line of each concurrent SQL*Loader session, but then it will globally apply to all tables being loaded with that session.

Enabling Constraints After A Parallel Direct Path Load

If REENABLE is set to TRUE, each SQL*Loader session attempts to re-enable constraints on a table after a direct path load. Each SQL*Loader session has a share lock on the table, however, so that another session cannot enable constraints before it is finished. When the last session completes, its attempt to re-enable constraints succeeds.



Warning: There is a danger that some constraints may not be re-enabled after a direct path load, you should check the status of the constraint after completing the load to ensure that it was enabled properly.

PRIMARY and UNIQUE KEY constraints

PRIMARY KEY and UNIQUE key constraints create indexes on a table when they are enabled, and subsequently can take a significantly long time to enable after a direct path loading session if the table is very large.

You should consider enabling these constraints manually after a load (and not specify the automatic enable feature). This allows you to manually create the required indexes in parallel to save time before enabling the constraint. See the *Oracle7 Server Administrator's Guide* for more information about creating indexes in parallel.

NLS Utilities

Part III explains how to use the NLS utilities:

- The NLS Data Installation utility which helps you convert text-format updates to NLS objects that you create or receive with a new Oracle distribution to binary format. It also aids you in merging these converted files into the existing NLS object set.
- The NLS Configuration utility which helps you configure your NLS boot files so that only the NLS objects that you require will be loaded.
- NLS Calendar utility which allows you to update existing NLS calendar data with additional ruler eras (imperial calendars) or add deviation days (lunar calendar).

National Language Support Utilities

This chapter describes three utilities:

- NLS Data Installation Utility
- NLS Configuration Utility
- NLS Calendar Utility



OSDoc

For platform-specific details on the use of these utilities, please see your operating system-specific Oracle documentation.

NLS Data Installation Utility

Overview

When you order an Oracle distribution set, a default set of NLS objects is included. In future you may receive updates to the data in the form of text files, or you may create your own text-format object files. These files must be converted into binary format and merged into the existing NLS object set. The NLS Data Installation Utility described here will allow you to do this.

Along with the binary object files, a boot file is generated by the NLS Data Installation Utility. This boot file is used by the modules to identify and locate all the NLS objects which it needs to load.

To facilitate boot file distribution and user configuration, three types of boot files are defined:

Installation Boot File	This is the boot file which is included as part of the distribution set.
System Boot File	This is the boot file which is generated by the NLS Data Installation Utility and which loads the NLS objects. If the user already has an installed installation boot file, its contents can be merged with the new system boot file during object generation.
User Boot File	This boot file only contains a subset of the System boot file. For a description on how this file is generated, please refer to the “NLS Configuration Utility” section on page 9 – 6.

Syntax

The NLS Data Installation Utility is invoked from the command line with the following syntax:

LXINST [ORANLS=*pathname*] [SYSDIR=*pathname*] [DESTDIR=*pathname*]
[HELP=[yes | no]]

where

ORANLS= <i>pathname</i>	Specifies where to find the text-format boot and object files and where to store the new binary-format boot and object files. If not specified, the value in the environment variable ORA_NLS (or the equivalent on your operating system) will be used. If both are specified, the command line parameter will override the environment variable. If neither is specified, the NLS Data Installation Utility will exit with an error message.
----------------------------	--

SYSDIR= <i>pathname</i>	Specifies where to find the existing system boot file. If not specified, the directory specified in the ORANLS parameter will be used. If no existing system boot file can be found (possibly a first-time installation), the installation boot file which has just been created will be moved or copied to the new system boot file without any changes.
DESTDIR= <i>pathname</i>	Specifies where to put the new (merged) system boot file. If not specified, the directory specified in the ORANLS parameter will be used. Any system boot file already existing in this directory will be overwritten.
HELP=[yes no]	If “yes”, a help message describing the syntax is printed.

Return Codes

You may receive the following return codes upon executing LXINST:

0	Generation of binary boot and object files and merge of the installation and system boot files successfully completed.
1	Installation has failed, reason of the failure has been indicated by means of an error message.

Note: The actual file names of the boot and object files are strictly imposed by Oracle Server release 7.2. Thus you can only specify the directory path name(s) where boot and object files are read from and written to.

Usage

The first step is to create or obtain your new text-format data object files. The distribution includes several example data files that you can use for reference. The next step is to create a text-format system boot file containing references to your new data objects; only those objects referenced in the boot file will be generated. An example system boot file is also included in the distribution. The names to be used for the text-format files are listed in the next section.

Once you have your new text-format files, call LXINST to generate the new binary_ object files. This will generate, in addition to the data object files, both an installation boot file and a system boot file. If you already have an NLS installation, you can set SYSDIR to point to the directory that contains your current data files, and the entries in the system Boot File there will merged into the new system boot file that is generated.

The binary object files will be generated in the directory specified by DESTDIR. If this is not your normal data directory, you must then move the new binary files into that directory. Be careful if DESTDIR is set to your normal data directory, as the current system boot file will be overwritten.

NLS Data Object Files

Read this section if you will be creating your own NLS data objects. It details the formats, contents, and restrictions expected by the NLS Data Installation Utility.

Text-Format Files

NLS data object files are specified in a system-independent, portable text format. In order to accommodate ASCII to EBCDIC conversions, only the following characters are allowed in the data file:

```
!"#$%&()*+,-./0123456789:;<=>?@
ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz {|}~
<space> character
```

All other characters, including 8-bit ASCII single-byte characters, multi-byte characters, and control/shift characters, must be specified as octal or hexadecimal escape sequences as defined in the ANSI C standard 6000 for octal numbers, \xhh for hexadecimal numbers). The backslash (\) and double quote (") may be escaped by \\ and \" respectively where necessary.

Some of the data in the object files will be tagged with a specific character set (for example, day/month names); this data should not be subject to ASCII/EBCDIC conversions when transported across hardware platforms (otherwise, a string such as "abc\123" will have its first three characters converted to EBCDIC but the last character will remain as "\123"). In order to ensure this, all of the characters in such strings should be specified in octal format when it is known that a data object file will be moved between platforms. If the file will be generated for only one specific platform, then this restriction is not necessary.

Note that the NLS Data Installation Utility does not convert any characters into octal or hexadecimal format. It is assumed that any characters outside of those listed above will already be in octal or hexadecimal format, and that any backslashes and double quotes are already properly escaped.

It is assumed that the character set used for the data object files will be US7ASCII for ASCII platforms or WE8EBCDIC500 for EBCDIC platforms. This is allowed because NLS data must always include US7ASCII or WE8EBCDIC500.

Object Types The following object types are currently supported:

Type	Name
0	Language
1	Territory
2	Character set
3	Linguistic definition

Object IDs NLS data objects are uniquely identified by a numeric object ID. The ID may never have a zero or negative value.

In general, you can define new objects as long as you specify the object ID within the range 9000–9999.



Warning: Character set IDs are an exception. When you want to create a new character set, you must report its ID to the Oracle National Language Support Group at Oracle Corporation in order to ensure that the ID is unique; otherwise, its functioning cannot be guaranteed.

Object Names Only a very restricted set of characters can be used in object names:

ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_– and <space>

Object names can only start with an alphabetic character. Language, territory and character set names cannot contain an underscore character, but linguistic definition names can. There is no case distinction in object names, and the maximum size of an object name is 30 bytes (excluding terminating null).

Data Object File Names The system-independent object file name is constructed from the generic boot file entry information:

pptddd

where

<i>pp</i>	NW for the system-independent text files or LX for the system-dependent binary files (files to be loaded by LX)
<i>t</i>	1 digit object type (hex)
<i>ddd</i>	4 digit object ID(hex)

The installation boot file name is *pp0BOOT*; the system boot file name is *pp1BOOT*; user boot files are named *pp2BOOT*, where *pp* is as stated above. The file extension for all files is *D*.

Examples:

Filename	Description
NW00001.D	Text-format language definition, ID = 1
NW203C2.D	Text-format character set definition, ID = 962
NW1BOOT.D	Text-format system boot file
1x00001.D	Binary language definition, ID = 1
1x30032.D	Binary linguistic definition, ID = 50
1x2BOOT.D	Binary user boot file

NLS Configuration Utility

Overview

At installation, all available NLS objects are stored and referenced in the system boot file. This file is used to load the available NLS dam.

The NLS Configuration Utility allows you to configure your boot files such that only the NLS objects that you require will be loaded. It does this by creating a user boot file, which contains a subset of the system boot file. Data loading by the kernel will then be performed according to the contents of this user boot file.

The NLS Configuration Utility allows you to configure a user boot file, either by selecting NLS objects from the installed system boot file which will then be included in a new user boot file, or by reading entries from an existing user boot file and possibly removing one or more of them and saving the remaining entries into a new user boot file. Note that you will not be allowed to actually “edit” an existing boot file as it may be in use by either the RDBMS or some other Oracle tool (that is, saving of boot file entries is never done to an existing one).

You may also use the NLS Data Installation Utility to check the integrity of an existing user boot file. This is necessary since the contents of existing NLS objects may change over time, and the installation of a new system boot file may cause user boot files to become out of date. Thus, a comparison function will notify you when it finds that the file is out of date and will allow you to create a new user boot file.

Syntax

The NLS Configuration Utility is invoked from the command line with the following syntax:

```
LXBCNF [ORANLS=pathname] [userbootdir=pathname]  
[DESTDIR=pathname] [HELP=[yes | no]]
```

where:

ORANLS= <i>pathname</i>	Specifies where to find the system boot file. If not specified, the value in the environment variable ORA_NLS (or the equivalent on your operating system) will be used. If both are specified, the command name parameter will override the environment variable. If neither is specified, the NLS Configuration Utility will exit with an error message.
SYSDIR= <i>pathname</i>	Specifies where to find an existing user boot file. If not specified, the directory specified in the ORANLS parameter will be used.
DESTDIR= <i>pathname</i>	Specifies where to put the new user boot file. If not specified, the directory specified in the ORANLS parameter will be used.
HELP=[yes no]	If “yes”, a help message describing the syntax is printed.

Menus

When the NLS Configuration Utility is started you are presented with the following top-level menu:

- File Menu
- Edit Menu
- Action Menu
- Windows Menu
- Help

File Menu

The file menu contains choices pertaining to file operations. Options are:

Menu Item	Options	Description
System Boot File	Open	This will open the current system boot file. Note that the Open menu item will be “grayed out” as soon as a system Boot File has been successfully read. Also note that you cannot perform any other functions until you have opened a system boot file.
User Boot File	New	Open a new user boot file.
	Read	Read the contents of an existing user boot file.
	Save	Save changes to the new user boot file.
	Revert	Undo the changes to the currently open user boot file made since the last “Save.”
Choose Printer		Not implemented in this release.
Page Setup		Not implemented in this release.
Print		Not implemented in this release.
Quit		Exit from the file.

Table 9 – 1 File Menu Options

Note: As long as the system boot file has not been opened and read, all these menu items will remain “grayed out”. That is, you cannot build a user boot file as long as there is no system boot file information available.

As soon as you select New to create a new user boot file, the following NLS objects will be created in the new file by default:

LANGUAGE AMERICAN
TERRITORY AMERICA
CHARACTER US7ASCII on an ASCII platform or WE8EBCDIC500
SET on an EBCDIC platform.

If you choose to read the contents of an existing user boot file, the entries read will be checked against the entries of the system boot file. If an entry is found which does not exist in the system boot file, you will receive a warning, and the entry will not be included.

Edit Menu

The Edit Menu contains choices for editing information that you enter in any of the dialogs and/or windows of the NLS Configuration Utility.

Action Menu	The Action Menu contains choices for performing operations on the user boot file. Note that this menu is available only in the character mode NLS Configuration Utility.	
	Copy Item	Copies the selected item from the system boot file to the user boot file.
	Delete Item	Deletes the selected item from the user boot file.
Windows Menu	The Windows Menu allows you to either activate certain windows or set the focus to an already open window (the latter is meant for character-mode platforms). Whenever a new window is opened, its name will be added to the Windows Menu automatically.	
	NLS Defaults	Not implemented in this release.
Help Menu	This menu provides functions which allow the user to retrieve various levels of help about the NLS Configuration Utility.	
	About	Shows version information of the NLS Configuration Utility.
	Help System	Not implemented in this release.

NLS Calendar Utility

Overview

A number of calendars besides Gregorian are supported. Although all of them are defined with data linked directly into NLS, some of them may require the addition of ruler eras (in the case of imperial calendars) or deviation days (in the case of lunar calendars) in the future. In order to do this without waiting for a new release, you can define the additional eras or deviation days in an external file, which is then automatically loaded when executing the calendar functions.

The calendar data is first defined in a text-format definition file. This file must be converted into binary format before it can be used. The Calendar Utility described here allows you to do this.

Syntax

The Calendar Utility is invoked directly from the command line:

```
LXEGEN
```

There are no parameters.

Usage

The Calendar Utility takes as input a text-format definition file. The name of the file and its location is hard-coded as a platform-dependent value. On UNIX platforms, the file name is `lxecal.dat`, and its location is `$ORACLE_HOME/ocommon/nls`. An example calendar definition file is included in the distribution.

The `LXEGEN` executable produces as output a binary file containing the calendar data in the appropriate format. The name of the output file is also hard-coded as a platform-dependent value; on UNIX the name is `lxecalhl.o`. The file will be generated in the same directory as the text-format file, and an already-existing file will be overwritten.

Once the binary file has been generated, it will automatically be loaded the next time an LX calendar function is called. Do not move or rename the file, as it is expected to be found in the same hard-coded name and location.

PART

IV

Offline Verification Utility

Offline Database Verification Utility

This chapter describes how to use the offline database verification utility, DB_VERIFY. Topics discussed include:

- Functionality
- Restrictions
- Usage
 - Command-Line Syntax
 - Parameters
- Server Manager Compatibility

DB_VERIFY

DB_VERIFY is an external command-line utility that performs a physical data structure integrity check on an offline database. It can be used against backup files and online files (or pieces of files). You use DB_VERIFY primarily when you need to insure that a backup database (or datafile) is valid before it is restored or as a diagnostic aid when you have encountered data corruption problems.

Because DB_VERIFY can be run against an offline database, integrity checks are significantly faster.



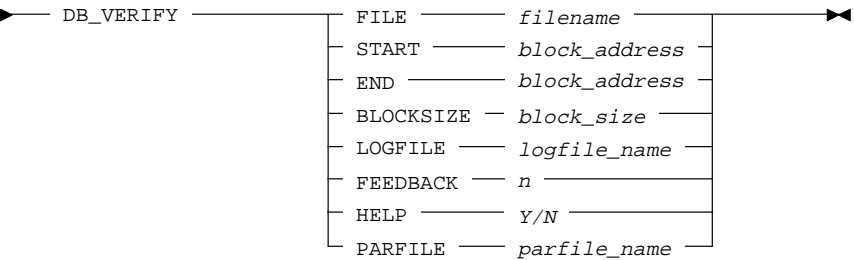
OSDoc

Additional Information: The name and location of DB_VERIFY is dependent on your operating system (for example, **dbv** on Sun/Sequent systems). See your operating system-specific Oracle documentation for the location of DB_VERIFY for your system.

Restrictions

DB_VERIFY checks are limited to cache managed blocks.

Syntax



Parameters

FILE	The name of the database file to verify
START	The starting block address to verify. Block addresses are specified in Oracle blocks (as opposed to operating system blocks). If START is not specified, assumes first block in the file.
END	The ending block address to verify. If END is not specified, assumes last block in the file.
BLOCKSIZE	BLOCKSIZE is only required if the file has a non 2Kb block size. In these cases the block size must be explicitly specified or the error DBV-00103 will be returned.

LOGFILE	Specifies the file to which logging information should be written. The default is to send output to the terminal display. If errors are reported, you should contact Oracle Technical Support.
FEEDBACK	Specifying FEEDBACK causes DB_VERIFY to display a single '.' for <i>n</i> pages verified. If <i>n</i> is 0, then feedback is disabled.
HELP	Provides onscreen help.
PARFILE	Specifies the name of the parameter file to use. DB_VERIFY parameters can be stored in a flat file in different combinations and with differing values. This way, you can have parameter files for specific types of integrity checks or for different datafiles.

Server Manager

Server Manager can be used to perform the verification process as well. The verification of the entire database or a tablespace will be managed by server manager in that it will invoke the verification process on each individual file.

For more information, see the *Oracle Server Manager User's Guide*

Sample DB_VERIFY Output

The following example shows how to get online help:

```
% dbv help=y
```

```
DBVERIFY: Release 7.3.1.0.0 - Wed Aug  2 09:14:36 1995
```

```
Copyright (c) Oracle Corporation 1979, 1994. All rights reserved.
```

Keyword	Description	(Default)
FILE	File to Verify	(NONE)
START	Start Block	(First Block of File)
END	End Block	(Last Block of File)
BLOCKSIZE	Logical Block Size	(2048)
LOGFILE	Output Log	(NONE)

This is sample output of verification for the file, t_db1.f. The feedback parameter has been given the value 100 to display one dot onscreen for every 100 pages processed:

```
% dbv file=t_db1.f feedback=100
```

```
DBVERIFY: Release 7.3.1.0.0 - Wed Aug  2 09:15:04 1995
```


Copyright (c) Oracle Corporation 1979, 1994. All rights reserved.

DBVERIFY - Verification starting : FILE = t_dbl.f

.....
.....

DBVERIFY - Verification complete

Total Pages Examined	: 9216
Total Pages Processed (Data)	: 2044
Total Pages Failing (Data)	: 0
Total Pages Processed (Index)	: 733
Total Pages Failing (Index)	: 0
Total Pages Empty	: 5686
Total Pages Marked Corrupt	: 0
Total Pages Influx	: 0

A

Summary of Changes

This appendix describes changes in the Oracle7 Server utility programs. It covers:

- Import/Export Changes
- SQL*Loader Changes

Export/Import Changes

This section describes changes to the Oracle7 Import/Export utility.

Release 7.2 Changes

These are the changes to Export in Release 7.2.

FEEDBACK parameter added to Import/Export to display an onscreen progress meter during imports and exports.

Release 7.1 Changes

These are the changes to Export in Release 7.1.

Change in Export File Format

In Release 7.1, the export file format was changed to accommodate stored procedures, functions, and packages that have comments embedded among the creation–statement keywords. In some cases, a patch is required to successfully migrate these code objects between a Release 7.0 database and a Release 7.1 database.

Read-Only Tablespaces

Read-only tablespaces can be exported.

Trusted Oracle Label Mapping

Trusted Oracle7 Import allows you to list the labels in the export file and specify a mapping that converts them into the labels used by the target Trusted Oracle7 database. For details, see the *Trusted Oracle7 Server Administrator's Guide*.

Version Incompatibility Error Message

When a higher level version of the Export utility is used with a lower level version of the Oracle7 Server with which it is incompatible, an EXP-37 error results.

New 7.1 Objects Supported

Release 7.1 of the Export/Import utilities supports these new Oracle7 objects:

- job queues
- refresh groups
- resource costs

Changes in Export for Release 7.0

The new features of Oracle7 export utility are described below.

Oracle7 Objects

New objects in Oracle7 (roles, profiles, triggers, stored procedures, and snapshots) can be exported. See page 1 – 6.

Log Files

Log files are supported in Oracle7. Systems that do not support I/O redirection can have a record of errors and warnings that occur during the export. See page 1 – 15.

Final Message	<p>In Version 6, any export that completed displayed the message “Export completed successfully”. In Oracle7, export ends with one of three messages:</p> <pre>Export terminated successfully without warnings Export terminated successfully with warnings Export terminated unsuccessfully</pre> <p>See page 1 – 5 for more information.</p>
CONSISTENT	The Oracle7 parameter CONSISTENT allows the creation of read-consistent exports. A read-consistent export is guaranteed not to include any partially completed transactions. It is used when exports and database updates must occur simultaneously. See page 1 – 13.
GRANTS=Y	The default for GRANTS export parameter is Y. In Version 6, it was N. See page 1 – 15.
CONSTRAINTS=Y	The default for the CONSTRAINTS export parameter is Y. In Version 6, it was N. See page 1 – 14.
Setting Up Export Views	In Oracle7, the script EXPVIEW.SQL must be run after CATALOG.SQL to establish the export views. In Version 6, the order made no difference. See page 1 – 2.
Trusted Oracle7 Server Parameters	The Trusted Oracle7 Server parameters MLS and MLS_LABEL_FORMAT have been added. See page 1 – 16.
Changes in Import for Release 7.0	The new features of Oracle7 Import are described below.
Oracle7 Objects	New objects in Oracle7 (roles, profiles, triggers, stored procedures, and snapshots) can be imported. See pages 2 – 9 to 2 – 10.
DESTROY Parameter	<p>The ALTER TABLESPACE statement stored in the Oracle7 export file sets REUSE=NO so that the default action on import is <i>not</i> to reuse the datafiles defined for the database. In Version 6, the default was REUSE=YES. This change minimizes the chance that a database can be inadvertently replaced when attempting to create a copy of it for testing or other purposes. This change resulted from the fact that the full pathname of datafiles in the database is included in the export file.</p> <p>A new import parameter allows you to specify DESTROY=YES, instructing Import to erase the existing datafiles and reuse them. This option allows you to re-create your database from backups, when necessary. The default value for DESTROY is NO. See page 2 – 19.</p>

Log Files	Log files are supported in Oracle7 so that systems that do not support I/O redirection can have a record of errors and warnings that occur during the import. See page 2 – 23.
Final Message	<p>In Version 6, any import that completed displayed the message “Import completed successfully”. In Oracle7, import ends with one of three messages:</p> <pre> Import terminated successfully without warnings Import terminated successfully with warnings Import terminated unsuccessfully </pre> <p>See page 2 – 5 for more information.</p>
User Definitions	In Version 6, user definitions did not exist separately from their GRANTS. User definitions were therefore imported by importing GRANT statements. In Oracle7, importing a user definition does not automatically grant connect access.
Importing Objects for Other Users	<p>Version 6 import adopted the target user’s privileges when importing objects. If the user did not have CREATE TABLE privilege, for example, a DBA might find that a user’s tables could not be imported—although the DBA had successfully exported them.</p> <p>Oracle Import uses the importer’s privileges to import objects into another user’s schema. This method preserves the importer’s privileges, allowing a fully privileged user to import objects for a limited–privilege user.</p>
IGNORE=N	In Oracle7, the default is to report all object creation errors. Previously they were ignored by default. To suppress the reporting of error messages that occur when tables and other objects already exist, specify IGNORE=Y. See page 2 – 21.
INDEXFILE Parameter	The Oracle7 parameter INDEXFILE makes it possible to siphon off index–creation statements, so they are placed in a file instead of being applied to the database. This procedure is more efficient and makes it possible to change index storage attributes. See page 2 – 22.
Character Set Conversion	In Version 6, Import/Export provided a limited capability to convert data between ASCII–based and EBCDIC–based systems. In Oracle7, that capacity has been extended to translate data between two systems using different character encoding schemes. See page 2 – 12.

CHARSET parameter	<p>In Version 6, the CHARSET parameter was used in limited ways to control the conversion from the character set used in the export file to the database character set. In Oracle7, the Export file records the character set used when the file was created. Oracle Import then automatically converts that character set to the database character set.</p> <p>In Oracle7, this parameter only checks that the Export file's character set matches the expected value. Use of the CHARSET parameter is no longer recommended as it will eventually become obsolete.</p>
Trusted Oracle7 Server Parameter	<p>The Trusted Oracle7 Server parameter MLS has been added. For details. See page 1 – 16 and page 2 – 23.</p>
Incremental Import and Export	<p>In Oracle7, an incremental import or export can be performed by any user who has been granted the EXP_FULL_DATABASE role. Previously, only SYS or SYSTEM could do an incremental import or export. See page 1 – 24.</p> <p>Similarly, any user who has been granted the IMP_FULL_DATABASE role can perform an incremental import. See page 2 – 29.</p>

SQL*Loader Changes

This section describes the changes to SQL*Loader and the releases in which they became effective.

Oracle7 Server Release 7.1

Changes for Release 7.1

- Data can be loaded in parallel with the PARALLEL direct path load. See page 8 – 21.
- Direct loads can load data into synonyms for tables (but not synonyms for views, or synonyms for synonyms). See page 8 – 7.
- Direct loads can specify UNRECOVERABLE to improve performance by disabling redo logging. See page 8 – 15.
- In Release 7.1, direct loads can specify a datafile character set which is different from the control file using the CHARACTERSET clause. In 7.0 this was possible for a conventional load, but not for a direct load. See page 5 – 24.

As before, the control file must be in the same character set as the session so that the session can interpret it. And, as before, the database character set can be different — data conversion happens automatically.

- The REPLACE option returns to using DELETE TABLE semantics. That was the standard behavior before Release 7.0. Release 7.1 returns to that standard, and no longer performs a table TRUNCATE when REPLACE is specified. With this change:
 - Integrity constraints on the table do not have to be disabled before the load.
 - Delete triggers that are defined on the table fire as rows are deleted.
 - DELETE privilege on the table is sufficient if the table is not in your own schema. See page 5 – 26 for more information.
- A new option, TRUNCATE, uses fast table truncation for a load. This option does not fire delete triggers for the truncated rows. Integrity constraints on the table must be disabled before loading with TRUNCATE. Finally, if the table is not in your own schema, the DELETE ANY TABLE privilege is required. See page 5 – 25.
- Previously, only one direct path load could be performed on any given table. With Oracle Server release 7.1, you can optionally specify a PARALLEL clause within the SQL*Loader control file or on the command line to indicate that multiple SQL*Loader direct path load sessions can share access to the table. See page 8 – 21.
- The FILE parameter allows different parts of a parallel load to allocate extents from different database files.

Oracle7

In Oracle7, SQL*Loader is part of the standard Oracle Server release. As a result, its release number jumps from “1.1” to “7.0”. New features include:

- Direct loads bypass referential integrity constraints and database triggers. See page 8 – 16.
- Input character conversion possible with the CHARACTERSET clause. See page 5 – 24.
- Multi-byte (Asian language) characters supported. See page 5 – 24.
- Fast table truncation used for REPLACE option. See page 5 – 25.

Release 1.1

Changes for Direct Path Loads:

- Maximum performance available with the direct path option. See Chapter 8.
- Operating system sorts can be used for high-speed sorting, fast indexing, and reduced need for temporary storage. See page 8 – 13.
- DIRECT command-line parameter added. See page 6 – 4.
- Meaning of ROWS parameter extended for direct loads. See page 8 – 9.
- CONTINUE_LOAD and table-level SKIP clauses added to continue a multiple-table direct path load after an interruption. See page 5 – 28.

Changes in File Management:

- The file processing string allows optimizing datafile reads. See page 5 – 18.
- The keywords RECLEN, STREAM, RECORD, FIXED, BLOCKSIZE, and VARIABLE are now obsolete. They are still recognized for upward compatibility, but they have no effect.
- Single quotation marks recommended for filenames. See page 5 – 13.
- When loading records that contain only generated data, SQL*Loader skips reading of input file. See page 5 – 45.
- The bad file is only created if it is needed. See page 5 – 19.

Changes in Data Management:

- SQL string allows use of SQL operators on fields. See page 6–73.
- RAW datatype added. See page 5 – 52.
- Insert current date/time with SYSDATE keyword. See page 5 – 46.
- Initial and trailing field delimiters can be different. See page 5 – 58.
- Mismatches in field length specifications generate warning messages in native datatype fields and character datatype fields. See page 5 – 55 and page 5 – 61.
- Default maximum field size for delimited fields is 255 bytes instead of 240 bytes. See page 5 – 58.

Changes in the Command Line:

- Command-line specifications override control file options. See page 5 – 12 and page 6 – 3.
- Bad file command-line specification overrides control-file specification. See page 5 – 19 and page 6 – 3.
- Discard file command-line specification overrides control-file specification. See page 5 – 21 and page 6 – 4.
- Arguments can be specified in a separate file with PARFILE command-line parameter. See page 6 – 5.

Obsolete Parameters:

- Because it has been replaced by the file processing options string, the RECLEN parameter is obsolete.

In addition, the old syntax

```
[ STREAM | RECORD | FIXED len [BLOCKSIZE size] | VARIABLE [len] ]
```

does not affect the way SQL*Loader reads the datafile, although the syntax is recognized to maintain upward compatibility.

Release 1.0.26

Changes in Release 1.0.26:

- Load character fields with all whitespace intact, using the PRESERVE BLANKS option. See page 5 – 74.
- Test to see if a field of undetermined length is all blank with the BLANKS keyword. See page 5 – 38.
- All-blank numeric fields no longer automatically loaded as NULL. See page 5 – 69.

Release 1.0.22

Changes in Release 1.0.22:

- ZONED DECIMAL datatype added. See page 5 – 51.
- Specifying how to handle missing fields at the end of a record with the TRAILING NULLCOLS keyword. See page 5 – 68.
- Putting special characters into quoted strings with the quoted-string escape character. See page 5 – 14.

B

Reserved Words

This appendix lists the words reserved by the Oracle utilities and explains how to avoid problems that can arise when you use them as names for database objects.

It is generally a good idea to avoid naming your tables or columns using terms that are reserved by any of the languages or utilities you are likely to use at your installation. Refer to the various language and reference manuals (PL/SQL, SQL*Plus, etc.) and to this appendix for lists of reserved words.

Consult the *Oracle7 Server SQL Reference* for a list of words that are reserved by SQL. Tables or columns that have these names must also be specified in double quotation marks.

SQL*Loader Reserved Words

When using SQL*Loader, you must follow the usual rules for naming tables and columns. A table or column name cannot be a *reserved word*, a word having special meaning for SQL*Loader. The following words must be enclosed in double quotation marks if they are table or column names:

AND	FLOAT	RECNUM
APPEND	FORMAT	RECORD
BADFILE	GENERATED	REPLACE
BDDN	GRAPHIC	RESUME
BEGINDATA	INDDN	SEQUENCE
BLANKS	INDEXES	SKIP
BLOCKSIZE	INFILE	SMALLINT
BY	INSERT	SORTDEVT
CHAR	INTEGER	SORTED
CONCATENATE	INTO	SORTNUM
CONSTANT	LAST	SQL/DS
CONTINUE_LOAD	LOAD	STREAM
CONTINUEIF	LOG	SYSDATE
COUNT	MAX	TABLE
DATA	NEXT	TERMINATED
DATE	NO	THIS
DECIMAL	NULLCOLS	TRAILING
DEFAULTIF	NULLIF	UNLOAD
DISCARDN	OPTIONALLY	VARCHAR
DISCARDFILE	OPTIONS	VARGRAPHIC
DISCARDMAX	PARALLEL	VARIABLE
DISCARDS	PART	WHEN
DOUBLE	PIECED	WHITESPACE
ENCLOSED	POSITION	WORKDDM
EXTERNAL	PRESERVE	YES
FIELDS	RAW	ZONED
FIXED	RECLN	

C

Notes for DB2/DXT Users

This appendix describes differences between SQL*Loader DDL syntax and DB2 Load Utility/DXT control file syntax. The topics discussed include:

- SQL*Loader Extensions to the DB2 Load Utility
- DB2 RESUME Option
- Options Included for Compatibility
- SQL*Loader Restrictions
- DB2/DXT Compatible Syntax

SQL*Loader Extensions to the DB2 Load Utility

SQL*Loader can use any DB2 Load Utility control file. SQL*Loader also offers numerous extensions to the DB2 loader by supporting the following features:

- The DATE datatype
- The automatic generation of unique sequential keys
- The ability to specify the record length explicitly
- Loading data from multiple data files of different file types
- Fixed-format, delimited-format, and variable-length records
- The ability to treat a single physical record as multiple logical records
- The ability to combine multiple physical records into one logical record via CONCATENATE, CONTINUEIF NEXT, and CONTINUEIF THIS (IBM supports only CONTINUEIF THIS)
- More thorough error reporting
- Bad file (DB2 stops on first error)
- Control over the number of records to skip, the number to load, and the number of errors to allow
- ANDed WHEN clause
- FIELDS clause for default field characteristics
- Direct path loads
- Parallel loads

Using the DB2 RESUME Option

You can use the DB2 syntax for RESUME, but you may prefer to use SQL*Loader’s equivalent keywords. See “Loading into Non-Empty Database Tables” on page 5 – 25 for more details about the SQL*Loader options summarized below.

DB2	SQL*Loader Options	Result
RESUME NO or no RESUME clause	INSERT	Data loaded only if table is empty. Otherwise an error is returned.
RESUME YES	APPEND	New data is appended to existing data in the table, if any.
RESUME NO REPLACE	REPLACE	New data replaces existing table data, if any.

Table C – 1 DB2 Functions and Equivalent SQL*Loader Operations

A description of the DB2 syntax follows.

If the tables you are loading already contain data, you have three choices for the disposition of that data. Indicate your choice using the RESUME clause. The argument to RESUME can be enclosed in parentheses.

```
RESUME { YES | NO [ REPLACE ] }
```

where:

- YES Appends the new rows to rows already in the table.
- NO Requires the table to be empty before loading. An error message results if the table contains rows and the run is terminated. This is the default.
- NO REPLACE Deletes any data in the table before loading new data. Thus, the new data will replace the old. This argument requires that the username invoking SQL*Loader have DELETE privilege on the table. You cannot recover the data that was in the table before the load, unless you saved it using Export or something comparable.

In SQL*Loader you can use one RESUME clause to apply to all loaded tables by placing the RESUME clause before any INTO TABLE clauses. Alternatively, you can specify your RESUME options on a table-by-table basis by putting a RESUME clause after the INTO TABLE specification. The RESUME option following a table name will override one placed earlier in the file. The earlier RESUME applies to all tables that do not have their own RESUME clause.

Inclusions for Compatibility

The IBM DB2 Load Utility contains certain elements that SQL*Loader does not use. In DB2, sorted indexes are created using external files, and specifications for these external files may be included in the load statement. For compatibility with the DB2 loader, SQL*Loader parses these options, but ignores them if they have no meaning for Oracle. The syntactical elements described below are allowed, but ignored, by SQL*Loader.

LOG Statement

This statement is included for compatibility with DB2. It is parsed but ignored by SQL*Loader. (This LOG option has nothing to do with the log file that SQL*Loader writes.) DB2 uses the log file for error recovery, and it may or may not be written.

SQL*Loader relies on Oracle's automatic logging, which may or may not be enabled as a warm start option.

```
[ LOG { YES | NO } ]
```

WORKDDN Statement This statement is included for compatibility with DB2. It is parsed but ignored by SQL*Loader. In DB2, this statement specifies a temporary file for sorting.

```
[ WORKDDN filename ]
```

SORTDEVT and SORTNUM Statements SORTDEVT and SORTNUM are included for compatibility with DB2. These statements are parsed but ignored by SQL*Loader. In DB2, these statements specify the number and type of temporary data sets for sorting.

```
[ SORTDEVT device_type ]  
[ SORTNUM n ]
```

DISCARD Specification

Multiple file handling requires that the DISCARD clauses (DISCARD DDN and DISCARDS) be in a different place in the control file — next to the datafile specification. However, when loading a single DB2 compatible file, these clauses can be in their old position — between the RESUME and RECLLEN clauses. Note that while DB2 Load Utility DISCARDS option zero (0) means no maximum number of discards, for SQL*Loader, option zero means to stop on the first discard.

Restrictions

Some aspects of the DB2 loader are not duplicated by SQL*Loader. For example, SQL*Loader does not load data from SQL/DS files nor from DB2 UNLOAD files. SQL*Loader gives an error upon encountering the DB2 Load Utility commands described below.

FORMAT Statement

The DB2 FORMAT statement must not be present in a control file to be processed by SQL*Loader. The DB2 loader will load DB2 UNLOAD format, SQL/DS format, and DB2 Load Utility format files. SQL*Loader does not support these formats. If this option is present in the command file, SQL*Loader will stop with an error. (IBM does not document the format of these files, so SQL*Loader cannot read them.)

```
FORMAT { UNLOAD | SQL/DS }
```

PART Statement

The PART statement is included for compatibility with DB2. There is no Oracle concept that corresponds to a DB2 partitioned table.

In SQL*Loader, the entire table is read. A warning indicates that partitioned tables are not supported, and that the entire table has been loaded.

```
[ PART n ]
```

SQL/DS Option

The option SQL/DS=*tablename* must not be used in the WHEN clause. SQL*Loader does not support the SQL/DS internal format. So if the SQL/DS option appears in this statement, SQL*Loader will terminate with an error.

DBCS Graphic Strings

Because Oracle does not support the double-byte character set (DBCS), graphic strings of the form G'***' are not permitted.

SQL*Loader Syntax with DB2-compatible Statements

In the following listing, DB2-compatible statements are in bold type:

```
OPTIONS (options)
{ LOAD | CONTINUE_LOAD } [DATA]
[ CHARACTERSET character_set_name ]
[ { { INFILE | INDDN } { filename | * }
    [ "OS-dependent file processing options string" ]
    [ { BADFILE | BADDN } filename ]
    [ { DISCARDFILE | DISCARDN } filename ]
    [ { DISCARDS | DISCARDMAX } n ] ]
[ { INFILE | INDDN } ] ...
[ APPEND | REPLACE | INSERT |
    RESUME [( { YES | NO [REPLACE] } [ ] ) ]
[ LOG { YES | NO } ]
[ WORKDDN filename ]
[ SORTDEVT device_type ]
[ SORTNUM n ]
[ { CONCATENATE [( n [ ] ) |
    CONTINUEIF { [ THIS | NEXT ]
                [( start [ { : | - } end ] ) | LAST ]
                operator { 'char_str' | X'hex_str' } [ ] } } ]
[ PRESERVE BLANKS ]
INTO TABLE tablename
    [ CHARACTERSET character_set_name ]
    [ SORTED [ INDEXES ] ( index_name [ ,index_name... ] ) ]
    [ PART n ]
    [ APPEND | REPLACE | INSERT |
        RESUME [( { YES | NO [REPLACE] } [ ] ) ]
    [ REENABLE [DISABLED_CONSTRAINTS] [EXCEPTIONS table_name] ]
    [ WHEN field_condition [ AND field_condition ... ] ]
    [ FIELDS [ delimiter_spec ] ]
    [ TRAILING [ NULLCOLS ] ]
    [ SKIP n ]
    (column_name
    { [ RECNUM
        | SYSDATE
        | CONSTANT value
        | SEQUENCE ( { n | MAX | COUNT } [ , increment ] )
        | [( POSITION ( { start [ { : | - } end ] | * [n] ) ) ]
          [ datatype_spec ]
          [ NULLIF field_condition ]
          [ DEFAULTIF field_condition ]
          [ "sql string" ] ] ] }
    [ , column_name ] ...)
[ INTO TABLE ] ... [ BEGINDATA ]
[ BEGINDATA]
```

Index

A

- access privileges, Export, 1 – 3
- APPEND keyword, SQL*Loader, 5 – 25, 5 – 36
- APPEND to table
 - example, 4 – 8
 - SQL*Loader, 5 – 26
- arrays, committing after insert, Import, 2 – 19
- ASCII
 - 7-bit format file conversion, 1 – 3
 - fixed-format files, reading, Export, 1 – 2
- ASCII to EBCDIC conversion, Import, 1 – 8, 2 – 3
- audit options, Oracle Version 6, 1 – 41

B

- backslash (\)
 - escape character in quoted strings, SQL*Loader, 5 – 14
 - quoted filenames and, SQL*Loader, 5 – 14
- backups, restoring dropped snapshots, Import, 2 – 11
- BAD, SQL*Loader command-line parameter, 6 – 3
- bad file
 - rejected records, SQL*Loader, 3 – 11
 - specifying, SQL*Loader, 5 – 19
- bad records, identifying, BAD, SQL*Loader command-line parameter, 6 – 3
- BADDN keyword, SQL*Loader, 5 – 19
- BADFILE keyword, SQL*Loader, 5 – 19

- base backup, Export. *See* Complete export
- base tables, exported during incremental export, 1 – 28
- BEGINDATA, control file keyword, SQL*Loader, 5 – 15
- bind array
 - conventional path load, SQL*Loader, 3 – 3
 - determining size, SQL*Loader, 5 – 63
 - determining size of, SQL*Loader, 5 – 64
 - minimizing memory requirements, SQL*Loader, 5 – 67
 - minimum requirements, SQL*Loader, 5 – 63
 - no space required for generated data, SQL*Loader, 5 – 67
 - performance implications, SQL*Loader, 5 – 63
 - size, with multiple INTO TABLE clauses, SQL*Loader, 5 – 67
 - specifying number of rows, conventional path load, SQL*Loader, 6 – 6
 - specifying size, SQL*Loader, 6 – 3
- BINDSIZE, SQL*Loader command-line parameter, 6 – 3
- BINDSIZE command-line parameter, SQL*Loader, 5 – 64
- blanks
 - BLANKS keyword for field comparison, SQL*Loader, 5 – 9, 5 – 38
 - loading fields consisting of blanks, SQL*Loader, 5 – 69
 - preserving, SQL*Loader, 5 – 74
 - trailing, loading with delimiters, SQL*Loader, 5 – 61

blanks *contents*

- trimming, SQL*Loader, 5 – 69

- whitespace, SQL*Loader, 5 – 69

BLANKS keyword, SQL*Loader, 5 – 38

BLOCKSIZE keyword, obsoleted, A – 8

BUFFER

- Export parameter, 1 – 13

- Import parameter, 2 – 18

buffers

- calculating for export, 1 – 13

- Export parameter, 1 – 13

- I/O for direct path load, SQL*Loader, 8 – 3,
8 – 13

- Import parameter, 2 – 18

size

- Import, 2 – 6, 2 – 25

- specifying with BINDSIZE parameter,
SQL*Loader, 5 – 64

space required by

- LONG DATA, SQL*Loader, 5 – 56

- VARCHAR data, SQL*Loader, 5 – 55

- specifying interactively, Export, 1 – 18

BYTEINT datatype, 5 – 50

- specification, SQL*Loader, 5 – 10

- SQL*Loader, 5 – 51

C

cached sequence numbers, Export, 1 – 7

case studies, SQL*Loader, 4 – 1

- associated files, 4 – 2

- file names, 4 – 2

- preparing tables, 4 – 3

CATALOG.SQL, 2 – 15

- preparing database for Export, 1 – 10

CATEXP.SQL, 2 – 15

- preparing database for Export, 1 – 10

CATEXP6.SQL, Export, 1 – 39

CATLDR.SQL, setup script, SQL*Loader, 8 – 7

CHAR columns, Oracle Version 6 export files,
2 – 36

CHAR datatype

- delimited form, SQL*Loader, 5 – 58

- reference, SQL*Loader, 5 – 56

- specification, SQL*Loader, 5 – 10

- trimming whitespace, SQL*Loader, 5 – 69

character datatypes, conflicting fields,
SQL*Loader, 5 – 61

character fields

- datatypes, SQL*Loader, 5 – 56

- delimiters, SQL*Loader, 5 – 58

- determining length, SQL*Loader, 5 – 61

- specified with delimiters, SQL*Loader, 5 – 56

character set, conversion, Export, 1 – 3

character sets

- 8-bit to 7-bit conversions, Import/Export,
2 – 36

- conversion between, SQL*Loader, 5 – 24

- conversion during import, 1 – 8, 2 – 3

- conversion for import, A – 4

- conversions, Export/Import, 1 – 9

- Import conversion, 2 – 12

multi-byte

- Import/Export, 2 – 36

- SQL*Loader, 5 – 24

- single-byte, Import/Export, 2 – 36

- version 6 conversions, Import/Export, 2 – 36

character strings

- as part of a field comparison, SQL*Loader,
5 – 9

- padded, when shorter than field,
SQL*Loader, 5 – 39

CHARACTERSET keyword, SQL*Loader,
5 – 24

CHARSET, Import parameter, 1 – 9, 2 – 19,
A – 5

- valid values, 2 – 19

check constraints, Import, 2 – 5

clusters

- Export, 1 – 29

- exporting definitions, 1 – 5

columns

- end of a field, SQL*Loader, 5 – 40

- import/export LONGs, 1 – 7

- naming, SQL*Loader, 5 – 38, 5 – 39

- null columns at the end of a record,

- SQL*Loader. *See* TRAILING NULLCOLS

- reordering before Import, 2 – 31

- setting to a constant value, SQL*Loader,
5 – 46

- setting to a unique sequence number, SQL*Loader, 5 – 47
- setting to datafile record number, SQL*Loader, 5 – 46
- setting to null, SQL*Loader, 5 – 68
- setting to null value, SQL*Loader, 5 – 46
- setting to the current date, SQL*Loader, 5 – 46
- setting value to zero, SQL*Loader, 5 – 67
- specifying as PIECED, SQL*Loader, 8 – 11
- specifying, SQL*Loader, 5 – 39
- starting position of a field, SQL*Loader, 5 – 40
- command-line parameters
 - description, SQL*Loader, 6 – 2
 - specifying defaults, SQL*Loader, 5 – 12
- comments
 - in Import parameter file, 2 – 17
 - in SQL*Loader control file, 4 – 9, 5 – 11
 - in the Export parameter file, 1 – 12
- COMMIT, Import parameter, 2 – 19
- complete exports, 1 – 24, 1 – 26
 - command syntax, 1 – 29
 - restrictions, 1 – 24
 - specifying, 1 – 15
- COMPRESS, Export parameter, 1 – 13, 2 – 3
- CONCATENATE keyword, SQL*Loader, 5 – 29 to 5 – 32
- connecting to a database, importing user definitions, Import, A – 4
- CONSISTENT, Export parameter, 1 – 13
- consistent views, exporting, 1 – 28
- consolidating extents, Export parameter COMPRESS, 1 – 13
- CONSTANT keyword
 - no space used in bind array, SQL*Loader, 5 – 67
 - SQL*Loader, 5 – 39, 5 – 46
- CONSTRAINTS, Export parameter, 1 – 14
- constraints
 - automatic, SQL*Loader, 8 – 18
 - check, Import, 2 – 5
 - CONSTRAINTS parameter on export, A – 3
 - direct path load, 8 – 16
 - disabling during a direct load, 8 – 17
 - disabling referential constraints, Import, 2 – 31
 - enabling after a direct load, 8 – 17
 - enforced on a direct load, 8 – 17
 - failed, Import, 2 – 5
 - load method, SQL*Loader, 8 – 6
 - not null, Import, 2 – 5
 - preventing Import errors due to uniqueness constraints, 2 – 19
 - PRIMARY KEY constraint, SQL*Loader, 8 – 23
 - referential integrity, Import, 2 – 5
 - UNIQUE constraint, SQL*Loader, 8 – 23
 - uniqueness, Import, 2 – 5
- continuation fields, SQL*Loader, 3 – 8
- CONTINUE_LOAD keyword, SQL*Loader, 5 – 28
- CONTINUEIF keyword
 - example, SQL*Loader, 4 – 11
 - SQL*Loader, 5 – 29 to 5 – 32
- continuing interrupted loads, SQL*Loader, 5 – 27
- CONTROL, SQL*Loader command-line parameter, 6 – 4
- control files
 - comments, SQL*Loader, 5 – 11
 - CONTROL, SQL*Loader command-line parameter, 6 – 4
 - creating, SQL*Loader, 3 – 5
 - data definition language syntax, SQL*Loader, 5 – 4
 - data definitions, basics, SQL*Loader, 3 – 3
 - definition, SQL*Loader, 3 – 5
 - editing, SQL*Loader, 3 – 5
 - field delimiters, SQL*Loader, 5 – 11
 - guidelines for creating, SQL*Loader, 3 – 5
 - location, SQL*Loader, 3 – 5
 - specifying data, SQL*Loader, 5 – 15
 - specifying discard file, SQL*Loader, 5 – 21
 - storing, SQL*Loader, 3 – 5
- conventional path loads
 - basics, 3 – 3, 8 – 2
 - bind array, SQL*Loader, 5 – 63
 - compared to direct path loads, 8 – 6
 - using, 8 – 2

- CREATE SESSION privilege
 - exporting database objects and, 1 – 3
 - Import, 2 – 4
- CREATE USER, Import, 2 – 9
- CTIME column, Export, 1 – 31
- cumulative exports, 1 – 24, 1 – 25
 - command syntax, 1 – 29
 - recording export in database tables, 1 – 16
 - restrictions, 1 – 24
 - specifying, 1 – 15
 - SYS.INCFIL table, 1 – 31
 - SYS.INCVID table, 1 – 31

D

- DATA, SQL*Loader command-line parameter, 6 – 4
- data
 - binary versus character format, SQL*Loader, 3 – 6
 - delimiter marks in data, SQL*Loader, 5 – 60
 - distinguishing different input formats, SQL*Loader, 5 – 42
 - enclosed, SQL*Loader, 3 – 7
 - formatted data, SQL*Loader, 4 – 22
 - generating unique values, SQL*Loader, 5 – 47
 - including in control files, SQL*Loader, 5 – 15
 - loading in sections, SQL*Loader, 8 – 11
 - loading into more than one table, SQL*Loader, 5 – 42
 - loading LONG, SQL*Loader, 5 – 56
 - loading without files, SQL*Loader, 5 – 45
 - mapping to Oracle format, SQL*Loader, 3 – 3
 - maximum length of delimited data, SQL*Loader, 5 – 61
 - methods of loading into tables, SQL*Loader, 5 – 25
 - moving between operating systems, SQL*Loader, 5 – 62
 - saving in a direct path load, 8 – 9
 - saving rows, SQL*Loader, 8 – 15
 - suppressing import of, 2 – 26
 - terminated, SQL*Loader, 3 – 7
 - unsorted, SQL*Loader, 8 – 14
 - values optimized for performance, SQL*Loader, 5 – 45

- data conversion, description, SQL*Loader, 3 – 12
- data definition language
 - APPEND keyword, 5 – 25
 - basics, SQL*Loader, 3 – 3
 - BEGINDATA keyword, 5 – 15
 - BLANKS keyword, SQL*Loader, 5 – 38
 - CHARACTERSET keyword, 5 – 24
 - column_name, SQL*Loader, 5 – 10
 - CONCATENATE keyword, 5 – 29
 - CONSTANT keyword, 5 – 39, 5 – 46
 - CONTINUEIF keyword, 5 – 29
 - datatype_spec, SQL*Loader, 5 – 10
 - date mask, SQL*Loader, 5 – 11
 - DEFAULTIF keyword, SQL*Loader, 5 – 67
 - delimiter_spec, SQL*Loader, 5 – 11
 - description, SQL*Loader, 3 – 4
 - DISABLED_CONSTRAINTS keyword, SQL*Loader, 8 – 17
 - DISCARD DDN keyword, 5 – 22
 - DISCARD MAX keyword, SQL*Loader, 5 – 23
 - example definition, SQL*Loader, 3 – 4
 - EXCEPTIONS keyword, SQL*Loader, 8 – 17
 - EXTERNAL keyword, 5 – 58
 - field_condition, SQL*Loader, 5 – 9
 - FILE keyword, SQL*Loader, 8 – 23
 - FLOAT keyword, 5 – 58
 - IN DDN keyword, 5 – 16
 - IN FILE keyword, 5 – 16
 - INSERT keyword, 5 – 25
 - length, SQL*Loader, 5 – 11
 - loading data in sections, SQL*Loader, 8 – 11
 - NULLIF keyword, SQL*Loader, 5 – 68
 - parallel keyword, SQL*Loader, 8 – 21
 - pos_spec, SQL*Loader, 5 – 9
 - POSITION keyword, 5 – 40
 - precision, SQL*Loader, 5 – 11
 - RECNUM keyword, 5 – 39
 - REENABLE keyword, SQL*Loader, 8 – 17
 - reference, keywords and parameters, SQL*Loader, 5 – 1
 - REPLACE keyword, 5 – 26
 - SEQUENCE keyword, 5 – 47
 - syntax diagrams
 - expanded, SQL*Loader, 5 – 9
 - high-level, SQL*Loader, 5 – 6
 - syntax reference, SQL*Loader, 5 – 1

- SYSDATE keyword. *See* SYSDATE datatype
- TERMINATED keyword, 5 – 58
- TRUNCATE keyword, 5 – 26
- UNRECOVERABLE keyword, SQL*Loader, 8 – 15
- WHITESPACE keyword, 5 – 58
- data field, specifying the datatype, SQL*Loader, 5 – 40
- data mapping, concepts, SQL*Loader, 3 – 3
- data path loads, direct and conventional, 8 – 2
- data recovery, direct path load, SQL*Loader, 8 – 10
- database administrator (DBA), privileges, for export, 1 – 3
- database objects
 - export privileges, 1 – 3
 - exporting all, 1 – 5
 - exporting LONG columns, 2 – 33
 - incremental export, 1 – 24
 - transferring across a network, Import, 2 – 33
- databases
 - data structures and exporting, 1 – 28
 - Export basics, 1 – 2
 - full export, 1 – 15
 - interactive mode, 1 – 18
 - full import, 2 – 27
 - importing full, 2 – 21
 - importing into secondary, Import, 2 – 20
 - preparing for Export, 1 – 10
 - privileges for exporting, 1 – 3
 - reducing fragmentation via full export/import, 2 – 35
 - reusing existing data files, Import, 2 – 19
- datafiles
 - preventing overwrite during import, 2 – 19
 - reusing during import, 2 – 19
 - specifying, SQL*Loader, 5 – 16, 6 – 4
 - specifying buffering, SQL*Loader, 5 – 18
 - specifying format, SQL*Loader, 5 – 18
 - storage, SQL*Loader, 3 – 6
- datatypes
 - BYTEINT, 5 – 51
 - SQL*Loader, 5 – 10
 - CHAR, 5 – 56
 - SQL*Loader. *See* CHAR datatype
 - conflicting character datatype fields, 5 – 61
 - converting, SQL*Loader, 3 – 12, 5 – 49

- DATE, 5 – 56
 - determining length, 5 – 62
 - SQL*Loader. *See* DATE datatype
- DECIMAL, 5 – 52
 - SQL*Loader. *See* DECIMAL datatype
- default, SQL*Loader, 5 – 40
- determining character field lengths, SQL*Loader, 5 – 61
- DOUBLE, 5 – 51
 - SQL*Loader, 5 – 10
- FLOAT, 5 – 51
 - SQL*Loader. *See* FLOAT datatype
- GRAPHIC, 5 – 53
 - SQL*Loader, 5 – 10
- GRAPHIC EXTERNAL, 5 – 53
- INTEGER, 5 – 50
 - SQL*Loader. *See* INTEGER datatype
- MLSLABEL, Trusted Oracle7 Server, 5 – 57, 5 – 62
- native
 - conflicting length specifications, SQL*Loader, 5 – 55
 - inter-operating system transfer issues, 5 – 62
 - SQL*Loader, 3 – 6, 5 – 50
- NUMBER, SQL*Loader, 5 – 49
- numeric EXTERNAL, 5 – 58
 - trimming, SQL*Loader, 5 – 69
- RAW, 5 – 52
 - SQL*Loader, 5 – 10
- SMALLINT, 5 – 50
 - SQL*Loader, 5 – 10
- specifications, SQL*Loader, 5 – 10
- specifying, SQL*Loader, 5 – 49
- specifying the datatype of a data field, SQL*Loader, 5 – 40
- VARCHAR, 5 – 54
 - SQL*Loader. *See* VARCHAR datatype
- VARCHAR2, SQL*Loader, 5 – 49
- VARGRAPHIC, 5 – 53
 - SQL*Loader, 5 – 10
- ZONED, 5 – 51
 - SQL*Loader. *See* ZONED datatype
- DATE datatype
 - delimited form, SQL*Loader, 5 – 58
 - determining length, SQL*Loader, 5 – 62
 - mask, SQL*Loader, 5 – 62
 - specification, SQL*Loader, 5 – 10

DATE datatype *continued*
 SQL*Loader, 5 – 56
 trimming whitespace, SQL*Loader, 5 – 69

date mask, SQL*Loader, 5 – 11

DB2 Load utility, use with SQL*Loader, 3 – 4

DB2 load utility, C – 1 to C – 7
 different placement of statements
 DISCARDDDN, C – 4
 DISCARDS, C – 4
 restricted capabilities of SQL*Loader, C – 5
 RESUME keyword, SQL*Loader equivalents,
 5 – 25
 SQL*Loader compatibility, ignored state-
 ments, C – 4

DBA role, EXP_FULL_DATATBASE role,
 1 – 10

DBCS (DB2 double-byte character set), not
 supported by Oracle, C – 5

DBHIGH system label, Trusted Oracle7 Server,
 2 – 9

DBMS MAC database, Trusted Oracle7 Server,
 2 – 15

DDL. *See* Data Definition Language

DECIMAL datatype
 (packed), 5 – 50
 EXTERNAL format
 SQL*Loader, 5 – 58
 trimming whitespace, SQL*Loader, 5 – 69
 length and precision, SQL*Loader, 5 – 11
 specification, SQL*Loader, 5 – 10
 SQL*Loader, 5 – 52

DEFAULT column values, Oracle Version 6 ex-
 port files, 2 – 37

DEFAULTIF keyword
 field condition, SQL*Loader, 5 – 37
 SQL*Loader, 5 – 67

DELETE ANY TABLE privilege, SQL*Loader,
 5 – 27

DELETE CASCADE, SQL*Loader, 5 – 26,
 5 – 27

DELETE privilege, SQL*Loader, 5 – 26

delimited data, maximum length, SQL*Loader,
 5 – 61

delimited fields, field length, SQL*Loader,
 5 – 61

delimited files, reading, Export, 1 – 2

delimiter_spec, SQL*Loader, 5 – 11

delimiters
 and SQL*Loader, 3 – 7
 control files, SQL*Loader, 5 – 11
 enclosure, SQL*Loader, 5 – 70
 field specifications, SQL*Loader, 5 – 70
 initial and trailing, case study, 4 – 22
 loading trailing blanks, SQL*Loader, 5 – 61
 marks in data, SQL*Loader, 5 – 60
 optional enclosure, SQL*Loader, 5 – 70
 specifying, SQL*Loader, 5 – 35, 5 – 58
 termination, SQL*Loader, 5 – 70

DESTROY, Import parameter, 2 – 19

DIRECT
 Export parameter, 1 – 14
 SQL*Loader command-line parameter, 6 – 4

direct load state, indexes left in direct load
 state, 8 – 9

Direct path exports, 1 – 32

direct path load
 advantages, 8 – 5
 case study, 4 – 20
 choosing sort order, SQL*Loader, 8 – 14
 compared to conventional path load, 8 – 6
 conditions for use, 8 – 6
 data saves, 8 – 9, 8 – 15
 DIRECT, SQL*Loader command-line
 parameter, 6 – 4
 DIRECT command line parameter,
 SQL*Loader, 8 – 7
 DISABLED_CONSTRAINTS keyword,
 8 – 17
 disabling media protection, SQL*Loader,
 8 – 15
 dropping indexes, 8 – 16
 to continue an interrupted load,
 SQL*Loader, 5 – 28
 EXCEPTIONS keyword, 8 – 17
 field defaults, 8 – 7
 improper sorting, SQL*Loader, 8 – 14
 index storage requirements, 8 – 8
 indexes, 8 – 7
 instance recovery, 8 – 10
 loading into synonyms, 8 – 7
 LONG data, 8 – 11

- media recovery, 8 – 11
- partitioned load, SQL*Loader, 8 – 21
- performance, 8 – 12
- performance issues, 8 – 7
- preallocating storage, 8 – 13
- presorting data, 8 – 13
- recovery, 8 – 10
- REENABLE keyword, 8 – 17
- referential integrity constraints, 8 – 17
- Release 7.0 changes, A – 7
- ROWS command line parameter, 8 – 9
- setting up, 8 – 7
- specifying, 8 – 7
- specifying number of rows to be read, SQL*Loader, 6 – 6
- SQL*Loader, 8 – 3
- SQL*Loader data loading method, 3 – 3
- table insert triggers, 8 – 18
- temporary segment storage requirements, 8 – 8
- triggers, 8 – 16
- using, 8 – 6, 8 – 7
- version requirements, 8 – 7
- DISABLED_CONSTRAINTS keyword, SQL*Loader, 8 – 17
- DISCARD, SQL*Loader command-line parameter, 6 – 4
- discard file
 - basics, SQL*Loader, 3 – 11
 - DISCARD DDN keyword
 - different placement from DB2, C – 4
 - SQL*Loader, 5 – 22
 - DISCARD FILE keyword, example, 4 – 11
 - DISCARD MAX keyword
 - example, 4 – 11
 - SQL*Loader, 5 – 23
 - DISCARDS control file clause, different placement from DB2, C – 4
 - DISCARDS keyword, SQL*Loader, 5 – 23
 - DISCARD MAX keyword, SQL*Loader, 5 – 23
 - SQL*Loader, 5 – 21
- discarded records
 - causes, SQL*Loader, 5 – 23
 - discard file, SQL*Loader, 5 – 21
 - limiting the number, SQL*Loader, 5 – 23
 - SQL*Loader, 3 – 10
- DISCARD MAX, SQL*Loader command-line parameter, 6 – 4

- DISCARD MAX keyword, discarded records, SQL*Loader, 5 – 23
- discontinued loads, continuing, SQL*Loader, 5 – 27
- DOUBLE datatype, 5 – 50
 - specification, SQL*Loader, 5 – 10
 - SQL*Loader, 5 – 51
- dropped snapshots, Import, 2 – 11
- dropping, indexes, to continue a direct path load, SQL*Loader, 5 – 28

E

- EBCDIC file conversion, 1 – 3
- EBCDIC to ASCII conversion, Import, 1 – 8, 2 – 3
- eight-bit character set support, See National Language Support, 1 – 9
- enclosed fields
 - and SQL*Loader, 3 – 7
 - ENCLOSED BY control file clause, SQL*Loader, 5 – 11
 - specified with enclosure delimiters, SQL*Loader, 5 – 58
 - whitespace in, SQL*Loader, 5 – 72
- enclosure delimiters
 - and SQL*Loader, 3 – 7
 - initial, SQL*Loader, 5 – 60
 - SQL*Loader, 5 – 70
 - trailing, SQL*Loader, 5 – 60
- ending column of a field, SQL*Loader, 5 – 40
- error handling
 - Export, 1 – 4
 - Import, 2 – 5
- error messages
 - caused by tab characters in data, SQL*Loader, 5 – 41
 - export log file, 1 – 15
 - fatal errors, 1 – 4
 - generated by DB2 load utility, C – 5
 - row errors during import, 2 – 5
- error messages
 - Trusted Oracle7 Server, 1 – 5
- ERRORS, SQL*Loader command-line parameter, 6 – 4

errors *continued*

- fatal Import errors, 2 – 7
- Import resource errors, 2 – 7
- LONG data, 2 – 6
- object creation, Import parameter IGNORE, 2 – 21
- object creation errors, 2 – 7
- escape character, quoted strings, SQL*Loader, 5 – 14
- EXCEPTIONS keyword, SQL*Loader, 8 – 17
- EXP-00024, fatal error message, 1 – 4
- EXP_FULL_DATABASE, export modes, 1 – 5
- EXP_FULL_DATABASE role, 1 – 3, 1 – 10, 1 – 15
 - about, 1 – 3
 - Import, 2 – 4
 - incremental export, A – 5
 - required for FULL database export, 1 – 5
- EXPDAT.DMP, Export output file, 1 – 15, 1 – 18
- EXPDATE column, Export, 1 – 31
- EXPFILE column, Export, 1 – 31
- EXPID column
 - Export, 1 – 31
- Export
 - 8-bit vs. 7-bit character sets, 1 – 9
 - all database objects, 1 – 5
 - all objects in user's schema, 1 – 5
 - and unclustered tables, 1 – 5
 - base backup, 1 – 24
 - basics, 1 – 2
 - BUFFER parameter, 1 – 13
 - buffer size, interactive prompt, 1 – 18
 - CATALOG.SQL, prepares database for Export, 1 – 10
 - CATEXP.SQL, prepares database for Export, 1 – 10
 - CATEXP6.SQL, preparing the database or Version 6 exports, 1 – 39
 - character set conversion, 1 – 3
 - complete, 1 – 24
 - example, 1 – 24
 - complete export, 1 – 26
 - privileges, 1 – 24
 - restrictions, 1 – 24
 - complete exports, 1 – 15
 - COMPRESS parameter, 1 – 13
 - CONSISTENT parameter, A – 3

- consolidating extents, interactive mode, 1 – 19
- CONSTRAINTS parameter, 1 – 14, A – 3
- creating necessary privileges, 1 – 10
- creating necessary views, 1 – 10, 1 – 34
- creating Version 6 export files, 1 – 37, 1 – 39
- cumulative, 1 – 24, 1 – 25
- cumulative export
 - example, 1 – 24
 - privileges, 1 – 24
 - restrictions, 1 – 24
- cumulative exports, 1 – 15
- data structures, 1 – 28
- database optimizer statistics, 1 – 16
- DIRECT parameter, 1 – 14
- direct path, 1 – 32
- displaying help message, 1 – 15
- error messages, A – 3
- establishing export views, 1 – 10, A – 3
- example sessions, 1 – 20
 - full database mode, 1 – 20
 - table mode, 1 – 22
 - user mode, 1 – 21
- EXPDAT.DMP, 1 – 18
- exporting all schema objects. *See* USER mode export
- exporting an entire database, 1 – 15
- exporting constraints, 1 – 14
- exporting grants, interactive mode, 1 – 18
- exporting indexes, 1 – 15
- exporting objects owned by SYS, 1 – 5
- exporting to another operating system, RECORDLENGTH parameter, 1 – 16
- fatal error messages, 1 – 4
- FEEDBACK parameter, 1 – 15
- file format, 1 – 38
- file formats, 1 – 3
- file handling, 1 – 3
- FILE parameter, 1 – 15
- full database export, interactive mode, 1 – 18
- full database mode, example session, 1 – 20
- FULL parameter, 1 – 15
- GRANTS parameter, 1 – 15
 - change in default value, A – 3
- HELP parameter, 1 – 15
- incremental, 1 – 24
- incremental export
 - example, 1 – 24
 - example session, 1 – 30

- privileges, 1 – 24
- restrictions, 1 – 24
- system tables, 1 – 31
- incremental export/import, A – 5
 - command syntax, 1 – 29
- incremental exports, 1 – 15
- INCTYPE parameter, 1 – 15
- INDEXES parameter, 1 – 15
- interactive mode, 1 – 10, 1 – 17
- interactive prompts, 1 – 18
- invoking, 1 – 10
- kinds of data exported, 1 – 28
- last valid export, SYS.INCVID table, 1 – 31
- listing Trusted Oracle7 Server labels, A – 2
- log files, LOG parameter, 1 – 15
- logging error messages, 1 – 15
- LONG columns, 1 – 7
- message log file, 1 – 4
- MLS parameter, Trusted Oracle7 Server, 1 – 16
- MLS_LABEL_FORMAT parameter, Trusted Oracle7 Server, 1 – 16
- multi-byte character sets, 1 – 9
- network issues, 1 – 8
- NLS support, 1 – 9
- NLS_LANG environment variable, 1 – 9
- objects, specifying export by username, 1 – 16
- objects owned by SYS, 1 – 5
- online help, 1 – 11
- Oracle Version 6
 - audit options, 1 – 41
 - datatype conversions, 1 – 40
 - LONG data errors, 1 – 40
 - truncated database link names, 1 – 40
 - VARCHAR errors, 1 – 40
- OWNER parameter, 1 – 16
- parameter conflicts, 1 – 17
- parameter CONSISTENT, 1 – 13
- parameter file, 1 – 11
 - maximum size, 1 – 11
- parameters, 1 – 12 to 1 – 42
 - conflicts, 1 – 17
- preparing for database for Export, 1 – 10
- previous versions, 1 – 37
- read-consistent view, 1 – 13
- read-consistent views, 1 – 28
- read-only tablespaces, A – 2

- RECORD parameter, 1 – 16
- RECORDLENGTH parameter, 1 – 16
- redirecting output to a log file, 1 – 4
- remote operation, 1 – 8
- restrictions, 1 – 3
- rollback segments, 1 – 29
- ROWS parameter, 1 – 16
- schema objects, interactive mode, 1 – 19
- sequence numbers, 1 – 7
- specified tables, 1 – 5
- specifying a list of tables to export, 1 – 17
- specifying export file name interactively, 1 – 18
- SQL*Net, 1 – 8
- STATISTICS parameter, 1 – 16
 - COMPUTE option, 1 – 16
 - ESTIMATE option, 1 – 16
- storage requirements, 1 – 10, 1 – 34
- SYS.INCEXP table
 - CTIME column, 1 – 31
 - EXPID column, 1 – 31
 - ITIME column, 1 – 31
 - NAME column, 1 – 31
 - OWNER# column, 1 – 31
 - TYPE column, 1 – 31
- SYS.INCFIL table, 1 – 31
 - EXPDATE column, 1 – 31
 - EXPFILE column, 1 – 31
 - EXPTYPE column, 1 – 31
 - EXPUSER column, 1 – 31
- SYS.INCVID table, 1 – 31
- table data, 1 – 18
- table mode, example session, 1 – 22
- table name restrictions, Export, 1 – 12
- table names, interactive mode, 1 – 19
- table of objects exported by mode, 1 – 6
- TABLES, 1 – 5
- TABLES parameter, 1 – 17
- tracking exported objects, 1 – 31
- tracking table for incremental export, 1 – 29
- transferring export files across a network, 1 – 8
- Trusted Oracle7 Server-specific information, 1 – 8
- user access privileges, 1 – 3
- USER mode, 1 – 5
- user mode, example session, 1 – 21

Export *continued*

- USER_SEGMENTS view, 1 – 10, 1 – 34
- USERID parameter, 1 – 17
- using, 1 – 10
- warning messages, 1 – 4
- export file
 - displaying contents, Import, 1 – 3
 - format, 1 – 2
 - importing entire file, 2 – 27
 - importing the entire file, 2 – 21
 - listing contents before importing, 2 – 23
 - listing contents of, 2 – 25
 - reading, 1 – 2
- Export modes
 - FULL database, 1 – 5
 - table, 1 – 5
 - user, 1 – 5
- export views, establishing, Export, A – 3
- Export/Import, using Oracle Version 6 files,
2 – 36 to 2 – 39
- exporting database objects, log files, A – 2
- EXPTYPE column, Export, 1 – 31
- EXPUSER column, Export, 1 – 31
- EXPVIEW.SQL, A – 3
- extent allocation, FILE, SQL*Loader command
line parameter, 6 – 5
- extents
 - consolidating, interactive mode, 1 – 19
 - consolidating into one extent, Export, 1 – 13
 - importing consolidated, 2 – 3
- EXTERNAL datatypes
 - DECIMAL, SQL*Loader, 5 – 58
 - FLOAT, SQL*Loader, 5 – 58
 - GRAPHIC, SQL*Loader, 5 – 53
 - INTEGER, 5 – 58
 - numeric
 - determining length, SQL*Loader, 5 – 61
 - SQL*Loader, 5 – 58
 - trimming, SQL*Loader, 5 – 69
 - SQL*Loader, 3 – 6
 - ZONED, SQL*Loader, 5 – 58
- EXTERNAL keyword, SQL*Loader, 5 – 58

F

- fatal errors
 - EXP-00024, 1 – 4
 - Import, 2 – 6, 2 – 7
- FEEDBACK
 - Export parameter, 1 – 15
 - Import parameter, 2 – 20
- field conditions, specifying, SQL*Loader, 5 – 37
- field length, specifications, SQL*Loader, 5 – 70
- fields
 - and SQL*Loader, 3 – 7
 - character, data length, SQL*Loader, 5 – 61
 - comparing, SQL*Loader, 5 – 9
 - comparing to literals, SQL*Loader, 5 – 39
 - continuation, SQL*Loader, 3 – 8
 - DECIMAL EXTERNAL, trimming
whitespace, SQL*Loader, 5 – 69
 - delimited
 - determining length, SQL*Loader, 5 – 61
 - specifications, SQL*Loader, 5 – 70
 - SQL*Loader, 5 – 58
 - enclosed, SQL*Loader, 5 – 58
 - FLOAT EXTERNAL, trimming whitespace,
SQL*Loader, 5 – 69
 - INTEGER EXTERNAL, trimming
whitespace, SQL*Loader, 5 – 69
 - length of, SQL*Loader, 5 – 11, 5 – 40
 - loading all blanks, SQL*Loader, 5 – 69
 - location, SQL*Loader, 5 – 40
 - numeric and precision versus length,
SQL*Loader, 5 – 11
 - numeric EXTERNAL, trimming whitespace,
SQL*Loader, 5 – 69
 - precision, SQL*Loader, 5 – 11
 - predetermined size
 - length, SQL*Loader, 5 – 61
 - SQL*Loader, 5 – 70
 - relative positioning, SQL*Loader, 5 – 71
 - specification of position, SQL*Loader, 5 – 9
 - specified with a termination delimiter,
SQL*Loader, 3 – 7, 5 – 58
 - specified with enclosure delimiters,
SQL*Loader, 3 – 7, 5 – 58
 - specifying, SQL*Loader, 5 – 39
 - specifying default delimiters, SQL*Loader, 5
– 35
 - terminated, SQL*Loader, 5 – 58

- VARCHAR, never trimmed, SQL*Loader, 5 – 69
 - ZONED EXTERNAL, trimming whitespace, SQL*Loader, 5 – 69
- FIELDS clause
 - SQL*Loader, 5 – 35
 - terminated by whitespace, SQL*Loader, 5 – 72
- FILE
 - Export parameter, 1 – 15
 - Import parameter, 2 – 20
 - keyword, SQL*Loader, 8 – 23
 - SQL*Loader command-line parameter, 6 – 5
- file format, export files, 1 – 3
- filenames
 - bad file, SQL*Loader, 5 – 19
 - datafile, SQL*Loader, 5 – 16
 - quotation marks, SQL*Loader, 5 – 13
 - specifying for export, 1 – 18
 - specifying more than one, SQL*Loader, 5 – 17
 - SQL*Loader, 5 – 13
- files
 - Export file format, 1 – 2
 - file processing options string, SQL*Loader, 5 – 18
 - in EBCDIC format, 1 – 3
 - logfile, SQL*Loader, 3 – 8
 - SQL*Loader
 - bad file, 3 – 11
 - discard file, 3 – 11
 - storage, SQL*Loader, 3 – 6
- fixed format records, SQL*Loader, 3 – 6
- FIXED keyword, obsoleted, A – 8
- fixed-format records
 - SQL*Loader, 5 – 70
 - vs. variable, SQL*Loader, 3 – 6
- FLOAT datatype, 5 – 50
 - EXTERNAL format
 - SQL*Loader, 5 – 58
 - trimming whitespace, SQL*Loader, 5 – 69
 - specification, SQL*Loader, 5 – 10
 - SQL*Loader, 5 – 51
- FLOAT EXTERNAL data values, SQL*Loader, 5 – 58
- FLOAT keyword, SQL*Loader, 5 – 58

- FORMAT statement in DB2, not allowed by SQL*Loader, C – 5
 - formats, and input records, SQL*Loader, 5 – 44
 - formatting errors, SQL*Loader, 5 – 19
 - fragmentation, reducing database fragmentation via full export/import, 2 – 35
 - FROMUSER, Import parameter, 2 – 20
 - FTP, Export files, 1 – 8
 - FULL, Import parameter, 2 – 21
 - FULL database export, 1 – 5
 - full database export, interactive mode, 1 – 18
 - FULL database import, 2 – 21
 - full database import, interactive mode, 2 – 27

G

GRANTS

- Export parameter, 1 – 15
- change in default value, A – 3
- Import parameter, 2 – 21

grants

- exporting, 1 – 15
- exporting database objects and, 1 – 18
- importing, 2 – 9, 2 – 26
- specifying for import, 2 – 21

GRAPHIC datatype, 5 – 50

- EXTERNAL format, SQL*Loader, 5 – 53
- specification, SQL*Loader, 5 – 10
- SQL*Loader, 5 – 53

GRAPHIC EXTERNAL datatype, 5 – 50

H

HELP

- Export parameter, 1 – 15
- Import parameter, 2 – 21

help, Import, 2 – 16

hexadecimal strings

- as part of a field comparison, SQL*Loader, 5 – 9
- padded, when shorter than field, SQL*Loader, 5 – 39

I

IBM Code Page 500 files, 1 – 3

IGNORE parameter, Import, 2 – 7, 2 – 21

IMP_FULL_DATABASE role, 2 – 21, 2 – 30

- created by CATEXP.SQL, 2 – 15

- Import, 2 – 4, 2 – 24

- incremental import, A – 5

Import

- backup files, 2 – 11

- basics, 2 – 2

- BUFFER parameter, 2 – 18

- buffer size, 2 – 6

- CATALOG.SQL, preparing the database, 2 – 15

- CATEXP.SQL, preparing the database, 2 – 15

- character set conversion, 1 – 8, 2 – 3

- character sets, 2 – 12

- CHARSET parameter, 2 – 19

- COMMIT parameter, 2 – 19

- committing after array insert, 2 – 19

- compatibility, 2 – 4

- complete export file, 2 – 29

- COMPRESS parameter, see also Export, 2 – 3

- consolidated extents, 2 – 3

- controlling size of rollback segments, 2 – 19

- conversion of Version 6 CHAR columns to VARCHAR2, 2 – 36

- creating an index-creation SQL script, 2 – 22

- cumulative export file, 2 – 29

- data files, reusing, 2 – 19

- database

 - importing a full database, 2 – 21

 - reusing existing data files, 2 – 19

- DESTROY parameter, 2 – 19

- disabling referential constraints, 2 – 31

- displaying onscreen help, 2 – 21

- dropping a tablespace, 2 – 34

- error handling, 2 – 5

- error messages, A – 4

 - when version incompatible with database, A – 2

- example session, 2 – 27

- export file

 - importing the entire file, 2 – 21

 - listing contents before import, 2 – 23

- export file option SHOW, 1 – 3

- failed integrity constraints, 2 – 5

- fatal errors, 2 – 6, 2 – 7

- FEEDBACK parameter, 2 – 20

- FILE parameter, 2 – 20

- filename prompt, 2 – 25

- FROMUSER parameter, 2 – 20

- FULL database import, 2 – 21

- full database import, 2 – 27

- full database mode, 2 – 13

- grants, specifying for import, 2 – 21

- GRANTS parameter, 2 – 21

- HELP parameter, 2 – 16, 2 – 21

- IGNORE parameter, 2 – 7, 2 – 21

- IMP_FULL_DATABASE role, 2 – 4

- importing grants, 2 – 9, 2 – 26

- importing into existing tables, 2 – 26

- importing objects into other schemas, 2 – 9

- importing read-only tablespace, 2 – 35

- importing rows, 2 – 23

- importing tables, 2 – 24

- incremental export file, 2 – 29

 - RESTORE option, 2 – 30

 - SYSTEM option, 2 – 30

- incremental export/import, A – 5

 - restrictions, 2 – 29

 - syntax, 2 – 30

- incremental import, 2 – 30

- INCTYPE parameter, 2 – 22

- INDEXES parameter, 2 – 22

- INDEXFILE parameter, 2 – 22

- INSERT errors, 2 – 6

- interactive mode, 2 – 25

- into a secondary database, 2 – 20

- invalid data, 2 – 6

- length of Oracle Version 6 export file

 - DEFAULT columns, 2 – 37

- listing contents of export file, 2 – 25

- log files, A – 4

 - LOG parameter, 2 – 23

- LONG columns, 1 – 7, 2 – 33

- LONG data errors, 2 – 6

- LONG RAW data, errors, 2 – 6

- LONG RAW data errors, 2 – 6

- manually ordering tables, 2 – 32

- master table trigger, 2 – 10

- MLS parameter, Trusted Oracle7 Server, 2 – 23

- MLS_LISTLABELS parameter, Trusted Oracle7 Server, 2 – 23
- MLS_MAPFILE parameter, Trusted Oracle7 Server, 2 – 23
- modes, 2 – 13
- multi-byte character sets, 2 – 36
- NLS considerations, 2 – 36
- NLS_LANG environment variable, 2 – 36
- object creation errors, 2 – 21
- object errors, 2 – 6
- OPTIMAL, storage parameter, 2 – 3
- OPTIMAL storage parameter, 2 – 35
- Oracle Version 6 integrity constraints, 2 – 37
- Oracle Version 6 LONG columns, 2 – 36
- parameter file, 2 – 16
- parameters, 2 – 18 to 2 – 38
- preparing the database, 2 – 15
- privileges required, 2 – 4, 2 – 8
- read-only tablespaces, 2 – 4
- recompiling stored procedures, 2 – 10
- RECORDLENGTH parameter, 2 – 23
- records, specifying length, 2 – 23
- reducing database fragmentation via full export/import, 2 – 35
- refresh error, 2 – 11
- Release 7.0 features, A – 3
- reorganizing tablespace during Import, 2 – 34
- resource errors, 2 – 7
- restrictions, 2 – 4
- rows, specifying for import, 2 – 23
- ROWS parameter, 2 – 23
- schema objects, 2 – 8, 2 – 9
- schemas, specifying for import, 2 – 20
- sequences, 2 – 7
- SHOW parameter, 2 – 23
- single-byte character sets, 2 – 36
- snapshot log, 2 – 10
- snapshot master table, 2 – 10
- snapshots, 2 – 10 to 2 – 12
 - restoring dropped, 2 – 11
- specifying buffer size, 2 – 25
- specifying by table, 2 – 13
- specifying by user, 2 – 13, 2 – 20
- specifying index-creation commands, 2 – 22
- specifying the export file to use, 2 – 20
- SQL statements without data, 2 – 26
- SQL*Net, 2 – 33
- starting, 2 – 16
- statistics on imported data, 2 – 32
- storage parameters, overriding, 2 – 35
- stored functions, 2 – 10
- stored packages, 2 – 10
- stored procedures, 2 – 10
- suppressing data import, 2 – 26
- system objects, 2 – 9, 2 – 21
- table data, 2 – 26
- table mode, 2 – 13
- table objects, import order, 2 – 2
- tables created before import, 2 – 31
- tablespaces, pre-created, 2 – 20
- TOUSER parameter, 2 – 24
- transferring files across networks, 2 – 33
- triggers, 2 – 10
- Trusted Oracle7 Server, 2 – 4, 2 – 15
- unique indexes, 2 – 22
- uniqueness constraints, preventing import errors, 2 – 19
- user definitions, 2 – 9, A – 4
- user mode, 2 – 13
- USERID parameter, 2 – 24
- Import/Export, using Oracle Version 6 files, 2 – 36 to 2 – 39
- incremental export, 1 – 24
 - recording export in database tables, 1 – 16
 - restrictions, 1 – 24
 - session example, 1 – 30
 - specifying, 1 – 15
- incremental export/import
 - backing up data, 1 – 29
 - command syntax, 1 – 29
 - data selected for export, 1 – 28
 - roles needed, A – 5
 - syntax, 2 – 30
 - SYS.INCFIL table, 1 – 31
 - SYS.INCVID table, 1 – 31
- incremental import parameter, 2 – 22
- INCTYPE
 - Export parameter, 1 – 15
 - Import parameter, 2 – 22
 - values
 - RESTORE, 2 – 22
 - SYSTEM, 2 – 22
- INDDN keyword, SQL*Loader, 5 – 16
- index options
 - SINGLEROW keyword, SQL*Loader, 5 – 36

- SORTED INDEXES, SQL*Loader, 5 – 36
- INDEXES
 - Export parameter, 1 – 15
 - Import parameter, 2 – 22
- indexes
 - creating manually, 2 – 22
 - direct path load, left in direct load state, 8 – 9
 - dropping
 - before continuing a direct path load,
 - SQL*Loader, 5 – 28
 - SQL*Loader, 8 – 16
 - exporting, 1 – 15
 - importing, 2 – 22
 - index-creation commands, Import, 2 – 22
 - left direct load state, SQL*Loader, 8 – 14
 - multiple column, SQL*Loader, 8 – 14
 - presorting data
 - case study, 4 – 20
 - SQL*Loader, 8 – 13
 - SQL*Loader, 5 – 36 to 5 – 37
 - state after discontinued load,
 - SQL*Loader, 5 – 27
 - unique, 2 – 22
- INDEXFILE, Import parameter, 2 – 22
- INFILE keyword, SQL*Loader, 5 – 16
- INSERT errors, Import, 2 – 6
- insert errors, specifying allowed number
 - before termination, SQL*Loader, 6 – 4
- INSERT into table, SQL*Loader, 5 – 26
- INSERT keyword, SQL*Loader, 5 – 25
- instance recovery, direct path load,
 - SQL*Loader, 8 – 10
- INTEGER datatype, 5 – 50
 - EXTERNAL format, 5 – 58
 - trimming whitespace, SQL*Loader, 5 – 69
 - specification, SQL*Loader, 5 – 10
- integrity constraints
 - failed, Import, 2 – 5
 - load method, SQL*Loader, 8 – 6
 - Oracle Version 6 export files, 2 – 37
- interrupted loads, continuing,
 - SQL*Loader, 5 – 27
- INTO TABLE clause, effect on bind array size,
 - SQL*Loader, 5 – 67
- INTO TABLE statement
 - column names, SQL*Loader, 5 – 39

- discards, SQL*Loader, 5 – 23
- multiple, SQL*Loader, 5 – 42
- SQL*Loader, 5 – 33
- invalid data, Import, 2 – 6
- invalid objects, warning messages, during
 - export, 1 – 4
- ITIME column, Export, 1 – 31

K

- key values, generating, SQL*Loader, 5 – 47

L

- labels
 - Multi-Level Security (MLS), MLS export
 - parameter, Trusted Oracle7 Server,
 - 1 – 16
 - Trusted Oracle7 Server, 2 – 23
- language support, see National Language
 - Support, 1 – 9
- leading whitespace
 - definition, SQL*Loader, 5 – 69
 - trimming, SQL*Loader, 5 – 71
- length, specifying record length for export,
 - 1 – 16
- length indicator, determining size, SQL*loader,
 - 5 – 65
- length of a numeric field, SQL*Loader, 5 – 11
- length subfield, VARCHAR DATA,
 - SQL*Loader, 5 – 54
- LOAD, SQL*Loader command-line parameter,
 - 6 – 5
- loading
 - combined physical records, SQL*Loader,
 - 4 – 11 to 4 – 14
 - datafiles containing TABs, SQL*Loader,
 - 5 – 41
 - delimited, free-format files, 4 – 8 to 4 – 11
 - direct path, Release 7.1 features, A – 5
 - fixed-length data, 4 – 6 to 4 – 8
 - negative numbers, SQL*Loader, 4 – 11
 - variable-length data, 4 – 4 to 4 – 6

LOG

- Export parameter, 1 – 4, 1 – 15
- Import parameter, 2 – 23
- SQL*Loader command-line parameter, 6 – 5

log file, specifying, SQL*Loader, 6 – 5

log files

- after a discontinued load, SQL*Loader, 5 – 27
- datafile information, SQL*Loader, 7 – 4
- example, 4 – 21, 4 – 25
- Export, 1 – 4
- exporting database objects and, A – 2
- global information, SQL*Loader, 7 – 2
- header information, SQL*Loader, 7 – 2
- Import, A – 4
- LOG
 - Export parameter, 1 – 15
 - import parameter, 2 – 23
- SQL*Loader, 3 – 8
- summary statistics, SQL*Loader, 7 – 5
- table information, SQL*Loader, 7 – 3
- table load information, SQL*Loader, 7 – 4

logical records

- consolidating multiple physical records into, SQL*Loader, 5 – 29
- versus physical records, SQL*Loader, 3 – 7

LONG columns

- exporting from one operating system to another, 1 – 7
- importing and exporting, 1 – 7
- Oracle Version 6 export files, 2 – 36

LONG data

- downgrading to Version 6.0, 1 – 40
- Import errors, 2 – 6
- importing and exporting, 2 – 25, 2 – 33
- loading, SQL*Loader, 5 – 56
- loading with direct path load, 8 – 11
- LONG FLOAT, C language datatype, 5 – 51

LONG RAW data, importing, errors, 2 – 6

LXBCNF executable, 9 – 7

LXEGEN executable, 9 – 9

LXINST executable, 9 – 2

M

Mandatory Access Control, MLS export parameter, Trusted Oracle7 Server, 1 – 16

master table

- snapshots, Import, 2 – 10
- trigger, snapshots, Import, 2 – 10

media protection, disabling for direct path loads, SQL*Loader, 8 – 15

media recovery, direct path load, 8 – 11
SQL*Loader, 8 – 10

memory, controlling usage, SQL*Loader, 5 – 18

messages

- fatal error, 1 – 4
- fatal errors, Trusted Oracle7 Server, 1 – 5
- warning, 1 – 4

missing data columns, SQL*Loader, 5 – 35

MLS

- Export parameter, Trusted Oracle7 Server, 1 – 16
- Import parameter, Trusted Oracle7 Server, 2 – 23

MLS_LABEL_FORMAT, Export parameter, Trusted Oracle7 Server, 1 – 16

MLS_LISTLABELS, Import parameter, Trusted Oracle7 Server, 2 – 23

MLS_MAPFILE, Import parameter, Trusted Oracle7 Server, 2 – 23

MLSLABEL, field masks, SQL*Loader, Trusted Oracle7 Server, 5 – 62

MLSLABEL datatype, SQL*Loader, Trusted Oracle7 Server, 5 – 50, 5 – 57

Multi Level-Security labels, MLS export parameter, Trusted Oracle7 Server, 1 – 16

multi-byte character sets

- blanks, SQL*Loader, 5 – 38
- Export/Import issues, 1 – 9
- Import, 2 – 36
- SQL*Loader, 5 – 24

multiple CPUs, SQL*Loader, 8 – 21

multiple table load

- control file specification, SQL*Loader, 5 – 42
- discontinued, SQL*Loader, 5 – 28
- generating unique sequence numbers for, SQL*Loader, 5 – 48

multiple-column indexes, SQL*Loader, 8 – 14

N

National Language Support

- Export/Import, 1 – 8

- Import, 1 – 8, 2 – 3, 2 – 12, 2 – 19

- See also* NLS

- SQL*Loader, 5 – 24

National Language Support (NLS)

- data object files, 9 – 4

- installation boot file, 9 – 2

- NLS Configuration Utility, 9 – 6

- NLS Data Installation Utility, 9 – 2

- system boot file, 9 – 2

- user boot file, 9 – 2

native datatypes

- and SQL*Loader, 5 – 50

- binary versus character data,

- SQL*Loader, 3 – 6

- conflicting length specifications,

- SQL*Loader, 5 – 55

- delimiters, SQL*Loader, 5 – 50

- inter-operating system transfer issues,

- SQL*Loader, 5 – 62

- negative numbers, loading, SQL*Loader, 4 – 11

- network issues, Export, 1 – 8

networks

- exporting database objects, 2 – 33

- transporting Export files across a

- network, 1 – 8

- NLS, *see* National Language Support, 1 – 8

- NLS Calendar Utility, 9 – 9

- NLS Data Installation Utility, 9 – 2

- NLS_LANG, environment variable

- Export, 1 – 9

- Import, 2 – 36

- import/export, 1 – 9

- SQL*Loader, 5 – 24

- Non-fatal errors, warning messages, 1 – 4

- normalizing data during a load,

- SQL*Loader, 4 – 14

- NOT NULL constraint, load method,

- SQL*Loader, 8 – 6

- not null constraints, Import, 2 – 5

null columns

- at end of record, SQL*Loader, 5 – 68

- setting, SQL*Loader, 5 – 68

null data

- missing columns at end of record,

- SQL*Loader, 5 – 35

- unspecified columns, SQL*Loader, 5 – 39

NULLIF keyword

- field condition, SQL*Loader, 5 – 37

- SQL*Loader, 5 – 68, 5 – 69

- NULLIF...BLANKS, case study, 4 – 20

NULLIF...BLANKS keyword,

- SQL*Loader, 5 – 38

- NUMBER datatype, SQL*Loader, 5 – 49

numeric EXTERNAL datatypes

- binary versus character data,

- SQL*Loader, 3 – 6

- delimited form, SQL*Loader, 5 – 58

- determining length, SQL*Loader, 5 – 61

- SQL*Loader, 5 – 58

- trimming, SQL*Loader, 5 – 69

- trimming whitespace, SQL*Loader, 5 – 69

- numeric fields, precision versus length,

- SQL*Loader, 5 – 11

O

- object names, SQL*Loader, 5 – 13

objects

- creation errors, 2 – 7

- Import, 2 – 6

- export all, 1 – 5

- exporting by username, 1 – 16

- exporting schema objects, interactive mode,

- 1 – 19

- exporting SYS-owned objects, 1 – 5

- ignoring creation errors, Import, 2 – 26

- ignoring existing objects during

- import, 2 – 21

- import creation errors, 2 – 21

- privileges. *See* schema objects

- restoring, Import, 2 – 30

- restoring sets, Import, 2 – 30

- system objects, importing, 2 – 9

- table of objects exported by import

- mode, 2 – 14

- table showing objects exported by

- mode, 1 – 6

- obsolete parameters, A – 8

- online help
 - Export, 1 – 11
 - Import, 2 – 16
- operating systems, moving data to different systems, SQL*Loader, 5 – 62
- OPTIMAL, storage parameter, 2 – 3
- optimizing
 - direct path loads, 8 – 12
 - input file processing, SQL*Loader, 5 – 18
- OPTIONALLY ENCLOSED BY, SQL*Loader, 5 – 11, 5 – 70
- OPTIONS keyword
 - for parallel loads, SQL*Loader, 5 – 34, 8 – 22
 - SQL*Loader, 5 – 12
- ORA_NLS environment variable, 9 – 2, 9 – 7
- Oracle Version 6
 - creating export files, 1 – 39
 - export
 - datatype conversions, 1 – 40
 - LONG data errors, 1 – 40
 - truncated link names, 1 – 40
 - VARCHAR errors, 1 – 40
 - exporting database objects, 2 – 36 to 2 – 39
 - unsupported Release 7.1 objects, 1 – 39
- ORANLS option, 9 – 2, 9 – 7
- OS MAC database, Trusted Oracle7 Server, 2 – 15
- output file, specifying for Export, 1 – 15
- OWNER, Export parameter, 1 – 16

P

- packed decimal data, SQL*Loader, 5 – 11
- padding of literal strings, SQL*Loader, 5 – 39
- PARALLEL, SQL*Loader command-line parameter, 6 – 5
- PARALLEL keyword, SQL*Loader, 8 – 21
- parallel loads
 - allocating extents, FILE, SQL*Loader command-line parameter, 6 – 5
 - PARALLEL, SQL*Loader command-line parameter, 6 – 5
- parameter file
 - comments, 2 – 17
 - Export, 1 – 12

- Export, 1 – 11
- Import, 2 – 16
- maximum size, Export, 1 – 11
- parameters
 - BUFFER
 - Export, 1 – 12
 - Import, 2 – 18
 - CHARSET, Import, 2 – 19
 - COMMIT, Import, 2 – 19
 - COMPRESS, Export, 1 – 12
 - conflicts between export parameters, 1 – 17
 - CONSISTENT, Export, 1 – 12
 - CONSTRAINTS, Export, 1 – 12
 - DESTROY, Import, 2 – 19
 - Export, 1 – 12
 - FEEDBACK
 - Export, 1 – 12
 - Import, 2 – 20
 - FILE
 - Export, 1 – 12
 - Import, 2 – 20
 - FROMUSER, Import, 2 – 20
 - FULL
 - Export, 1 – 12
 - Import, 2 – 21
 - GRANTS, 2 – 21
 - Export, 1 – 12
 - HELP
 - Export, 1 – 12
 - Import, 2 – 21
 - IGNORE, Import, 2 – 21
 - INCTYPE
 - Export, 1 – 12
 - Import, 2 – 22
 - INDEXES
 - Export, 1 – 12
 - Import, 2 – 22
 - INDEXFILE, Import, 2 – 22
 - list
 - Export, 1 – 12 to 1 – 17
 - Import, 2 – 18 to 2 – 24
 - LOG, 1 – 4
 - Export, 1 – 12
 - Import, 2 – 23
 - MLS
 - Export, Trusted Oracle7 Server, 1 – 12
 - Trusted Oracle7 Server import, 2 – 23

- MLS_LABEL_FORMAT, Export, Trusted Oracle7 Server, 1 – 12
- MLS_LISTLABELS, Trusted Oracle7 Server import, 2 – 23
- MLS_MAPFILE, Trusted Oracle7 Server import, 2 – 23
- OWNER, Export, 1 – 12
- RECORD, Export, 1 – 12
- RECORDLENGTH
 - Export, 1 – 12
 - Import, 2 – 23
- ROWS
 - Export, 1 – 12
 - Import, 2 – 23
- SHOW, 2 – 23
- STATISTICS, Export, 1 – 12
- TABLES
 - Export, 1 – 12
 - Import, 2 – 24
- TOUSER, import, 2 – 24
- USERID
 - Export, 1 – 12
 - Import, 2 – 24
- PARFILE
 - Export command line option, 1 – 10, 1 – 11
 - Import command line option, 2 – 16
 - SQL*Loader command-line parameter, 6 – 5
- PART statement in DB2, not allowed by SQL*Loader, C – 5
- partitioned load, SQL*Loader, 8 – 21
- partitioned tables in DB2, no Oracle equivalent, C – 5
- passwords, hiding, 2 – 16
- performance
 - direct path loads, 8 – 12
 - optimizing reading of data files, SQL*Loader, 5 – 18
 - partitioned load, SQL*Loader, 8 – 21
- performance improvement, conventional path for small loads, 8 – 18
- physical versus logical records, SQL*Loader, 3 – 7
- PIECED keyword, SQL*Loader, 8 – 11

- POSITION keyword
 - specification of field position, SQL*Loader, 5 – 9
- SQL*Loader, 5 – 40
- tabs, 5 – 41
- with multiple INTO TABLE clauses, SQL*Loader, 5 – 41, 5 – 44
- precision of a numeric field versus length, SQL*Loader, 5 – 11
- predetermined size fields, SQL*Loader, 5 – 70
- prerequisites, SQL*Loader, 3 – 9
- PRESERVE BLANKS keyword, SQL*Loader, 5 – 74
- presorting data for a direct path load, case study, 4 – 20
- primary keys, Import, 2 – 5
- privileges
 - complete export, 1 – 24
 - creating privileges necessary for Export, 1 – 10
 - cumulative export, 1 – 24
- DELETE, SQL*Loader, 5 – 26
- DELETE ANY TABLE, SQL*Loader, 5 – 27
- for export
 - database objects, 1 – 3
 - schema objects, 1 – 3
- for normal export, 1 – 3
- Import, 2 – 4, 2 – 8
- incremental export, 1 – 24
- required for SQL*Loader, 3 – 9
- SQL*Loader and Trusted Oracle7 Server, 3 – 9

Q

- quotation marks
 - backslash
 - escape character, SQL*Loader, 5 – 14
 - in filenames, SQL*Loader, 5 – 14
 - filenames, SQL*Loader, 5 – 13
 - SQL string, SQL*Loader, 5 – 13
 - use with database object names, SQL*Loader, 5 – 13
 - within quoted strings, SQL*Loader, 5 – 13

R

- RAW datatype, 5 – 50
 - specification, SQL*Loader, 5 – 10
- SQL*Loader, 5 – 52
- read-consistent export, 1 – 13, A – 3
- read-only tablespaces
 - Export, A – 2
 - exporting/importing, 2 – 35
 - Import, 2 – 4
- READBUFFERS keyword, SQL*Loader, 5 – 18, 8 – 12
- RECLEN keyword, obsoleted, A – 8
- RECNUM keyword
 - no space used in bind array, SQL*Loader, 5 – 67
 - SQL*Loader, 5 – 39
 - use with SKIP, SQL*Loader, 5 – 46
- recompiling, stored functions, procedures, and packages, 2 – 10
- RECORD
 - Export parameter, 1 – 16
 - SQL*Loader parameter, obsoleted, A – 8
- RECORDLENGTH
 - Export parameter, 1 – 16
 - Import parameter, 2 – 23
- records
 - consolidating into a single logical record, SQL*Loader, 5 – 29
 - discarded
 - DISCARD, SQL*Loader command-line parameter, 6 – 4
 - DISCARDMAX, SQL*Loader command-line parameter, 6 – 4
 - SQL*Loader, 5 – 21
 - discarded by SQL*Loader, 3 – 10, 5 – 23
 - distinguishing different formats, SQL*Loader, 5 – 44
 - extracting multiple logical records, SQL*Loader, 5 – 42
 - fixed format, SQL*Loader, 3 – 6
 - formats, SQL*Loader, 5 – 70
 - loading methods, SQL*Loader, 3 – 3
 - null columns at end, SQL*Loader, 5 – 68
 - physical versus logical, SQL*Loader, 3 – 7
 - rejected, SQL*Loader, 5 – 19
 - rejected by Oracle, SQL*Loader, 3 – 11
 - rejected by SQL*Loader, 3 – 10, 3 – 11
 - setting column to record number, SQL*Loader, 5 – 46
 - short, missing data columns, SQL*Loader, 5 – 35
 - skipping, SQL*Loader, 6 – 7
 - specifying how to load, LOAD, SQL*Loader command-line parameter, 6 – 5
 - specifying length for export, 1 – 16
 - specifying length for import, 2 – 23
 - stream format, SQL*Loader, 3 – 6
 - variable format, SQL*Loader, 3 – 6
- RECOVERABLE keyword, SQL*Loader, 5 – 12
- recovery
 - direct path load, SQL*Loader, 8 – 10
 - replacing rows, 5 – 26
- redo log files
 - direct path load, 8 – 11
 - instance and media recovery, SQL*Loader, 8 – 11
 - saving space, direct path load, 8 – 15
- REENABLE keyword, SQL*Loader, 8 – 17
- referential integrity constraints
 - disabling for import, 2 – 31
 - Import, 2 – 5
 - SQL*Loader, 8 – 16
- refresh error, snapshots, Import, 2 – 11
- reject file, specifying, SQL*Loader. *See* bad file
- rejected records, SQL*Loader, 3 – 10, 5 – 19
- relative field positioning
 - where a field starts, SQL*Loader, 5 – 71
 - with multiple INTO TABLE clauses, SQL*Loader, 5 – 43 to 5 – 45
- Release 7.0 features
 - Export, A – 2
 - SQL*Loader, A – 6
- Release 7.1 features
 - Export/Import, A – 2
 - SQL*Loader, A – 5 to A – 6
- Release 7.2 features, Export/Import, A – 2
- remote operation, Export/Import, 1 – 8
- REPLACE keyword, SQL*Loader, 5 – 26
- REPLACE table
 - example, 4 – 11

- to replace table during a load,
SQL*Loader, 5 – 26
- reserved words, SQL*Loader, B – 2
- resource errors, Import, 2 – 7
- restrictions
 - DB2 load utility, C – 5
 - Export, 1 – 3
 - Import, 2 – 4
 - importing an incremental export file, 2 – 29
 - on importing grants, 2 – 9
 - on importing into another user's
schema, 2 – 9
 - on importing into own schema, 2 – 8
 - on table names, Export, 1 – 12
 - on table names in import parameter
file, 2 – 17
- RESUME, DB2 keyword, SQL*Loader
equivalents, 5 – 25
- roles
 - EXP_FULL_DATABASE, 1 – 3, 1 – 5, 1 – 10, 2
– 4, A – 5
 - IMP_FULL_DATABASE, 2 – 4, 2 – 15, 2 – 21,
2 – 30, A – 5
 - Import, 2 – 24
- rollback segments
 - CONSISTENT Export parameter, 1 – 13
 - controlling size during import, 2 – 19
 - during loads, SQL*Loader, 5 – 21
 - Export, 1 – 29
 - Import, 2 – 4
- row errors, Import, 2 – 5
- ROWID, Import, 2 – 11
- ROWS
 - command line parameter, SQL*Loader, 8 – 9
 - Export parameter, 1 – 16
 - Import parameter, 2 – 23
 - performance issues, SQL*Loader, 8 – 15
 - SQL*Loader command-line parameter, 6 – 6
- rows
 - choosing which to load, SQL*Loader, 5 – 34
 - exporting, 1 – 16
 - specifying for import, 2 – 23
 - specifying number to insert before save,
SQL*Loader, 8 – 9
 - updates to existing, SQL*Loader, 5 – 26

S

- schemas
 - exporting, interactive mode, 1 – 19
 - objects, export privileges, 1 – 3
 - specifying for import, 2 – 20
- scientific notation for FLOAT
EXTERNAL, 5 – 58
- secondary database, importing, 2 – 20
- secure databases
 - Import parameter MAPFILE, 2 – 23
 - Import parameter MLS, 2 – 23
 - Import parameter MLS_LISTLABELS, 2 – 23
- segments, temporary, FILE keyword,
SQL*Loader, 8 – 23
- SEQUENCE keyword, SQL*Loader, 5 – 47
- sequence numbers
 - cached, Export, 1 – 7
 - exporting, 1 – 7
 - for multiple tables, SQL*Loader, 5 – 48
 - generated by SEQUENCE clause
 - example, 4 – 8
 - SQL*Loader, 5 – 47
 - generated, not read, SQL*Loader, 5 – 39
 - no space used in bind array,
SQL*Loader, 5 – 67
 - setting column to a unique number,
SQL*Loader, 5 – 47
 - skipped, 1 – 7
- sequences, 2 – 7
- short records with missing data, SQL*Loader,
5 – 35
- SHORTINT, C Language datatype, 5 – 50
- SHOW, Import parameter, 1 – 3, 2 – 23
- SILENT, SQL*Loader command-line
parameter, 6 – 6
- single table load, discontinued,
SQL*Loader, 5 – 28
- single-byte character sets, Import, 2 – 36
- SINGLEROW, SQL*Loader, 5 – 36
- SKIP
 - control file keyword, SQL*Loader, 5 – 64
 - effect on RECNUM specification,
SQL*Loader, 5 – 46

- SQL*Loader, 5 – 29
- SQL*Loader command-line parameter, 6 – 7
- SMALLINT datatype, 5 – 50
 - specification, SQL*Loader, 5 – 10
 - SQL*Loader, 5 – 50
- snapshot log, Import, 2 – 11
- snapshots
 - importing, 2 – 10 to 2 – 12
 - log, Import, 2 – 10
 - master table, Import, 2 – 10
 - restoring dropped, Import, 2 – 11
 - restoring dropped snapshots, Import, 2 – 11
- SORTED INDEXES
 - case study, 4 – 20
 - direct path loads, SQL*Loader, 5 – 36
 - SQL*Loader, 8 – 14
- sorting
 - multiple column indexes, SQL*Loader, 8 – 14
 - optimum sort order, SQL*Loader, 8 – 14
 - presorting in direct path load, 8 – 13
 - SORTED INDEXES statement, SQL*Loader, 8 – 14
- SQL operators, applying to fields, SQL*Loader, 5 – 75
- SQL statements, executing, Import, 2 – 26
- SQL string
 - applying SQL operators to fields, SQL*Loader, 5 – 75
 - example of, 4 – 22
 - quotation marks, SQL*Loader, 5 – 13
- SQL*Loader
 - appending rows to tables, 5 – 26
 - BAD, command-line parameter, 6 – 3
 - bad file, 3 – 11
 - BADDN keyword, 5 – 19
 - BADFILE keyword, 5 – 19
 - basics, 3 – 2
 - bind arrays and performance, 5 – 63
 - BINDSIZE, command-line parameter, 6 – 3
 - BINDSIZE command-line parameter, 5 – 64
 - case studies, 4 – 1
 - associated files, 4 – 2
 - direct path load, 4 – 20 to 4 – 22
 - extracting data from a formatted report, 4 – 22 to 4 – 28
 - loading combined physical records, 4 – 11 to 4 – 14
 - loading data into multiple tables, 4 – 14 to 4 – 19
 - loading delimited, free-format files, 4 – 8 to 4 – 11
 - loading fixed-length data, 4 – 6 to 4 – 8
 - loading variable-length data, 4 – 4 to 4 – 6
 - preparing tables, 4 – 3
 - choosing which rows to load, 5 – 34
 - command-line arguments, 6 – 3 to 6 – 8
 - command-line parameters, 6 – 2
 - summary, 6 – 2
 - CONCATENATE keyword, 5 – 29
 - concepts, 3 – 1
 - concurrent sessions, 8 – 21
 - conditions for loading, 3 – 9
 - CONTINUE_LOAD keyword, 5 – 28
 - CONTINUEIF keyword, 5 – 29
 - CONTROL, command-line parameter, 6 – 4
 - control file, creating, 3 – 5
 - controlling memory usage, 5 – 18
 - conventional path loads, 8 – 2
 - DATA, command-line parameter, 6 – 4
 - data conversion, 3 – 12
 - data definition language
 - expanded syntax diagrams, 5 – 9
 - high-level syntax diagrams, 5 – 6
 - data definition language (DDL), 3 – 4
 - data definition language syntax, 5 – 4
 - data mapping concepts, 3 – 3
 - datafiles, specifying, 5 – 16
 - datatype specifications, 3 – 12
 - DB2 load utility. *See* DB2 load utility
 - DDL syntax reference, 5 – 1
 - delimiters, 3 – 7
 - DIRECT, command-line parameter, 6 – 4
 - DIRECT command line parameter, 8 – 7
 - direct path load, 8 – 3
 - DISCARD, command-line parameter, 6 – 4
 - discard file, 3 – 11
 - discarded records, 3 – 10
 - DISCARDFILE keyword, 5 – 22
 - DISCARDMAX, command-line parameter, 6 – 4
 - DISCARDMAX keyword, 5 – 23
 - DISCARDS keyword, 5 – 23
 - enclosed data, 3 – 7
 - ERRORS, command-line parameter, 6 – 4
 - errors caused by tabs, 5 – 41

SQL*Loader *continued*

- example sessions, 4 – 1
- exclusive access, 8 – 20
- fields, 3 – 7
- FILE, command-line parameter, 6 – 5
- filenames, 5 – 13
- high-water mark, 8 – 10
- index options, 5 – 36 to 5 – 37
- INTO TABLE statement, 5 – 33
- keywords and parameters, reference, 5 – 1
- LOAD, command-line parameter, 6 – 5
- load methods, 8 – 2
- loading data
 - conventional path method, 3 – 3
 - direct path method, 3 – 3
- loading data without files, 5 – 45
- loading LONG data, 5 – 56
- LOG, command-line parameter, 6 – 5
- log file
 - datafile information, 7 – 4
 - global information, 7 – 2
 - header information, 7 – 2
 - summary statistics, 7 – 5
 - table information, 7 – 3
 - table load information, 7 – 4
- log file entries, 7 – 1
- log files, 3 – 8
- mapping data, 3 – 3
- methods for loading data into tables, 5 – 25
- methods of loading data, 3 – 3
- multiple INTO TABLE statements, 5 – 42
- National Language Support, 5 – 24
- native datatype handling, 3 – 6
- NULLIF...BLANKS clause, case study, 4 – 20
- object names, 5 – 13
- PARALLEL, command-line parameter, 6 – 5
- parallel data loading, 8 – 21
- parallel loading, 8 – 21
- PARFILE, command-line parameter, 6 – 5
- READBUFFERS keyword, 5 – 18
- rejected records, 3 – 10
- Release 7.0 features, A – 6
- Release 7.1 features, A – 5 to A – 6
- replacing rows in tables, 5 – 26
- required privileges, 3 – 9
- reserved words, B – 2
- ROWS, command-line parameter, 6 – 6
- rows, inserting into tables, 5 – 26
- SILENT, command-line parameter, 6 – 6

- SINGLEROW index keyword, 5 – 36
- SKIP, command-line parameter, 6 – 7
- SKIP keyword, 5 – 29
- SORTED INDEXES
 - case study, 4 – 20
 - direct path loads, 5 – 36
- specifying a single load method for all tables, 5 – 27
- specifying columns, 5 – 39
- specifying data format, 3 – 4
- specifying data location, 3 – 4
- specifying datatypes, 5 – 49
- specifying field conditions, 5 – 37
- specifying fields, 5 – 39
- specifying more than one data file, 5 – 17
- suppressing messages, SILENT, 6 – 6
- terminated data, 3 – 7
- updating rows, 5 – 26
- USERID, command-line parameter, 6 – 7

SQL*Net

- connect string, 1 – 8
- Export/Import, 1 – 8, 2 – 33
- SQL/DS option (DB2 file format), not supported by SQL*Loader, C – 5
- starting column of a field, SQL*Loader, 5 – 40
- STATISTICS, export parameter, 1 – 16
- statistics
 - generating on imported data, 2 – 32
 - STATISTICS export parameter, 1 – 16
- storage parameters, 2 – 3
 - consolidating extents, interactive mode, 1 – 19
 - estimating export requirements, 1 – 10, 1 – 34
 - exporting tables, 1 – 17
 - OPTIMAL parameter, 2 – 3, 2 – 35
 - overriding, Import, 2 – 35
 - preallocating, direct path load, 8 – 13
 - temporary for a direct path load, 8 – 8
- stored functions, importing, 2 – 10
- stored packages, importing, 2 – 10
- stored procedures
 - direct path load, 8 – 20
 - importing, 2 – 10
- stream format records, SQL*Loader, 3 – 6, 5 – 70

- STREAM keyword, obsoleted, A – 8
- string comparisons, SQL*Loader, 5 – 9, 5 – 39
- synonyms
 - direct path load, 8 – 7
 - Export, 1 – 29
- syntax, data definition language,
 - SQL*Loader, 5 – 1
- syntax diagrams, SQL*Loader
 - expanded, 5 – 9
 - high-level, 5 – 6
- SYS, exporting objects owned by, 1 – 5
- SYS.INCEXP
 - CTIME column, Export, 1 – 31
 - EXPID column, Export, 1 – 31
 - export table, 1 – 31
 - incremental export tracking table, 1 – 29
 - incremental/cumulative export database table, 1 – 16
 - ITIME column, Export, 1 – 31
 - NAME column, Export, 1 – 31
 - OWNER# column, Export, 1 – 31
 - TYPE column, Export, 1 – 31
- SYS.INCFIL, 1 – 31
 - EXPDATE column, Export, 1 – 31
 - EXPFILE column, Export, 1 – 31
 - EXPID column, Export, 1 – 31
 - export table, 1 – 29
 - EXPTYPE column, Export, 1 – 31
 - EXPUSER column, Export, 1 – 31
 - incremental/cumulative export database table, 1 – 16
- SYS.INCVID
 - Export, 1 – 31
 - export table, 1 – 29
 - incremental/cumulative export database table, 1 – 16
- SYSDATE datatype
 - case study, 4 – 22
 - no space used in bind array,
 - SQL*Loader, 5 – 67
- SYSDATE keyword, SQL*Loader, 5 – 46
- system objects, importing, 2 – 9, 2 – 21
- system tables, incremental
 - export, 1 – 31 to 1 – 32

T

- TABLES, Export parameter, 1 – 17
- tables
 - appending rows to, SQL*Loader, 5 – 26
 - backing up and export, 1 – 29
 - continuing a multiple table load,
 - SQL*Loader, 5 – 28
 - continuing a single table load,
 - SQL*Loader, 5 – 28
 - definitions, creating before import, 2 – 31
 - exclusive access during direct path loads,
 - SQL*Loader, 8 – 20
 - export mode, 1 – 5
 - exporting, TABLES parameter, 1 – 17
 - exporting by name, interactive mode, 1 – 19
 - exporting data, Export, 1 – 18
 - exporting specified, 1 – 5
 - Import table names, 2 – 24
 - importing data, Import, 2 – 26
 - importing into existing tables, 2 – 26
 - initial extents and exporting, interactive mode, 1 – 19
 - insert triggers, direct path load, SQL*Loader, 8 – 18
 - inserting rows, SQL*Loader, 5 – 26
 - loading data into more than one table,
 - SQL*Loader, 5 – 42
 - loading data into tables, SQL*Loader, 5 – 25
 - loading method, for individual tables,
 - SQL*Loader, 5 – 33
 - loading more than one table, SQL*Loader.
 - See multiple table load
 - maintaining consistency, 1 – 13
 - manually ordering for import, 2 – 32
 - master table, Import, 2 – 10
 - name restrictions, Export, 1 – 12
 - name restrictions in parameter file,
 - Import, 2 – 17
 - object import order, Import, 2 – 2
 - partitioned in DB2, no Oracle equivalent, C – 5
 - predefining before Import, 2 – 31
 - replacing rows in, SQL*Loader, 5 – 26
 - size, USER_SEGMENT_VIEW,
 - Export, 1 – 10, 1 – 34

- specifying a single load method for all tables,
 - SQL*Loader, 5 – 27
- specifying export by tablename, 1 – 17
- system, incremental export, 1 – 31
- truncating, SQL*Loader, 5 – 27
- unclustered, 1 – 5
- updating existing rows, SQL*Loader, 5 – 26
- tablespaces
 - dropping during import, 2 – 34
 - Export, 1 – 29
 - export/import read-only, 2 – 35
 - pre-created, 2 – 20
 - read-only, Import, 2 – 4
 - reorganizing, Import, 2 – 34
- tabs
 - loading data files and, SQL*Loader, 5 – 41
 - trimming, SQL*Loader, 5 – 69
 - whitespace, SQL*Loader, 5 – 69
- temporary segments
 - FILE keyword, SQL*Loader, 8 – 23
 - not exported during backup, 1 – 29
- temporary storage in a direct path load, 8 – 8
- TERMINATED BY
 - SQL*Loader, 5 – 11, 5 – 58
 - WHITESPACE, SQL*Loader, 5 – 58, 5 – 72
 - with OPTIONALLY ENCLOSED BY,
 - SQL*Loader, 5 – 70
- terminated fields
 - and SQL*Loader, 3 – 7
 - specified with a delimiter,
 - SQL*Loader, 5 – 58, 5 – 70
- TOUSER, Import parameter, 2 – 24
- trailing, whitespace, trimming,
 - SQL*Loader, 5 – 72
- trailing blanks, loading with delimiters,
 - SQL*Loader, 5 – 61
- TRAILING NULLCOLS
 - case study, 4 – 22
 - control file keyword, SQL*Loader, 5 – 36
- triggers
 - database insert triggers, SQL*Loader, 8 – 18
 - Import, 2 – 10
 - permanently disabled, direct path load,
 - SQL*Loader, 8 – 20
 - replacing with integrity constraints,
 - SQL*Loader, 8 – 18
 - update triggers, SQL*Loader, 8 – 19
- trimming
 - summary, SQL*Loader, 5 – 73
 - trailing whitespace, SQL*Loader, 5 – 72
 - VARCHAR fields, SQL*Loader, 5 – 69
- TRUNCATE keyword, SQL*Loader, 5 – 26
- Trusted Oracle7 Server
 - DBHIGH system label, 2 – 9
 - error messages, 1 – 5
 - Export issues, 1 – 8
 - Import issues, 2 – 4, 2 – 15
 - labels, 1 – 3
 - listing labels in export file, A – 2
 - MLS
 - Export parameter, 1 – 16
 - import parameter, 2 – 23
 - MLS_LABEL_FORMAT, Export
 - parameter, 1 – 16
 - MLS_LISTLABELS import parameter, 2 – 23
 - MLS_MAPFILE import parameter, 2 – 23
 - privileges for SQL*Loader, 3 – 9
 - write access to labels, 2 – 8

U

- unclustered tables, 1 – 5
- unique indexes, Import, 2 – 22
- unique values, generating, SQL*Loader, 5 – 47
- uniqueness constraints, Import, 2 – 5
- uniqueness constraints, preventing errors
 - during import, 2 – 19
- UNLOAD (DB2 file format), not supported by
 - SQL*Loader, C – 5
- UNRECOVERABLE keyword,
 - SQL*Loader, 5 – 12, 8 – 15
- unsorted data, direct path load, SQL*Loader, 8 – 14
- updating rows in a table, SQL*Loader, 5 – 26
- user definitions, importing, 2 – 9
- user export mode, 1 – 5
- USER_SEGMENTS view, table sizes, 1 – 10, 1 – 34
- USERID
 - Export parameter, 1 – 17

- Import parameter, 2 – 24
- SQL*Loader command-line parameter, 6 – 7
- users, user definition, Import, A – 4

V

- VARCHAR datatype, 5 – 50
 - downgrading to Version 6.0, 1 – 40
 - specification, SQL*Loader, 5 – 10
 - SQL*Loader, 5 – 54
 - trimming whitespace, SQL*Loader, 5 – 69
- VARCHAR2 datatype, SQL*Loader, 5 – 49
- VARCHAR2 datatype, 2 – 36
- VARGRAPHIC datatype, 5 – 50
 - specification, SQL*Loader, 5 – 10
 - SQL*Loader, 5 – 53
- VARIABLE, SQL*Loader keyword, obsoleted, A – 8
- variable format records
 - SQL*Loader, 3 – 6, 5 – 70
 - vs. fixed, SQL*Loader, 3 – 6
- views
 - creating views necessary for Export, 1 – 10, 1 – 34
 - Export, 1 – 29
 - export parameter CONSISTENT, 1 – 13

W

- warning messages, 1 – 4
- WHEN clause
 - discards resulting from, SQL*Loader, 5 – 23
 - example, 4 – 14
 - field condition, SQL*Loader, 5 – 37
 - SQL*Loader, 5 – 34
- whitespace
 - included in a field, or not, SQL*Loader, 5 – 71
 - leading, SQL*Loader, 5 – 69
 - preserving, SQL*Loader, 5 – 74
 - terminating a field with, SQL*Loader, 5 – 72
 - trailing, 5 – 69
 - trimming, SQL*Loader, 5 – 69
 - WHITESPACE, SQL*Loader, 5 – 11
- WHITESPACE keyword, SQL*Loader, 5 – 58

Z

- ZONED datatype, 5 – 50
 - EXTERNAL format
 - SQL*Loader, 5 – 58
 - trimming whitespace, SQL*Loader, 5 – 69
 - length versus precision, SQL*Loader, 5 – 11
 - specification, SQL*Loader, 5 – 10
 - SQL*Loader, 5 – 51

Reader’s Comment Form

Oracle7 Server Utilities, Release 7.3 Part No. A32541–1

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the topic, chapter, and page number below:

Please send your comments to:

Oracle Server Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065
Fax: (415) 506–7228

If you would like a reply, please give your name, address, and telephone number below:

Thank you for helping us improve our documentation.