Oracle8*i*

Utilities

Release 8.1.5

February, 1999

Part No. A67792-01

ORACLE

Oracle 8i Utilities, Release 8.1.5

Part No. A67792-01

Primary Author: Jason Durbin

Contributors: Karleen Aghevli, Lee Barton, Allen Brumm, George Claborn, William Fisher, Paul Lane, Tracy Lee, Vishnu Narayana, Visar Nimani, Joan Pearson, Mike Sakayeda, James Stenois, Chao Wang , Gail Ymanaka, Hiro Yoshioka

Graphic Designer: Valarie Moore

Contents

2
Import

3 SQL*Loader Concepts

4 SQL*Loader Case Studies

5 SQL*Loader Control File Reference

6 SQL*Loader Command-Line Reference

Index

Send Us Your Comments

Oracle8i Utilities, Release 8.1.5

Part No. A67792-01

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 have any comments or suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- The Information Development department at infodev@us.oracle.com
- FAX - 650.506.7228. Attn: Oracle Utilities
- postal service:
  Server Technologies Documentation Manager
  Oracle Corporation
  500 Oracle Parkway
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

Preface

This manual describes how to use the Oracle8*i* utilities for data transfer, maintenance, and database administration.

Oracle8i Utilities contains information that describes the features and functionality of the Oracle8*i* and the Oracle8*i* Enterprise Edition products. Oracle8*i* and Oracle8*i* Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional.

For information about the differences between Oracle8*i* and the Oracle8*i* Enterprise Edition and the features and options that are available to you, see *Getting to Know Oracle8i and the Oracle8i Enterprise Edition*.

The Oracle Utilities

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

Audience

This manual is for database administrators (DBAs), application programmers, security administrators, system operators, and other Oracle users who 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
- create and maintain user-defined character sets (NLS Utilities) and other Oracle NLS data

To use this manual, you need a working knowledge of SQL and Oracle fundamentals, information that is contained in *Oracle8i Concepts*. In addition, SQL*Loader requires that you know how to use your operating system's file management facilities.

Note: This manual does not contain instructions for installing the utilities, which is operating system-specific. Installation instructions for the utilities can be found in your operating system-specific Oracle documentation.

How Oracle8*i* Utilities Is Organized

This manual is divided into three 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 export guidelines, export modes, interactive and command-line methods, parameter specifications, and describes Export object support. It also provides example Export sessions.

Chapter 2, "Import"

This chapter describes how to use Import to read data from Export files into an Oracle database. It discusses import guidelines, interactive and command-line methods, parameter specifications, and describes Import object support. 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 (including object support). It discusses input to SQL*Loader, database preparation, and output from SQL*Loader.

Chapter 4, "SQL*Loader Case Studies"

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, direct path loads, and loading objects, collections, and REF columns.

Chapter 5, "SQL*Loader Control File Reference"

This chapter describes the control file syntax you use to configure SQL*Loader and to describe to SQL*Loader how to map your data to Oracle format. It provides detailed syntax diagrams and information about specifying data files, tables and columns, the location of data, the type and format of data to be loaded, and more.

Chapter 6, "SQL*Loader Command-Line Reference"

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

Chapter 7, "SQL*Loader: Log File Reference"

This chapter describes the information contained in SQL*Loader log file output.

Chapter 8, "SQL*Loader: Conventional and Direct Path Loads"

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

Part III: Offline Database Verification Utility

Chapter 9, "Offline Database Verification Utility"

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

Appendix A, "SQL*Loader Reserved Words"

This appendix lists the words reserved for use only by SQL*Loader.

Appendix B, "DB2/DXT User Notes"

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 manual follows textual and typographic conventions explained in the following sections.

Text of the Manual

The following conventions are used in the text of this manual:

UPPERCASE Words	Uppercase text is used to call attention to command keywords, object names, parameters, filenames, and so on, for example: "If you create a private rollback segment, its name must be included in the ROLLBACK_SEGMENTS parameter in the PARAMETER file."
Italicized Words	Italicized words are used at the first occurrence and definition of a term, as in the following example: "A *database* is a collection of data to be treated as a unit. The general purpose of a database is to store and retrieve related information, as needed." Italicized words are used also to indicate emphasis, book titles, and to highlight names of performance statistics.

PL/SQL, SQL, and SQL*Plus commands and statements are displayed in a fixed-width font using the following conventions, separated from normal text as in the following example:

```
ALTER TABLESPACE users   ADD DATAFILE 'users2.ora' SIZE 50K;
```

Punctuation: , ' "	Example statements may include punctuation such as commas or quotation marks. All punctuation given in example statements is required. All statement examples end with a semicolon. Depending on the application in use, a semicolon or other terminator may or may not be required to end a statement.
UPPERCASE Words: INSERT, SIZE	Uppercase words in example statements indicate the keywords in Oracle SQL. However, when you issue statements, keywords are not case-sensitive.

Lowercase Words: emp, users2.ora	Lowercase words in example statements indicate words supplied only for the context of the example. For example, lowercase words may indicate the name of a table, column, or file. Some operating systems are case sensitive, so refer to your installation or user's manual to find whether you must pay attention to case.

We Welcome Your Comments

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

Oracle8*i* Documentation Manager
Oracle Corporation
500 Oracle Parkway
Redwood City, CA 94065
U.S.A.
FAX: 650.506.7228 Attn.: Oracle8*i* Utilities

You can also e-mail your comments to the Information Development department:

infodev@us.oracle.com

Part I

Export/Import

1

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 is stored outside the database, and it can be read into another Oracle database using the Import utility (described in Chapter 2, "Import"). This chapter covers the following topics:

- What is the Export Utility?
- Export Modes
- Using Export
- Export Parameters
- Example Export Sessions
- Using the Interactive Method
- Warning, Error, and Completion Messages
- Direct Path Export
- Incremental, Cumulative, and Complete Exports
- Network Considerations
- Character Set and NLS Considerations
- Considerations in Exporting Database Objects
- Transportable Tablespaces
- Using Different Versions of Export
- Creating Oracle Release 7 Export Files from an Oracle8i Database

What is the Export Utility?

Export provides a simple way for you to transfer data objects between Oracle databases, even if they reside on platforms with different hardware and software configurations. Export extracts the object definitions and table data from an Oracle database and stores them in an Oracle binary-format Export dump file located typically on disk or tape.

Such files can then be FTPed or physically transported (in the case of tape) to a different site and used, with the Import utility, to transfer data between databases that are on machines not connected via a network or as backups in addition to normal backup procedures.

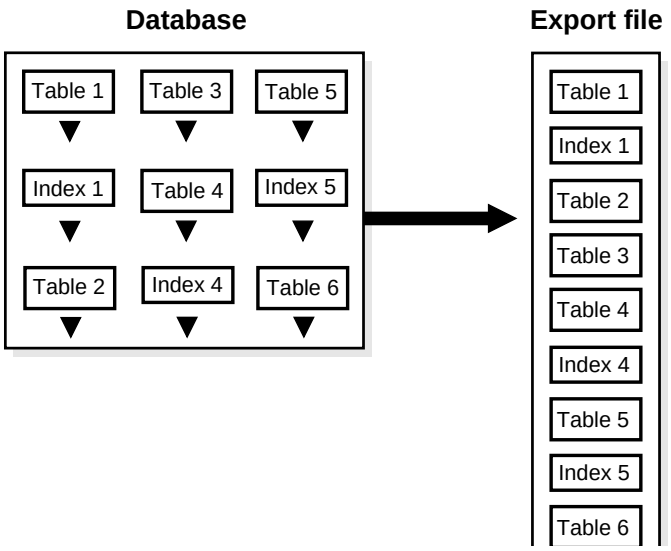
The Export and Import utilities can also facilitate certain aspects of Oracle Advanced Replication functionality like offline instantiation. See *Oracle8i Replication* for more information.

Note that, Export dump files can only be read by the Oracle utility, Import (see Chapter 2, "Import"). If you need to read load data from ASCII fixed-format or delimited files, see Part II, SQL*Loader of this manual.

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

> **Note:** If you are working with the Advanced Replication Option, refer to the information about migration and compatibility in *Oracle8i Replication*.

Figure 1–1 Exporting a Database



New in this Release

The following Export features are new as of this release of Oracle8*i*:

- Export of subpartitions. see Understanding Table-Level and Partition-Level Export on page 1-8.
- The ability to specify multiple dump files for an export command. See the parameters FILE on page 1-19 and FILESIZE on page 1-19.
- The ability to specify a query for the select statements that Export uses to unload tables. See QUERY on page 1-21.
- The maximum number of bytes in an export file on each volume of tape has been increased. See VOLSIZE on page 1-27.
- The ability to export tables containing LOBs and objects, even if direct path is specified on the command line. See Considerations in Exporting Database Objects on page 1-55.
- The ability to export and import precalculated optimizer statistics instead of recomputing the statistics at import time. (This feature is only applicable to certain exports and tables.) See STATISTICS on page 1-23.

- Developers of domain indexes can export application-specific metadata associated with an index using the new ODCIIndexGetMetadata method on the ODCIIndex interface. See the *Oracle8i Data Cartridge Developer's Guide* for more information.
- Export of procedural objects. The data definition language for procedural objects is now implemented as PL/SQL rather than SQL, for example, Advanced Queues and Resource Scheduler objects.
- Export of transportable tablespace metadata. See TRANSPORT_TABLESPACE on page 1-26.

Reading the Contents of an Export File

Export files are stored in Oracle-binary format. Export files generated by Export cannot be read by utilities other than Import. Export files created by Export cannot be read by earlier versions of the Import utility. However, Import can read files written by the current and previous releases of Export, but cannot read files in other formats. To load data from ASCII fixed-format or delimited files, see Part II of this manual for information about SQL*Loader.

You can, however, display the contents of an export file by using the Import SHOW parameter. For more information, see SHOW on page 2-28.

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 created a synonym for it.

Note also that the following schema names are reserved and will not be processed by Export:

- ORDSYS
- MDSYS
- CTXSYS
- ORDPLUGINS

Export Modes

The Export utility provides four modes of export. All users can export in table mode and user mode. A user with the EXP_FULL_DATABASE role (a *privileged user*) can export in table mode, user mode, tablespace, and full database mode. The database objects that are exported depend on the mode you choose. Tablespace mode allows you to move a set of tablespaces from one Oracle database to another. See Transportable Tablespaces on page 1-57 and the *Oracle8i Administrator's Guide* for details about how to move or copy tablespaces to another database. For an introduction to the transportable tablespaces feature, see *Oracle8i Concepts*.

See Export Parameters on page 1-14 for information on specifying each mode.

You can use conventional path Export or direct path Export to export in any of the first three modes. The differences between conventional path export and direct path Export are described in Direct Path Export on page 1-41.

Table 1–1 shows the objects that are exported and imported in each mode.

Table 1–1 Objects Exported and Imported in Each Mode

Table Mode	User Mode	Full Database Mode	Tablespace Mode
For each table in the TABLES list, users can export and import:	**For each user in the Owner list, users can export and import:**	**Privileged users can export and import all database objects except those owned by SYS, and those in the ORDSYS, CTXSYS, MDSYS and ORDPLUGINS schemas:**	**For each tablespace in the TABLESPACES list, a privileged user can export and import the DDL for the following objects:**
pre-table procedural actions	foreign function libraries	tablespace definitions	cluster definitions
object type definitions used by table	object types	profiles	
table definitions	database links	user definitions	**For each table within the current tablespace, the following objects' DDL is included:**
pre-table actions	sequence numbers	roles	
table data by partition	cluster definitions	system privilege grants	pre-table procedural actions

Table 1–1 Objects Exported and Imported in Each Mode (Cont.)

Table Mode	User Mode	Full Database Mode	Tablespace Mode
nested table data	**In addition, for each table that the specified user owns, users can export and import:**	role grants default roles tablespace quotas	object type definitions used by the table
owner's table grants owner's table indexes table constraints (primary, unique, check)	pre-table procedural actions	resource costs	table definition (table rows are not included)
analyze tables	object type definitions used by table	rollback segment definitions	pre-table actions
column and table comments	table definitions	database links	table grants
auditing information	pre-table actions	sequence numbers	table indexes
security policies for table	table data by partition	all directory aliases	table constraints (primary, unique, check)
table referential constraints	nested table data	application contexts	column and table comments
owner's table triggers	owner's table grants	all foreign function libraries	referential integrity constraints
post-table actions	owner's table indexes **(1)**	all object types all cluster definitions	bitmap indexes (**note:** not functional or domain indexes)
post-table procedural actions and objects	table constraints (primary, unique, check)	default and system auditing	post-table actions
	analyze table		triggers
In addition, privileged users can export and import:	column and table comments	**For each table, the privileged user can export and import**:	post-table procedural actions and objects
triggers owned by other users	auditing information	pre-table procedural actions	
indexes owned by other users	security policies for table	object type definitions used by table	
	table referential constraints	table definitions	

Table 1–1 Objects Exported and Imported in Each Mode (Cont.)

Table Mode	User Mode	Full Database Mode	Tablespace Mode
	private synonyms	pre-table actions	
	user views	table data by partition	
	user stored procedures, packages, and functions	nested table data	
	referential integrity constraints	table grants	
	operators	table indexes	
	triggers (**2)**	table constraints (primary, unique, check)	
	post-table actions	analyze table	
	indextypes	column and table comments	
	snapshots and materialized views	auditing information	
	snapshot logs	all referential integrity constraints	
	job queues	all synonyms	
	refresh groups	all views	
	dimensions	all stored procedures, packages, and functions	
	procedural objects	post-table actions	
	post-table procedural actions and objects	operators	
	post-schema procedural actions and objects	indextypes	
		post-table actions	
		all triggers	
		analyze cluster	
		all snapshots and materialized views	
		all snapshot logs	
		all job queues	

Table 1–1 Objects Exported and Imported in Each Mode (Cont.)

Table Mode	User Mode	Full Database Mode	Tablespace Mode
		all refresh groups and children	
		dimensions	
		password history	
		system auditing	
		post-table procedural actions and objects	
		post-schema procedural actions and objects	

1. Non-privileged users can export and import only indexes they own on tables they own. They cannot export indexes they own that are on tables owned by other users, nor can they export indexes owned by other users on their own tables. Privileged users can export and import indexes on the specified users' tables, even if the indexes are owned by other users. Indexes owned by the specified user on other users' tables are not included, unless those other users are included in the list of users to export.
2. Non-privileged and privileged users can export and import all triggers owned by the user, even if they are on tables owned by other users.

Understanding Table-Level and Partition-Level Export

In table-level Export, an entire partitioned or non-partitioned table, along with its indexes and other table-dependent objects, is exported. All the partitions and subpartitions of a partitioned table are exported. (This applies to both direct path Export and conventional path Export.) All Export modes (full, user, table, transportable tablespace) support table-level Export.

In partition-level Export, the user can export one or more specified partitions or subpartitions of a table. Full database, user, and transportable tablespace mode Export do not support partition-level Export; only table mode Export does. Because incremental Exports (incremental, cumulative, and complete) can be done only in full database mode, partition-level Export cannot be specified for incremental exports.

In all modes, partitioned data is exported in a format such that partitions or subpartitions can be imported selectively.

For information on how to specify a partition-level Export, see TABLES on page 1-24.

Using Export

This section describes how to use the Export utility, including what you need to do before you begin exporting and how to invoke Export.

Before Using Export

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

> **Note:** 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 needs to be run only once on a database. You do not need to run it again before you perform the export. The script performs the following tasks to prepare the database for Export:

- creates the necessary export views
- assigns all necessary privileges to the EXP_FULL_DATABASE role
- assigns EXP_FULL_DATABASE to the DBA role

Before you run Export, ensure that there is sufficient disk or tape storage space to write the export file. If there is not enough space, Export terminates 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 of the Oracle data dictionary. The following query displays disk usage for all tables:

```
select sum(bytes) from user_segments where segment_type='TABLE';
```

The result of the query does not include disk space used for data stored in LOB (large object) or VARRAY columns or partitions.

See the *Oracle8i Reference* for more information about dictionary views.

Invoking Export

You can invoke Export in one of the following ways:

- Enter the following 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 parameter files. This is the recommended method.

- Enter the command

  exp username/password

  followed by the parameters you need.

  Note: The number of parameters cannot exceed the maximum length of a command line on the 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 method provides less functionality than the parameter-driven method. It exists for backward compatibility.

You can use a combination of the first and second options. That is, you can list parameters both in the parameters file and on the command line. In fact, you can specify the same parameter in both places. The position of the PARFILE parameter and other parameters on the command line determines what parameters override others. For example, assume the parameters file `params.dat` contains the parameter INDEXES=Y and Export is invoked with the following line:

```
exp system/manager PARFILE=params.dat INDEXES=N
```

In this case, because `INDEXES=N` occurs *after* `PARFILE=params.dat`, `INDEXES=N` overrides the value of the INDEXES parameter in the PARFILE.

You can specify the username and password in the parameter file, although, for security reasons, this is not recommended. If you omit the username / password combination, Export prompts you for it.

See Export Parameters on page 1-14 for descriptions of the parameters.

To see how to specify an export from a database that is not the default database, refer to Exporting and Importing with Net8 on page 1-52.

Invoking Export as SYSDBA

Typically, you should not need to invoke Export as SYSDBA. You may need to do so at the request of Oracle technical support. To invoke Export as SYSDBA, use the following syntax:

```
exp username/password AS SYSDBA
```

or, optionally

```
exp username/password@instance AS SYSDBA
```

> **Note:** Since the string "AS SYSDBA" contains a blank, most operating systems require that entire string 'username/password AS SYSDBA' be placed in quotes or marked as a literal by some method. Note that some operating systems also require that quotes on the command line be escaped as well. Please see your operating system-specific documentation for information about special and reserved characters on your system. Note that if either the username or password is omitted, Export will prompt you for it.

If you prefer to use the Export interactive mode, please see Interactively Invoking Export as SYSDBA on page 1-36 for more information.

Getting Online Help

Export provides online help. Enter `exp help=y` on the command line to see a help screen like the one shown below.

When you invoke the help display, you will see something similar to the following:

```
> exp help=y

Export: Release 8.1.5.0.0 - Production on Wed Oct 28 15:00:10 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.


You can let Export prompt you for parameters by entering the EXP
command followed by your username/password:

     Example: EXP SCOTT/TIGER

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

```
     Format:  EXP KEYWORD=value or KEYWORD=(value1,value2,...,valueN)
     Example: EXP SCOTT/TIGER GRANTS=Y TABLES=(EMP,DEPT,MGR)
              or TABLES=(T1:P1,T1:P2), if T1 is partitioned table

USERID must be the first parameter on the command line.

Keyword  Description (Default)       Keyword      Description (Default)
------------------------------------------------------------------------
USERID   username/password           FULL         export entire file (N)
BUFFER   size of data buffer         OWNER        list of owner usernames
FILE     output files (EXPDAT.DMP)   TABLES       list of table names
COMPRESS import into one extent (Y)  RECORDLENGTH length of IO record
GRANTS   export grants (Y)           INCTYPE      incremental export type
INDEXES  export indexes (Y)          RECORD       track incr. export (Y)
ROWS     export data rows (Y)        PARFILE      parameter filename
CONSTRAINTS export constraints (Y)   CONSISTENT   cross-table consistency
LOG      log file of screen output   STATISTICS   analyze objects (ESTIMATE)
DIRECT   direct path (N)             TRIGGERS     export triggers (Y)
FEEDBACK display progress every x rows (0)
FILESIZE maximum size of each dump file
QUERY    select clause used to export a subset of a table
VOLSIZE  number of bytes to write to each tape volume

The following keywords only apply to transportable tablespaces
TRANSPORT_TABLESPACE export transportable tablespace metadata (N)
TABLESPACES list of tablespaces to transport

Export terminated successfully without warnings.
```

Specifying Export Parameters on the Command Line or in the Parameter File

You can specify Export parameters in three ways: from a command-line entry, allow Export to prompt you for parameter values, or in the parameter file.

Command-Line Parameter Entry

You can specify all valid parameters and their values from the command line using the following syntax:

```
exp KEYWORD=value
```

or

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

Export Parameter Prompts

If you prefer to let Export prompt you for the value of each parameter, you can use the following syntax:

```
exp username/password
```

Export will display each parameter with a request for you to enter a value.

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 one of the following:

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

The following example shows a partial parameter file listing:

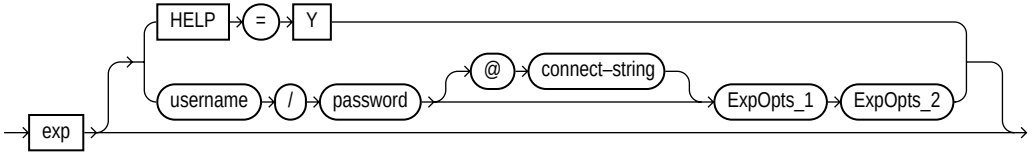
```
FULL=Y
FILE=DBA.DMP
GRANTS=Y
INDEXES=Y
CONSISTENT=Y
```

> **Additional Information:** The maximum size of the parameter file may be limited by the operating system. The name of the parameter file is subject to the file naming conventions of the operating system. See your Oracle operating system-specific documentation for more information.

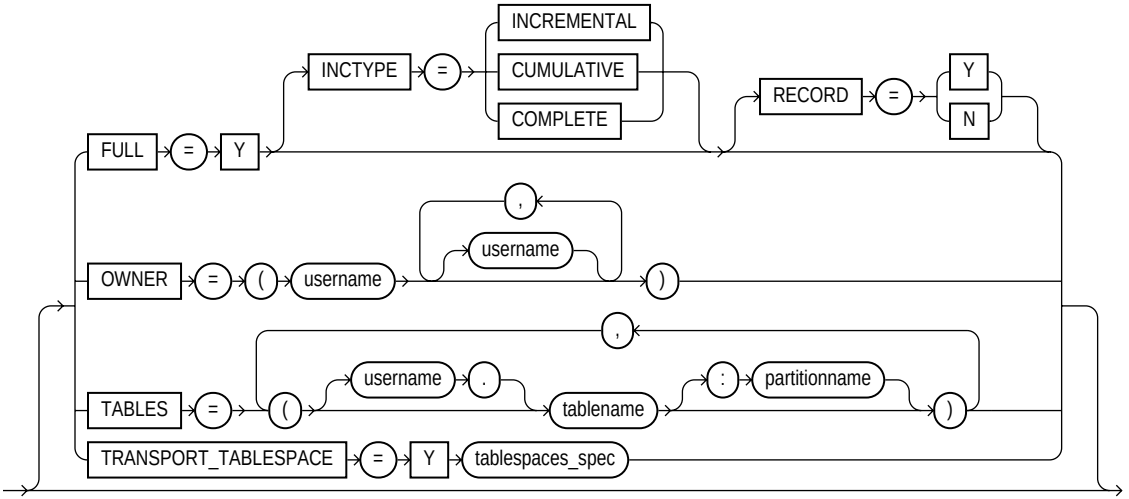
You can add comments to the parameter file by preceding them with the pound (#) sign. Export ignores all characters to the right of the pound (#) sign.

Export Parameters

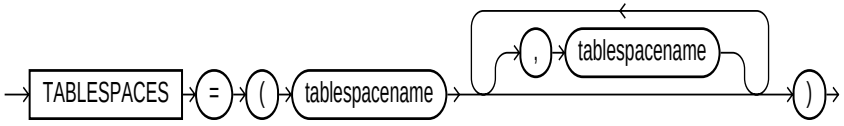
The following three diagrams show the syntax for the parameters that you can specify in the parameter file or on the command line. The remainder of this section describes each parameter.



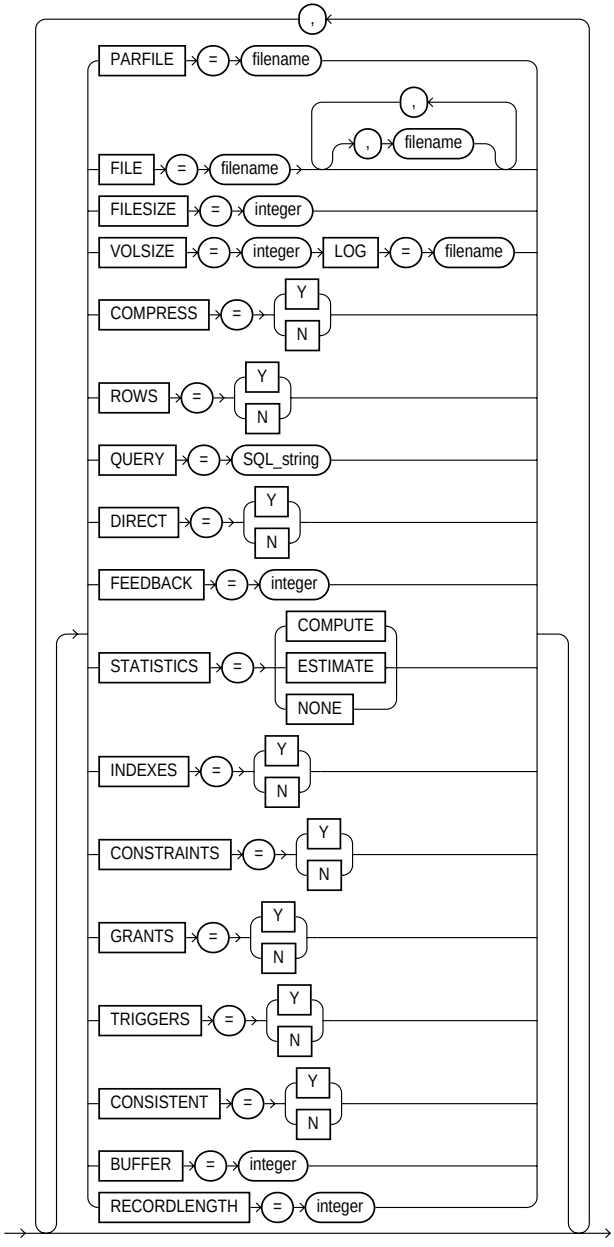
expopts_1



tablespaces_spec



Expopts_2


,
PARFILE
=
filename
,
,
filename
FILE
=
filename
FILESIZE
=
integer
VOLSIZE
=
integer
LOG
=
filename
COMPRESS
=
Y
N
ROWS
=
Y
N
QUERY
=
SQL_string
DIRECT
=
Y
N
FEEDBACK
=
integer
STATISTICS
=
COMPUTE
ESTIMATE
NONE
INDEXES
=
Y
N
CONSTRAINTS
=
Y
N
GRANTS
=
Y
N
TRIGGERS
=
Y
N
CONSISTENT
=
Y
N
BUFFER
=
integer
RECORDLENGTH
=
integer

BUFFER

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

Specifies the size, in bytes, of the buffer used to fetch rows. As a result, this parameter determines the maximum number of rows in an array fetched by Export. Use the following formula to calculate the buffer size:

```
buffer_size = rows_in_array * maximum_row_size
```

If you specify zero, the Export utility fetches only one row at a time.

Tables with LONG, LOB, BFILE, REF, ROWID, LOGICAL ROWID, DATE, or type columns are fetched one row at a time.

> **Note:** The BUFFER parameter applies only to conventional path Export. It has no effect on a direct path Export.

COMPRESS

Default: Y

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

The default, COMPRESS=Y, causes Export to flag table data for consolidation into one initial extent upon Import. If extent sizes are large (for example, because of the PCTINCREASE parameter), the allocated space will be larger than the space required to hold the data.

If you specify COMPRESS=N, Export uses the current storage parameters, including the values of initial extent size and next extent size. The values of the parameters may be the values specified in the CREATE TABLE or ALTER TABLE statements or the values modified by the database system. For example, the NEXT extent size value may be modified if the table grows and if the PCTINCREASE parameter is nonzero.

Note: Although the actual consolidation is performed upon import, you can specify the COMPRESS parameter only when you export, not when you import. The Export utility, not the Import utility, generates the data definitions, including the storage parameter definitions. Thus, if you specify COMPRESS=Y when you export, you can import the data in consolidated form only.

> **Note:** LOB data is not compressed. For LOB data, the original values of initial extent size and next extent size are used.

CONSISTENT

Default: N

Specifies whether or not Export uses the SET TRANSACTION READ ONLY statement to ensure that the data seen by Export is consistent to a single point in time and does not change during the execution of the export command. You should specify CONSISTENT=Y when you anticipate that other applications will be updating the target data after an export has started.

If you specify CONSISTENT=N (the default), each table is usually exported in a single transaction. However, if a table contains nested tables, the outer table and each inner table are exported as separate transactions. If a table is partitioned, each partition is exported as a separate transaction.

Therefore, if nested tables and partitioned tables are being updated by other applications, the data that is exported could be inconsistent. To minimize this possibility, export those tables at a time when updates are not being done.

The following chart shows a sequence of events by two users: USER1 exports partitions in a table and USER2 updates data in that table.

Time Sequence	USER1	USER2
1	Begins export of TAB:P1	
2		Updates TAB:P2 Updates TAB:P1 Commit transaction
3	Ends export of TAB:P1	
4	Exports TAB:P2	

If the export uses CONSISTENT=Y, none of the updates by USER2 are written to the export file.

If the export uses CONSISTENT=N, the updates to TAB:P1 are not written to the export file. However, the updates to TAB:P2 are written to the export file because the update transaction is committed before the export of TAB:P2 begins. As a result, USER2's transaction is only partially recorded in the export file, making it inconsistent.

If you use CONSISTENT=Y and the volume of updates is large, the rollback segment will be large. In addition, the export of each table will be slower because the rollback segment must be scanned for uncommitted transactions.

Keep in mind the following points about using CONSISTENT=Y:

- To minimize the time and space required for such exports, you should export tables that need to remain consistent separately from those that do not.

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

- To reduce the chances of encountering a "snapshot too old" error, export the minimum number of objects that must be guaranteed consistent.

  The "snapshot too old" 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, a "snapshot too old" error results.

  To avoid this error, you should minimize the time taken by a read-consistent export. (Do this 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.

  Note: You cannot specify CONSISTENT=Y with an incremental export.

CONSTRAINTS

Default: Y

Specifies whether or not the Export utility exports table constraints.

DIRECT

Default: N

Specifies whether you use direct path or conventional path Export.

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.

For more information about direct path exports, see Direct Path Export on page 1-41.

FEEDBACK

Default: 0 (zero)

Specifies that Export should display a progress meter in the form of a dot for *n* number of rows exported. For example, if you specify FEEDBACK=10, Export displays a dot each time 10 rows are exported. The FEEDBACK value applies to all tables being exported; it cannot be set on a per-table basis.

FILE

Default: expdat.dmp

Specifies the names of the export files. The default extension is .dmp, but you can specify any extension. Since Export supports multiple export files (see the parameter FILESIZE on page 1-19), you can specify multiple filenames to be used.

When Export reaches the value you have specified for the maximum FILESIZE, Export stops writing to the current file, opens another export file with the next name specified by the parameter FILE and continues until complete or the maximum value of FILESIZE is again reached. If you do not specify sufficient export filenames to complete the export, Export will prompt you to provide additional filenames.

FILESIZE

Export supports writing to multiple export files and Import can read from multiple export files. If you specify a value (byte limit) for the FILESIZE parameter, Export will write only the number of bytes you specify to each dump file.

When the amount of data Export must write exceeds the maximum value you specified for FILESIZE, it will get the name of the next export file from the FILE parameter (see FILE on page 1-19 for more information) or, if it has used all the names specified in the FILE parameter, it will prompt you to provide a new export filename. If you do not specify a value for FILESIZE (note that a value of 0 is equivalent to not specifying FILESIZE), then Export will write to only one file, regardless of the number of files specified in the FILE parameter.

> **Note:** If your export file(s) requirements exceed the available diskspace, Export will abort and you will have to repeat the Export after making sufficient diskspace available.

The FILESIZE parameter has a maximum value equal to the maximum value that can be stored in 64 bits.

> **Note:** The maximum value that can be stored in a file is operating system dependent. You should verify this maximum value in your operating-system specific documentation before specifying FILESIZE. You should also ensure that the file size you specify for Export is supported on the system on which Import will run.

The FILESIZE value can also be specified as a number followed by K (number of kilobytes). For example, FILESIZE=2K is the same as FILESIZE=2048. Similarly, M specifies megabytes (1024 * 1024) while G specifies gigabytes (1024**3). B remains the shorthand for bytes; the number is not multiplied to get the final file size (FILESIZE=2048b is the same as FILESIZE=2048)

FULL

Default: N

Indicates that the Export is a full database mode Export (that is, it exports the entire database.) Specify FULL=Y to export in full database mode. You need the EXP_FULL_DATABASE role to export in this mode.

GRANTS

Default: Y

Specifies whether or not the Export utility exports object grants. The object grants that are exported depend on whether you use full database 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. Note that system privilege grants are always exported.

HELP

Default: N

Displays a help message with descriptions of the Export parameters.

INCTYPE

Default: none

Specifies the type of incremental Export. The options are COMPLETE, CUMULATIVE, and INCREMENTAL. See Incremental, Cumulative, and Complete Exports on page 1-44 for more information.

INDEXES

Default: Y

Specifies whether or not the Export utility exports indexes.

LOG

Default: none

Specifies a file name to receive informational and error messages. For example:

```
exp system/manager LOG=export.log
```

If you specify this parameter, messages are logged in the log file *and* displayed to the terminal display.

OWNER

Default: undefined

Indicates that the Export is a user-mode Export and lists the users whose objects will be exported. If the user initiating the export is the DBA, multiple users may be listed.

PARFILE

Default: undefined

Specifies a filename for a file that contains a list of Export parameters. For more information on using a parameter file, see Specifying Export Parameters on the Command Line or in the Parameter File on page 1-12.

QUERY

Default: none

This parameter allows you to select a subset of rows from a set of tables when doing a table mode export. The value of the query parameter is a string that contains a WHERE clause for a SQL select statement which will be applied to all tables (or table partitions) listed in the TABLE parameter.

For example, if user SCOTT wants to export only those employees whose job title is SALESMAN and whose salary is greater than 1600, he could do the following (note that this example is Unix-based):

```
exp scott/tiger tables=emp query=\"where job=\'SALESMAN\' and sal\<1600\"
```

> **Note:** Since the value of the QUERY parameter contains blanks, most operating systems require that the entire strings `where job=\'SALESMAN\'` and `sal\<1600` be placed in double quotes or marked as a literal by some method. Also note that operating system reserved characters need to be escaped as are single quotes, double quotes and '<' in the Unix example above. Please see your operating system-specific documentation for information about special and reserved characters on your system.

When executing this command, Export builds a select statement similar to this:

```
SELECT * FROM EMP where job='SALESMAN' and sal <1600;
```

The QUERY is applied to all tables (or table partitions) listed in the TABLE parameter. So, for example,

```
exp scott/tiger tables=emp,dept query=\"where job=\'SALESMAN\' and sal\<1600\"
```

will unload rows in both EMP and DEPT that match the query.

Again, the SQL statements that Export executes are similar to these:

```
SELECT * FROM EMP where where job='SALESMAN' and sal <1600;

SELECT * FROM DEPT where where job='SALESMAN' and sal <1600;
```

Restrictions

- The parameter QUERY cannot be specified for full, user, or transportable tablespace mode exports.
- The parameter QUERY must be applicable to all specified tables.
- The parameter QUERY cannot be specified in a direct path export (DIRECT=Y)
- The parameter QUERY cannot be specified for tables with inner nested tables.
- You will not be able to determine from the contents of the export file whether the data is the result of a QUERY export.

RECORD

Default: Y

Indicates whether or not to record an incremental or cumulative export in the system tables SYS.INCEXP, SYS.INCFIL, and SYS.INCVID. For information about these tables, see System Tables on page 1-50.

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.

If you do not define this parameter, it defaults to your platform-dependent value for BUFSIZ. For more information about the BUFSIZ default value, see your operating system-specific documentation.

You can set RECORDLENGTH to any value equal to or greater than your system's BUFSIZ. (The highest value is 64KB.) Changing the RECORDLENGTH parameter affects only the size of data that accumulates before writing to the disk. It does not affect the operating system file block size.

> **Note:** You can use this parameter to specify the size of the Export I/O buffer.
>
> **Additional Information:** See your Oracle operating system-specific documentation to determine the proper value or to create a file with a different record size.

ROWS

Default: Y

Specifies whether or not the rows of table data are exported.

STATISTICS

Default: ESTIMATE

Specifies the type of database optimizer statistics to generate when the exported data is imported. Options are ESTIMATE, COMPUTE, and NONE. See the *Oracle8i Concepts* manual for information about the optimizer and the statistics it uses. See also the Import parameter RECALCULATE_STATISTICS on page 2-27 and Importing Statistics on page 2-63.

In some cases, Export will place the precomputed statistics in the export file as well as the ANALYZE commands to regenerate the statistics.

However, the precomputed optimizer statistics will not be used at export time if:

- A table has indexes with system generated names (including LOB indexes)
- A table has columns with system generated names
- There were row errors while exporting
- The client character set or NCHARSET does not match server character set or NCHARSET
- You have specified a QUERY clause
- Only certain partitions or subpartitions are to be exported
- Tables have indexes based upon constraints that have been analyzed (check, unique, and primary key constraints)
- Tables have indexes with system generated names that have been analyzed (IOTs, nested tables, type tables which have specialized constraint indexes)

  Note: Specifying ROWS=N does not preclude saving the precomputed statistics in the Export file. This allows you to tune plan generation for queries in a non-production database using statistics from a production database.

TABLES

Default: none

Specifies that the Export is a table-mode Export and lists the table names and partition and subpartition names to export. You can specify the following when you specify the name of the table:

- *schema* specifies the name of the user's schema from which to export the table or partition. Note that the schema names ORDSYS, MDSYS, CTXSYS, and ORDPLUGINS are reserved by Export.
- *tablename* specifies the name of the table to be exported. Table-level Export lets you export entire partitioned or non-partitioned tables. If a table in the list is partitioned and you do not specify a partition name, all its partitions and subpartitions are exported.
- *partition* or *subpartition name* indicates that the export is a partition-level Export. Partition-level Export lets you export one or more specified partitions or subpartitions within a table.

The syntax you use to specify the above is in the form:

```
schema.tablename:partitionname
schema.tablename:subpartitionname
```

If you use tablename:partition name, the specified table must be partitioned, and partition-name must be the name of one of its partitions or subpartitions.

See Example Export Session Using Partition-Level Export on page 1-33 for several examples of partition-level exports.

> **Additional Information:** Some operating systems, such as UNIX, require that you use escape characters before special characters, such as a parenthesis, so that the character is not treated as a special character. On UNIX, use a backslash (\) as the escape character, as shown in the following example:
>
> ```
> TABLES=\(EMP,DEPT\)
> ```

Table-Name Restrictions

Table names specified on the command line cannot include a pound (#) sign, unless the table name is enclosed in quotation marks. Similarly, in the parameter file, if a table name includes a pound (#) sign, the Export utility interprets the rest of the line as a comment, unless the table name is enclosed in quotation marks.

For example, if the parameter file contains the following line, Export interprets everything on the line after EMP# as a comment, and therefore does not export the tables DEPT and MYDATA:

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

However, given the following line, the Export utility exports all three tables:

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

> **Attention:** When you specify the table name using quotation marks, the name is case sensitive. The name must exactly match the table name stored in the database. By default, table names in a database are stored as uppercase.

In the previous example, a table named EMP# is exported, not a table named emp#. Because the tables DEPT and MYDATA are not specified in quotation marks, the names are not case sensitive.

> **Additional Information:** Some operating systems require single quotation marks rather than double quotation marks, or vice versa; see your Oracle operating system-specific documentation. Different operating systems also have other restrictions on table naming.

For example, the UNIX C shell attaches a special meaning to a dollar sign ($) or pound sign (#) (or certain other special characters). You must use escape characters to get such characters in the name past the shell and into Export.

TABLESPACES

Default: none

When TRANSPORT_TABLESPACE is specified as Y, use this parameter to provide a list of the tablespaces to be exported from the database into the export file.

See Transportable Tablespaces on page 1-57 for more information.

TRANSPORT_TABLESPACE

Default: N

When specified as Y, this parameter enables the export of transportable tablespace metadata. See the *Oracle8i Administrator's Guide* and *Oracle8i Concepts* for more information.

USERID

Default: none

Specifies the `username/password` (and optional connect string) of the user initiating the export. If you omit the password Export will prompt you for it.

USERID can also be:

```
username/password AS SYSDBA
```

or

```
username/password@instance AS SYSDBA
```

See Invoking Export as SYSDBA on page 1-11 for more information. Note also that your operating system may require you to treat AS SYSDBA as a special string requiring you to enclose the entire string in quotes as described on page 1-11.

Optionally, you can specify the *@connect_string* clause for Net8. See the user's guide for your Net8 protocol for the exact syntax of *@connect_string*. See also *Oracle8i Distributed Database Systems*.

VOLSIZE

Specifies the maximum number of bytes in an export file on each volume of tape.

The VOLSIZE parameter has a maximum value equal to the maximum value that can be stored in 64 bits. See your Operating system-specific documentation for more information.

The VOLSIZE value can be specified as number followed by K (number of kilobytes). For example, VOLSIZE=2K is the same as VOLSIZE=2048. Similarly, M specifies megabytes (1024 * 1024) while G specifies gigabytes (1024**3). B remains the shorthand for bytes; the number is not multiplied to get the final file size (VOLSIZE=2048b is the same as VOLSIZE=2048)

Parameter Interactions

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

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

Similarly, OWNER and TABLE conflict 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.

Example Export Sessions

The following examples show you how to use the command line and parameter file methods in the full database, user, and table modes.

Example Export Session in Full Database Mode

Only users with the DBA role or the EXP_FULL_DATABASE role can export in full database mode. In this example, an entire database is exported to the file dba.dmp with all GRANTS and all data.

Parameter File Method

```
> exp system/manager parfile=params.dat
```

The params.dat file contains the following information:

```
FILE=dba.dmp
GRANTS=y
FULL=y
ROWS=y
```

Command-Line Method

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

Export Messages

```
Export: Release 8.1.5.0.0 - Production on Fri Oct 30 09:34:00 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production
Export done in WE8DEC character set and WE8DEC NCHAR character set

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 directory aliases
. exporting context namespaces
. exporting foreign function library names
. exporting object type definitions
. exporting system procedural objects and actions
. exporting pre-schema procedural objects and actions
. exporting cluster definitions
. about to export SYSTEM's tables via Conventional Path ...
. . exporting table                      DEF$_AQCALL          0 rows exported
. . exporting table                     DEF$_AQERROR          0 rows exported
. . exporting table                    DEF$_CALLDEST          0 rows exported
. . exporting table                DEF$_DEFAULTDEST          0 rows exported
. . exporting table                DEF$_DESTINATION          0 rows exported
```

```
. . exporting table                      DEF$_ERROR          0 rows exported
. . exporting table                        DEF$_LOB          0 rows exported
. . exporting table                     DEF$_ORIGIN          0 rows exported
. . exporting table                 DEF$_PROPAGATOR          0 rows exported
. . exporting table       DEF$_PUSHED_TRANSACTIONS          0 rows exported
. . exporting table                  DEF$_TEMP$LOB          0 rows exported
. . exporting table        SQLPLUS_PRODUCT_PROFILE          0 rows exported
. about to export OUTLN's tables via Conventional Path ...
. . exporting table                            OL$          0 rows exported
. . exporting table                      OL$HINTS          0 rows exported
. about to export DBSNMP's tables via Conventional Path ...
. about to export SCOTT's tables via Conventional Path ...
. . exporting table                          BONUS          0 rows exported
. . exporting table                           DEPT          4 rows exported
. . exporting table                            EMP         14 rows exported
. . exporting table                       SALGRADE          5 rows exported
. about to export ADAMS's tables via Conventional Path ...
. about to export JONES's tables via Conventional Path ...
. about to export CLARK's tables via Conventional Path ...
. about to export BLAKE's tables via Conventional Path ...
. . exporting table                           DEPT          8 rows exported
. . exporting table                        MANAGER          4 rows exported
. exporting referential integrity constraints
. exporting synonyms
. exporting views
. exporting stored procedures
. exporting operators
. exporting indextypes
. exporting bitmap, functional and extensible indexes
. exporting posttables actions
. exporting triggers
. exporting snapshots
. exporting snapshot logs
. exporting job queues
. exporting refresh groups and children
. exporting dimensions
. exporting post-schema procedural objects and actions
. exporting user history table
. 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 or more database users. For example, a DBA may want to back up the tables of deleted users 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. In this example, user SCOTT is exporting his own tables.

Parameter File Method

```
> exp scott/tiger parfile=params.dat
```

The params.dat file contains the following information:

```
FILE=scott.dmp
OWNER=scott
GRANTS=y
ROWS=y
COMPRESS=y
```

Command-Line Method

```
> exp scott/tiger file=scott.dmp owner=scott grants=Y rows=Y compress=y
```

Export Messages

```
Export: Release 8.1.5.0.0 - Production on Fri Oct 30 09:35:33 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production
Export done in WE8DEC character set and WE8DEC NCHAR character set
. exporting pre-schema procedural objects and actions
. exporting foreign function library names for user SCOTT
. exporting object type definitions for user SCOTT
About to export SCOTT's objects ...
. exporting database links
. exporting sequence numbers
. exporting cluster definitions
. about to export SCOTT's tables via Conventional Path ...
. . exporting table                          BONUS          0 rows exported
. . exporting table                           DEPT          4 rows exported
. . exporting table                            EMP         14 rows exported
```

```
. . exporting table                          SALGRADE          5 rows exported
. exporting synonyms
. exporting views
. exporting stored procedures
. exporting operators
. exporting referential integrity constraints
. exporting triggers
. exporting indextypes
. exporting bitmap, functional and extensible indexes
. exporting posttables actions
. exporting snapshots
. exporting snapshot logs
. exporting job queues
. exporting refresh groups and children
. exporting dimensions
. exporting post-schema procedural objects and 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 rows are 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 table mode to export tables from any user's schema by specifying `TABLES=schema.table`

If `schema` is not specified, Export defaults to the previous schema from which an object was exported. If there is not a previous object, Export defaults to the exporter's schema. In the following example, Export defaults to the SYSTEM schema 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 the user owns. A user with the EXP_FULL_DATABASE role can export dependent objects that are owned by other users. A non-privileged user can export only dependent objects for the specified tables that the user owns.

Exports in table mode do not include cluster definitions. As a result, the data is exported as unclustered tables. Thus, you can use table mode to uncluster tables.

Example 1

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

Parameter File Method

```
> exp system/manager parfile=params.dat
```

The params.dat file contains the following information:

```
FILE=expdat.dmp
TABLES=(scott.emp,blake.dept)
GRANTS=y
INDEXES=y
```

Command-Line Method

```
> exp system/manager tables=(scott.emp,blake.dept) grants=Y indexes=Y
```

Export Messages

```
Export: Release 8.1.5.0.0 - Production on Fri Oct 30 09:35:59 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production
Export done in WE8DEC character set and WE8DEC NCHAR character set

About to export specified tables via Conventional Path ...
Current user changed to SCOTT
. . exporting table                            EMP         14 rows exported
Current user changed to BLAKE
. . exporting table                           DEPT          8 rows exported
Export terminated successfully without warnings.
```

Example 2

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

Parameter File Method

```
> exp blake/paper parfile=params.dat
```

The params.dat file contains the following information:

```
FILE=blake.dmp
TABLES=(dept,manager)
ROWS=Y
COMPRESS=Y
```

Command-Line Method

```
> exp blake/paper file=blake.dmp tables=(dept, manager) rows=y compress=Y
```

Export Messages

```
Export: Release 8.1.5.0.0 - Production on Fri Oct 30 09:36:08 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production
Export done in WE8DEC character set and WE8DEC NCHAR character set

About to export specified tables via Conventional Path ...
. . exporting table                            DEPT          8 rows exported
. . exporting table                         MANAGER          4 rows exported
Export terminated successfully without warnings.
```

Example Export Session Using Partition-Level Export

In partition-level export, you can specify the partitions and subpartitions of a table that you want to export.

Example 1

Assume EMP is a partitioned table with two partitions M and Z (partitioned on employee name). As this example shows, if you export the table without specifying a partition, all of the partitions are exported.

Parameter File Method

```
> exp scott/tiger parfile=params.dat
```

The params.dat file contains the following:

```
TABLES=(emp)
ROWS=y
```

Command-Line Method

```
> exp scott/tiger tables=emp rows=Y
```

Export Messages

```
Export: Release 8.1.5.0.0 - Production on Fri Oct 30 09:36:23 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production
Export done in WE8DEC character set and WE8DEC NCHAR character set

About to export specified tables via Conventional Path ...
. . exporting table                            EMP
. . exporting partition                        M               8 rows exported
. . exporting partition                        Z               6 rows exported
Export terminated successfully without warnings.
```

Example 2

Assume EMP is a partitioned table with two partitions M and Z (partitioned on employee name). As this example shows, if you export the table and specify a partition, only the specified partition is exported.

Parameter File Method

```
> exp scott/tiger parfile=params.dat
```

The params.dat file contains the following:

```
TABLES=(emp:m)
ROWS=y
```

Command-Line Method

```
> exp scott/tiger tables=emp:m rows=Y
```

Export Messages

```
Export: Release 8.1.5.0.0 - Production on Fri Oct 30 09:36:29 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production
Export done in WE8DEC character set and WE8DEC NCHAR character set

About to export specified tables via Conventional Path ...
. . exporting table                            EMP
. . exporting partition                              M          8 rows exported
Export terminated successfully without warnings.
```

Example 3

Assume EMP is a partitioned table with two partitions M and Z. Table EMP is partitioned using composite method. M has subpartitions sp1 and sp2, and Z has subpartitions sp3 and sp4. As the example shows, if you export the composite partition M, all its subpartitions (sp1 and sp2) will be exported. If you export the table and specify a subpartition (sp4), only the specified subpartition is exported.

Parameter File Method

```
> exp scott/tiger partfile=params.dat
```

The params.dat file contains the following:

```
TABLES=(emp:m,emp:sp4)
ROWS=Y
```

Command-line Method

```
> exp scott/tiger tables=(emp:m, emp:sp4) rows=Y
```

Export Messages

```
Export: Release 8.1.5.0.0 - Development on Fri Oct 30 09:36:29 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.


Connected to: Oracle8i Enterprise Edition Release 8.1.5.0.0 - Development
With the Partitioning option
```

```
PL/SQL Release 8.1.5.0.0 - Production
Export done in WE8DEC character set and WE8DEC NCHAR character set

About to export specified tables via Conventional Path ...

. . exporting table                            EMP
. . exporting composite partition                M
. . exporting subpartition                       SP1        4 rows exported
. . exporting subpartition                       SP2        0 rows exported
. . exporting composite partition                Z
. . exporting subpartition                       SP4        1 rows exported
Export terminated successfully without warnings.
```

Using the Interactive Method

Starting Export from the command line with no parameters initiates the interactive method. The interactive method does not provide prompts for all Export functionality. The interactive method is provided only for backward compatibility.

If you do not specify a username/password combination on the command line, the Export utility prompts you for this information.

Interactively Invoking Export as SYSDBA

Typically, you should not need to invoke Export as SYSDBA. However, you may have occasion to do so under specific circumstances at the request of Oracle Technical Support.

If you use the Export interactive mode, you will not be prompted to specify whether you want to connect as SYSDBA or @instance. You must specify "AS SYSDBA" and/or "@instance" with the username.

The response to the Export interactive username prompt could be for example:

```
username/password@instance as sysdba
username/password@instance
username/password as sysdba
username/password
username@instance as sysdba  (prompts for password)
username@instance            (prompts for password)
username                     (prompts for password)
username AS sysdba           (prompts for password)
/    as sysdba               (no prompt for password, OS authentication
                             is used)
```

```
/                          (no prompt for password, OS authentication
                           is used)
/@instance as sysdba       (no prompt for password, OS authentication
                           is used)
/@instance                 (no prompt for password, OS authentication
                           is used)
```

Note: if you omit the password and allow Export to prompt you for it, you cannot specify the @instance string as well. You can specify @instance only with username.

Then, Export displays the following prompts:

```
Enter array fetch buffer size: 30720 >
Export file: expdat.dmp >
(1)E(ntire database), (2)U(sers), or (3)T(ables): (1)E >
Export grants (yes/no): yes >
Export table data (yes/no): yes >
Compress extents (yes/no): yes >
Export done in WE8DEC character set and WE8DEC NCHAR character set

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 directory aliases
. exporting context namespaces
. exporting foreign function library names
. exporting object type definitions
. exporting system procedural objects and actions
. exporting pre-schema procedural objects and actions
. exporting cluster definitions
. about to export SYSTEM's tables via Conventional Path ...
. . exporting table                       DEF$_AQCALL          0 rows exported
. . exporting table                      DEF$_AQERROR          0 rows exported
. . exporting table                     DEF$_CALLDEST          0 rows exported
. . exporting table                 DEF$_DEFAULTDEST          0 rows exported
. . exporting table                 DEF$_DESTINATION          0 rows exported
. . exporting table                       DEF$_ERROR          0 rows exported
. . exporting table                         DEF$_LOB          0 rows exported
```

```
. . exporting table                    DEF$_ORIGIN          0 rows exported
. . exporting table                DEF$_PROPAGATOR          0 rows exported
. . exporting table       DEF$_PUSHED_TRANSACTIONS          0 rows exported
. . exporting table                  DEF$_TEMP$LOB          0 rows exported
. . exporting table        SQLPLUS_PRODUCT_PROFILE          0 rows exported
. about to export OUTLN's tables via Conventional Path ...
. . exporting table                            OL$          0 rows exported
. . exporting table                       OL$HINTS          0 rows exported
. about to export DBSNMP's tables via Conventional Path ...
. about to export SCOTT's tables via Conventional Path ...
. . exporting table                          BONUS          0 rows exported
. . exporting table                           DEPT          4 rows exported
. . exporting table                            EMP         14 rows exported
. . exporting table                      SALGRADE          5 rows exported
. about to export ADAMS's tables via Conventional Path ...
. about to export JONES's tables via Conventional Path ...
. about to export CLARK's tables via Conventional Path ...
. about to export BLAKE's tables via Conventional Path ...
. . exporting table                           DEPT          8 rows exported
. . exporting table                        MANAGER          4 rows exported
. exporting referential integrity constraints
. exporting synonyms
. exporting views
. exporting stored procedures
. exporting operators
. exporting indextypes
. exporting bitmap, functional and extensible indexes
. exporting posttables actions
. exporting triggers
. exporting snapshots
. exporting snapshot logs
. exporting job queues
. exporting refresh groups and children
. exporting dimensions
. exporting post-schema procedural objects and actions
. exporting user history table
. exporting default and system auditing options
Export terminated successfully without warnings.
```

You may not see all 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].

Restrictions

Keep in mind the following points when you use the interactive method:

- In user mode, Export prompts for all user names to be included in the export before exporting any data. To indicate the end of the user list and begin the current Export session, press [Return].
- In table mode, if you do not specify a schema prefix, 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, Export assumes that all tables are in BETH's schema until another schema is specified. Only a privileged user (someone with the EXP_FULL_DATABASE role) can export tables in another user's schema.
- If you specify a null table list to the prompt "Table to be exported," the Export utility exits.

Warning, Error, and Completion Messages

This section discusses the messages that Export issues in certain situations.

Log File

You can capture all Export messages in a log file, either by using the LOG parameter (see LOG on page 1-21) or, for those systems that permit it, by redirecting Export's output to a file. The Export utility writes a log of detailed information about successful unloads and any errors that may occur. Refer to the operating system-specific Oracle documentation for information on redirecting output.

Warning Messages

Export does 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 issues a warning whenever it encounters an invalid object.

For example, if a non-existent table is specified as part of a table-mode export, the Export utility exports all other tables.

Then, it issues a warning and terminates successfully, as shown in the following listing:

```
> exp scott/tiger tables=xxx,emp

Export: Release 8.1.5.0.0 - Production on Fri Oct 30 09:38:11 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production
Export done in WE8DEC character set and WE8DEC NCHAR character set

About to export specified tables via Conventional Path ...
EXP-00011: SCOTT.XXX does not exist
. . exporting table                            EMP         14 rows exported
Export terminated successfully with warnings.
```

Fatal Error Messages

Some errors are *fatal* and 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, if the CATEXP.SQL script is not executed, Export issues the following fatal error message:

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

> **Additional Information:** Messages are documented in the *Oracle8i Error Messages* manual and in your Oracle operating system-specific documentation.

Completion Messages

When Export completes without errors, Export displays the message "Export terminated successfully without warnings." If one or more non-fatal errors occurs but Export is able to continue to completion, Export displays the message "Export terminated successfully with warnings." If a fatal error occurs, Export terminates immediately with the message "Export terminated unsuccessfully."

Direct Path Export

Export provides two methods for exporting table data:

- conventional path Export
- direct path Export

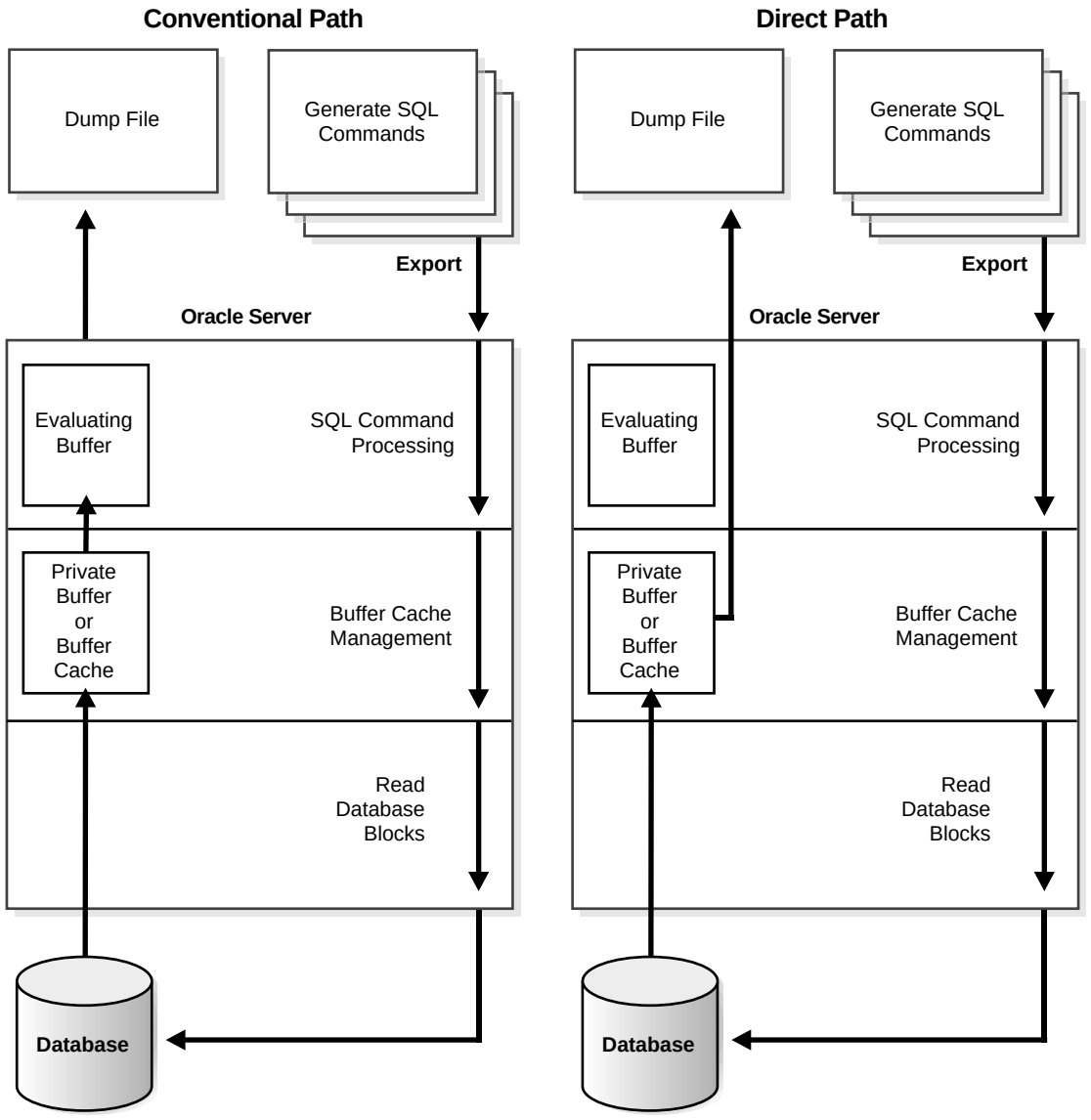
Conventional path Export uses the SQL SELECT statement to extract data from tables. Data is read from disk into a buffer cache, and rows are transferred to the evaluation buffer. The data, after passing expression evaluation, is transferred to the Export client, which then writes the data into the export file.

Direct path Export 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 and saves on data copies whenever possible.

Figure 1–2 on page 1-42 shows how data extraction differs between conventional path and direct path Export.

In a direct path Export, data is read from disk into the buffer cache and rows are transferred *directly* 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 to the Export client, which then writes the data into the export file.

Figure 1–2 Database Reads on Conventional Path and Direct Path



Invoking a Direct Path Export

To use direct path Export, specify the DIRECT=Y parameter on the command line or in the parameter file. The default is DIRECT=N, which extracts the table data using the conventional path.

Note: The Export parameter BUFFER applies only to conventional path exports. For direct path Export, use the parameter RECORDLENGTH to specify the size of the buffer that Export uses for writing to the export file.

Restrictions: You cannot export certain tables using direct path. For example, you cannot export tables using object features on LOBs. If you specify direct path for export, tables containing objects and LOBs will be exported using conventional path.

Character Set Conversion

Direct path Export exports in the database server character set only. If the character set of the export session is not the same as the database character set when an export is initiated, Export displays a warning and aborts. Using the NLS_LANG parameter, specify the session character set to be the same as that of the database before retrying the export.

Performance Issues

You may be able to improve performance by increasing the value of the RECORDLENGTH parameter when you invoke a direct path Export. Your exact performance gain varies depending upon the following factors:

- DB_BLOCK_SIZE
- the types of columns in your table
- your I/O layout (The drive receiving the export file should be separate from the disk drive where the database files reside.)

When using direct path Export, set the RECORDLENGTH parameter equal to the DB_BLOCK_SIZE database parameter, so that each table scan returns a full database block worth of data. If the data does not fit in the export I/O buffer, the Export utility performs multiple writes to the export file for each database block.

The following values are generally recommended for RECORDLENGTH:

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

Restriction: You cannot use the interactive method to invoke direct path Export.

Incremental, Cumulative, and Complete Exports

Important: Incremental, cumulative, and complete Exports are obsolete features that will be phased out in a subsequent release. You should begin now to migrate to Oracle's Backup and Recovery Manager for database backups. See Oracle8i Backup and Recovery Guide for more information.

Restrictions:

- You can do incremental, cumulative, and complete exports only in full database mode (FULL=Y). Only users who have the role EXP_FULL_DATABASE can run incremental, cumulative, and complete Exports. This role contains the privileges needed to modify the system tables that track incremental exports. System Tables on page 1-50 describes those tables.
- You cannot specify incremental Exports as read-consistent.

Base Backups

If you use cumulative and incremental Exports, you should periodically perform a complete Export to create a *base backup*. Following the complete Export, perform frequent incremental Exports and occasional cumulative Exports. After a given period of time, you should begin the cycle 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 definition and all its data, *not just the changed rows*. Typically, you perform incremental Exports more often than cumulative or complete Exports.

Assume that a complete Export was done at Time 1. Figure 1–3 on page 1-45 shows an incremental Export at Time 2, after three tables have been modified. Only the modified tables and associated indexes are exported.

Figure 1–3 Incremental Export at Time 2

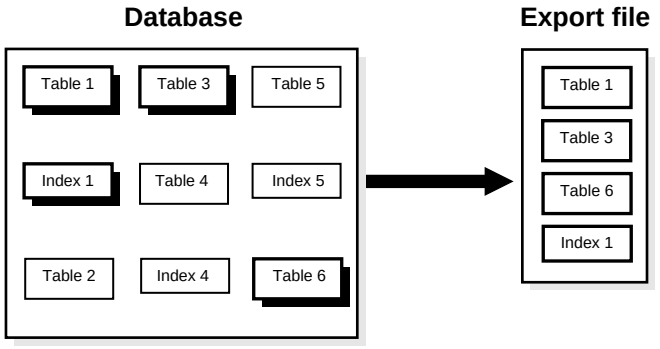
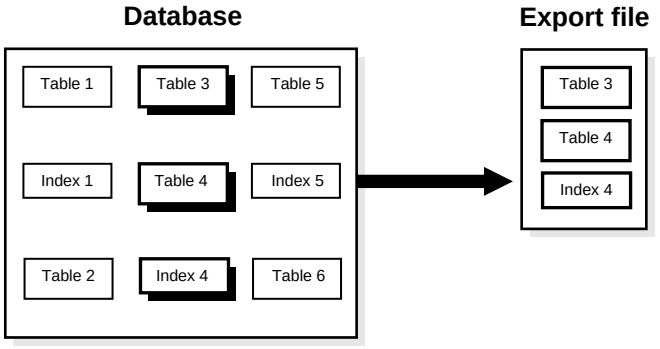


Figure 1–4 shows another incremental Export at Time 3, after two tables have been modified since Time 2. Because Table 3 was modified a second time, it is exported at Time 3 as well as at Time 2.

Figure 1–4 Incremental Export at Time 3

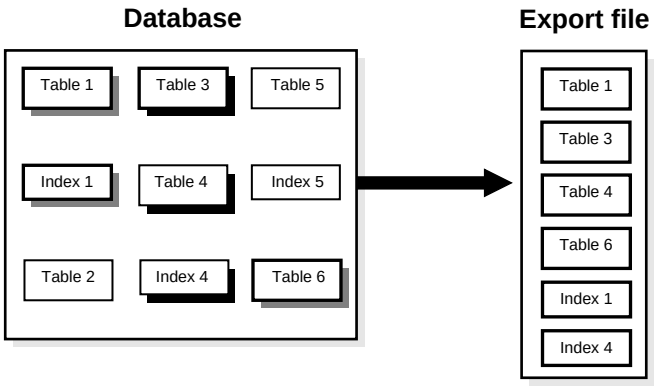


Cumulative Exports

A *cumulative* Export backs up tables that have changed since the last cumulative or complete Export. 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 4. Tables 1 and 6 have been modified since Time 3. All tables modified since the complete Export at Time 1 are exported.

Figure 1–5 Cumulative Export at Time 4



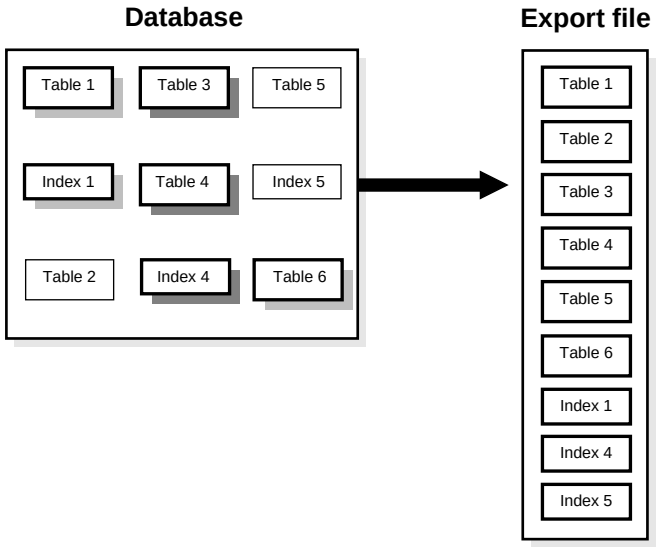
This cumulative export file includes the changes from the incremental Exports from Time 2 and Time 3. Table 3, which was modified at both times, occurs only once in the export file. In this way, cumulative exports save space over multiple incremental Exports.

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 1–6 on page 1-47 shows a complete Export at Time 5. With the complete Export, all objects in the database are exported regardless of when (or if) they were modified.

Figure 1–6 Complete Export at Time 5



A Scenario

The scenario described in this section 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 Sunday
- 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 22
     X  I  I  I  I  I  I  C  I  I  I  I  I  I  C  I  I  I  I  I  I  X
    Sun               Sun               Sun               Sun
```

To restore through day 18, first you import the *system information* from the incremental Export taken on day 18. Then, you import the *data* from:

1. the complete Export taken on day 1
2. the cumulative Export taken on day 8
3. the cumulative Export taken on day 15
4. three incremental Exports taken on days 16, 17, and 18

The incremental Exports on days 2 through 7 can be discarded on day 8, after the cumulative Export is done, because the cumulative Export incorporates all the incremental Exports. Similarly, the incremental Exports on days 9 through 14 can be discarded after the cumulative Export on day 15.

Note: The section INCTYPE on page 1-20 explains the syntax to specify incremental, cumulative, and complete Exports.

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, such as the data, indexes, grants, audits, triggers, and comments.

The entire grant structure for tables or views is exported with the underlying base tables. Indexes are exported with their base table, regardless of who created the index. If the base view is included, "instead of" triggers on views are included.

Any modification (UPDATE, INSERT, or DELETE) on a table automatically qualifies that table for incremental Export. When a table is exported, all of its inner nested tables and LOB columns are exported also. Modifying an inner nested table column causes the outer table to be exported. Modifying a LOB column causes the entire table containing the LOB data to be exported.

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

- 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 to the base tables and data, the following data is exported:

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

  Note: Export does not export grants on data dictionary views for security reasons that affect Import. If such grants were exported, access privileges would be changed and the user would not be aware of this. Also, not forcing grants on import allows the user more flexibility to set up appropriate grants on import.

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 full=y inctype=incremental

Export: Release 8.1.5.0.0 - Production on Fri Oct 30 09:40:11 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production
Export done in WE8DEC character set and WE8DEC NCHAR character set

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 directory aliases
. exporting context namespaces
. exporting foreign function library names
. exporting object type definitions
. exporting system procedural objects and actions
. exporting pre-schema procedural objects and actions
. exporting cluster definitions
. about to export SYSTEM's tables via Conventional Path ...
. about to export OUTLN's tables via Conventional Path ...
. about to export DBSNMP's tables via Conventional Path ...
. about to export SCOTT's tables via Conventional Path ...
. . exporting table                           DEPT          8 rows exported
. . exporting table                            EMP         23 rows exported
. about to export ADAMS's tables via Conventional Path ...
. about to export JONES's tables via Conventional Path ...
. about to export CLARK's tables via Conventional Path ...
. about to export BLAKE's tables via Conventional Path ...
. exporting referential integrity constraints
. exporting synonyms
. exporting views
. exporting stored procedures
. exporting operators
. exporting indextypes
. exporting bitmap, functional and extensible indexes
. exporting posttables actions
. exporting triggers
. exporting snapshots
. exporting snapshot logs
. exporting job queues
. exporting refresh groups and children
. exporting dimensions
. exporting post-schema procedural objects and actions
. exporting user history table
. exporting default and system auditing options
. exporting information about dropped objects
Export terminated successfully without warnings.
```

System Tables

The user SYS owns three tables (INCEXP, INCFIL, and INCVID) that are maintained by Export. These tables are updated when you specify RECORD=Y (the default). You should not alter these tables in any way.

SYS.INCEXP

The table SYS.INCEXP tracks which objects were exported in specific exports

This table contains the following columns:

OWNER# The userid of the schema containing the table.

NAME The object name. The primary key consists of OWNER#, NAME, and TYPE.

TYPE The type of the object (a code specifying INDEX, TABLE, CLUSTER, VIEW, SYNONYM, SEQUENCE, PROCEDURE, FUNCTION, PACKAGE, TRIGGER, DIMENSION, OPERATOR, INDEXTYPE, 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 the table SYS.INCFIL.

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. You can use the views DBA_EXP_OBJECTS, DBA_EXP_VERSION, and DBA_EXP_FILES to display information about incremental exports.

SYS.INCFIL

The table SYS.INCFIL tracks the incremental and cumulative exports and assigns a unique identifier to each.

This table contains the following columns:

EXPID The ID of the incremental or cumulative export, also found in the 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 individual who initiated the export.

When you export 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.

SYS.INCVID

The table SYS.INCVID contains one column for the EXPID of the last valid export. This information determines the EXPID of the next export.

Network Considerations

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

Transporting Export Files Across a Network

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

Exporting and Importing with Net8

With Net8 (and SQL*Net V2), you can perform exports and imports over a network. For example, if you run Export locally, you can write data from a remote Oracle database into a local export file. If you run Import locally, you can read data into a remote Oracle database.

To use Export with Net8, include the `@connect_string` after the `username/password` when you enter the `exp` command, as shown in the following example:

```
exp scott/tiger@SUN2 FILE=export.dmp FULL=Y
```

> **Additional Information:** For the exact syntax of this clause, see the user's guide for your Net8 or SQL*Net protocol. For more information on Net8 or Oracle Names, see the *Net8 Administrator's Guide*.

Character Set and NLS Considerations

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

Character Set Conversion

In conventional mode, the Export utility writes to the export file using the character set specified for the user session, such as 7-bit ASCII, IBM Code Page 500 (EBCDIC), or an Oracle NLS character set like JA16EUC converting from the database server character set as necessary. Import then 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, the data is automatically converted to the character encoding scheme specified for the user session during import, as specified by the NLS_LANG environment variable. After the data is 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.

Some 8-bit characters can be lost (that is, converted to 7-bit equivalents) when you import an 8-bit character set export file. This occurs if the client machine has a native 7-bit character set or if the NLS_LANG operating system environment variable is set to a 7-bit character set. Most often, you notice that accented characters lose their accent mark.

Both Export and Import provide descriptions of any required character set conversion before exporting or importing the data.

When you use direct path Export, the character set of the user's session must be the same as the database character set.

For more information, see the *Oracle8i National Language Support Guide*.

NCHAR Conversion During Export and Import

The Export utility always exports NCHAR data in the national character set of the Export server. (You specify the national character set with the NATIONAL character set statement at database creation.)

The Import utility automatically converts the data to the national character set of the Import server.

Multi-Byte Character Sets and Export and Import

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 the same character set or where the ratio of the width of the widest character in the import character set to the width of the smallest character in the export character set is 1. If the ratio is not 1, Import cannot translate the character data to the Import character set.

> **Caution:** When the character set width differs between the export client and the export server, truncation of data can occur if conversion causes expansion of data. If truncation occurs, Export displays a warning message.

Instance Affinity and Export

If you use instance affinity to associate jobs with instances in databases you plan to import/export, you should refer to the information in the *Oracle8i Administrator's Guide*, the *Oracle8i Reference*, and *Oracle8i Parallel Server Concepts and Administration* for information about use of instance affinity with the Import/Export utilities and, if you are using both release 8.0 and 8.1, to *Oracle8i Migration* for possible compatibility issues.

Fine-Grained Access Support

You can export tables with fine-grain access policies enabled.

Note, however, that the user who imports from an export file containing such tables must have the appropriate privileges (specifically, execute privilege on the DBMS_RLS package so that the tables' security policies can be reinstated). If a user without the correct privileges attempts to export a table with fine-grained access policies enabled, only those rows that the exporter is privileged to read will be exported.

Considerations in Exporting Database Objects

The following sections describe points you should take into consideration when you export particular database objects.

Exporting Sequences

If transactions continue to access sequence numbers during an export, sequence numbers can be skipped. 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 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 LONG and LOB Datatypes

On export, LONG datatypes are fetched in sections. However, enough memory must be available to hold all of the contents of each row, including the LONG data.

LONG columns can be up to 2 gigabytes in length.

> **Note:** All data in a LOB column does not need to be held in memory at the same time. LOB data is loaded and unloaded in sections.

Exporting Foreign Function Libraries

The contents of foreign function libraries are not included in the export file. Instead, only the library specification (name, location) is included in full database and user mode export. The database administrator must move the library and update the library specification if the database is moved to a new location.

Exporting Offline Bitmapped Tablespaces

If the data you are exporting contains offline bitmapped tablespace(s), Export will not be able to export the complete tablespace definition and will display an error message. You can still import the data, however, you must precreate the offline bitmapped tablespace(s) before importing to prevent DDL commands that may reference the missing tablespaces from failing.

Exporting Directory Aliases

Directory alias definitions are included only in a full database mode Export. To move a database to a new location, the database administrator must update the directory aliases to point to the new location.

Directory aliases are not included in user or table mode Export. Therefore, you must ensure that the directory alias has been created on the target system before the directory alias is used.

Exporting BFILE Columns and Attributes

The export file does not hold the contents of external files referenced by BFILE columns or attributes. Instead, only the names and directory aliases for files are copied on Export and restored on Import. If you move the database to a location where the old directories cannot be used to access the included files, the database administrator must move the directories containing the specified files to a new location where they can be accessed.

Exporting Object Type Definitions

In all Export modes, the Export utility includes information about object type definitions used by the tables being exported. The information, including object name, object identifier, and object geometry, is needed to verify that the object type on the target system is consistent with the object instances contained in the export file. This ensures that the object types needed by a table are created with the same object identifier at import time.

Note however, that in table, user, and tablespace mode, the export file does not include a full object type definition needed by a table if the user running Export does not have execute access to the object type. In this case only enough information is written to verify that the type exists, with the same object identifier and the same geometry, on the import target system.

The user must ensure that the proper type definitions exist on the target system, either by working with the DBA to create them, or by importing them from full database or user mode exports performed by the DBA.

It is important to perform a full database mode export regularly to preserve all object type definitions. Alternatively, if object type definitions from different schemas are used, the DBA should perform a user mode export of the appropriate set of users. For example, if SCOTT's table TABLE1 contains a column on BLAKE's type TYPE1, the DBA should perform a user mode export of both BLAKE and SCOTT to preserve the type definitions needed by the table.

Exporting Nested Tables

Inner nested table data is exported whenever the outer containing table is exported. Although inner nested tables can be named, they cannot be exported individually.

Exporting Advanced Queue (AQ) Tables

Queues are implemented on tables. The export and import of queues constitutes the export and import of the underlying queue tables and related dictionary tables. You can export and import queues only at queue table granularity.

When you export a queue table, both the table definition information and queue data is exported. Because the queue table data is exported as well as the table definition, the user is responsible for maintaining application-level data integrity when queue table data is imported.

See *Oracle8i Application Developer's Guide - Advanced Queuing* for more information.

Transportable Tablespaces

The transportable tablespace feature enables you to move a set of tablespaces from one Oracle database to another.

To move or copy a set of tablespaces, you must make the tablespaces read-only, copy the datafiles of these tablespaces, and use Export/Import to move the database information (metadata) stored in the data dictionary. Both the datafiles and the metadata export file must be copied to the target database. The transport of these files can be done using any facility for copying binary files, such as the operating system copying facility, binary-mode FTP, or publishing on CDs.

After copying the datafiles and exporting the metadata, you can optionally put the tablespaces in read-write mode. See Transportable Tablespaces on page 2-63 for more information about importing from an export file that contains transportable tablespace metadata.

Export provides the following parameter keywords you can use to enable export of transportable tablespace metadata.

- TRANSPORT_TABLESPACE
- TABLESPACES

See TRANSPORT_TABLESPACE and TABLESPACES on page 1-26 for more information.

Additional Information: See the *Oracle8i Administrator's Guide* for details about managing transportable tablespaces. For an introduction to the transportable tablespaces feature, see *Oracle8i Concepts*.

Using Different Versions of Export

This section describes the general behavior and restrictions of running an Export version that is different from Oracle8*i*.

Using a Previous Version of Export

In general, you can use the Export utility from any Oracle release 7 to export from an Oracle8*i* server and create an Oracle release 7 export file. (This procedure is described in Creating Oracle Release 7 Export Files from an Oracle8i Database on page 1-60.)

Oracle Version 6 (or earlier) Export cannot be used against an Oracle8*i* database.

Whenever a lower version Export utility runs with a higher version of the Oracle Server, categories of database objects that did not exist in the lower version are excluded from the export. (See Excluded Objects on page 1-60 for a complete list of Oracle8*i* objects excluded from an Oracle release 7 Export.)

Attention: When backward compatibility is an issue, use the earlier release or version of the Export utility against the Oracle8*i* database, and use conventional path export.

Attention: Export files generated by Oracle8*i* Export, either direct path or conventional path, are incompatible with earlier releases of Import and can be imported only with Oracle8*i* Import.

Using a Higher Version Export

Attempting to use a higher version of Export with an earlier 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.

Creating Oracle Release 8.0 Export Files from an Oracle8*i* Database

You do not need to take any special steps to create an Oracle Release 8.0 export file from an Oracle8*i* database, however, certain features are not supported.

- Export does not export rows from tables containing objects and LOBs when you use Export release 8.0 on an Oracle8*i* database and have specified a direct path load (DIRECT=Y).
- Export does not export dimensions when you use Export release 8.0 on an Oracle8*i* database.
- Functional and domain indexes will not be exported when you use Export release 8.0 on an Oracle8*i* database.
- Secondary objects (tables, indexes, sequences, etc. created in support of a domain index) will not be exported when you use Export release 8.0 on an Oracle8*i* database.
- Views, procedures, functions, packages, type bodies, and types containing references to new release 8.1 features may not compile when you use Export release 8.0 on an Oracle8*i* database.
- Objects whose DDL is implemented as a stored procedure rather than SQL will not be exported when you use Export release 8.0 on an Oracle8*i* (or earlier) database.
- Triggers whose action is a CALL statement will not be exported when you use Export release 8.0 on an Oracle8*i* database.
- Tables containing logical ROWID columns, primary key refs, or user-defined OID columns will not be exported when you use Export release 8.0 on an Oracle8*i* database.
- Temporary tables will not be exported when you use Export release 8.0 on an Oracle8*i* database.
- Index Organized Tables (IOTs) will revert to an uncompressed state when you use Export release 8.0 on an Oracle 8*i* database.
- Partitioned IOTs will lose their partitioning information when you use Export release 8.0 on an Oracle8*i* database.
- Indextypes and operators will not be exported when you use Export release 8.0 on an Oracle8*i* database.
- Bitmapped and temporary tablespaces will not be exported when you use Export release 8.0 on an Oracle8*i* database.

- Java source / class / resource will not be exported when you use Export release 8.0 on an Oracle8*i* database.
- Varying-width CLOBs, collection enhancements, and LOB-storage clauses for VARRAY columns or nested table enhancements will not be exported when you use Export release 8.0 on an Oracle8*i* database.
- Fine-grained access security policies are not preserved when you use Export release 8.0 on an Oracle8*i* database.

Creating Oracle Release 7 Export Files from an Oracle8*i* Database

You can create an Oracle release 7 export file from an Oracle8*i* database by running Oracle release 7 Export against an Oracle8*i* server. To do so, however, the user SYS must first run the CATEXP7.SQL script, which creates the export views that make the database look, to Export, like an Oracle release 7 database.

> **Note:** An Oracle8*i* Export requires that the CATEXP.SQL script is run against the database before performing the Export. CATEXP.SQL is usually run automatically when the user SYS runs CATALOG.SQL to create the necessary views. CATEXP7.SQL, however, is not run automatically and must be executed manually. CATEXP7.SQL and CATEXP.SQL can be run in any order; after one of these scripts has been run, it need not be run again.

Excluded Objects

The Oracle release 7 Export utility produces an Oracle release 7 export file by issuing queries against the views created by CATEXP7.SQL. These views are fully compatible with Oracle release 7 and consequently do not contain the new Oracle8*i* objects listed in Creating Oracle Release 8.0 Export Files from an Oracle8i Database on page 1-59 or the following Oracle8 objects:

- directory aliases
- foreign function libraries
- object types
- tables containing objects introduced in Oracle8 (such objects include LOB, REF, and BFILE columns and nested tables)
- partitioned tables
- Index Organized Tables (IOT)
- tables containing more than 254 columns

- tables containing NCHAR columns
- tables containing VARCHAR columns longer than 2,000 characters
- reverse indexes
- password history
- system / schema event triggers
- tables with universal ROWID columns
- bitmap indexes

Enterprise Manager and Oracle7 Export If you want to use Enterprise Manager to export 7.3.2 databases, you must use Enterprise Manager version 1.4.0 or above.

2

Import

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

Import reads only 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:

- What is the Import Utility?
- Import Modes
- Using Import
- Privileges Required to Use Import
- Importing into Existing Tables
- Import Parameters
- Using Table-Level and Partition-Level Export and Import
- Example Import Sessions
- Using the Interactive Method
- Importing Incremental, Cumulative, and Complete Export Files
- Controlling Index Creation and Maintenance
- Reducing Database Fragmentation
- Warning, Error, and Completion Messages
- Error Handling
- Network Considerations

What is the Import Utility?

The basic concept behind Import is very simple. Import inserts the data objects extracted from one Oracle database by the Export utility (and stored in an Export dump file) into another Oracle database. Export dump files can only be read by Import. See Chapter 1, "Export" for more information about Oracle's Export utility.

Import reads the object definitions and table data that the Export utility extracted from an Oracle database and stored in an Oracle binary-format Export dump file located typically on disk or tape.

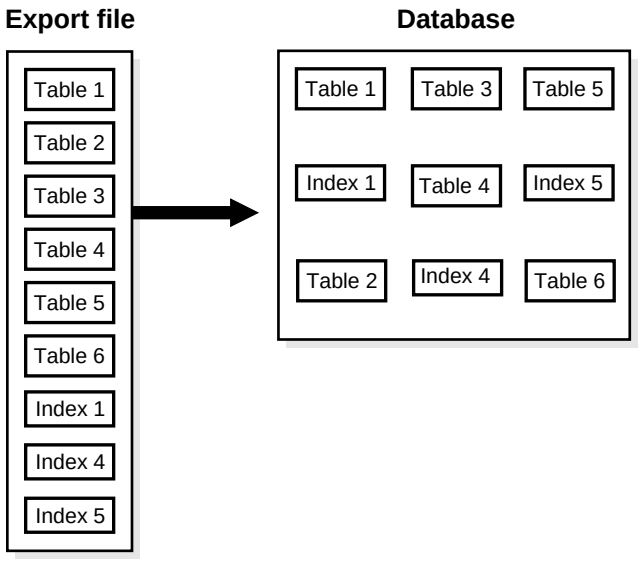
Such files are typically FTPed or physically transported (in the case of tape) to a different site and used, with the Import utility, to transfer data between databases that are on machines not connected via a network or as backups in addition to normal backup procedures.

> **Note:** Export dump files can only be read by the Oracle utility Import. If you need to load data from ASCII fixed-format or delimited files, see Part II, SQL*Loader of this manual.

The Export and Import utilities can also facilitate certain aspects of Oracle Advanced Replication functionality like offline instantiation. See *Oracle8i Replication* for more information.

Figure 2–1 illustrates the process of importing from an Export dump file.

Figure 2–1 Importing an Export File



New in this Release

The following Import features are new as of this release of Oracle:

- Import of subpartitions. See Using Table-Level and Partition-Level Export and Import on page 2-33.
- The ability to specify multiple dump files for an import command. See the parameters FILE on page 2-22 and FILESIZE on page 2-22.
- The Import parameter TOID_NOVALIDATE which allows you to cause Import to omit validation of object types (used typically when the types were created by a cartridge installation). See TOID_NOVALIDATE on page 2-30.
- The maximum number of bytes in an export file on each volume of tape has been increased. See VOLSIZE on page 2-32.
- Fine-grained access support. See Fine-Grained Access Support on page 2-52.
- The ability to export and import precalculated optimizer statistics instead of recomputing the statistics at import time. (This feature is only applicable to certain exports and tables.) See RECALCULATE_STATISTICS on page 2-27.

- Import of transportable tablespace metadata. See TRANSPORT_TABLESPACE on page 2-31.

Table Objects: Order of Import

Table objects are imported as they are read from the export file. The export file contains objects in the following order:

1. type definitions
2. table definitions
3. table data
4. table indexes
5. integrity constraints, views, procedures, and triggers
6. bitmap, functional, and domain indexes

First, new tables are created. Then, data is imported and indexes are built. Then triggers are imported, integrity constraints are enabled on the new tables, and any bitmap, functional, and/or domain indexes are built. 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, all EMP rows that reference departments that have not yet been imported into DEPT would be rejected if the constraints were enabled.

When data is imported into existing tables, however, the order of import can still produce referential integrity failures. In the situation just given, 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, SCOTT's row will fail, even though it would be valid at the end of the import.

> **Suggestion:** For the reasons mentioned previously, 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.

Compatibility

Import can read export files created by Export Version 5.1.22 and later.

Import Modes

The Import utility provides four modes of import. The objects that are imported depend on the Import mode you choose and the mode that was used during the export. All users have two choices of import mode. A user with the IMP_FULL_DATABASE role (a privileged user) has four choices:

Table	This mode allows you to import specific tables and partitions. A privileged user can qualify the tables by specifying the schema that contains them.
User	This mode allows you to import all objects that belong to you (such as tables, grants, indexes, and procedures). 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 which imports a Full Database Export dump file.
Transportable Tablespace	This mode allows a privileged user to move a set of tablespaces from one Oracle database to another.

See Import Parameters on page 2-16 for information on specifying each mode.

A user with the IMP_FULL_DATABASE role must specify one of these options or specify an incremental import. Otherwise, an error results. If a user without the IMP_FULL_DATABASE role fails to specify one of these options, a user-level import is performed.

Table 1–1 on page 1-5 shows the objects that are exported and imported in each mode.

Understanding Table-Level and Partition-Level Import

You can import tables, partitions and subpartitions in the following ways:

- **Table-level Import:** imports all data from the specified tables in an Export file.
- **Partition-level Import**: imports only data from the specified source partitions or subpartitions.

You must set the parameter IGNORE=Y when loading data into an existing table. See IGNORE on page 2-24 for more information.

Table-Level Import

For each specified table, table-level Import imports all of the table's rows. With table-level Import:

- All tables exported using any Export mode (Full, User, Table) can be imported.
- Users can import the entire (partitioned or non-partitioned) table, partitions or subpartitions from a table-level export file into a (partitioned or non-partitioned) target table with the same name.

If the table does not exist, and if the exported table was partitioned, table-level Import creates a partitioned table. If the table creation is successful, table-level Import reads all of the source data from the export file into the target table. After Import, the target table contains the partition definitions of *all* of the partitions and subpartitions associated with the source table in the Export file. This operation ensures that the physical and logical attributes (including partition bounds) of the source partitions are maintained on Import.

Partition-Level Import

Partition-level Import imports a set of partitions or subpartitions from a source table into a target table. Note the following points:

- Import always stores the rows according to the partitioning scheme of the target table.
- Partition-level Import lets you selectively load data from the specified partitions or subpartitions in an export file.
- Partition-level Import inserts only the row data from the specified source partitions or subpartitions.
- If the target table is partitioned, partition-level Import rejects any rows that fall above the highest partition of the target table.
- Partition-level Import can be specified only in table mode.

For information see Using Table-Level and Partition-Level Export and Import on page 2-33.

Using Import

This section describes what you need to do before you begin importing and how to invoke and use the Import utility.

Before Using Import

To use Import, you must run either the script CATEXP.SQL or CATALOG.SQL (which runs CATEXP.SQL) after the database has been created or migrated to release 8.1.

> **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 need to be run only once on a database. You do not need to run either script again before performing future import operations. Both scripts perform the following tasks to prepare the database for Import:

- assign all necessary privileges to the IMP_FULL_DATABASE role
- assign IMP_FULL_DATABASE to the DBA role
- create required views of the data dictionary

Invoking Import

You can invoke Import in three ways:

- Enter the following 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 parameter files. This is the recommended method. See The Parameter File on page 2-10 for information on how to use the parameter file.

- Enter the command

  ```
  imp username/password <parameters>
  ```

  replacing <parameters> with 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. Note that the interactive method does not provide as much functionality as the parameter-driven method. It exists for backward compatibility.

You can use a combination of the first and second options. That is, you can list parameters both in the parameters file and on the command line. In fact, you can specify the same parameter in both places. The position of the PARFILE parameter and other parameters on the command line determines what parameters override others. For example, assume the parameters file `params.dat` contains the parameter INDEXES=Y and Import is invoked with the following line:

```
imp system/manager PARFILE=params.dat INDEXES=N
```

In this case, because `INDEXES=N` occurs *after* `PARFILE=params.dat`, `INDEXES=N` overrides the value of the INDEXES parameter in the PARFILE.

You can specify the username and password in the parameter file, although, for security reasons, this is not recommended.

If you omit the username and password, Import prompts you for it.

See Import Parameters on page 2-16 for a description of each parameter.

Invoking Import as SYSDBA

Typically, you should not need to invoke Import as SYSDBA. However, there may be a few situations in which you need to do so, usually with the help of Oracle Technical Support.

To invoke Import as SYSDBA, use the following syntax:

```
imp username/password AS SYSDBA
```

or, optionally

```
imp username/password@instance AS SYSDBA
```

> **Note:** Since the string "AS SYSDBA" contains a blank, most operating systems require that entire string 'username/password AS SYSDBA' be placed in quotes or marked as a literal by some method. Note that some operating systems also require that quotes on the command line be escaped as well. Please see your operating system-specific Oracle documentation for information about special and reserved characters on your system.

Note that if either the username or password is omitted, Import will prompt you for it.

If you use the Import interactive mode, you will not be prompted to specify whether you want to connect as SYSDBA or @instance. You must specify "AS SYSDBA" and/or "@instance" with the username.

Getting Online Help

Import provides online help. Enter `imp help=y` on the command line to see a help printout like the one shown below.

```
> imp help=y

Import: Release 8.1.5.0.0 - Production on Wed Oct 28 15:00:44 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.


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
               or TABLES=(T1:P1,T1:P2), if T1 is partitioned table

USERID must be the first parameter on the command line.

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     input files (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   CONSTRAINTS  import constraints (Y)
DESTROY  overwrite tablespace data file (N)
```

```
INDEXFILE write table/index info to specified file
SKIP_UNUSABLE_INDEXES  skip maintenance of unusable indexes (N)
ANALYZE  execute ANALYZE statements in dump file (Y)
FEEDBACK display progress every x rows(0)
TOID_NOVALIDATE  skip validation of specified type ids
FILESIZE maximum size of each dump file
RECALCULATE_STATISTICS recalculate statistics (N)
VOLSIZE  number of bytes in file on each volume of a file on tape

The following keywords only apply to transportable tablespaces
TRANSPORT_TABLESPACE import transportable tablespace metadata (N)
TABLESPACES tablespaces to be transported into database
DATAFILES datafiles to be transported into database
TTS_OWNERS users that own data in the transportable tablespace set

Import terminated successfully without warnings.
```

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
```

or

```
imp username/password parfile=filename
```

The syntax for parameter file specifications is one of the following:

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

You can add comments to the parameter file by preceding them with the pound (#) sign. All characters to the right of the pound (#) sign are ignored. The following is an example of a partial parameter file listing:

```
FULL=y
FILE=DBA.DMP
GRANTS=Y
INDEXES=Y # import all indexes
```

See Import Parameters on page 2-16 for a description of each parameter.

Privileges Required to Use Import

This section describes the privileges you need to use the Import utility and to import objects into your own and others' schemas.

Access Privileges

To use Import, you need the privilege CREATE SESSION to log on to the Oracle server. This privilege belongs to the CONNECT role established during database creation.

You can do an import even if you did not create the export file. However, if the export file was created by someone other than you, you can import that file only if you have the IMP_FULL_DATABASE role.

Importing Objects into Your Own Schema

Table 2–1 lists the privileges required to import objects into your own schema. All of these privileges initially belong to the RESOURCE role.

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

Object	Privileges		Privilege Type
clusters		CREATE CLUSTER	system
	And:	tablespace quota, or	
		UNLIMITED TABLESPACE	system
database links		CREATE DATABASE LINK	system
	And:	CREATE SESSION on remote db	system
triggers on tables		CREATE TRIGGER	system
triggers on schemas		CREATE ANY TRIGGER	system
indexes		CREATE INDEX	system
	And:	tablespace quota, or	
		UNLIMITED TABLESPACE	system
integrity constraints		ALTER TABLE	object

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

Object	Privileges		Privilege Type
libraries		CREATE ANY LIBRARY	system
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
(including comments and audit options)	And:	tablespace quota, or UNLIMITED TABLESPACE	system
views		CREATE VIEW	system
	And:	SELECT on the base table, or	object
		SELECT ANY TABLE	system
object types		CREATE TYPE	system
foreign function libraries		CREATE LIBRARY	system
dimensions		CREATE DIMENSION	system
operators		CREATE OPERATOR	system
indextypes		CREATE INDEXTYPE	system

Importing Grants

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. Table 2–2 shows the required conditions for the authorizations to be valid on the target system.

Table 2–2 Privileges Required to Import Grants

Grant	Conditions
object privileges	Object must exist in the user's schema, *or* 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.

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 role IMP_FULL_DATABASE must be enabled. The parameter FULL specifies that these system objects are included in the import when the export file is a full export:

- profiles
- public database links
- public synonyms
- roles
- rollback segment definitions
- resource costs
- foreign function libraries
- context objects
- system procedural objects
- system audit options
- system privileges

- tablespace definitions
- tablespace quotas
- user definitions
- directory aliases
- system event triggers

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.

Importing into Existing Tables

This section describes factors to take into account when you import data 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 use either the same table definition previously used or a compatible format. For example, while you can increase the width of columns and change their order, you cannot do the following:

- add NOT NULL columns
- change the datatype of a column to an incompatible datatype (LONG to NUMBER, for example)
- change the definition of object types used in a table

Disabling Referential Constraints

In the normal import order, referential constraints are imported only after all tables are imported. This sequence prevents 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, 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.

Referential constraints between tables can also cause problems. For example, if the EMP table appears before the DEPT table in the export file, but a referential check exists from the EMP table into the DEPT table, some of the rows from the EMP table may not be imported due to a referential constraint violation.

To prevent errors like these, you should disable referential integrity constraints when importing data into existing tables.

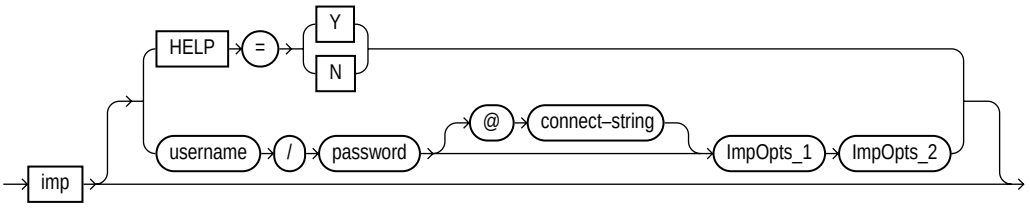
Manually Ordering the Import

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.

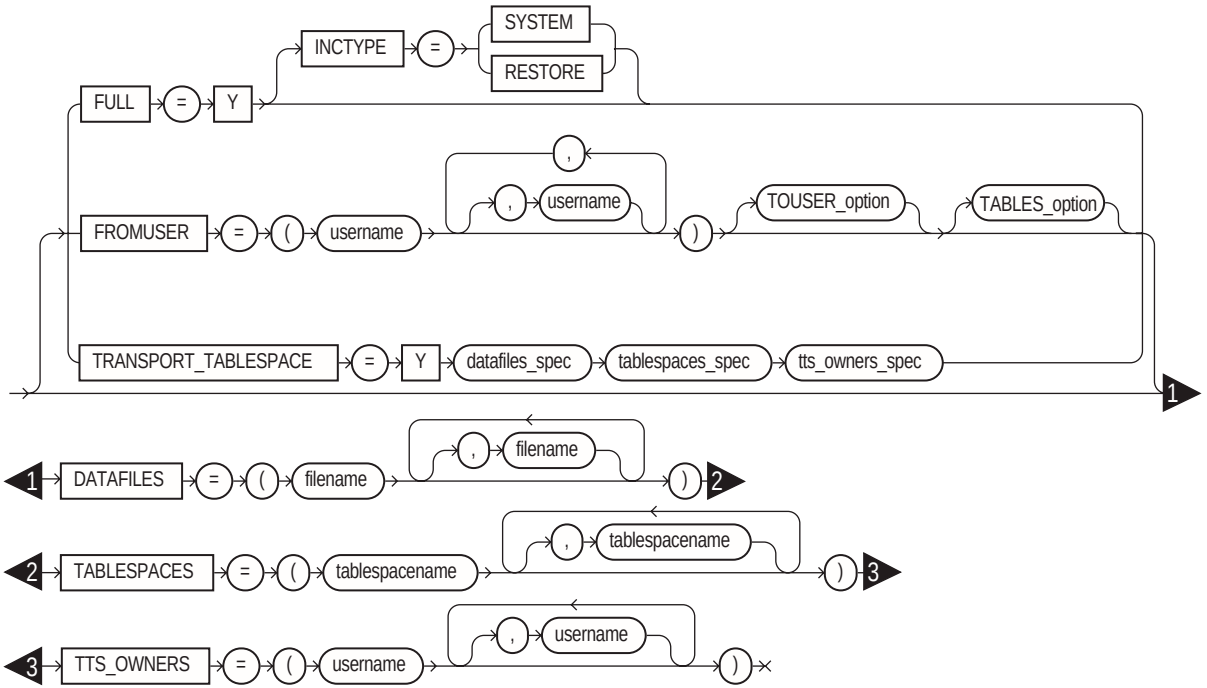
To do so, do several imports from an export file instead of one. First, import tables that are the targets of referential checks, before importing the tables that reference them. This option works if tables do not reference each other in circular fashion, and if a table does not reference itself.

Import Parameters

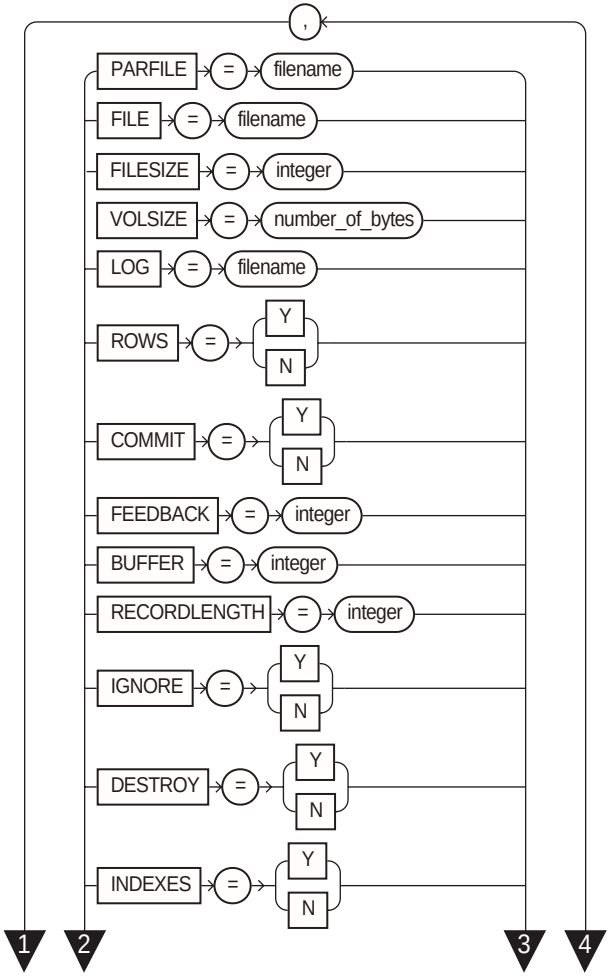
The following diagrams show the syntax for the parameters that you can specify in the parameter file or on the command line:



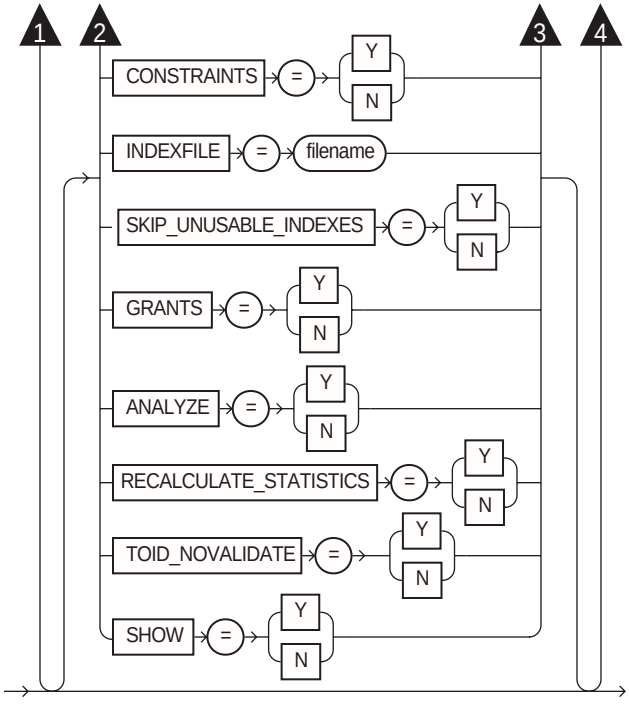
impopts_1



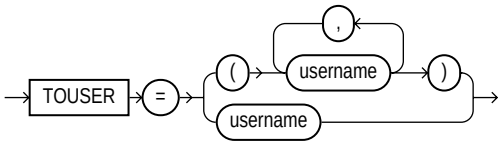
impopts_2


,
PARFILE
=
filename
FILE
=
filename
FILESIZE
=
integer
VOLSIZE
=
number_of_bytes
LOG
=
filename
ROWS
=
Y
N
COMMIT
=
Y
N
FEEDBACK
=
integer
BUFFER
=
integer
RECORDLENGTH
=
integer
IGNORE
=
Y
N
DESTROY
=
Y
N
INDEXES
=
Y
N
1
2
3
4

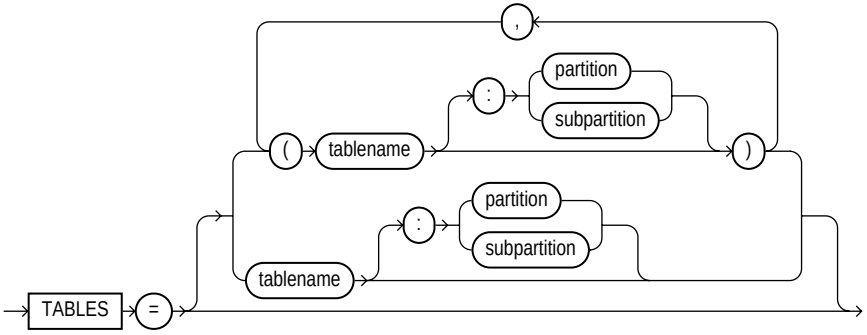
impopts_2 (continued)



TOUSER_option



TABLES_option



The following sections describe parameter functionality and default values.

ANALYZE

Default: Y

Specifies whether or not the Import utility executes SQL ANALYZE statements found in the export file or loads optimizer statistics for tables, indexes, and columns that were precomputed on the Export system. See also the Import parameter RECALCULATE_STATISTICS on page 2-27 and Importing Statistics on page 2-63.

BUFFER

Default: operating system-dependent

The *buffer-size* is the size, in bytes, of the buffer through which data rows are transferred.

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

```
buffer_size = rows_in_array * maximum_row_size
```

For tables containing LONG, LOB, BFILE, REF, ROWID, DATE, or type columns, rows are inserted individually. The size of the buffer must be large enough to contain the entire row, except for LOB and LONG columns. If the buffer cannot hold the longest row in a table, Import attempts to allocate a larger buffer.

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

CHARSET

Note: This parameter applies to Oracle Version 5 and 6 export files only. Use of this parameter is *not* recommended. It is provided only for compatibility with previous versions. Eventually, it will no longer be supported. See The CHARSET Parameter on page 2-66 if you still need to use this parameter.

COMMIT

Default: N

Specifies whether Import should commit after each array insert. By default, Import commits only after loading each table, and Import performs a rollback when an error occurs, before continuing with the next object.

If a table has nested table columns or attributes, the contents of the nested tables are imported as separate tables. Therefore, the contents of the nested tables are always committed in a transaction distinct from the transaction used to commit the outer table.

If COMMIT=N and a table is partitioned, each partition and subpartition in the Export file is imported in a separate transaction.

Specifying COMMIT=Y prevents rollback segments from growing inordinately large and improves the performance of large imports. Specifying COMMIT=Y is advisable if the table has a uniqueness constraint. If the import is restarted, any rows that have already been imported are rejected with a non-fatal error.

Note that, if a table does not have a uniqueness constraint, Import could produce duplicate rows when you re-import the data.

For tables containing LONG, LOB, BFILE, REF, ROWID, UROWID, DATE or type columns, array inserts are not done. If COMMIT=Y, Import commits these tables after each row.

CONSTRAINTS

Default: Y

Specifies whether or not table constraints are to be imported. Note that the default is to import constraints. If you do not want constraints to be imported, you must set this parameter's value to N.

DATAFILES

Default: none

When TRANSPORT_TABLESPACE is specified as Y, use this parameter to list the datafiles to be transported into the database.

See Transportable Tablespaces on page 2-63 for more information.

DESTROY

Default: N

Specifies whether or not the existing data files making up the database should be reused. That is, specifying DESTROY=Y causes Import to include the REUSE option in the datafile clause of the CREATE TABLESPACE command which causes Import to reuse the original database's data files after deleting their contents.

Note that the export file contains the datafile names used in each tablespace. If you specify DESTROY=Y and attempt to create a second database on the same machine (for testing or other purposes), the Import utility will overwrite the first database's data files when it creates the tablespace. In this situation you should use the default, DESTROY=N, so that an error occurs if the data files already exist when the tablespace is created. Also, when you need to import into the original database, you will need to specify IGNORE=Y to add to the existing data files without replacing them.

Warning: If datafiles are stored on a raw device, DESTROY=N *does not prevent* files from being overwritten.

FEEDBACK

Default: 0 (zero)

Specifies that Import should display a progress meter in the form of a dot for *n* number of rows imported. For example, if you specify FEEDBACK=10, Import displays a dot each time 10 rows have been imported. The FEEDBACK value applies to all tables being imported; it cannot be set on a per-table basis.

FILE

Default: expdat.dmp

Specifies the names of the export files to import. The default extension is .dmp. Since Export supports multiple export files (see the parameter FILESIZE below), you may need to specify multiple filenames to be imported.

You need not be the user who exported the export files, however, you must have read access to the files. If you were not the exporter of the export files, you must also have the IMP_FULL_DATABASE role granted to you.

FILESIZE

Export supports writing to multiple export files and Import can read from multiple export files. If, on export, you specify a value (byte limit) for the Export FILESIZE parameter, Export will write only the number of bytes you specify to each dump file. On import, you must use the Import parameter FILESIZE to tell Import the maximum dump file size you specified on export.

> **Note:** The maximum value that can be stored in a file is operating system dependent. You should verify this maximum value in your operating-system specific documentation before specifying FILESIZE.

The FILESIZE value can be specified as a number followed by K (number of kilobytes). For example, FILESIZE=2K is the same as FILESIZE=2048. Similarly, M specifies megabytes (1024 * 1024) while G specifies gigabytes (1024**3). B remains the shorthand for bytes; the number is not multiplied to get the final file size (FILESIZE=2048b is the same as FILESIZE=2048)

FROMUSER

Default: none

A comma-separated list of schemas to import. This parameter is relevant only to users with the IMP_FULL_DATABASE role. The parameter enables you to import a subset of schemas from an export file containing multiple schemas (for example, a full export dump file or a multi-schema, user mode export dump file).

You will typically use FROMUSER in conjunction with the Import parameter TOUSER which you use to specify a list of usernames whose schemas will be targets for import (see TOUSER on page 2-31). However, if you omit specifying TOUSER, Import will:

- import objects into the FROMUSER's schema if the export file is a full dump or a multi-schema, user mode export dump file.
- create objects in the importer's schema (regardless of the presence of or absence of the FROMUSER schema on import) if the export file is a single schema, user mode export dump file created by an unprivileged user.

  Note: Specifying FROMUSER=SYSTEM causes only schema objects belonging to user SYSTEM to be imported, it does not cause system objects to be imported.

FULL

Default: N

Specifies whether to import the entire export file.

GRANTS

Default: Y

Specifies whether to import object grants.

By default, the Import utility imports any object grants that were exported. If the export was a user-mode Export, the export file contains only first-level object grants (those granted by the owner).

If the export was a full database mode Export, the export file contains all object grants, including lower-level grants (those granted by users given a privilege with the WITH GRANT OPTION). If you specify GRANTS=N, the Import utility does not import object grants. (Note that system grants *are* imported even if GRANTS=N.

Note: Export does not export grants on data dictionary views for security reasons that affect Import. If such grants were exported, access privileges would be changed and the importer would not be aware of this.

HELP

Default: N

Displays a description of the Import parameters.

IGNORE

Default: N

Specifies how object creation errors should be handled. If you specify IGNORE=Y, Import overlooks object creation errors when it attempts to create database objects. If you specify IGNORE=Y, Import continues without reporting the error. Note that even if IGNORE=Y, Import does not replace an existing object, instead, it will skip the object. Note that, even if you specify IGNORE=Y, Import does not replace an existing object, instead, it skips the object.

If you accept the default, IGNORE=N, Import logs and / or displays the object creation error before continuing.

For tables, IGNORE=Y causes rows to be imported into existing tables. No message is given. If a table already exists, IGNORE=N causes an error to be reported, and the table is skipped with no rows inserted. Also, objects dependent on tables, such as indexes, grants, and constraints, won't be created if a table already exists and IGNORE=N.

Note that only *object creation errors* are ignored; other errors, such as operating system, database, and SQL errors, *are not* ignored and may cause processing to stop.

In situations where multiple refreshes from a single export file are done with IGNORE=Y, certain objects can be created multiple times (although they will have unique system-defined names). You can prevent this for certain objects (for example, constraints) by doing an import with the value of the parameter CONSTRAINTS set to N. Note that, if you do a full import with the CONSTRAINTS parameter set to N, no constraints for any tables are imported.

If you want to import data into tables that already exist— perhaps because you want to use new storage parameters, or because you have already created the table in a cluster — specify IGNORE=Y. The Import utility imports the rows of data into the existing table.

Warning: When you import into existing tables, if no column in the table is uniquely indexed, 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.

The options are:

SYSTEM	Imports the most recent version of system objects. You should specify the most recent incremental export file when you use this option. A SYSTEM import imports system objects such as foreign function libraries and object type definitions, but does not import user data or objects.
RESTORE	Imports all user database objects and data contained in the export file, excluding system objects.

See Importing Incremental, Cumulative, and Complete Export Files on page 2-43 for more information about the INCTYPE parameter.

INDEXES

Default: Y

Specifies whether or not to import indexes. System-generated indexes such as LOB indexes, OID indexes, or unique constraint indexes are re-created by Import regardless of the setting of this parameter.

You can postpone all user-generated index creation until after Import completes by specifying INDEXES = N.

If indexes for the target table already exist at the time of the import, Import performs index maintenance when data is inserted into the table.

INDEXFILE

Default: none

Specifies a file to receive index-creation commands.

When this parameter is specified, index-creation commands for the requested mode are extracted and written to the specified file, rather than used to create indexes in the database. No database objects are imported.

If the Import parameter CONSTRAINTS is set to Y, Import also writes table constraints to the index file.

The file can then be edited (for example, to change storage parameters) and used as a SQL script to create the indexes.

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.

Perform the following steps to use this feature:

1. Import using the INDEXFILE parameter to create a file of index-creation commands.
2. Edit the file, making certain to add a valid password to the CONNECT strings.
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. Execute the file of index-creation commands as a SQL script to create the index.

The INDEXFILE parameter can be used only with the FULL=Y, FROMUSER, TOUSER, or TABLES parameters.

LOG

Default: none

Specifies a file to receive informational and error messages. If you specify a log file, the Import utility writes all information to the log in addition to the terminal display.

PARFILE

Default: undefined

Specifies a filename for a file that contains a list of Import parameters. For more information on using a parameter file, see The Parameter File on page 2-10.

RECALCULATE_STATISTICS

Default: N

Setting this parameter to Y will cause database optimizer statistics to generate when the exported data is imported. See the *Oracle8i Concepts* manual for information about the optimizer and the statistics it uses. See also the Export parameter STATISTICS on page 1-23, the Import parameter ANALYZE on page 2-19 and Importing Statistics on page 2-63

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.

If you do not define this parameter, it defaults to your platform-dependent value for BUFSIZ. For more information about the BUFSIZ default value, see your operating system-specific documentation.

You can set RECORDLENGTH to any value equal to or greater than your system's BUFSIZ. (The highest value is 64KB.) Changing the RECORDLENGTH parameter affects only the size of data that accumulates before writing to the database. It does not affect the operating system file block size.

> **Note:** You can use this parameter to specify the size of the Import I/O buffer.
>
> **Additional Information:** See your Oracle operating system-specific documentation to determine the proper value or to create a file with a different record size.

ROWS

Default: Y

Specifies whether or not to import the rows of table data.

SHOW

Default: N

When you specify SHOW, the contents of the export file are listed to the display and not imported. The SQL statements contained in the export are displayed in the order in which Import will execute them.

The SHOW parameter can be used only with the FULL=Y, FROMUSER, TOUSER, or TABLES parameters.

SKIP_UNUSABLE_INDEXES

Default: N

Specifies whether or not Import skips building indexes that were set to the Index Unusable state (set by either system or user). Refer to "ALTER SESSION SET SKIP_UNUSABLE_INDEXES=TRUE" in the *Oracle8i SQL Reference* manual for details. Other indexes (not previously set Index Unusable) continue to be updated as rows are inserted.

This parameter allows you to postpone index maintenance on selected index partitions until after row data has been inserted. You then have the responsibility to rebuild the affected index partitions after the Import.

You can use the INDEXFILE parameter in conjunction with INDEXES = N to provide the SQL scripts for re-creating the index. Without this parameter, row insertions that attempt to update unusable indexes fail.

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 in Table 1–1 on page 1-5. The number of tables that can be specified at the same time is dependent on command line limits.

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 is:

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

> **Additional Information:** Some operating systems, such as UNIX, require that you use escape characters before special characters, such as a parenthesis, so that the character is not treated as a special character. On UNIX, use a backslash (\) as the escape character, as shown in the following example:
>
> ```
> TABLES=\(EMP,DEPT\)
> ```

Table Name Restrictions

Table names specified on the command line or in the parameter file cannot include a pound (#) sign, unless the table name is enclosed in quotation marks.

For example, if the parameter file contains the following line, Import interprets everything on the line after EMP# as a comment. As a result, DEPT and MYDATA are not imported.

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

However, if the parameter file contains the following line, the Import utility imports all three tables:

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

> **Attention:** When you specify the table name in quotation marks, it is case sensitive. The name must exactly match the table name stored in the database. By default, database names are stored as uppercase.
>
> **Additional Information:** Some operating systems require single quotes instead of double quotes. See your Oracle operating system-specific documentation.

TABLESPACES

Default: none

When TRANSPORT_TABLESPACE is specified as Y, use this parameter to provide a list of tablespaces to be transported into the database.

See Transportable Tablespaces on page 2-63 for more information.

TOID_NOVALIDATE

Default: none

When you import a table that references a type, but a type of that name already exists in the database, Import attempts to verify that the pre-existing type is in fact the type used by the table (rather than a different type that just happens to have the same name).

To do this, Import compares the type's unique identifier (TOID) with the identifier stored in the export file, and will not import the table rows if the TOIDs do not match.

In some situations, you may not want this validation to occur on specified types (for example, if the types were created by a cartridge installation). You can use the TOID_NOVALIDATE parameter to specify types to exclude from TOID comparison.

The syntax is

```
toid_novalidate=([schema-name.]type-name [, ...])
```

For example:

```
imp scott/tiger table=foo toid_novalidate=bar
imp scott/tiger table=foo toid_novalidate=(fred.type0,sally.type2,type3)
```

If you do not specify a schema-name for the type, it defaults to the schema of the importing user. For example, in the first example above, the type "bar" defaults to "scott.bar".

The output of a typical import with excluded types would contain entries similar to the following:

```
[...]
. importing IMP3's objects into IMP3
. . skipping TOID validation on type IMP2.TOIDTYP0
. . importing table                  "TOIDTAB3"
[...]
```

> **Note:** When you inhibit validation of the type identifier, it is your responsibility to ensure that the attribute list of the imported type matches the attribute list of the existing type. If these attribute lists do not match, results are unpredictable.

TOUSER

Default: none

Specifies a list of usernames whose schemas will be targets for import. 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, the schema names are paired. The following example imports SCOTT's objects into JOE's schema, and FRED's objects into TED's schema:

```
imp system/manager FROMUSER=scott,fred TOUSER=joe,ted
```

> **Note:** If the FROMUSER list is longer than the TOUSER list, the remaining schemas will be imported into either the FROMUSER schema, or into the importer's schema, based on normal defaulting rules. You can use the following syntax to ensure that any extra objects go into the TOUSER schema:
>
> ```
> imp system/manager FROMUSER=scott,adams TOUSER=ted,ted
> ```
>
> Note that user Ted is listed twice.

TRANSPORT_TABLESPACE

Default: N

When specified as Y, instructs Import to import transportable tablespace metadata from an export file.

See Transportable Tablespaces on page 2-63 for more information.

TTS_OWNERS

Default: none

When TRANSPORT_TABLESPACE is specified as Y, use this parameter to list the users who own the data in the transportable tablespace set.

See Transportable Tablespaces on page 2-63 for more information.

USERID

Default: undefined

Specifies the *username/password* (and optional connect string) of the user performing the import.

USERID can also be:

```
username/password AS SYSDBA
```

or

```
username/password@instance AS SYSDBA
```

See Invoking Import as SYSDBA on page 2-8 for more information. Note also that your operating system may require you to treat AS SYSDBA as a special string requiring you to enclose the entire string in quotes as described on page 2-8.

Optionally, you can specify the *@connect_string* clause for Net8. See the user's guide for your Net8 protocol for the exact syntax of *@connect_string*.

VOLSIZE

Specifies the maximum number of bytes in an export file on each volume of tape.

The VOLSIZE parameter has a maximum value equal to the maximum value that can be stored in 64 bits. See your Operating system-specific documentation for more information.

The VOLSIZE value can be specified as number followed by K (number of kilobytes). For example, VOLSIZE=2K is the same as VOLSIZE=2048. Similarly, M specifies megabytes (1024 * 1024) while G specifies gigabytes (1024**3). B remains the shorthand for bytes; the number is not multiplied to get the final file size (VOLSIZE=2048b is the same as VOLSIZE=2048)

Using Table-Level and Partition-Level Export and Import

Both table-level Export and partition-level Export can migrate data across tables, partitions, and subpartitions.

Guidelines for Using Partition-Level Import

This section provides more detailed information about partition-level Import. For general information, see Understanding Table-Level and Partition-Level Import on page 2-5.

Partition-level Import cannot import a non-partitioned exported table. However, a partitioned table can be imported from a non-partitioned exported table using table-level Import. Partition-level Import is legal only if the source table (that is, the table called *tablename* at export time) was partitioned and exists in the Export file.

- If the partition or subpartition name is not a valid partition in the export file, Import generates a warning.
- The partition or subpartition name in the parameter refers to only the partition or subpartition in the Export file, which may not contain all of the data of the table on the export source system.

If ROWS = Y (default), and the table does not exist in the Import target system, the table is created and all rows from the source partition or subpartition are inserted into the target table's partition or subpartition.

If ROWS = Y (default) and IGNORE=Y, but the table already existed before Import, all the rows for the specified partition or subpartition in the table are inserted into the table. The rows are stored according to the existing partitioning scheme of the target table.

If the target table is partitioned, Import reports any rows that are rejected because they fall above the highest partition of the target table.

If ROWS = N, Import does not insert data into the target table and continues to process other objects associated with the specified table and partition or subpartition in the file.

If the target table is non-partitioned, the partitions and subpartitions are imported into the entire table. Import requires IGNORE = Y to import one or more partitions or subpartitions from the Export file into a non-partitioned table on the import target system.

Migrating Data Across Partitions and Tables

The presence of a table-name:partition-name with the TABLES parameter results in reading from the Export file only data rows from the specified source partition or subpartition. If you do not specify the partition or subpartition name, the entire table is used as the source. If you specify a partition name for a composite partition, all subpartitions within the composite partition are used as the source.

Import issues a warning if the specified partition or subpartition is not in the export file.

Data exported from one or more partitions or subpartitions can be imported into one or more partitions or subpartitions. Import inserts rows into partitions or subpartitions based on the partitioning criteria in the target table.

In the following example, the partition specified by the partition-name is a composite partition. All of its subpartitions will be imported:

```
imp system/manager FILE = export.dmp FROMUSER = scott TABLES=b:py
```

The following example causes row data of partitions qc and qd of table scott.e to be imported into the table scott.e:

```
imp scott/tiger FILE = export.dmp TABLES = (e:qc, e:qd) IGNORE=y
```

If table "e" does not exist in the Import target database, it is created and data is inserted into the same partitions. If table "e" existed on the target system before Import, the row data is inserted into the partitions whose range allows insertion. The row data can end up in partitions of names other than qc and qd.

> **Note:** With partition-level Import to an existing table, you *must* set up the target partitions or subpartitions properly and use IGNORE=Y.

Example Import Sessions

This section gives some examples of import sessions that show you how to use the parameter file and command-line methods. The examples illustrate four 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
- tables imported using partition-level Import

Example Import of Selected Tables for a Specific User

In this example, using a full database export file, an administrator imports the DEPT and EMP tables into the SCOTT schema.

Parameter File Method

```
> imp system/manager parfile=params.dat
```

The params.dat file contains the following information:

```
FILE=dba.dmp
SHOW=n
IGNORE=n
GRANTS=y
FROMUSER=scott
TABLES=(dept,emp)
```

Command-Line Method

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

Import Messages

```
Import: Release 8.1.5.0.0 - Production on Fri Oct 30 09:41:18 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production

Export file created by EXPORT:V08.01.05 via conventional path
import done in WE8DEC character set and WE8DEC NCHAR character set
. importing SCOTT's objects into SCOTT
. . importing table                          "DEPT"          4 rows imported
. . importing table                           "EMP"         14 rows imported
Import terminated successfully without warnings.
```

Example Import of Tables Exported by Another User

This example illustrates importing the UNIT and MANAGER tables from a file exported by BLAKE into the SCOTT schema.

Parameter File Method

```
> imp system/manager parfile=params.dat
```

The params.dat file contains the following information:

```
FILE=blake.dmp
SHOW=n
IGNORE=n
GRANTS=y
ROWS=y
FROMUSER=blake
TOUSER=scott
TABLES=(unit,manager)
```

Command-Line Method

```
> imp system/manager fromuser=blake touser=scott file=blake.dmp tables=(unit,manager)
```

Import Messages

```
Import: Release 8.1.5.0.0 - Production on Fri Oct 30 09:41:34 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production

Export file created by EXPORT:V08.01.05 via conventional path

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

import done in WE8DEC character set and WE8DEC NCHAR character set
. . importing table                          "UNIT"          4 rows imported
. . importing table                       "MANAGER"          4 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 SCOTT into user BLAKE's account.

Parameter File Method

```
> imp system/manager parfile=params.dat
```

The params.dat file contains the following information:

```
FILE=scott.dmp
FROMUSER=scott
TOUSER=blake
TABLES=(*)
```

Command-Line Method

```
> imp system/manager file=scott.dmp fromuser=scott touser=blake tables=(*)
```

Import Messages

```
Import: Release 8.1.5.0.0 - Production on Fri Oct 30 09:41:36 1998

(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production

Export file created by EXPORT:V08.01.05 via conventional path

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

import done in WE8DEC character set and WE8DEC NCHAR character set
. . importing table                          "BONUS"          0 rows imported
. . importing table                           "DEPT"          4 rows imported
. . importing table                            "EMP"         14 rows imported
. . importing table                        "SALGRADE"          5 rows imported
Import terminated successfully without warnings.
```

Example Import Session Using Partition-Level Import

This section describes an import of a table with multiple partitions, a table with partitions and subpartitions, and repartitioning a table on different columns.

Example 1: A Partition-level Import

In this example, emp is a partitioned table with three partitions, p1, p2 and p3.

A table-level export file was created using the following command:

```
> exp scott/tiger tables=emp file=exmpexp.dat rows=y

About to export specified tables via Conventional Path --
. . exporting table                         EMP
. . exporting partition                          P1          7 rows exported
. . exporting partition                          P2         12 rows exported
. . exporting partition                          P3          3 rows exported

Export terminated successfully without warnings.
```

In a partition-level import you can specify the specific partitions of an exported table that you want to import. In this example, p1 and p3 of table emp:

```
> imp scott/tiger tables=(emp:p1,emp:p3) file=exmpexp.dat rows=y

Export file created by EXPORT:V08.01.05 via direct path
import done in WE8DEC character set and WE8DEC NCHAR character set
. importing SCOTT's objects into SCOTT
. . importing partition            "EMP":"P1"                    7 rows imported
. . importing partition            "EMP":"P3"                    3 rows imported
Import terminated successfully without warnings.
```

Example 2: A Partition-level Import of a Composite Partitioned Table.

This example demonstrates that the partitions and subpartitions of a composite partitioned table are imported. Emp is a partitioned table with two composite partitions p1 and p2. P1 has three subpartitions p1_sp1, p1_sp2 and p1_sp3 and p2 has two subpartitions p2_sp1 and p2_sp2.

A table-level export file was created using the following command:

```
> exp scott/tiger tables=emp file=exmpexp.dat rows=y

About to export specified tables via Conventional Path --
. . exporting table                         EMP
. . exporting partition                          P1
```

```
. . exporting subpartition                    P1_SP1        11 rows exported
. . exporting subpartition                    P1_SP2        17 rows exported
. . exporting subpartition                    P1_SP3         3 rows exported
. . exporting partition                       P2
. . exporting subpartition                    P2_SP1         5 rows exported
. . exporting subpartition                    P2_SP2        12 rows exported

Export terminated successfully without warnings.
```

The following import command results in the importing of subpartition p1_sp2 and p1_sp3 of composite partition p1 in table emp and all the subpartitions of composite partition p2 in table emp.

```
> imp scott/tiger tables=(emp:p1_sp2,emp:p1_sp3,emp:p2) file=exmpexp.dat rows=y

Export file created by EXPORT:V08.01.05 via conventional path
import done in WE8DEC character set and WE8DEC NCHAR character set
. importing SCOTT's objects into SCOTT
. . importing table                 EMP
. . importing subpartition          "EMP":"P1_SP2"              17 rows imported
. . importing subpartition          "EMP":"P1_SP3"               3 rows imported
. . importing subpartition          "EMP":"P2_SP1"               5 rows imported
. . importing subpartition          "EMP":"P2_SP2"              12 rows imported

Import terminated successfully without warnings.
```

Example 3: Repartitioning a Table on a Different Column

This example assumes the EMP table has two partitions, based on the EMPNO column. This example repartitions the EMP table on the DEPTNO coumn.

Perform the following steps to repartition a table on a different column:

1. Export the table to save the data.
2. Delete the table from the database.
3. Create the table again with the new partitions.
4. Import the table data.

The following example shows how to repartition a table on a different column:

```
> exp scott/tiger tables=emp file=empexp.dat
```

```
About to export specified tables via Conventional Path ...
. . exporting table                          EMP
. . exporting partition                      EMP_LOW          4 rows exported
. . exporting partition                      EMP_HIGH        10 rows exported
Export terminated successfully without warnings.



SQL> drop table emp cascade constraints;
Table dropped.
SQL>
SQL> create table emp
   2    (
   3    empno    number(4) not null,
   4    ename    varchar2(10),
   5    job      varchar2(9),
   6    mgr      number(4),
   7    hiredate date,
   8    sal      number(7,2),
   9    comm     number(7,2),
  10    deptno   number(2)
  11    )
  12   partition by range (deptno)
  13     (
  14     partition dept_low values less than (15)
  15       tablespace tbs_d1,
  16     partition dept_mid values less than (25)
  17       tablespace tbs_d2,
  18     partition dept_high values less than (35)
  19       tablespace tbs_d3
  20     );
Table created.
SQL> exit
> imp scott/tiger tables=emp file=empexp.dat ignore=y



Export file created by EXPORT:V08.01.05 via conventional path
. importing SCOTT's objects into SCOTT
. . importing table           EMP
. . importing partition       "EMP":"EMP_LOW"          4 rows imported
. . importing partition       "EMP":"EMP_HIGH"        10 rows imported
Import terminated successfully without warnings.
```

The following SELECT statements show that the data is partitioned on the DEPTNO column:

```
SQL> select empno, deptno from emp partition (dept_low);
     EMPNO     DEPTNO
---------- ----------
      7934         10
      7782         10
      7839         10
3 rows selected.
SQL> select empno, deptno from emp partition (dept_mid);
     EMPNO     DEPTNO
---------- ----------
      7369         20
      7566         20
      7902         20
      7788         20
      7876         20
5 rows selected.

SQL> select empno, deptno from emp partition (dept_high);

     EMPNO     DEPTNO
---------- ----------
      7499         30
      7521         30
      7900         30
      7654         30
      7698         30
      7844         30
6 rows selected.
```

Using the Interactive Method

Starting Import from the command line with no parameters initiates the interactive method. The interactive method does not provide prompts for all Import functionality. The interactive method is provided only for backward compatibility.

If you do not specify a username/password on the command line, the Import utility prompts you for this information. The following example shows the interactive method:

```
> imp system/manager

Import: Release 8.1.5.0.0 - Production on Fri Oct 30 09:42:54 1998
```

```
(c) Copyright 1998 Oracle Corporation.  All rights reserved.

Connected to: Oracle8 Enterprise Edition Release 8.1.5.0.0 - Production
With the Partitioning option
PL/SQL Release 8.1.5.0.0 - Production

Import file: expdat.dmp >
Enter insert buffer size (minimum is 8192) 30720>
Export file created by EXPORT:V08.01.05 via conventional path

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

import done in WE8DEC character set and WE8DEC NCHAR character set
List contents of import file only (yes/no): no >
Ignore create error due to object existence (yes/no): no >
Import grants (yes/no): yes >
Import table data (yes/no): yes >
Import entire export file (yes/no): yes >
. importing BLAKE's objects into SYSTEM
. . importing table                          "DEPT"          4 rows imported
. . importing table                       "MANAGER"          3 rows imported
Import terminated successfully without warnings.
```

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].

Note: If you specify No at the previous prompt, Import prompts you for a schema name and the table names you want to import for that schema:

```
Enter table(T) or partition(T:P) names. Null list means all tables for user
```

Entering a null table list causes all tables in the schema to be imported. You can only specify one schema at a time when you use the interactive method.

Importing Incremental, Cumulative, and Complete Export Files

Because 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.*

Because imports from incremental export files are dependent on the method used to export the data, you should also read Incremental, Cumulative, and Complete Exports on page 1-44.

It is important to note that, because 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 usually generated if the object already exists.

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 has been run on a database. Once that has been done, the process of restoring objects follows the steps listed below.

1. Import the most recent incremental export file (specify INCTYPE=SYSTEM for the import) or cumulative export file, if no incremental exports have been taken. This step imports the correct system objects (for example, users, object types, etc.) for the database.
2. Import the most recent complete export file. (Specify INCTYPE=RESTORE for the import)
3. Import all cumulative export files after the last complete export. (Specify INCTYPE=RESTORE for the import)
4. Import all incremental export files after the last cumulative export. (Specify INCTYPE=RESTORE for the import)

For example, if you have the following:

- 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 always start with a clean database (that is, no user tables) before starting the import sequence.

Importing Object Types and Foreign Function Libraries from an Incremental Export File

For incremental imports only, object types and foreign function libraries are handled as system objects. That is, their definitions are only imported with the other system objects when INCTYPE = SYSTEM. This imports the most recent definition of the object type (including the object identifier) and the most recent definition of the library specification.

Then, as tables are imported from earlier incremental export files using INCTYPE=RESTORE, Import verifies that any object types needed by the table exist and have the same object identifier. If the object type does not exist, or if it exists but its object identifier does not match, the table is not imported.

This indicates the object type had been dropped or replaced subsequent to the incremental export, requiring that all tables dependent on the object also had been dropped.

Controlling Index Creation and Maintenance

This section describes the behavior of Import with respect to index creation and maintenance.

Index Creation and Maintenance Controls

If SKIP_UNUSABLE_INDEXES=Y, the Import utility postpones maintenance on all indexes that were set to Index Unusable before Import. Other indexes (not previously set Index Unusable) continue to be updated as rows are inserted. This approach saves on index updates during Import of existing tables.

Delayed index maintenance may cause a violation of an existing unique integrity constraint supported by the index. The existence of a unique integrity constraint on a table does not prevent existence of duplicate keys in a table that was imported with INDEXES = N. The supporting index will be in UNUSABLE state until the duplicates are removed and the index is rebuilt.

Delaying Index Creation

Import provides you with the capability of delaying index creation and maintenance services until after completion of the import and insertion of exported data. Performing index (re)creation or maintenance after import completes is generally faster than updating the indexes for each row inserted by Import.

Index creation can be time consuming, and therefore can be done more efficiently after the import of all other objects has completed. You can postpone creation of indexes until after the Import completes by specifying INDEXES = N (INDEXES = Y is the default). You can then store the missing index definitions in a SQL script by running Import while using the INDEXFILE parameter. The index-creation commands that would otherwise be issued by Import are instead stored in the specified file.

After the import is complete, you must create the indexes, typically by using the contents of the file (specified with INDEXFILE) as a SQL script after specifying passwords for the connect statements.

If the total amount of index updates are smaller during data insertion than at index rebuild time after import, users can choose to update those indexes at table data insertion time by setting INDEXES = Y.

Example of Postponing Index Maintenance

For example, assume that partitioned table `t` with partitions `p1` and `p2` exists on the Import target system. Assume that local indexes `p1_ind` on partition `p1` and `p2_ind` on partition `p2` exist also. Assume that partition `p1` contains a much larger amount of data in the existing table `t`, compared with the amount of data to be inserted by the Export file (expdat.dmp). Assume that the reverse is true for `p2`.

Consequently, performing index updates for `p1_ind` during table data insertion time is more efficient than at partition index rebuild time. The opposite is true for `p2_ind`.

Users can postpone local index maintenance for `p2_ind` during Import by using the following steps:

1. Issue the following SQL statement before Import:

```
ALTER TABLE t MODIFY PARTITION p2 UNUSABLE LOCAL INDEXES;
```

2. Issue the following Import command:

```
imp scott/tiger FILE=export.dmp TABLES = (t:p1, t:p2) IGNORE=Y SKIP_UNUSABLE_INDEXES=Y
```

This example executes the ALTER SESSION SET SKIP_UNUSABLE_INDEXES=Y statement before performing the import.

3. Issue the following SQL statement after Import:

```
ALTER TABLE t MODIFY PARTITION p2 REBUILD UNUSABLE LOCAL INDEXES;
```

In this example, local index `p1_ind` on `p1` will be updated when table data is inserted into partition `p1` during Import. Local index `p2_ind` on `p2` will be updated at index rebuild time, after Import.

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 and import as follows:

1. Do a full database export (FULL=Y) to back up the entire database.
2. Shut down Oracle after all users are logged off.
3. Delete the database. See your Oracle operating system-specific documentation for information on how to delete a database.

4. Re-create the database using the CREATE DATABASE command.
5. Do a full database import (FULL=Y) to restore the entire database.

See the *Oracle8i Administrator's Guide* for more information about creating databases.

Warning, Error, and Completion Messages

By default, Import displays all error messages. If you specify a log file by using the LOG parameter, Import writes the error messages to the log file in addition to displaying them on the terminal. You should always specify a log file when you import. (You can redirect Import's output to a file on those systems that permit I/O redirection.)

> **Additional Information:** See LOG on page 2-26. Also see your operating system-specific 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, and Import was able to continue to completion, the message "Import terminated successfully with warnings" occurs. If a fatal error occurs, Import ends immediately with the message "Import terminated unsuccessfully."

> **Additional Information:** Messages are documented in *Oracle8i Error Messages* and your operating system-specific documentation.

Error Handling

This section describes errors that can occur when you import database objects.

Row Errors

If a row is rejected due to an integrity constraint violation or invalid data, Import displays a warning message but continues processing the rest of the table. Some errors, such as "tablespace full," apply to all subsequent rows in the table. These errors cause Import to stop processing the current table and skip to the next 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 the *Oracle8i Application Developer's Guide - Fundamentals* and *Oracle8i Concepts* for more information on integrity constraints.

Invalid Data

Row errors can also occur when the column definition for a table in a database is different from the column definition in the export file. 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.

Errors Importing Database Objects

Errors can occur for many reasons when you import database objects, as described in this section. When such an error occurs, import of the current database object is discontinued. Import then attempts to continue with the next database object in the export file.

Object Already Exists

If a database object to be imported already exists in the database, 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 database object. The current database 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. The database object is not replaced. If the object is a table, rows are imported into it. Note that only *object creation errors* are ignored, all other errors (such as operating system, database, and SQL) *are* 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

If sequence numbers need to be reset to the value in an export file as part of an import, you should drop sequences. A sequence that is not dropped before the import is not set to the value captured in the export file, because Import does not drop and re-create a sequence that already exists. If the sequence already exists, 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 you are 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 you are importing a row, Import stops processing the current table and skips to the next table. If you have specified COMMIT=Y, Import commits the partial import of the current table.

If not, a rollback of the current table occurs before Import continues. (See the description of COMMIT on page 2-20 for information about the COMMIT parameter.)

Domain Index Metadata

Domain indexes can have associated application-specific metadata that is imported via anonymous PL/SQL blocks. These PL/SQL blocks are executed at import time prior to the CREATE INDEX statement. If a PL/SQL block causes an error, the associated index is not created because the metadata is considered an integral part of the index.

Fatal Errors

When a fatal error occurs, Import terminates. For example, if you enter an invalid username/password combination or attempt to run Export or Import without having prepared the database by running the scripts CATEXP.SQL or CATALOG.SQL, a fatal error occurs and causes Import to terminate.

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, when 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 and Importing with Net8

Net8 lets you export and import over a network. For example, running Import locally, you can read data into a remote Oracle database.

To use Import with Net8, you must include the connection qualifier string *@connect_string* when entering the username/password in the exp or imp command. For the exact syntax of this clause, see the user's guide for your Net8 protocol. For more information on Net8, see the *Net8 Administrator's Guide*. See also *Oracle8i Distributed Database Systems*.

Import and Snapshots

Note: In certain situations, particularly those involving data warehousing, snapshots may be referred to as *materialized views*. This section retains the term snapshot.

The three interrelated objects in a snapshot system are the master table, optional snapshot log, and the snapshot itself. The tables (master table, snapshot log table definition, and snapshot tables) can be exported independently of one another. Snapshot logs can be exported only if you export the associated master table. You can export snapshots using full database or user-mode Export; you cannot use table-mode Export.

This section discusses how fast refreshes are affected when these objects are imported. *Oracle8i Replication* provides more information about snapshots and snapshot logs. See also *Oracle8i Replication*, Appendix B, "Migration and Compatibility" for Import-specific information.

Master Table

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

Snapshot Log

When a ROWID snapshot log is exported, ROWIDs stored in the snapshot log have no meaning upon import. As a result, each ROWID snapshot's first attempt to do a fast refresh fails, generating an error indicating that a complete refresh is required.

To avoid the refresh error, do a complete refresh after importing a ROWID snapshot log. After you have done a complete refresh, subsequent fast refreshes will work properly. In contrast, when a primary key snapshot log is exported, the keys' values do retain their meaning upon Import. Therefore, primary key snapshots can do a fast refresh after the import. See *Oracle8i Replication* for information about primary key snapshots.

Snapshots and Materialized Views

A snapshot that has been restored from an export 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 Snapshot

When you restore a snapshot from an export file, you may encounter a problem under certain circumstances.

Assume that a snapshot is refreshed at time A, exported at time B, and refreshed again at time C. Then, because of corruption or other problems, the snapshot needs to be restored by dropping the snapshot and importing it again. The newly imported version has the last refresh time recorded as time A. However, 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), they are used, and the fast refresh completes successfully. Otherwise, the fast refresh fails, generating an error that says a complete refresh is required.

Importing a Snapshot into a Different Schema

Snapshots, snapshot logs, and related items are exported with the schema name explicitly given in the DDL statements, therefore, snapshots and their related items cannot be imported into a different schema.

If you attempt to use FROMUSER/TOUSER to import snapshot data, an error will be written to the Import log file and the items will not be imported.

Import and Instance Affinity

If you use instance affinity to associate jobs with instances in databases you plan to import/export, you should refer to the information in the *Oracle8i Administrator's Guide*, the *Oracle8i Reference*, and *Oracle8i Parallel Server Concepts and Administration* for information about the use of instance affinity with the Import/Export utilities.

Fine-Grained Access Support

You can export tables with fine-grain access policies enabled.

Note, however, in order to restore the policies, the user who imports from an export file containing such tables must have the appropriate privileges (specifically execute privilege on the DBMS_RLS package so that the tables' security policies can be reinstated). If a user without the correct privileges attempts to import from an export file that contains tables with fine-grain access policies, a warning message will be issued. Therefore, it is advisable for security reasons that the exporter/importer of such tables be the DBA.

Storage Parameters

By default, a table is imported into its original tablespace.

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 unless the table:

- is partitioned
- is a type table
- contains LOB or VARRAY columns
- has an Index-Only Table (IOT) overflow segment

If the user does not have sufficient quota in the default tablespace, the user's tables are not imported. (See Reorganizing Tablespaces on page 2-54 to see how you can use this to your advantage.)

The OPTIMAL Parameter

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

Storage Parameters for OID INDEXes and LOB Columns

Tables are exported with their current storage parameters. For object tables, the OIDINDEX is created with its current storage parameters and name, if given. For tables that contain LOB or VARRAY columns, LOB or VARRAY data is created with their current storage parameters.

If you alter the storage parameters of existing tables prior to export, the tables are exported using those altered storage parameters. Note, however, that storage parameters for LOB data cannot be altered prior to export (for example, chunk size for a LOB column, whether a LOB column is CACHE or NOCACHE, etc.).

Note that LOB data might not reside in the same tablespace as the containing table. The tablespace for that data must be read / write at the time of import or the table will not be imported.

If LOB data reside in a tablespace that does not exist at the time of import or the user does not have the necessary quota in that tablespace, the table will not be imported. Because there can be multiple tablespace clauses, including one for the table, Import cannot determine which tablespace clause caused the error.

Overriding Storage Parameters

Before Import, you may want to pre-create large tables with different storage parameters before importing the data. If so, you must specify IGNORE=Y on the command line or in the parameter file.

The Export COMPRESS Parameter

By default at export time, storage parameters are adjusted to consolidate all data into its initial extent. To preserve the original size of an initial extent, you must specify at export time that extents are *not* to be consolidated (by setting COMPRESS=N.) See COMPRESS on page 1-16 for a description of the COMPRESS parameter.

Read-Only Tablespaces

Read-only tablespaces can be exported. On import, if the tablespace does not already exist in the target database, 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 read-only, you must make it read / write before the import.

Dropping a Tablespace

You can drop a tablespace by redefining the objects to use different tablespaces before the import. You can then issue the import command and specify IGNORE=Y.

In many cases, 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, with IGNORE=Y, 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 with the exception of partitioned tables, type tables, and tables that contain LOB or VARRAY columns or index-only tables with overflow segments. Import cannot determine which tablespace caused the error. Instead, the user must precreate the table and import the table again, specifying IGNORE=Y.

Objects are not imported into the default tablespace if the tablespace does not exist or the user does not have the necessary quotas for their 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 as long as the table is unpartitioned, contains no LOB or VARRAY columns, the table is not a type table, and is not an index-only table with an overflow segment. This scenario can be used to move a 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 tablespace A.
4. Give JOE a quota on tablespace B and make it the default tablespace.
5. Import JOE's tables. (By default, Import puts JOE's tables into tablespace B.)

Character Set and NLS Considerations

This section describes the character set conversions that can take place during export and import operations.

Character Set Conversion

CHAR Data

Up to three character set conversions may be required for character data during an export/import operation:

1. Export writes export files using the character set specified in the NLS_LANG environment variable for the user session. A character set conversion is performed if the value of NLS_LANG differs from the database character set.
2. If the character set in the export file is different than the Import user session character set, Import performs a character set conversion to its user session character set. Import can perform this conversion only if the ratio of the width of the *widest* character in its user session character set to the width of the *smallest* character in the export file character set is one (1).
3. A final character set conversion may be performed if the target database's character set is different from Import's user session character set.

To minimize data loss due to character set conversions, it is advisable to ensure that the export database, the export user session, the import user session, and the import database all use the same character set.

NCHAR Data

Data of datatypes NCHAR, NVARCHAR2, and NCLOB are written to the export file directly in the national character set of the source database. If the national character set of the source database is different than the national character set of the import database, a conversion is performed.

Import and Single-Byte Character Sets

Some 8-bit characters can be lost (that is, 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 5 or 6 export file with a character set different from that of the native operating system or the setting for NLS_LANG, you must set the CHARSET import parameter to specify the character set of the export file.

Refer to the sections Character Set Conversion on page 1-53.

Import and Multi-Byte Character Sets

For multi-byte character sets, Import can convert data to the user-session character set only if the ratio of the width of the widest character in the import character set to the width of the smallest character in the export character set is 1. If the ratio is not 1, the user-session character set should be set to match the export character set, so that Import does no conversion.

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 must be a superset (or equivalent) of the source character set.

For more information, refer to the *Oracle8i National Language Support Guide*.

Considerations when Importing Database Objects

This section describes the behavior of various database objects during Import.

Importing Object Identifiers

The Oracle server assigns object identifiers to uniquely identify object types, object tables, and rows in object tables. These object identifiers are preserved by import.

When you import a table that references a type, but a type of that name already exists in the database, Import attempts to verify that the pre-existing type is in fact the type used by the table (rather than a different type that just happens to have the same name).

To do this, Import compares the type's unique identifier (TOID) with the identifier stored in the export file, and will not import the table rows if the TOIDs do not match.

In some situations, you may not want this validation to occur on specified types (for example, if the types were created by a cartridge installation). You can use the parameter TOID_NOVALIDATE to specify types to exclude from TOID comparison. See TOID_NOVALIDATE on page 2-30 for more information.

Attention: You should be very careful about using TOID_NOVALIDATE as type validation provides a very important capability that helps avoid data corruption. Be sure you feel very confident of your knowledge of type validation and how it works before attempting to import with this feature disabled.

Import uses the following criteria to decide how to handle object types, object tables, and rows in object tables

For object types, if IGNORE=Y and the object type already exists and the object identifiers match, no error is reported. If the object identifiers do not match and the parameter TOID_NOVALIDATE has not been set to ignore the object type, an error is reported and any tables using the object type are not imported.

For object types, if IGNORE=N and the object type already exists, an error is reported. If the object identifiers do not match and the parameter TOID_ NOVALIDATE has not been set to ignore the object type, any tables using the object type are not imported.

For object tables, if IGNORE=Y and the table already exists and the object identifiers match, no error is reported. Rows are imported into the object table. Import of rows may fail if rows with the same object identifier already exist in the object table. If the object identifiers do not match and the parameter TOID_NOVALIDATE has not been set to ignore the object type, an error is reported and the table is not imported.

For object tables, if IGNORE = N and the table already exists, an error is reported and the table is not imported.

Because Import preserves object identifiers of object types and object tables, note the following considerations when importing objects from one schema into another schema, using the FROMUSER and TOUSER parameters:

- If the FROMUSER's object types and object tables already exist on the target system, errors occur because the object identifiers of the TOUSER's object types and object tables are already in use. The FROMUSER's object types and object tables must be dropped from the system before the import is started.
- If an object table was created using the OID AS option to assign it the same object identifier as another table, both tables cannot be imported. One may be imported, but the second receives an error because the object identifier is already in use.

Importing Existing Object Tables and Tables That Contain Object Types

Users frequently pre-create tables before import to reorganize tablespace usage or change a table's storage parameters. The tables must be created with the same definitions as were previously used or a compatible format (except for storage parameters). For object tables and tables that contain columns of object types, format compatibilities are more restrictive.

For tables containing columns of object types, the same object type must be specified and that type must have the same object identifier as the original. If the parameter TOID_NOVALIDATE has been set to ignore the object type, the object IDs do not need to match.

Export writes information about object types used by a table in the Export file, including object types from different schemas. Object types from different schemas used as top level columns are verified for matching name and object identifier at import time. Object types from different schemas that are nested within other object types are not verified.

If the object type already exists, its object identifier is verified. If the parameter TOID_NOVALIDATE has been set to ignore the object type, the object IDs do not need to match. Import retains information about what object types it has created, so that if an object type is used by multiple tables, it is created only once.

Note: In all cases, the object type must be compatible in terms of the internal format used for storage. Import does not verify that the internal format of a type is compatible. If the exported data is not compatible, the results can be unpredictable.

Importing Nested Tables

Inner nested tables are exported separately from the outer table. Therefore, situations may arise where data in an inner nested table might not be properly imported:

- Suppose a table with an inner nested table is exported and then imported without dropping the table or removing rows from the table. If the IGNORE=Y parameter is used, there will be a constraint violation when inserting each row in the outer table. However, data in the inner nested table may be successfully imported, resulting in duplicate rows in the inner table.
- If fatal errors occur inserting data in outer tables, the rest of the data in the outer table is skipped, but the corresponding inner table rows are not skipped. This may result in inner table rows not being referenced by any row in the outer table.
- If an insert to an inner table fails after a non-fatal error, its outer table row will already have been inserted in the outer table and data will continue to be inserted in it and any other inner tables of the containing table. This circumstance results in a partial logical row.
- If fatal errors occur inserting data into an inner table, the import skips the rest of that inner table's data but does not skip the outer table or other nested tables.
- You should always carefully examine the logfile for errors in outer tables and inner tables. To be consistent, table data may need to be modified or deleted.

Because inner nested tables are imported separately from the outer table, attempts to access data from them while importing may produce unexpected results. For example, if an outer row is accessed before its inner rows are imported, an incomplete row may be returned to the user.

Importing REF Data

REF columns and attributes may contain a hidden ROWID that points to the referenced type instance. Import does not automatically recompute these ROWIDs for the target database. You should execute the following command to reset the ROWIDs to their proper values:

```
ANALYZE TABLE [schema.]table VALIDATE REF UPDATE
```

See the *Oracle8i SQL Reference* manual for more information about the ANALYZE TABLE command.

Importing BFILE Columns and Directory Aliases

Export and Import do not copy data referenced by BFILE columns and attributes from the source database to the target database. Export and Import only propagate the names of the files and the directory aliases referenced by the BFILE columns. It is the responsibility of the DBA or user to move the actual files referenced through BFILE columns and attributes.

When you import table data that contains BFILE columns, the BFILE locator is imported with the directory alias and file name that was present at export time. Import does not verify that the directory alias or file exists. If the directory alias or file does not exist, an error occurs when the user accesses the BFILE data.

For operating system directory aliases, if the directory syntax used in the export system is not valid on the import system, no error is reported at import time. Subsequent access to the file data receives an error.

It is the responsibility of the DBA or user to ensure the directory alias is valid on the import system.

Importing Foreign Function Libraries

Import does not verify that the location referenced by the foreign function library is correct. If the formats for directory and file names used in the library's specification on the export file are invalid on the import system, no error is reported at import time. Subsequent usage of the callout functions will receive an error.

It is the responsibility of the DBA or user to manually move the library and ensure the library's specification is valid on the import system.

Importing Stored Procedures, Functions, and Packages

When a local stored procedure, function, or package is imported, it retains its original specification timestamp. The procedure, function, or package is recompiled upon import. If the compilation is successful, it can be accessed by remote procedures without error.

Procedures are exported after tables, views, and synonyms, therefore they usually compile successfully since all dependencies will already exist. However, procedures, functions, and packages are *not* exported in dependency order. If a procedure, function, or package depends on a procedure, function, or package that is stored later in the Export dump file, it will not compile successfully. Later use of the procedure, function, or package will automatically cause a recompile and, if successful, will change the timestamp. This may cause errors in the remote procedures that call it.

Importing Java Objects

When a Java source or class is imported, it retains its original *resolver* (the list of schemas used to resolve Java full names). If the object is imported into a different schema, that resolver may no longer be valid. For example, the default resolver for a Java object in SCOTT's schema is ((* SCOTT) (* PUBLIC)). If the object is imported into BLAKE's schema, it may be necessary to alter the object so that the resolver references BLAKE's schema.

Importing Advanced Queue (AQ) Tables

Importing a queue also imports any underlying queue tables and the related dictionary tables. A queue can be imported only at the granularity level of the queue table. When a queue table is imported, export pre-table and post-table action procedures maintain the queue dictionary.

See *Oracle8i Application Developer's Guide - Advanced Queuing* for more information.

Importing LONG Columns

LONG columns can be up to 2 gigabytes in length. In importing and exporting, the LONG columns must fit into memory with the rest of each row's data. The memory used to store LONG columns, however, does not need to be contiguous because LONG data is loaded in sections.

Importing Views

Views are exported in dependency order. In some cases, Export must determine the ordering, rather than obtaining the order from the server database. In doing so, Export may not always be able to duplicate the correct ordering, resulting in compilation warnings when a view is imported and the failure to import column comments on such views.

In particular, if VIEWA uses the stored procedure PROCB and PROCB uses the view VIEWC, Export cannot determine the proper ordering of VIEWA and VIEWC. If VIEWA is exported before VIEWC and PROCB already exists on the import system, VIEWA receives compilation warnings at import time.

Grants on views are imported even if a view has compilation errors. A view could have compilation errors if an object it depends on, such as a table, procedure, or another view, does not exist when the view is created. If a base table does not exist, the server cannot validate that the grantor has the proper privileges on the base table with the GRANT OPTION.

Therefore, access violations could occur when the view is used, if the grantor does not have the proper privileges after the missing tables are created.

Importing views that contain references to tables in other schemas require that the importer have SELECT ANY TABLE privilege. If the importer has not been granted this privilege, the views will be imported in an uncompiled state. Note that granting the privilege to a role is insufficient. For the view to be compiled, the privilege must be granted directly to the importer.

Importing Tables

Import attempts to create a partitioned table with the same partition or subpartition names as the exported partitioned table, including names of the form SYS_P*nnn*. If a table with the same name already exists, Import processing depends on the value of the IGNORE parameter.

Unless SKIP_UNUSABLE_INDEXES=Y, inserting the exported data into the target table fails if Import cannot update a non-partitioned index or index partition that is marked Indexes Unusable or otherwise not suitable.

Transportable Tablespaces

The transportable tablespace feature enables you to move a set of tablespaces from one Oracle database to another.

To do this, you must make the tablespaces read-only, copy the datafiles of these tablespaces, and use Export/Import to move the database information (metadata) stored in the data dictionary. Both the datafiles and the metadata export file must be copied to the target database. The transport of these files can be done using any facility for copying flat, binary files, such as the operating system copying facility, binary-mode FTP, or publishing on CDs.

After copying the datafiles and importing the metadata, you can optionally put the tablespaces in read-write mode.

See also Transportable Tablespaces on page 1-57 for information on creating an Export file containing transportable tablespace metadata.

Import provides the following parameter keywords to enable import of transportable tablespaces metadata.

- TRANSPORT_TABLESPACE
- TABLESPACES
- DATAFILES
- TTS_OWNERS

See TRANSPORT_TABLESPACE on page 2-31, TABLESPACES on page 2-29, DATAFILES on page 2-21, and TTS_OWNERS on page 2-31 for more information.

> **Additional Information:** See the *Oracle8i Administrator's Guide* for details about how to move or copy tablespaces to another database. For an introduction to the transportable tablespaces feature, see *Oracle8i Concepts*.

Importing Statistics

If statistics are requested at Export time and analyzer statistics are available for a table, Export will place the ANALYZE command to recalculate the statistics for the table into the dump file. In certain circumstances, Export will also write the precalculated optimizer statistics for tables, indexes, and columns to the dump file. See also the description of the Export parameter STATISTICS on page 1-23 and the Import parameter RECALCULATE_STATISTICS on page 2-27.

Because of the time it takes to perform an ANALYZE statement, it is usually preferable for Import to use the precalculated optimizer statistics for a table (and its indexes and columns) rather than executing the ANALYZE statement saved by Export. However, in the following cases, Import will ignore the precomputed statistics because they are potentially unreliable:

- Character set translations between the dump file and the import client and the import database could potentially change collating sequences that are implicit in the precalculated statistics.
- Row errors occurred while importing the table.
- A partition level import is performed (column statistics will no longer be accurate).

Note: Specifying ROWS=N will not prevent the use of precomputed statistics. This feature allows plan generation for queries to be tuned in a non-production database using statistics from a production database.

In certain situations, the importer might want to always use ANALYZE commands rather than using precomputed statistics. For example, the statistics gathered from a fragmented database may not be relevant when the data is Imported in a compressed form. In these cases the importer may specify RECALCULATE_STATISTICS=Y to force the recalculation of statistics.

If you do not want any statistics to be established by Import, you can specify ANALYZE=N, in which case, the RECALCULATE_STATISTICS parameter is ignored. See ANALYZE on page 2-19.

Using Export Files from a Previous Oracle Release

Using Oracle Version 7 Export Files

This section describes guidelines and restrictions that apply when you import data from an Oracle version 7 database into an Oracle8*i* server. Additional information may be found in *Oracle8i Migration*.

Check Constraints on DATE Columns

In Oracle8*i*, check constraints on DATE columns must use the TO_DATE function to specify the format of the date. Because this function was not required in earlier Oracle versions, data imported from an earlier Oracle database might not have used the TO_DATE function. In such cases, the constraints are imported into the Oracle8*i* database, but they are flagged in the dictionary as invalid.

The catalog views DBA_CONSTRAINTS, USER_CONSTRAINTS, and ALL_CONSTRAINTS can be used to identify such constraints. Import issues a warning message if invalid date constraints are in the database.

Using Oracle Version 6 Export Files

This section describes guidelines and restrictions that apply when you import data from an Oracle Version 6 database into an Oracle8*i* server. Additional information may be found in the *Oracle8i Migration* manual.

CHAR columns

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

Syntax of Integrity Constraints

Although the SQL syntax for integrity constraints in Oracle Version 6 is different from the Oracle7 and Oracle8*i* syntax, integrity constraints are correctly imported into Oracle8*i*.

Status of Integrity Constraints

NOT NULL constraints are imported as ENABLED. All other constraints are imported as DISABLED.

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 Oracle8*i*:

```
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 Oracle8*i* server does make this check, however. As a result, tables that could be imported into a Version 6 database may not import into Oracle8*i*.

If the DEFAULT is a value returned by a function, 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 Oracle7, so columns with a default of USER may not be long enough. To determine the maximum size that the USER function returns, 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

Oracle8*i* Import reads Export dump files created by Oracle Version 5.1.22 and later. Keep in mind the following:

- CHAR columns are automatically converted to VARCHAR2
- NOT NULL constraints are imported as ENABLED
- Import automatically creates an index on any clusters to be imported

The CHARSET Parameter

Default: none

Note: This parameter applies to Oracle Version 5 and 6 export files only. Use of this parameter is *not* recommended. It is provided only for compatibility with previous versions. Eventually, it will no longer be supported.

Oracle Version 5 and 6 export files do not contain the NLS character set identifier. However, a version 5 or 6 export file does indicate whether the user session character set was ASCII or EBCDIC.

Use this parameter to indicate the actual character set used at export time. The import utility will verify whether the specified character set is ASCII or EBCDIC based on the character set in the export file.

If you do not specify a value for the CHARSET parameter, Import will verify that the user session character set is ASCII, if the export file is ASCII, or EBCDIC, if the export file is EBCDIC.

If you are using an Oracle7 or Oracle8*i* export file, 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.

Part II

SQL*Loader

3

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
- SQL*Loader Control File
- Input Data and Datafiles
- Data Conversion and Datatype Specification
- Discarded and Rejected Records
- Log File and Logging Information
- Conventional Path Load versus Direct Path Load
- Loading Objects, Collections, and LOBs
- Partitioned and Sub-Partitioned Object Support
- Application Development: Direct Path Load API

SQL*Loader Basics

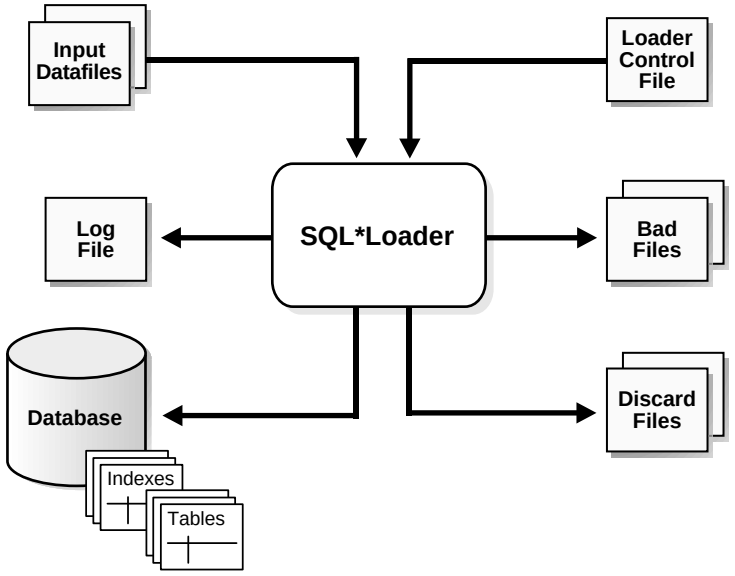
SQL*Loader loads data from external files into tables of an Oracle database.

SQL*Loader:

- Has a powerful data parsing engine which puts little limitation on the format of the data in the datafile.
- Can load data from multiple datafiles during the same load session.
- Can load data into multiple tables during the same load session.
- Is character set aware (you can specify the character set of the data).
- Can selectively load data (you can load records based on the records' values).
- Can manipulate the data before loading it, using SQL functions.
- Can generate unique sequential key values in specified columns.
- Can use the operating system's file system to access the datafile(s).
- Can load data from disk, tape, or named pipe.
- Does sophisticated error reporting which greatly aids troubleshooting.
- Supports two loading "paths" -- Conventional and Direct. While conventional path loading is very flexibility, direct path loading provides superior loading performance (see Chapter 8, "SQL*Loader: Conventional and Direct Path Loads")
- Can load arbitrarily complex object-relational data.
- Supports secondary datafiles for loading of LOBs and collections.
- Is to a large degree compatible with the DB2 Load Utility from IBM; consequently, with no or few changes a DB2 Load Utility control file can be used as a SQL*Loader control file. See Appendix B, "DB2/DXT User Notes".

Figure 3–1, "SQL*Loader Overview" shows the basic components of a SQL*Loader session.

Figure 3–1 SQL*Loader Overview



SQL*Loader takes as input a *control file*, which controls the behavior of SQL*Loader, and one or more *datafiles*. Output of the SQL*Loader is an Oracle database (where the data is loaded), a *log file*, a *bad file*, and potentially a *discard file*.

SQL*Loader Control File

The control file is a text file written in a language that SQL*Loader understands. The control file describes the task that the SQL*Loader is to carry out. The control file tells SQL*Loader where to find the data, how to parse and interpret the data, where to insert the data, and more. See Chapter 4, "SQL*Loader Case Studies" for example control files.

Although not precisely defined, a control file can be said to have three sections:

1. The first section contains session-wide information, for example:
   - global options such as bindsize, rows, records to skip, etc.
   - INFILE clauses to specify where the input data is located
   - data character set specification

2. The second section consists of one or more "INTO TABLE" blocks. Each of these blocks contains information about the table into which the data is to be loadedsuch as the table name and the columns of the table.
3. The third section is optional and, if present, contains input data.

Some control file syntax considerations to keep in mind are:

- The syntax is free-format (statements can extend over multiple lines)
- It is case insensitive, however, strings enclosed in single or double quotation marks are taken literally, including case.
- In control file syntax, comments extend from the two hyphens (--), which mark the beginning of the comment, to the end of the line. Note that the optional third section of the control file is interpreted as data rather than as control file syntax; consequently, comments in this section are not supported.
- Certain words have special meaning to SQL*Loader and are therefore reserved (see Appendix A, "SQL*Loader Reserved Words" for a complete list). If a particular literal or a database object name (column name, table name, etc.) is also a reserved word (keyword), it must be enclosed in single or double quotation marks.

See also Chapter 5, "SQL*Loader Control File Reference" for details about control file syntax and its semantics.

Input Data and Datafiles

The other input to SQL*Loader, other than the control file, is the data. SQL*Loader reads data from one or more files (or operating system equivalents of files) specified in the control file. See INFILE: Specifying Datafiles on page 5-22. From SQL*Loader's perspective, the data in the datafile is organized as *records*. A particular datafile can be in fixed record format, variable record format, or stream record format.

Important: If data is specified inside the control file (that is, INFILE * was specified in the control file), then tthe data is interpreted in the stream record format with the default record terminator.

Fixed Record Format

When all the records in a datafile are of the same byte length, the file is in fixed record format. Although this format is the least flexible, it does result in better performance than variable or stream format. Fixed format is also simple to specify, for example:

```
INFILE <datafile_name> "fix n"
```

specifies that SQL*Loader should interpret the particular datafile as being in fixed record format where every record is *n* bytes long.

Example 3–1 shows a control file that specifies a datafile that should be interpreted in the fixed record format. The datafile in the example contains five physical records. The first physical record is [001, od,] which is exactly eleven bytes (assuming a single-byte character set). The second record is [0002,fghi,] followed by the newline character which is the eleventh byte, etc.

Example 3–1 Loading Data in Fixed Record Format

```
load data
infile 'example.dat'  "fix 11"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1 char(5),
 col2 char(7))

example.dat:
001,   cd, 0002,fghi,
00003,lmn,
1, "pqrs",
0005,uvwx,
```

Variable Record Format

When you specify that a datafile is in variable record format, SQL*Loader expects to find the length of each record in a character field at the beginning of each record in the datafile. This format provides some added flexibility over the fixed record format and a performance advantage over the stream record format. For example, you can specify a datafile which is to be interpreted as being in variable record format as follows:

```
INFILE "datafile_name" "var n"
```

where n specifies the number of bytes in the record length field. Note that if n is not specified, it defaults to five. If it is not specified, SQL*Loader assumes a length of five. Also note that specifying *n* larger than 2^32 -1 will result in an error.

Example 3–2 shows a control file specification that tells SQL*Loader to look for data in the datafile example.dat and to expect variable record format where the record length fields are 3 bytes long. The example.dat datafile consists of three physical records, first specified to be 009 (i.e. 9) bytes long, the second 010 bytes long and the third 012 bytes long. This example also assumes a single-byte character set for the datafile.

Example 3–2 Loading Data in Variable Record Format

```
load data
infile 'example.dat'  "var 3"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1 char(5),
 col2 char(7))

example.dat:
009hello,cd,010world,im,
012my,name is,
```

Stream Record Format (SRF)

Stream record format is the most flexible format. There is, however, some performance impact. In stream record format, records are not specified by size, rather SQL*Loader forms records by scanning for the *record terminator*.

The specification of a datafile to be interpreted as being in stream record format looks like:

```
INFILE <datafile_name> ["str 'terminator_string'"]
```

where the 'terminator_string' is a string specified using alpha-numeric characters. However, in the following cases the terminator_string should be specified as a hexadecimal string (which, if character encoded in the character set of the datafile, would form the desired terminator_string):

- when the terminator_string contains special (non-printable) characters.
- when the terminator_string contains newline or carriage return characters.
- when specifying the terminator_string for a datafile in a character set different than that of the client's (control file's).

If no terminator_string is specified, it defaults to the newline (end-of-line) character(s) (line-feed in Unix-based platforms, carriage return followed by a line-feed on Microsoft platforms, etc.).

Example 3–3 illustrates loading in stream record format where the terminator string is specified using a hex-string. The string X'7c0a', assuming an ASCII character set, translates to '|' followed by the newline character '\n'. The datafile in the example, consists of two records, both properly terminated by the '|\n' string (i.e. X'7c0a').

Example 3–3 Loading Data in Stream Record Format

```
load data
infile 'example.dat'  "str X'7c0a'"
into table example
fields terminated by ',' optionally enclosed by '"'
(col1 char(5),
 col2 char(7))

example.dat:
hello,world,|
james,bond,|
```

Logical Records

SQL*Loader organizes the input data into physical records, according to the specified record format. By default a physical record is a logical record, but for added flexibility, SQL*Loader can be instructed to combine a number of physical records into a logical record.

SQL*Loader can be instructed to follow one of the following two logical record forming strategies:

- Combine a fixed number of physical records to form each logical record.
- Combine physical records into logical records while a certain condition is true.

Case 4: Loading Combined Physical Records on page 4-15 demonstrates using continuation fields to form one logical record from multiple physical records.

For more information see Assembling Logical Records from Physical Records on page 5-36

Data Fields

Once a logical record is formed, field setting on the logical record is done. Field setting is the process where SQL*Loader, based on the control file field specifications, determines what part of the data in the logical record corresponds to which field in the control file. Note that it is possible for two or more field specifications to claim the same data; furthermore, a logical record can contain data which is claimed by no control file field specification.

Most control file field specifications claim a particular part of the logical record. This mapping takes the following forms:

- The byte position of the datafield's beginning, end, or both, can be specified. This specification form is not the most flexible, but it enjoys high field setting performance. See Specifying the Position of a Data Field on page 5-48.
- The string(s) delimiting (enclosing and / or terminating) a particular datafield can be specified. Note that a delimited datafield is assumed to start where the last datafield ended; unless, the byte position of the start of the datafield is specified. See Specifying Delimiters on page 5-69.

- The byte offset and / or the length of the datafield can be specified. This way each field starts a specified number of bytes from where the last one ended and continues for a specified length. See Specifying the Position of a Data Field on page 5-48.
- Length-value datatypes can be used. In this case the first x number of bytes of the data field contain information about how long the rest of the data field is. See SQL*Loader Datatypes on page 5-57.

Data Conversion and Datatype Specification

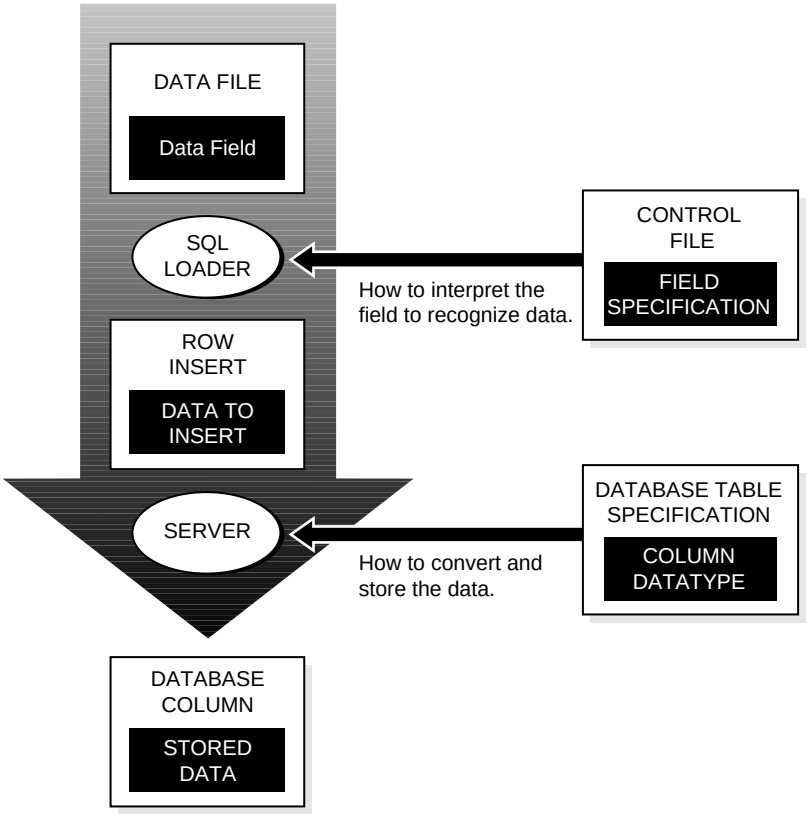
Figure 3–2 shows the stages in which *datafields* in the datafile are converted into *columns* in the database during a conventional path load (direct path loads are conceptually similar, but the implementation is different.) The top of the diagram shows a data record containing one or more datafields. The bottom shows the destination database column. It is important to understand the intervening steps when using SQL*Loader.

Figure 3–2 depicts the "division of labor" between SQL*Loader and the Oracle server. The field specifications tell SQL*Loader how to interpret the format of the datafile. The Oracle server then converts that data and inserts it into the database columns, using the column datatypes as a guide. Keep in mind the distinction between a *field* in a datafile and a *column* in the database. Remember also 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 parse the input data and populate the bind arrays which correspond to a SQL insert statement using that data. The insert statement is then executed by the Oracle server to be stored in the table. The Oracle server uses the datatype of the column to convert the data into its final, stored form. There are two conversion steps:

1. SQL*Loader identifies a field in the datafile, interprets the data, and passes it to the Oracle server via a bind buffer.
2. The Oracle server accepts the data and stores it in the database.

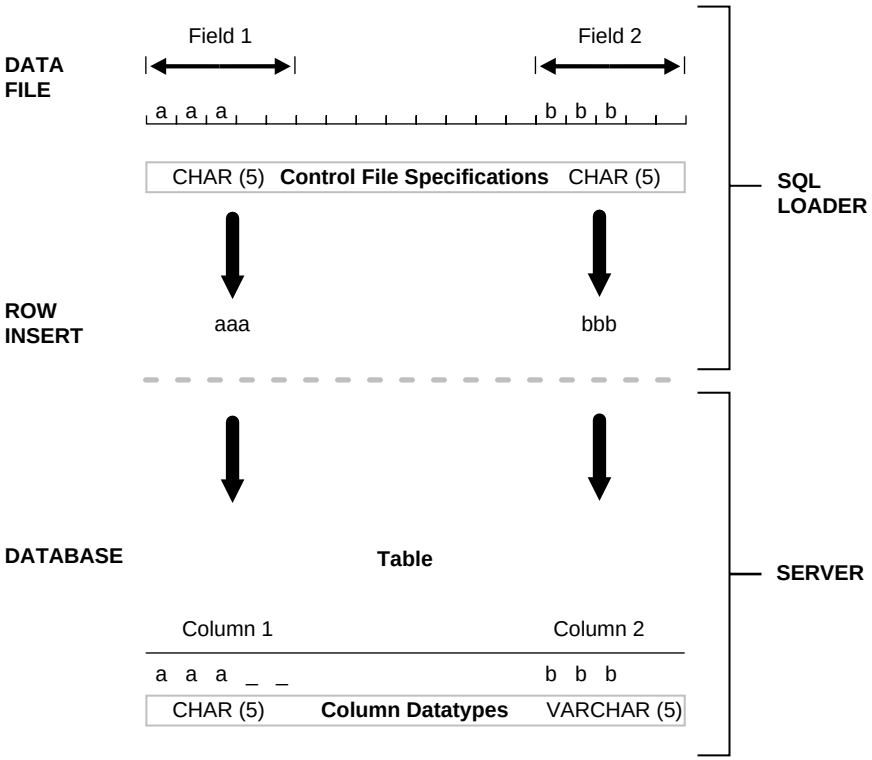
Figure 3–2 Translation of Input Data Field to Oracle Database Column



In Figure 3–3, 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, NCHAR, NVARCHAR, or even a NUMBER column in the database, with the Oracle8*i* server handling any necessary conversions.

By default, SQL*Loader removes trailing spaces from CHAR data before passing it to the database. So, in Figure 3–3, 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.

Figure 3–3 Example of Field Conversion



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 is 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 to be loaded.
- The datatype of the field tells SQL*Loader how to treat the data in the datafile (e.g. bind type). It is *not* the same as the column datatype. SQL*Loader input datatypes are independent of 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 converts data stored in VARRAYs before storing the VARRAY data.
- The distinction between logical records and physical records.

Discarded and Rejected Records

Records read from the input file might not be inserted into the database. Figure 3–4 shows the stages at which records may be *rejected* or *discarded*.

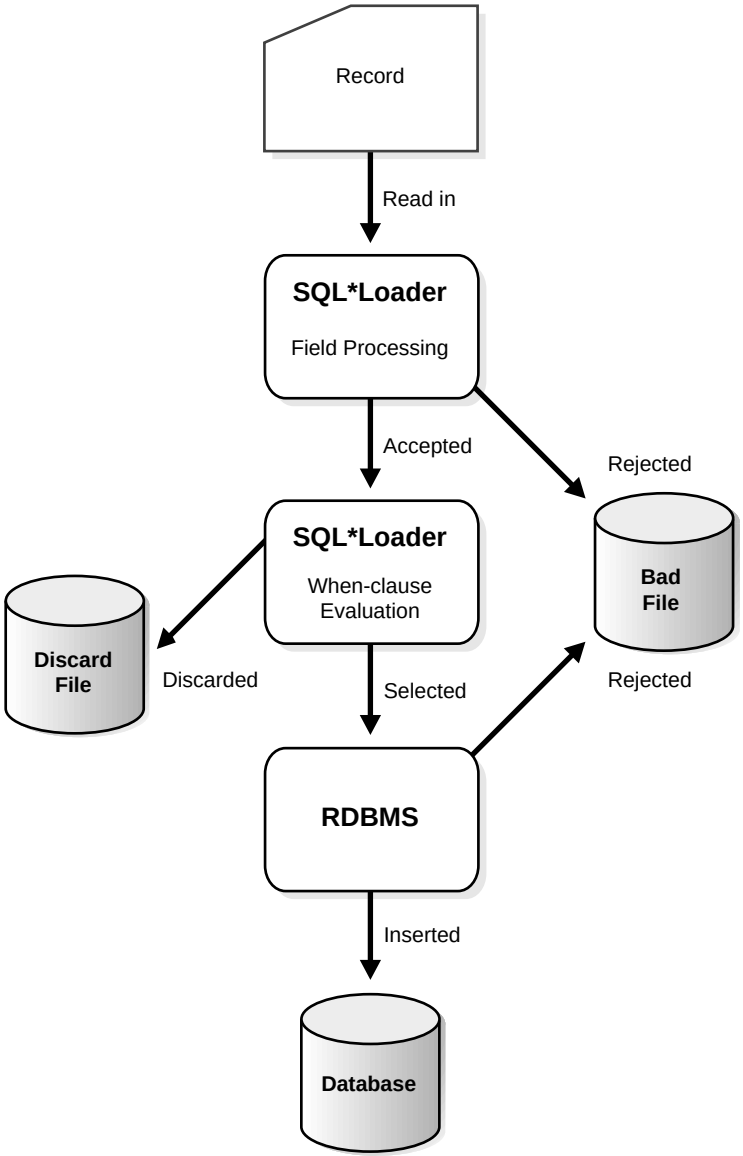
The Bad File

The *bad file* contains records 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 BADFILE: Specifying the Bad File on page 5-25.

Figure 3–4 Record Filtering



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 rejected data can be loaded with the existing control file after necessary corrections are made.

Case 4: Loading Combined Physical Records on page 4-15 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 (see Specifying the Discard File on page 5-27). The discard file contains records that were filtered out of the load because they did not match any record-selection criteria specified in the control file.

The discard file therefore contains records that were not inserted into any table in the database. You can specify the maximum number of such records that the discard file can accept. Data written to any database table 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: Loading Combined Physical Records on page 4-15 shows how the discard file is used. For more details, see Specifying the Discard File on page 5-27.

Log File and 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 Chapter 7, "SQL*Loader: Log File Reference". All of the case studies in Chapter 4 also contain sample log files.

Conventional Path Load versus Direct Path Load

SQL*Loader provides two methods to load data: Conventional Path, which uses a a SQL INSERT statement with a bind array, and Direct Path, which loads data directly into a database. These modes are discussed below and, more thoroughly, in Chapter 8, "SQL*Loader: Conventional and Direct Path Loads". The tables to be loaded must already exist in the database, SQL*Loader never creates tables, it loads existing tables. Tables may already contain data, or they may be empty.

The following privileges are required for a 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 or TRUNCATE option to empty out the table's old data before loading the new data in its place.

Additional Information: For Trusted Oracle, in addition to the above privileges, you must also have write access to all labels you are loading data into in the Trusted Oracle database. See your Trusted Oracle documentation."

Conventional Path

During conventional path loads, the input records are parsed according to the field specifications, and each data field is copied to its corresponding bind array. When the bind array is full (or there is no more data left to read), an array insert is executed. For more information on conventional path loads, see Data Loading Methods on page 8-2. For information on the bind array, see Determining the Size of the Bind Array on page 5-74.

Note that SQL*Loader stores LOB fields after a bind array insert is done. Thus, if there are any errors in processing the LOB field (for example, the LOBFILE could not be found), the LOB field is left empty.

There are no special requirements for tables being loaded via the conventional path.

Direct Path

A direct path load parses the input records according to the field specifications, converts the input field data to the column datatype and builds a column array. The column array is passed to a block formatter which creates data blocks in Oracle database block format. The newly formatted database blocks are written directly to the database bypassing most RDBMS processing. Direct path load is much faster than conventional path load, but entails several restrictions. For more information on the direct path, see Data Loading Methods on page 8-2.

Note: You cannot use direct path for LOBs, VARRAYs, objects, or nested tables.

Parallel Direct Path

A parallel direct path load allows multiple direct path load sessions to concurrently load the same data segments (allows intra-segment parallelism). Parallel Direct Path is more restrictive than Direct Path. For more information on the parallel direct path, see Data Loading Methods on page 8-2.

Loading Objects, Collections, and LOBs

You can use SQL*Loader to bulk load objects, collections, and LOBs. It is assumed that you are familiar with the concept of objects and with Oracle's implementation of object support as described in *Oracle8i Concepts*, and the *Oracle8i Administrator's Guide*.

Supported Object Types

SQL*Loader supports loading of the following two object types:

column-objects

When a column of a table is of some object type, the objects in that column are referred to as *column-objects*. Conceptually such objects are stored in entirety in a single column position in a row. These objects do not have object identifiers and cannot be referenced.

row objects

These objects are stored in tables, known as object tables, that have columns corresponding to the attributes of the object. The object tables have an additional system generated column, called SYS_NC_OID$, that stores system generated unique identifiers (OID) for each of the objects in the table. Columns in other table can refer to these objects by using the OIDs.

Please see Loading Column Objects on page 5-90 and Loading Object Tables on page 5-95 for details on using SQL*Loader control file data definition language to load these object types.

Supported Collection Types

SQL*Loader supports loading of the following two collection types:

Nested Tables

A nested table is a table that appears as a column in another table. All operations that can be performed on other tables can also be performed on nested tables.

VARRAYs

VARRYs are variable sized arrays. An array is an ordered set of built-in types or objects, called *elements*. Each array element is of the same type and has an *index* which is a number corresponding to the element's position in the VARRAY.

When creating a VARRAY type, you must specify the maximum size. Once you have declared a varray type, it can be used as the datatype of a column of a relational table, as an object type attribute, or as a PL/SQL variable.

Please see Loading Collections (Nested Tables and VARRAYs) on page 5-107 for details on using SQL*Loader control file data definition language to load these collection types.

Supported LOB Types

A LOB is a *large object type*. This release of SQL*Loader supports loading of four LOBs types:

- **BLOB:** a LOB containing unstructured binary data.
- **CLOB:** a LOB containing single-byte character data.
- **NCLOB:** a LOB containing fixed size characters from a national character set.
- **BFILE:** a BLOB stored outside of the database tablespaces in a server-side OS file.

LOBs can be column datatypes, and with the exception of the NCLOB, they can be an object's attribute datatypes. LOBs can have an actual value, they can be NULL, or they can be "empty".

Please see Loading LOBs on page 5-98 for details on using SQL*Loader control file data definition language to load these LOB types.

New SQL*Loader DDL Behavior and Restrictions

Note that, in order to provide object support the behavior of certain DDL clauses and certain restrictions are different than in previous releases. These changes apply in all cases, not just when you are loading objects, collections, or LOBs. For example:

- Records:
  - There is no requirement that a LOB from a LOBFILE fit in memory. SQL*Loader reads LOBFILEs in 64K chunks. To load physical records larger than 64K, you can use the READSIZE parameter to specify a larger physical record size. See Secondary Data Files (SDFs) and LOBFILES on page 3-20, READSIZE (read buffer) on page 6-7, and SDF_spec on page 5-14.
  - Logical records must fit completely into the client's available memory. This excludes any data that is part of a particular record, but which is read from a secondary datafile. This logical record size restriction also applies to sub-records within SDFs See Secondary Data Files (SDFs) and LOBFILES on page 3-20.
- Record Formats:
  - Stream Record Format

    In stream record format, the newline character marks the end of a physical record. Starting with release 8.1, you can specify a custom record separator in the OS-file-processing string. See New SQL*Loader DDL Support for Objects, Collections, and LOBs on page 3-20 for more details.
  - Variable Record Format

    The usual syntax of following the INFILE directive with the "var" string (see *Oracle8i Concepts*) has been extended to include the number of characters, at the beginning of each record, which are to be interpreted as the record length specifiers. See the syntax information in Chapter 5.

    Note that the default, in the case that no value is specified, is 5 characters. Also note that the max size of a variable record is 2^32-1; specifying larger values will result in an error.

- DEFAULTIF and NULLIF:

  If the field_condition is true, the DEFAULTIF clause initializes the LOB / Collection to empty (not null).

  If the field_condition is true, the NULLIF clause initializes the LOB / Collection to null as it does for other datatypes.

  Note also that you can chain field_conditions arguments using the AND logical operator. See Chapter 5 for syntax details.

  Notes:

  - A NULLIF / DEFAULTIF clause cannot refer to a field in a secondary datafile (SDF) unless the clause is on a field in the same secondary data file.
  - NULLIF / DEFAULTIF field conditions cannot be based on fields read from LOBFILEs.

- Field Delimiters

  In previous version of SQL Loader, you could load fields which were delimited (terminated or enclosed) by a character. Beginning with this release, the delimiter can be one or more characters long. The syntax to specify delimited fields remains the same except that you can specify entire strings of characters as delimiters.

  As with single character delimiters, when specifying string delimiters, one should take into consideration the character set of the datafile. When the character set of the datafile is different than that of the control file, you can specify the delimiters in hexadecimal (i.e. X'<hexadecimal string>'). If the delimiters are indeed specified in hex., the specification must consist of characters that are valid in the character set of the input datafile. On the other hand, if hexadecimal specification is not used, the delimiter specification is considered to be in the client's (i.e. control file's) character set. In this case, the delimiter is converted into the datafile's character set before searching for the delimiter in the datafile.

  Note the following:

  - Stutter syntax is supported with string delimiters as it was with one character delimiters (i.e. the closing enclosure delimiter can be stuttered).
  - No leading whitespaces in the initial multi-character enclosure delimiter is allowed.
  - If a field is terminated by WHITESPACE, the leading whitespaces are trimmed.

- SQL Strings

  SQL Strings are not supported for LOBs, BFILEs, object columns, nested tables, or varrays, therefore, you cannot specify SQL Strings as part of a filler field specification.

- Filler Fields

  To facilitate loading, you have available a new keyword, FILLER. You use this keyword to specify a filler field which is *a datafile mapped field which does not correspond to a database column*.

  The filler field is assigned values from the datafield to which it is mapped. The filler field can be used as an argument to a number of functions, for example, NULLIF. See the specification for a function's syntax in SQL*Loader's Data Definition Language (DDL) Syntax Diagrams on page 5-3 to see if a filler field can be used as an argument.

  The syntax for a filler field is same as that for a column based field except that a filler field's name is followed by the keyword FILLER.

  Filler fields can be used in field condition specifications in NULLIF, DEFAULTIF, and WHEN clauses. However, they cannot be used in SQL strings.

  Filler field specifications cannot contain a NULLIF / DEFAULTIF clause. **See** Chapter 5, "SQL*Loader Control File Reference" for more detail on the filler field syntax.

  Filler fields are initialized to NULL if the TRAILING NULLCOLS is specified and applicable. Note that if another field references a nullified filler field, an error is generated.

New SQL*Loader DDL Support for Objects, Collections, and LOBs

The following sections discuss new concepts specific to using SQL*Loader to load objects, collections, and LOBs.

Secondary Data Files (SDFs) and LOBFILES

The data to be loaded into some of the new datatypes, like LOBs and collections, can potentially be very lengthy; consequently, it is likely that you will want to have such data instances out of line from the rest of the data. LOBFILES and Secondary Data Files (SDFs) provide a method to separate lengthy data.

LOBFILES LOBFILES are relatively simple datafiles that facilitate LOB loading. The attribute that distinguishes LOBFILEs from the primary datafiles is that in LOBFILEs there is no concept of a record. In LOBFILEs the data is in any of the following type fields:

- Predetermined size fields (fixed length fields).
- Delimited fields (i.e. TERMINATED BY or ENCLOSED BY)

  Note: The clause PRESERVE BLANKS is not applicable to fields read from a LOBFILE.
- Length-Value pair fields (variable length fields) -- VARRAW, VARCHAR, or VARCHARC loader datatypes are used for loading from this type of fields.
- A single LOB field into which the entire contents of a file can be read.

See Chapter 5 for LOBFILE syntax.

Note: A field read from a LOBFILE cannot be used as an argument to a clause (for example, the NULLIF clause).

Secondary Data Files (SDFs) **S**econdary-**D**ata-**F**ile are files similar in concept to the primary datafiles. Like primary datafiles, SDFs are a collection of records and each record is made up of fields. The SDFs are specified on a per control-file-field basis.

You use the SDF keyword to specify SDFs. The SDF keyword can be followed by either the file specification string (see Specifying Filenames and Objects Names on page 5-18) or a filler field (see Secondary Data Files (SDFs) and LOBFILES on page 3-20) which is mapped to a datafield containing file specification string(s).

Note that as for a primary datafile, the following can be specified for each SDF:

- The record format (fixed, stream, or variable). Also, if stream record format is used, you can specify the record separator (see Secondary Data Files (SDFs) and LOBFILES on page 3-20).
- The RECORDSIZE.
- The character set for a SDF can be specified using the CHARACTERSET clause (see Handling Different Character Encoding Schemes, on page 5-30).
- A default delimiter (using the delimiter specification) for the fields which inherit a particular SDF specification (all member fields / attributes of the collection which contain the SDF specification, with exception of the fields containing their own LOBFILE specification).

- To load SDFs larger than 64K, you must use the READSIZE parameter to specify a larger physical record size. You can specify the READSIZE parameter either from the command line or as part of an OPTIONS directive (see OPTIONS on page 5-18). See READSIZE (read buffer) on page 6-7 and SDF_spec on page 5-14.

See Chapter 5 for the SDF syntax.

Full Field Names

Be aware that with SQL*Loader support for complex datatypes like column-objects, the possibility arises that two identical field names could exist in the control file, one corresponding to a column, the other corresponding to a column object's attribute. Certain clauses can refer to fields (for example, WHEN, NULLIF, DEFAULTIF, SID, OID, REF, BFILE, etc.) causing a naming conflict if identically named fields exist in the control file.

Therefore, if you use clauses that refer to fields, you must specify the full name (for example, if field fld1 is specified to be a COLUMN OBJECT and it contains field fld2, when specifying fld2 in a clause such as NULLIF, you must use the full field name fld1.fld2).

When to Use LOBFILEs or SDFs

Say for example, you need to load employee names, employee ids, and employee resumes. You can read the employee names and ids from the main datafile(s), while you could read the resumes, which can be quite lengthy, from LOBFILEs.

Dynamic Versus Static LOBFILE and SDF Specifications

You can specify SDFs and LOBFILEs either statically (you specify the actual name of the file) or dynamically (you use a filler field as the source of the filename). In either case, when the EOF of a SDF/LOBFILE is reached, the file is closed and further attempts at sourcing data from that particular file produce results equivalent to sourcing data from an empty field.

In the case of the dynamic secondary file specification this behavior is slightly different. Whenever the specification changes to reference a new file, the old file is closed and the data is read from the beginning of the newly referenced file.

Note that this dynamic switching of the datasource files has a resetting effect. For example, when switching from the current file to a previously opened file, the previously opened file is reopened, and the data is read from the beginning of the file.

You should not specify the same SDF / LOBFILE as the source of two different fields. If you do so, typically, the two fields will read the data independently.

Restrictions

- Note that, if a non-existent SDF or LOBFILE is specified as a data source for a particular field, that field is initialized to empty, or, if the concept of empty does not apply to the particular field type, the field is initialized to null.
- Note that the POSITION directive cannot be used in fields which read data from LOBFILEs
- Table level delimiters are not inherited by fields which are read from an SDF / LOBFILE.

Partitioned and Sub-Partitioned Object Support

The Oracle8*i* SQL*Loader supports loading partitioned objects in the database. A partitioned object in Oracle is a table or index consisting of *partitions* (pieces) that have been grouped, typically by common logical attributes. For example, sales data for the year 1997 might be partitioned by month. The data for each month is stored in a separate partition of the sales table. Each partition is stored in a separate segment of the database and can have different physical attributes.

Oracle8*i* SQL*Loader Partitioned Object Support enables SQL*Loader to load the following:

- A single partition of a partitioned table
- All partitions of a partitioned table
- Non-partitioned table

Oracle8*i* SQL*Loader supports partitioned objects in all three paths (modes):

- *Conventional Path*: changed minimally from Oracle7, as mapping a row to a partition is handled transparently by SQL.
- *Direct Path*: changed significantly from Oracle7 to accommodate mapping rows to partitions of tables and composite partitions, to support local indexes, functional indexes, and to support global indexes, which can also be partitioned; direct path bypasses SQL and loads blocks directly into the database.
- *Parallel Direct Path*: changed from Oracle7 to include support for concurrent loading of an individual partition and also a partitioned table; allows multiple

direct path load sessions to load the same segment or set of segments concurrently.

Parallel direct path loads are used for intra-segment parallelism. Note that inter-segment parallelism can be achieved by concurrent single partition direct path loads with each load session loading a different partition of the same table.

Application Development: Direct Path Load API

Oracle provides a direct path load API for application developers. Please see the *Oracle Call Interface Programmer's Guide* for more information.

4

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.

This chapter contains the following sections:

The Case Studies

This chapter contains the following case studies:

Case 1: Loading Variable-Length Data 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: Loading Fixed-Format Fields: Loads a datafile with fixed-length fields, stream-format records, all records the same length.

Case 3: Loading a Delimited, Free-Format File 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: Loading Combined Physical Records Combines multiple physical records into one logical record corresponding to one database row

Case 5: Loading Data into Multiple Tables Loads data into multiple tables in one run

Case 6: Loading Using the Direct Path Load Method Loads data using the direct path load method

Case 7: Extracting Data from a Formatted Report Extracts data from a formatted report

Case 8: Loading Partitioned Tables Loads partitioned tables.

Case 9: Loading LOBFILEs (CLOBs) Adds a CLOB column called RESUME to the table emp, uses a FILLER field (RES_FILE), and loads multiple LOBFILEs into the emp table.

Case 10: Loading REF Fields and VARRAYs Loads a customer table, which has a primary key as its OID and which stores order items in a VARRAY and loads an order table which has a REF to the customer table and the order times in a VARRAY.

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.

Table 4–1 lists the files associated with each case:

Table 4–1 Case Studies and Their Related Files

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
8	x	x	x
9	x	x	x
10	x		x

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 and Notes

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.

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 ULCASE10.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 Identifying Data in the Control File with BEGINDATA on page 5-21.
- Including data to be loaded from the control file itself, so there is no separate datafile. See Identifying Data in the Control File with BEGINDATA on page 5-21.
- Loading data in stream format, with both types of delimited fields — terminated and enclosed. See Delimited Fields on page 5-73.

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. Since no datatype is specified, the default is a character of length 255.
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.

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.

Log File

The following shows a portion of the log file:

```
Control File:   ulcase1.ctl
Data File:      ulcase1.ctl
  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 65536 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     *   ,  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:                  65016 bytes(84 rows)
Space allocated for memory besides bind array:       0 bytes

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

Run began on Sun Nov 08 11:08:19 1998
Run ended on Sun Nov 08 11:08:20 1998

Elapsed time was:     00:00:01.16
CPU time was:         00:00:00.10
```

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 Fields

Case 2 demonstrates

- A separate datafile. See INFILE: Specifying Datafiles on page 5-22.
- Data conversions. See Datatype Conversions on page 5-68.

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

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
```

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).

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.

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 65536 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:                  65520 bytes(1092 rows)
Space allocated for memory besides bind array:       0 bytes

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

Run began on Sun Nov 08 11:09:31 1998
Run ended on Sun Nov 08 11:09:32 1998

Elapsed time was:     00:00:00.63
CPU time was:         00:00:00.16
```

Case 3: Loading a Delimited, Free-Format File

Case 3 demonstrates

- Loading data (enclosed and terminated) in stream format. See Delimited Fields on page 5-73.
- Loading dates using the datatype DATE. See DATE on page 5-64.
- Using SEQUENCE numbers to generate unique keys for loaded data. See Setting a Column to a Unique Sequence Number on page 5-55.
- Using APPEND to indicate that the table need not be empty before inserting new records. See Loading into Empty and Non-Empty Tables on page 5-32.
- Using Comments in the control file set off by double dashes. See Control File Basics on page 5-17.
- Overriding general specifications with declarations for individual fields. See Specifying Field Conditions on page 5-44.

Control File

This control file loads the same table as in 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 a maximum of 20. If a length is not specified, the length is a maximum of 20. If a length is not specified, then the length depends on the length of the date mask.
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
```

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.

Log File

The following shows a portion of the log file:

```
Control File:   ulcase3.ctl
Data File:      ulcase3.ctl
  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 65536 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:                  65379 bytes(31 rows)
Space allocated for memory besides bind array:       0 bytes

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

Run began on Sun Nov 08 11:13:41 1998
Run ended on Sun Nov 08 11:13:46 1998

Elapsed time was:     00:00:04.83
CPU time was:         00:00:00.09
```

Case 4: Loading Combined Physical Records

Case 4 demonstrates:

- Combining multiple physical records to form one logical record with CONTINUEIF; see Assembling Logical Records from Physical Records on page 5-36.
- Inserting negative numbers.
- Indicating with REPLACE that the table should be emptied before the new data is inserted; see Loading into Empty and Non-Empty Tables on page 5-32.
- Specifying a discard file in the control file using DISCARDFILE; see Specifying the Discard File on page 5-27.
- Specifying a maximum number of discards using DISCARDMAX; see Specifying the Discard File on page 5-27.
- Rejecting records due to duplicate values in a unique index or due to invalid data values; see Rejected Records on page 5-26.

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 from the logical record. Note that column 1 in each physical record should then contain either an asterisk or a non-data value.

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
*7658 CHIN
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
```

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.

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.dis
 (Allow 999 discards)

Number to load: ALL
Number to skip: 0
Errors allowed: 50
Bind array:     64 rows, maximum of 65536 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.
ORA-01400: cannot insert NULL into ("SCOTT"."EMP"."EMPNO")

Record 9: Rejected - Error on table EMP.
```

```
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:                  65520 bytes(910 rows)
Space allocated for memory besides bind array:       0 bytes

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

Run began on Sun Nov 08 11:49:42 1998
Run ended on Sun Nov 08 11:49:42 1998

Elapsed time was:     00:00:00.69
CPU time was:         00:00:00.13
```

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 Loading Data into Multiple Tables on page 5-52.
- 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 Using Multiple INTO TABLE Statements on page 5-50.
- Using a WHEN clause. See Choosing which Rows to Load on page 5-40.
- Loading the same field (EMPNO) into multiple tables.

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.

Data File

```
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
```

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.

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.dis
 (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 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.EMPIX) violated

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:                 65536 bytes(1024 rows)
```

```
Space allocated for memory besides bind array:        0 bytes

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

Run began on Sun Nov 08 11:54:39 1998
Run ended on Sun Nov 08 11:54:40 1998

Elapsed time was:     00:00:00.67
CPU time was:         00:00:00.16
```

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.

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       ALBERT          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, "SQL*Loader: Conventional and Direct Path Loads".
- How to specify the indexes for which the data is pre-sorted. See Pre-sorting Data for Faster Indexing on page 8-16.
- Loading all-blank numeric fields as null. See Loading All-Blank Fields on page 5-81.
- The NULLIF clause. See NULLIF Keyword on page 5-80.

  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.

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 indexes:presorting data:case study 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 5-81.

Invoking SQL*Loader

Run the script ULCASE6.SQL as SCOTT/TIGER then enter the following at the command line:

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

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.

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
    NULL if MGR = BLANKS
SAL                                32:39    8           CHARACTER
    NULL if SAL = BLANKS
```

```
COMM                                41:48     8           CHARACTER
    NULL if COMM = BLANKS
DEPTNO                              50:51     2           CHARACTER
    NULL if EMPNO = BLANKS


The following index(es) on table EMP were processed:
index SCOTT.EMPIX loaded successfully with 7 keys

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:        0 bytes

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

Run began on Sun Nov 08 11:15:28 1998
Run ended on Sun Nov 08 11:15:31 1998

Elapsed time was:     00:00:03.22
CPU time was:         00:00:00.10
```

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 the chapter on database triggers in *Oracle8i Application Developer's Guide - Fundamentals*
- Use of the SQL string to manipulate data; see Applying SQL Operators to Fields on page 5-87.
- Different initial and trailing delimiters; see Specifying Delimiters on page 5-69.
- Use of SYSDATE; see Setting a Column to the Current Date on page 5-55.
- Use of the TRAILING NULLCOLS clause; see TRAILING NULLCOLS on page 5-42.
- Ambiguous field length warnings; see Conflicting Native Datatype Field Lengths on page 5-68 and Conflicting Character Datatype Field Lengths on page 5-72.

Note: This example creates a trigger that uses the last value of unspecified fields.

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.

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
```

Log File

The following is a portion of the log file:

```
1) SQL*Loader-307: Warning: conflicting lengths 2 and 3 specified for column
DEPTNO table EMP
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
------------------------------ ---------- ----- ---- ---- ---------------------
HIREDATE                                                  SYSDATE
DEPTNO                              1:2     3           CHARACTER
    NULL if DEPTNO = BLANKS
JOB                                7:14     8  WHT      CHARACTER
    NULL if JOB = BLANKS
    SQL string for column : "UPPER(:job)"
```

```
MGR                                 28:31     4  WHT      CHARACTER
    NULL if MGR = BLANKS
ENAME                               34:41     8  WHT      CHARACTER
    SQL string for column : "UPPER(:ename)"
EMPNO                                NEXT     *  WHT      CHARACTER
SAL                                    51     *  WHT      CHARACTER
    SQL string for column : "TO_NUMBER(:sal,'$99,999.99')"
COMM                                 NEXT     *        (  CHARACTER
                                                          %
    SQL string for column : ":comm * 100"

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:                  65286 bytes(81 rows)
Space allocated for memory besides bind array:       0 bytes

   Total logical records skipped:          0
   Total logical records read:            13
   Total logical records rejected:         0
2) Total logical records discarded:        7

Run began on Sun Nov 08 11:16:30 1998
Run ended on Sun Nov 08 11:16:31 1998

Elapsed time was:     00:00:00.75
CPU time was:         00:00:00.09
```

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.

Case 8: Loading Partitioned Tables

Case 8 demonstrates

- Partitioning of data. See *Oracle8i Concepts* for more information on partitioned data concepts.
- Explicitly defined field positions and datatypes.
- Loading using the fixed record length option.

Control File

The control file is ULCASE8.CTL. It loads the lineitem table with fixed length records, partitioning the data according to shipdate.

```
LOAD DATA
1)  INFILE 'ulcase10.dat' "fix 129"
BADFILE 'ulcase10.bad'
TRUNCATE
INTO TABLE lineitem
PARTITION (ship_q1)
2)  (l_orderkey      position    (1:6) char,
     l_partkey       position   (7:11) char,
     l_suppkey       position  (12:15) char,
     l_linenumber    position  (16:16) char,
     l_quantity      position  (17:18) char,
     l_extendedprice position  (19:26) char,
     l_discount      position  (27:29) char,
     l_tax           position  (30:32) char,
     l_returnflag    position  (33:33) char,
     l_linestatus    position  (34:34) char,
     l_shipdate      position  (35:43) char,
     l_commitdate    position  (44:52) char,
     l_receiptdate   position  (53:61) char,
     l_shipinstruct  position  (62:78) char,
     l_shipmode      position  (79:85) char,
     l_comment       position (86:128) char)
```

Notes:

1. Specifies that each record in the datafile is of fixed length (129 characters in this example). See Input Data and Datafiles on page 3-5.
2. Identifies the column name and location of the data in the datafile to be loaded into each column.

Table Creation

In order to partition the data the lineitem table is created using four (4) partitions according to the shipment date:

```
create table lineitem
(l_orderkey     number,
l_partkey       number,
l_suppkey       number,
l_linenumber    number,
l_quantity      number,
l_extendedprice number,
l_discount      number,
l_tax           number,
l_returnflag    char,
l_linestatus    char,
l_shipdate      date,
l_commitdate    date,
l_receiptdate   date,
l_shipinstruct  char(17),
l_shipmode      char(7),
l_comment       char(43))
partition by range (l_shipdate)
(
partition ship_q1 values less than (TO_DATE('01-APR-1996', 'DD-MON-YYYY'))
tablespace p01,
partition ship_q2 values less than (TO_DATE('01-JUL-1996', 'DD-MON-YYYY'))
tablespace p02,
partition ship_q3 values less than (TO_DATE('01-OCT-1996', 'DD-MON-YYYY'))
tablespace p03,
partition ship_q4 values less than (TO_DATE('01-JAN-1997', 'DD-MON-YYYY'))
tablespace p04
)
```

Input Data File

The datafile for this case, ULCASE8.DAT, is listed below. Each record is 129 characters in length. Note that five(5) blanks precede each record in the file.

```
     1 151978511724386.60 7.04.0NO09-SEP-6412-FEB-9622-MAR-96DELIVER IN
PERSONTRUCK  iPBw4mMm7w7kQ zNPL i261OPP
     1 2731 73223658958.28.09.06NO12-FEB-9628-FEB-9620-APR-96TAKE BACK RETURN
MAIL   5wM04SNyl0AnghCP2nx lAi
     1 3370 3713 810210.96 .1.02NO29-MAR-9605-MAR-9631-JAN-96TAKE BACK RETURN
REG AIRSQC2C 5PNCy4mM
     1 5214 46542831197.88.09.06NO21-APR-9630-MAR-9616-MAY-96NONE
AIR    Om0L65CSAwSj5k6k
     1 6564  6763246897.92.07.02NO30-MAY-9607-FEB-9603-FEB-96DELIVER IN
PERSONMAIL   CB0SnyOL PQ32B70wB75k 6Aw10m0wh
     1 7403 160524 31329.6 .1.04NO30-JUN-9614-MAR-9601 APR-96NONE
FOB    C2gOQj OB6RLk1BS15 igN
     2 8819 82012441659.44  0.08NO05-AUG-9609-FEB-9711-MAR-97COLLECT COD
AIR    O52M70MRgRNnmm476mNm
     3 9451 721230 41113.5.05.01AF05-SEP-9629-DEC-9318-FEB-94TAKE BACK RETURN
FOB    6wQnO0Llg6y
     3 9717  1834440788.44.07.03RF09-NOV-9623-DEC-9315-FEB-94TAKE BACK RETURN
SHIP   LhiA7wygz0k4g4zRhMLBAM
     3 9844 1955 6 8066.64.04.01RF28-DEC-9615-DEC-9314-FEB-94TAKE BACK RETURN
REG AIR6nmBmjQkgiCyzCQBkxPPOx5j4hB 0lRywgniP1297
```

Invoking SQL*Loader

Invoke SQL*Loader with a command such as:

```
sqlldr scott/tiger control=ulcase8.ctl data=ulcase8.dat
```

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

Log File

The following shows a portion of the log file:

```
Control File:   ulcase8.ctl
Data File:      ulcase8.dat
  File processing option string: "fix 129"
  Bad File:     ulcase10.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 LINEITEM, partition SHIP_Q1, loaded from every logical record.
Insert option in effect for this partition: TRUNCATE

   Column Name                  Position   Len  Term Encl Datatype
------------------------------ ---------- ----- ---- ---- ---------------------
L_ORDERKEY                          1:6     6           CHARACTER
L_PARTKEY                           7:11    5           CHARACTER
L_SUPPKEY                          12:15    4           CHARACTER
L_LINENUMBER                       16:16    1           CHARACTER
L_QUANTITY                         17:18    2           CHARACTER
L_EXTENDEDPRICE                    19:26    8           CHARACTER
L_DISCOUNT                         27:29    3           CHARACTER
L_TAX                              30:32    3           CHARACTER
L_RETURNFLAG                       33:33    1           CHARACTER
L_LINESTATUS                       34:34    1           CHARACTER
L_SHIPDATE                         35:43    9           CHARACTER
L_COMMITDATE                       44:52    9           CHARACTER
L_RECEIPTDATE                      53:61    9           CHARACTER
L_SHIPINSTRUCT                     62:78   17           CHARACTER
L_SHIPMODE                         79:85    7           CHARACTER
L_COMMENT                         86:128   43           CHARACTER

Record 4: Rejected - Error on table LINEITEM, partition SHIP_Q1.
ORA-14401: inserted partition key is outside specified partition

Record 5: Rejected - Error on table LINEITEM, partition SHIP_Q1.
ORA-14401: inserted partition key is outside specified partition

Record 6: Rejected - Error on table LINEITEM, partition SHIP_Q1.
ORA-14401: inserted partition key is outside specified partition

Record 7: Rejected - Error on table LINEITEM, partition SHIP_Q1.
ORA-14401: inserted partition key is outside specified partition

Record 8: Rejected - Error on table LINEITEM, partition SHIP_Q1.
ORA-14401: inserted partition key is outside specified partition
```

```
Record 9: Rejected - Error on table LINEITEM, partition SHIP_Q1.
ORA-14401: inserted partition key is outside specified partition

Record 10: Rejected - Error on table LINEITEM, partition SHIP_Q1.
ORA-14401: inserted partition key is outside specified partition


Table LINEITEM, partition SHIP_Q1:
  3 Rows successfully loaded.
  7 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:                 65532 bytes(381 rows)
Space allocated for memory besides bind array:      0 bytes

Total logical records skipped:          0
Total logical records read:            10
Total logical records rejected:         7
Total logical records discarded:        0

Run began on Sun Nov 08 11:30:49 1998
Run ended on Sun Nov 08 11:30:50 1998

Elapsed time was:     00:00:01.11
CPU time was:         00:00:00.14
```

Case 9: Loading LOBFILEs (CLOBs)

Case 9 demonstrates

- Adding a CLOB column called RESUME to the table emp.
- Using a FILLER field (RES_FILE).
- Loading multiple LOBFILEs into the emp table.

Control File

The control file is ULCASE9.CTL. It loads new emp records with the resume for each employee coming from a different file.

```
LOAD DATA
INFILE *
INTO TABLE EMP
REPLACE
FIELDS TERMINATED BY ',' 
( EMPNO    INTEGER EXTERNAL,
  ENAME    CHAR,
  JOB      CHAR,
  MGR      INTEGER EXTERNAL,
  SAL      DECIMAL EXTERNAL,
  COMM     DECIMAL EXTERNAL,
  DEPTNO   INTEGER EXTERNAL,
1)  RES_FILE FILLER CHAR,
2)  "RESUME" LOBFILE (RES_FILE) TERMINATED BY EOF NULLIF RES_FILE = 'NONE'
)
BEGINDATA
7782,CLARK,MANAGER,7839,2572.50,,10,ulcase91.dat
7839,KING,PRESIDENT,,5500.00,,10,ulcase92.dat
7934,MILLER,CLERK,7782,920.00,,10,ulcase93.dat
7566,JONES,MANAGER,7839,3123.75,,20,ulcase94.dat
7499,ALLEN,SALESMAN,7698,1600.00,300.00,30,ulcase95.dat
7654,MARTIN,SALESMAN,7698,1312.50,1400.00,30,ulcase96.dat
7658,CHAN,ANALYST,7566,3450.00,,20,NONE
```

Notes:

1. This is a filler field. The filler field is assigned values from the datafield to which it is mapped. See Secondary Data Files (SDFs) and LOBFILES on page 3-20 for more information.

2. RESUME is loaded as a CLOB. The LOBFILE function is used to specify the name of the field that specifies name of the file which contains the data for the LOB field. See Loading LOB Data Using LOBFILEs on page 5-101 for more information.

Input Data Files

```
>>ulcase91.dat<<
                          Resume for Mary Clark

Career Objective: Manage a sales team with consistent record breaking
                  performance.
Education:        BA Business University of Iowa 1992
Experience:       1992-1994 - Sales Support at MicroSales Inc.
                  Won "Best Sales Support" award in 1993 and 1994
                  1994-Present - Sales Manager at MicroSales Inc.
                  Most sales in mid-South division for 2 years

>>ulcase92.dat<<

                       Resume for Monica King
Career Objective: President of large computer services company
Education:        BA English Literature Bennington, 1985
Experience:       1985-1986 - Mailroom at New World Services
                  1986-1987 - Secretary for sales management at
                              New World Services
                  1988-1989 - Sales support at New World Services
                  1990-1992 - Saleman at New World Services
                  1993-1994 - Sales Manager at New World Services
                  1995      - Vice President of Sales and Marketing at
                              New World Services
                  1996-Present - President of New World Services

>>ulcase93.dat<<

                         Resume for Dan Miller

Career Objective: Work as a sales support specialist for a services
                  company
Education:        Plainview High School, 1996
Experience:       1996 - Present: Mail room clerk at New World Services

>>ulcase94.dat<<
```

```
                        Resume for Alyson Jones

Career Objective: Work in senior sales management for a vibrant and
                  growing company
Education:        BA Philosophy Howard Univerity 1993
Experience:       1993 - Sales Support for New World Services
                  1994-1995 - Salesman for New World Services.  Led in
                  US sales in both 1994 and 1995.
                  1996 - present - Sales Manager New World Services.  My
                  sales team has beat its quota by at least 15% each
                  year.

>>ulcase95.dat<<

                          Resume for David Allen

Career Objective: Senior Sales man for agressive Services company
Education:        BS Business Administration, Weber State 1994
Experience:       1993-1994 - Sales Support New World Services
                  1994-present - Salesman at New World Service.  Won sales
                  award for exceeding sales quota by over 20%
                  in 1995, 1996.

>>ulcase96.dat<<

                         Resume for Tom Martin

Career Objective: Salesman for a computing service company
Education:        1988 - BA Mathematics, University of the North
Experience:       1988-1992 Sales Support, New World Services
                  1993-present Salesman New World Services
```

Invoking SQL*Loader

Invoke SQL*Loader with a command such as:

```
sqlldr sqlldr/test control=ulcase9.ctl data=ulcase9.dat
```

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

Log File

The following shows a portion of the log file:

```
Control File:   ulcase9.ctl
Data File:      ulcase9.ctl
  Bad File:     ulcase9.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 from every logical record.
Insert option in effect for this table: REPLACE

   Column Name                  Position   Len  Term Encl Datatype
------------------------------ ---------- ----- ---- ---- ---------------------
EMPNO                               FIRST     *   ,       CHARACTER
ENAME                                NEXT     *   ,       CHARACTER
JOB                                  NEXT     *   ,       CHARACTER
MGR                                  NEXT     *   ,       CHARACTER
SAL                                  NEXT     *   ,       CHARACTER
COMM                                 NEXT     *   ,       CHARACTER
DEPTNO                               NEXT     *   ,       CHARACTER
RES_FILE                             NEXT     *   ,       CHARACTER
  (FILLER FIELD)
"RESUME"                          DERIVED     *  WHT      CHARACTER
    Dynamic LOBFILE.  Filename in field RES_FILE
    NULL if RES_FILE = 0X4e4f4e45(character 'NONE')


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:                  63984 bytes(31 rows)
Space allocated for memory besides bind array:       0 bytes
```

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

Run began on Sun Nov 08 11:31:11 1998
Run ended on Sun Nov 08 11:31:19 1998

Elapsed time was:     00:00:08.14
CPU time was:         00:00:00.09
```

Case 10: Loading REF Fields and VARRAYs

Case 10 demonstrates

- Loading a customer table, which has a primary key as its OID and which stores order items in a VARRAY.
- Loading an order table which has a REF to the customer table and the order times in a VARRAY.

Control File

```
LOAD DATA
INFILE *
CONTINUEIF THIS (1) = '*'
INTO TABLE customers
replace
fields terminated by ","
(
  cust_no                       char,
  name                          char,
  addr                          char
)
INTO TABLE orders
replace
fields terminated by ","
(
  order_no                      char,
1)  cust_no             FILLER    char,
2)  cust                          REF (CONSTANT 'CUSTOMERS', cust_no),
1)  item_list_count     FILLER    char,
3) item_list                       varray count (item_list_count)
  (
4)  item_list                    column object
    (
5)    item                      char,
      cnt                       char,
      price                     char
    )
  )
)
6) BEGINDATA
*00001,Spacely Sprockets,15 Space Way,
*00101,00001,2,
*Sprocket clips, 10000, .01,
```

```
Sprocket cleaner, 10, 14.00
*00002,Cogswell Cogs,12 Cogswell Lane,
*00100,00002,4,
*one quarter inch cogs,1000,.02,
*one half inch cog, 150, .04,
*one inch cog, 75, .10,
*Custom coffee mugs, 10, 2.50
```

Notes:

1. This is a filler field. The filler field is assigned values from the datafield to which it is mapped. See Secondary Data Files (SDFs) and LOBFILES on page 3-20 for more information.
2. This field is created as a REF field. See
3. `item_list` is stored in a VARRAY.
4. The second occurence of item_list identifies the datatype of each element of the VARRAY. Here, the datatype is a column object.
5. This list shows all attributes of the column object that are loaded for the VARRAY. The list is enclosed in parenthesis. See section Loading Column Objects on page 5-90 for more information about loading column objects.
6. The data is contained in the control file and is preceded by the keyword BEGINDATA.

Invoking SQL*Loader

Invoke SQL*Loader with a command such as:

```
sqlldr sqlldr/test control=ulcase10.ctl
```

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

Log File

The following shows a portion of the log file:

```
Control File:   ulcase10.ctl
Data File:      ulcase10.ctl
  Bad File:     ulcase10.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:   1:1 = 0X2a(character '*'), in current physical record
Path used:      Conventional

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

   Column Name                  Position   Len  Term Encl Datatype
------------------------------ ---------- ----- ---- ---- ---------------------
CUST_NO                            FIRST     *   ,       CHARACTER
NAME                                NEXT     *   ,       CHARACTER
ADDR                                NEXT     *   ,       CHARACTER

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

   Column Name                  Position   Len  Term Encl Datatype
------------------------------ ---------- ----- ---- ---- ---------------------
ORDER_NO                            NEXT     *   ,       CHARACTER
CUST_NO                             NEXT     *   ,       CHARACTER
  (FILLER FIELD)
CUST                             DERIVED                 REF
    Arguments are:
        CONSTANT 'CUSTOMERS'
        CUST_NO
ITEM_LIST_COUNT                     NEXT     *   ,       CHARACTER
  (FILLER FIELD)
ITEM_LIST                        DERIVED     *           VARRAY
    Count for VARRAY
        ITEM_LIST_COUNT

*** Fields in ITEM_LIST
ITEM_LIST                        DERIVED     *           COLUMN OBJECT

*** Fields in ITEM_LIST.ITEM_LIST
ITEM                               FIRST     *   ,       CHARACTER
CNT                                 NEXT     *   ,       CHARACTER
PRICE                               NEXT     *   ,       CHARACTER
*** End of fields in ITEM_LIST.ITEM_LIST
```

```
*** End of fields in ITEM_LIST


Table CUSTOMERS:
  2 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.


Table ORDERS:
  2 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:                 65240 bytes(28 rows)
Space allocated for memory besides bind array:      0 bytes

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

Run began on Sun Nov 08 11:46:13 1998
Run ended on Sun Nov 08 11:46:14 1998

Elapsed time was:     00:00:00.65
CPU time was:         00:00:00.16
```

5

SQL*Loader Control File Reference

This chapter describes the SQL*Loader control file syntax. The following topics are included:

SQL*Loader's Data Definition Language (DDL)

SQL*Loader's Control File: Load Configuration

SQL*Loader's Control File: Loading Data

SQL*Loader's Control File: Loading Objects, LOBs, and Collections

SQL*Loader's Data Definition Language (DDL) Syntax Diagrams

You use SQL*Loader's data definition language (DDL) to control how SQL*Loader performs a data load into your database. You can also use DDL to manipulate the data you are loading.

The SQL*Loader Control File

The SQL*Loader control file is a repository that contains the DDL instructions that you have created to control where SQL*Loader will find the data to load, how SQL*Loader expects that data to be formatted, how SQL*Loader will be configured (memory management, rejecting records, interrupted load handling, etc.) as it loads the data, and how it will manipulate the data being loaded. You create the SQL*Loader control file and its contents using a simple text editor like vi, or xemacs.

The rest of this chapter explains how to use DDL to achieve your required data load.

SQL*Loader DDL Syntax Diagram Notation

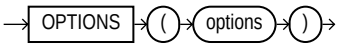
The SQL*Loader DDL diagrams (sometimes called "railroad diagrams") used in this chapter to illustrate syntax use standard SQL syntax notation. For more information about the syntax notation used in this chapter, see the *PL/SQL User's Guide and Reference* or the preface in the *Oracle8i SQL Reference*.

See Control File Basics on page 5-17 for more information

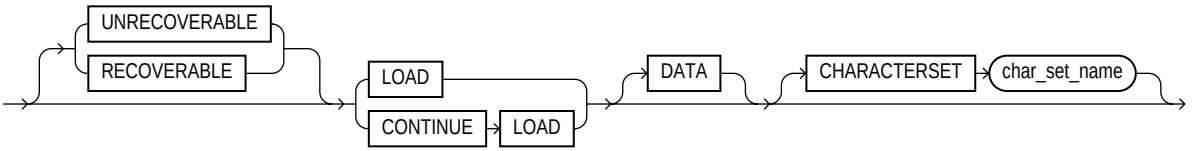
High-Level Syntax Diagrams

The following diagrams of DDL syntax are shown with certain clauses collapsed (position_spec, into_table clause, etc.). These diagrams are expanded and explained in more detail in Expanded DDL Syntax on page 5-15.

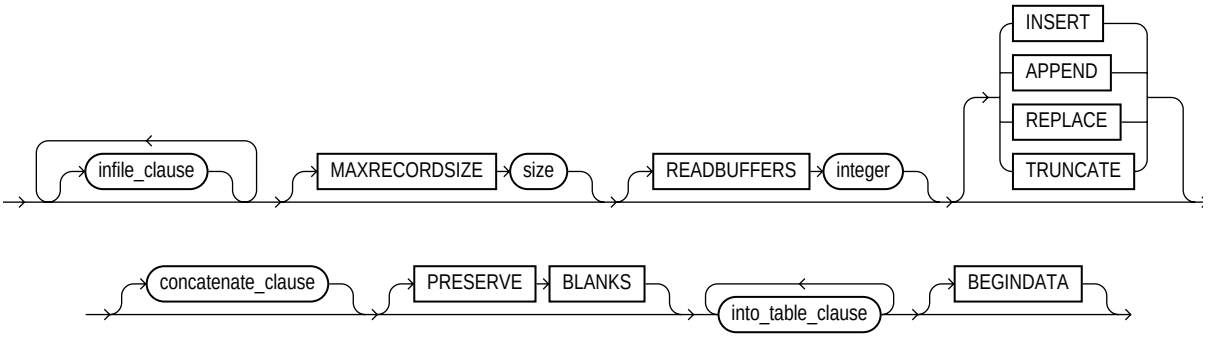
Options Clause



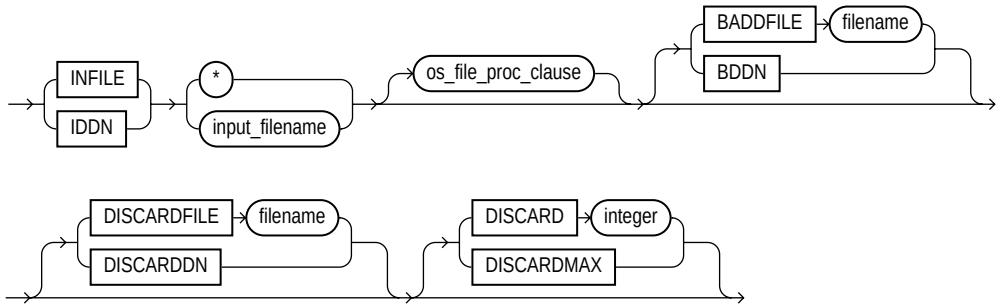
Load Statement



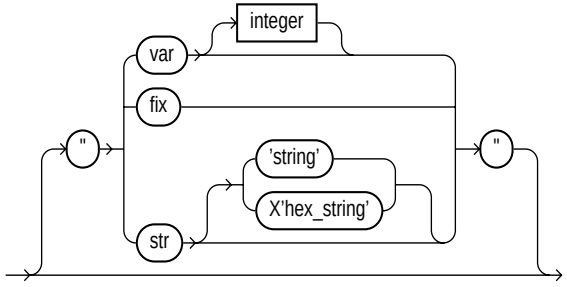
Note: The characterset specified does not apply to data in the control file.



infile_clause

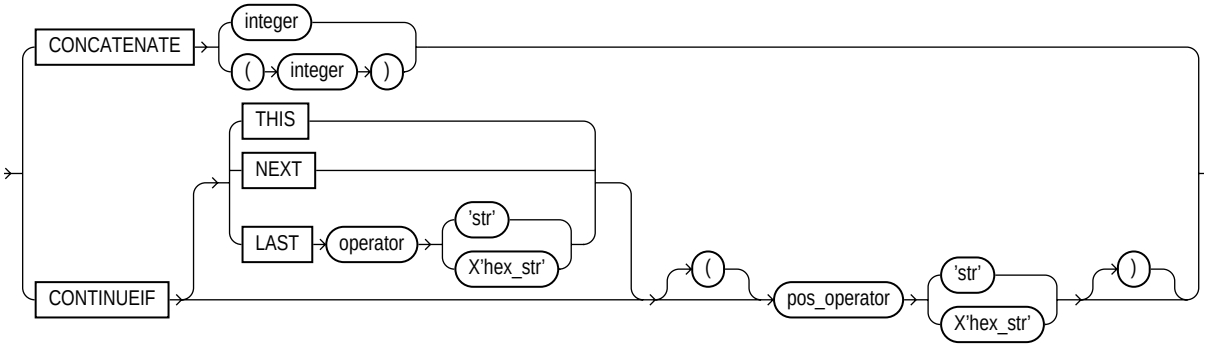


os_file_proc_clause

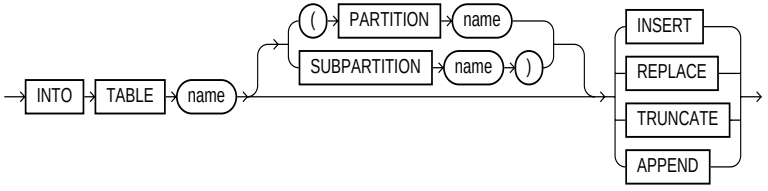


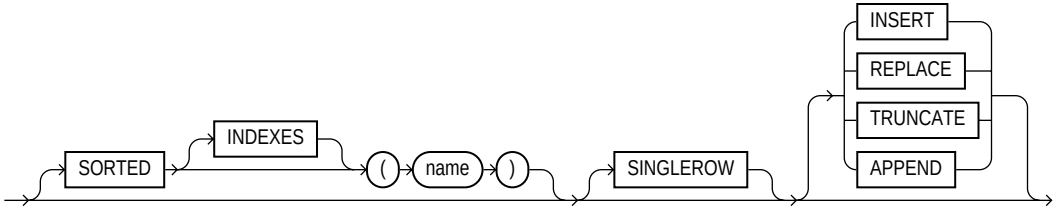
Important: The syntax above is specific to the Unix platform. Please see your Oracle operating system-specific documentation for the syntax required by your platform.

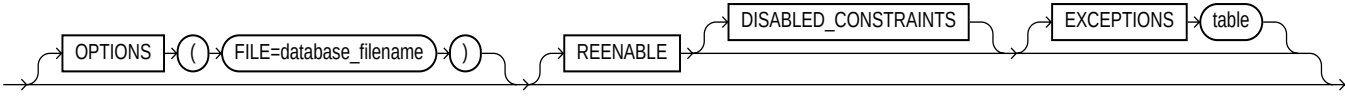
concatenate_clause

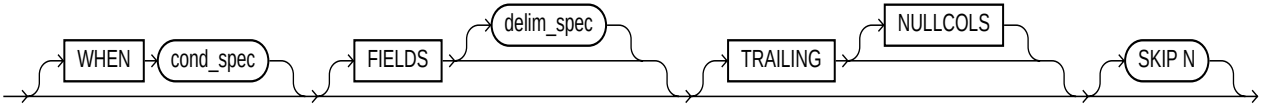


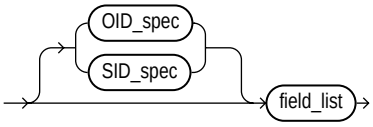
into_table_clause

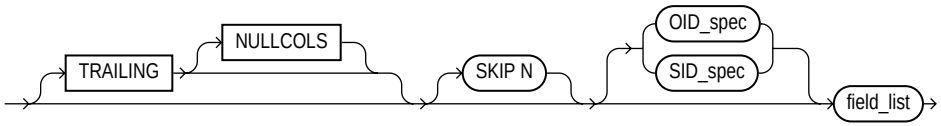




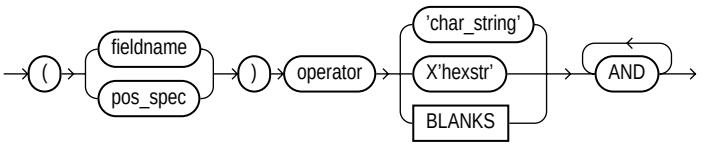




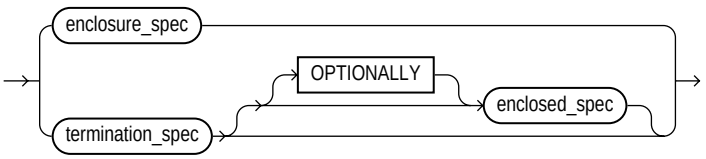




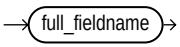
cond_spec



delim_spec

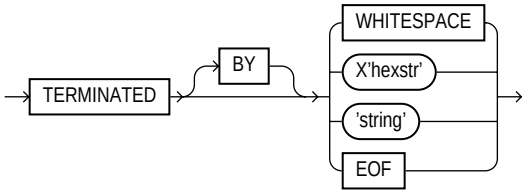


full_fieldname



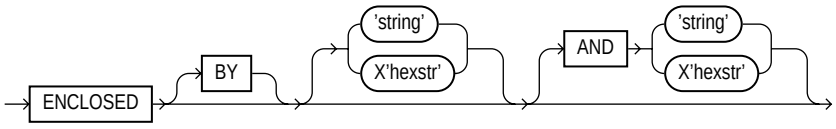
Note: `full_fieldname` is the full name of a field specified using dot notation. If the field `col2` is an attribute of a column object `col1`, when referring to `col2` in one of the directives, you must use the notation `col1.col2`. The `column_name` and the `full_fieldname` referencing/naming the same entity can be different because `column_name` never includes the full name of the entity (no dot notation).

termination_spec



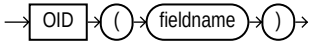
Note: Only fields which are loaded from a LOBFILE can be terminated by EOF.

enclosure_spec



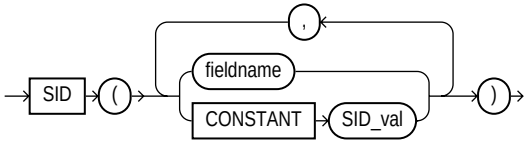
Note: Fields terminated by EOF cannot be enclosed.

OID_spec

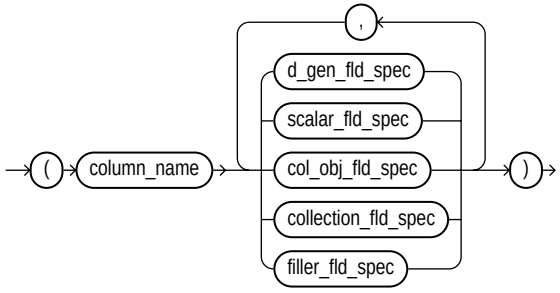


Note: If the table uses primary key OIDs instead of system-generated OIDs, do not specify an OID clause.

SID_spec

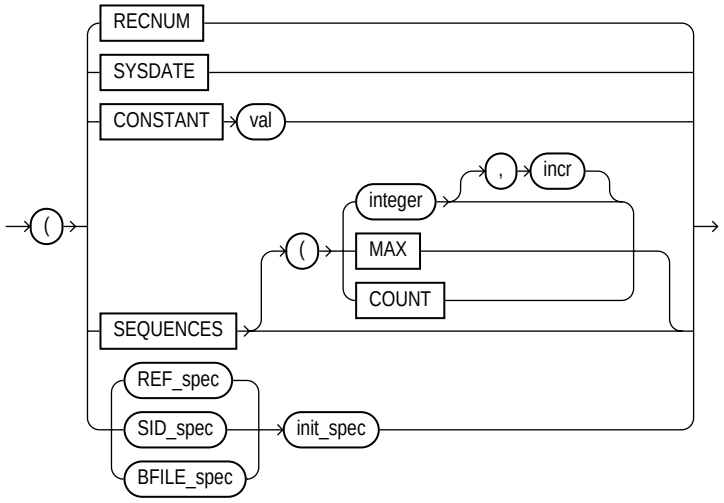


field_list

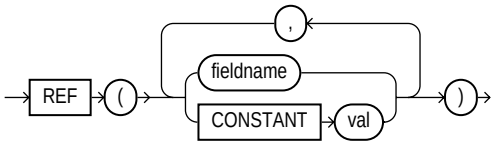


Note: The `column_name` and the `fieldname` referencing / naming the same entity can be different because `column_name` never includes the full name of the entity (no dot notation).

d_gen_fld_spec



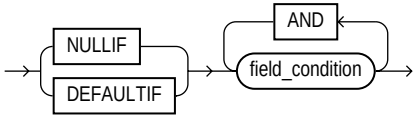
REF_spec



Notes:

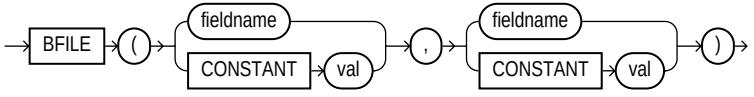
- The first argument to the REF directive is assumed to be the table name.
- If the REF column is a primary key REF, then the relative ordering of the arguments to the REF directive must match the relative ordering of the columns making up the primary key REF (i.e. the relative ordering of the columns making up the primary key OID in the object table).

init_spec



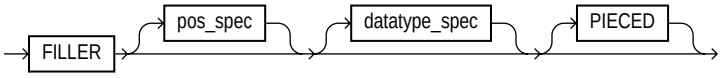
Note: No field_condition can be based on fields in a secondary data file (SDF).

BFILE_spec



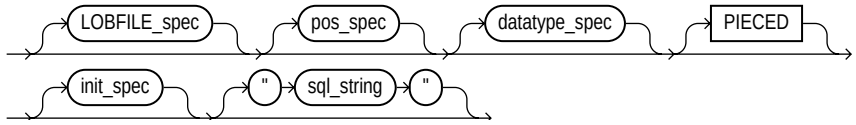
Note: The first argument to the BFILE directive contains the DIRECTORY OBJECT (the server_directory alias). The second argument contains the filename.

filler_fld_spec



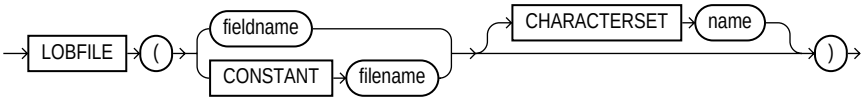
Note: Conventional path loading does piecing when necessary. During direct path loads, piecing is done automatically, therefore, it is unnecessary to specify the PIECED keyword.

scalar_fld_spec



Note: Conventional path loading does piecing when necessary. During direct path loads, piecing is done automatically, therefore, it is unnecessary to specify the PIECED keyword. Note also that you cannot specify `sql_string` for LOB fields (regardless of whether LOBFILE_spec is specified).

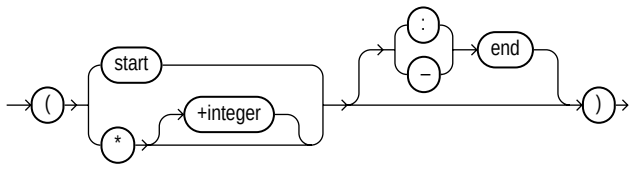
LOBFILE_spec



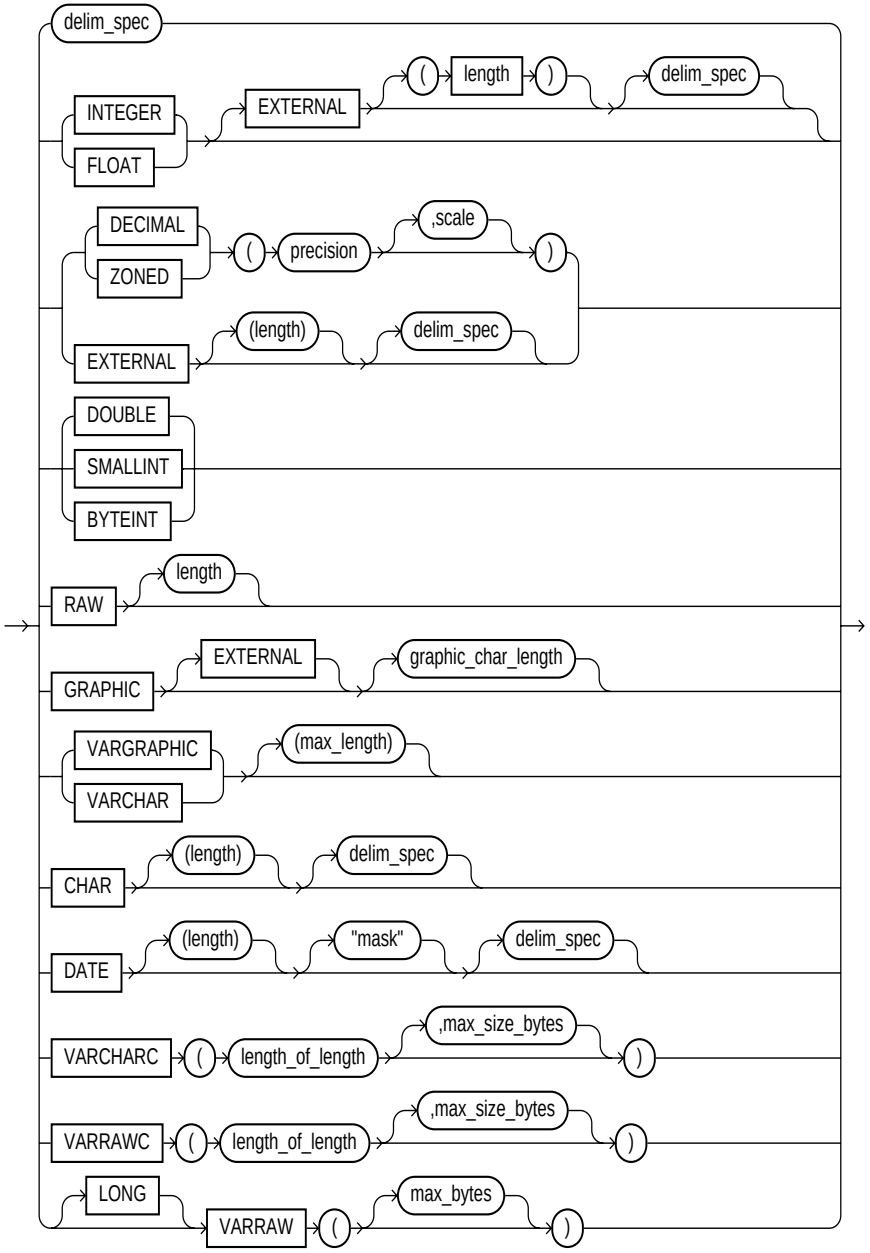
Notes:

- You cannot use pos_spec if the data is loaded from a LOBFILE.
- Only LOBs can be loaded from LOBFILES.

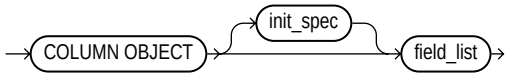
pos_spec



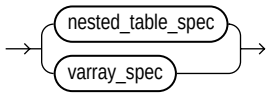
datatype_spec



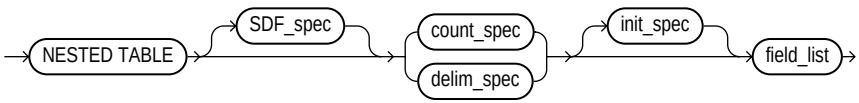
col_obj_fld_spec



collection_fld_spec

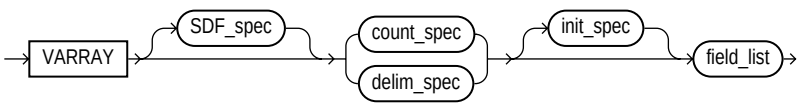


nested_table_spec



Note: `field_list` cannot contain a `collection_fld_spec`

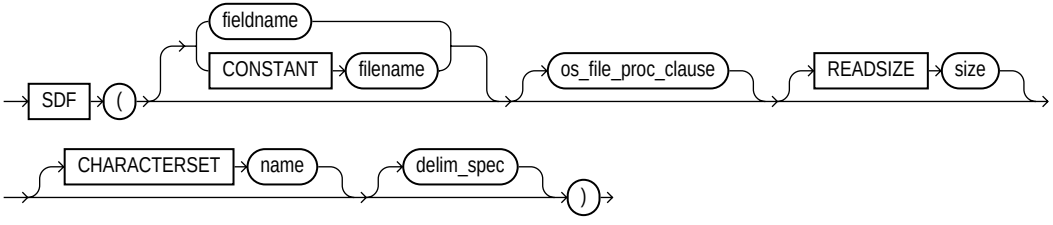
VARRAY_spec



Notes: A `col_obj_spec` nested within a VARRAY cannot contain a `collection_fld_spec`.

The `<column_name>` specified as part of the `field_list` must be the same as the `<column_name>` preceding the keyword VARRAY.

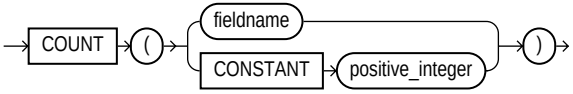
SDF_spec



Notes:

- Only a collection_fld_spec can name a SDF as its datasource.
- The delim_spec is used as the default delimiter for all the fields described as part of the field_list of a collection_fld_spec

count_spec



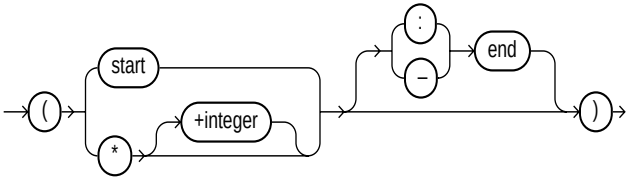
Note: If a field is specified as the argument to the COUNT clause, that field must be mapped into the datafile data which is convertible to an integer (e.g. the string of characters "124").

Expanded DDL Syntax

Position Specification

pos_spec

A position specification (*pos_spec*) provides the starting location for a field and, optionally, the ending location. *pos_spec* syntax is:

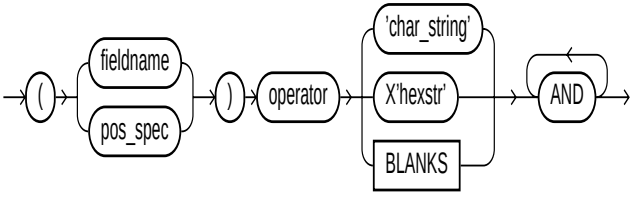


The position must be surrounded by parentheses. The starting location can be specified as a column number, as * (next column), or *+n (next column plus an offset). The *start* and *end* locations can be separated with a colon (:) or a dash (-).

Field Condition

field_condition

A field condition compares a named field or an area of the record to a specified 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. *field_condition* syntax is:



char_string and *hex_string* can be enclosed in either single quotation marks or double quotation marks. *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. BLANKS is required when you are loading delimited data and you cannot predict the length of the field, or when you use a multi-byte character set that has multiple blanks.

There must not be any spaces between the operator and the operands. For example:

```
(1)='x'
```

is legal, while

```
(1) = 'x'
```

generates an error.

Column Name
column_name

The column name you specify in a field condition must be one of the columns defined for the input record. It must be specified with double quotation marks if its name is a reserved word. See Specifying Filenames and Objects Names on page 5-18 for more details.

Precision vs. Length
precision
length

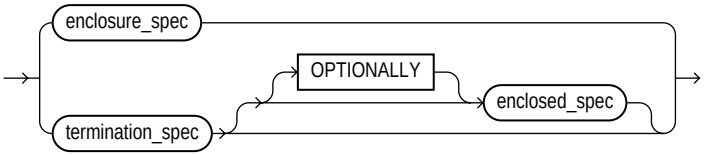
The precision of a numeric field is the number of digits it contains. The length of a numeric field is the number of byte positions on the record. The byte length of a ZONED decimal field is the same as its precision. However, the byte length of a (packed) DECIMAL field is (p+1)/2, rounded up, where p is the number's precision, because packed numbers contain two digits (or digit and sign) per byte.

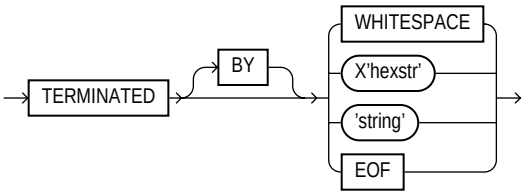
Date Mask

The date mask specifies the format of the date value. For more information, see the DATE datatype on page 5-64.

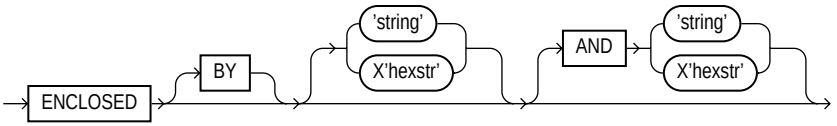
Delimiter Specification
delimiter_spec

delimiter_spec can specify a termination delimiter, enclosure delimiters, or a combination of the two, as shown below:





Note: Only fields which are loaded from a LOB file can be TERMINATED BY EOF.



Note: Fields TERMINATED BY EOF cannot also be enclosed.

For more information, see Specifying Delimiters on page 5-69.

Control File Basics

The following sections describe the various DDL entries in the SQL*Loader control file and their function. All statements use the data definition language syntax described in the previous sections. The control file statements are presented in the approximate order they would appear in the control file.

Comments in the Control File

Comments can appear anywhere in the command section of the file, but they should not appear within the data. Precede any comment with two hyphens. For example,

```
--This is a Comment
```

All text to the right of the double hyphen is ignored, until the end of the line. Appendix Case 3: Loading a Delimited, Free-Format File on page 4-11 shows comments 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. The OPTION statement precedes the LOAD DATA statement.

OPTIONS

The OPTIONS parameter allows you to specify runtime arguments in the control file, rather than on the command line. The following arguments can be specified using the OPTIONS parameter. These arguments are described in greater detail 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) )
```

Note: Values specified on the command line override values specified in the OPTIONS statement in the control file.

Specifying Filenames and Objects Names

SQL*Loader follows the SQL standard for specifying object names (for example, table and column names). This section explains certain exceptions to that standard and how to specify database objects and filenames in the SQL*Loader control file that require special treatment. It also shows how the escape character is used in quoted strings.

Filenames that Conflict with SQL and SQL*Loader Reserved Words

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 A, "SQL*Loader Reserved Words", for a complete list of reserved words.

You must use double quotation marks if the object name contains special characters other than those recognized by SQL ($, #, _), or if the name is case sensitive.

Specifying SQL Strings

You must specify SQL strings within double quotation marks. The SQL string applies SQL operators to data fields. See Applying SQL Operators to Fields on page 5-87 for more information.

Restrictions A control file entry cannot specify a SQL string for any field in the control file that uses a BFILE, SID, OID, or REF directive.

SQL strings cannot be used with column objects or collections, or attributes of column objects or collections.

Operating System Considerations

Specifying a Complete Path

If you encounter problems when trying to specify a complete pathname, it may be due to an operating system-specific incompatibility caused by special characters in the specification. In many cases, specifying the pathname within single quotation marks prevents errors.

If not, please see your operating system-specific documentation for possible solutions.

The Backslash Escape Character

In DDL syntax, you can place a double quotation mark inside a string delimited by double quotation marks by preceding it with the escape character, "\" (if the escape is allowed on your operating system). The same rule applies when single quotation marks are required in a string delimited by single quotation marks.

For example, `homedir\data"norm\myfile` contains a double quotation mark. Preceding the double quote with a backslash indicates that the double quote is to be taken literally:

```
INFILE 'homedir\data\"norm\mydata'
```

You can also put the escape character itself into a string by entering it twice:

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.

Non-Portable Strings

There are two kinds of character strings in a SQL*Loader control file that are not portable between operating systems: *filename* and *file processing option* strings. When converting to a different operating system, these strings will likely need to be modified. All other strings in a SQL*Loader control file should be 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 must specify double backslashes in your pathnames and use single quotation marks.

Additional Information: Please see your Oracle operating system-specific documentation for information about which escape characters are required or allowed.

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.

Identifying Data in the Control File with BEGINDATA

If your data is contained in the control file itself and not in a separate datafile, you must include it following the load configuration specifications.

Specify the BEGINDATA keyword before the first data record. The syntax is:

```
BEGINDATA data
```

BEGINDATA is used in conjunction with the INFILE keyword, as described on page 5-22 by specifying INFILE *. Case 1: Loading Variable-Length Data on page 4-5 provides an example.

Notes:

- If you omit the BEGINDATA keyword but include data in the control file, SQL*Loader tries to interpret your data as control information and issues an error message. If your data is in a separate file, do not use the BEGINDATA keyword.
- Do not use spaces or other characters on the same line as the BEGINDATA parameter because the line containing BEGINDATA will be interpreted as the first line of data.
- Do not put Comments after BEGINDATA as they will also be interpreted as data.

INFILE: Specifying Datafiles

You use the INFILE keyword to specify a datafile or datafiles fully followed by a file-processing options string. You can specify multiple files by using multiple INFILE keywords. You can also specify the datafile from the command line, using the DATA parameter described in Command-Line Keywords on page 6-3.

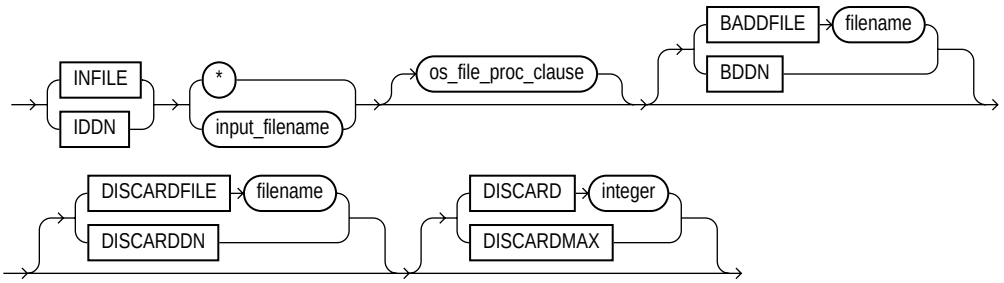
Note: the command-line parameter DATA overrides the INFILE keyword contained in the control file.

Naming the File

To specify a file that contains the data to be loaded, use the INFILE keyword, followed by the filename and optional processing options string. *Remember that a filename specified on the command line overrides the first INFILE 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 itself contains the data to be loaded, specify an asterisk (*). This specification is described in Identifying Data in the Control File with BEGINDATA on page 5-21.

Note: IDDN has been retained for compatibility with DB2.



where:

INFILE or INDDN	(Use INDDN when DB2 compatibility is required.) This keyword specifies that a datafile specification follows.
filename	Name of the file containing the data. Any spaces or punctuation marks in the filename must be enclosed in single quotation marks. See Specifying Filenames and Objects Names on page 5-18.

*	If your data is in the control file itself, use an asterisk instead of the filename. If you have data in the control file as well as datafiles, you must specify the asterisk first in order for the data to be read.
processing_options	This is the file-processing options string. It specifies the datafile format. It also optimizes datafile reads. See Specifying Datafile Format and Buffering on page 5-24.

Specifying Multiple Datafiles

To load data from multiple datafiles in one SQL*Loader run, use an INFILE statement for each datafile. Datafiles need not have the same file processing options, 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.

You can also specify a separate discard file and bad file for each datafile. However, the separate bad files and discard files must be declared after each datafile name. For example, the following excerpt from a control file specifies four datafiles with separate bad and discard files:

```
INFILE  mydat1.dat  BADFILE  mydat1.bad  DISCARDFILE mydat1.dis
INFILE  mydat2.dat
INFILE  mydat3.dat  DISCARDFILE  mydat3.dis
INFILE  mydat4.dat  DISCARDMAX  10 0
```

- For MYDAT1.DAT, both a bad file and discard file are explicitly specified. Therefore both files are created, as needed.
- For MYDAT2.DAT, neither a bad file nor a discard file is specified. Therefore, only the bad file is created, as needed. If created, the bad file has a default filename and extension. The discard file is *not* created, even if rows are discarded.
- For MYDAT3.DAT, the default bad file is created, if needed. A discard file with the specified name (`mydat3.dis`) is created, as needed.
- For MYDAT4.DAT, the default bad file is created, if needed. Because the DISCARDMAX option is used, SQL*Loader assumes that a discard file is required and creates it with the default name `mydat4.dsc`, if it is needed.

Examples

Data Contained In The Control File Itself

```
INFILE  *
```

Data Contained in File WHIRL with Default Extension .dat

```
INFILE  WHIRL
```

Data in File datafile.dat: Full Path Specified

```
INFILE 'c:/topdir/subdir/datafile.dat'
```

Note: Filenames that include spaces or punctuation marks must be enclosed in single quotation marks. For more details on filename specification, see Specifying Filenames and Objects Names on page 5-18.

Specifying READBUFFERS

The READBUFFERS keyword controls memory usage by SQL*Loader. *This keyword is used for direct path loads only.* For more information, Maximizing Performance of Direct Path Loads on page 8-16.

Specifying Datafile Format and Buffering

When configuring SQL*Loader, you can specify an operating system-dependent *file processing options string* in the control file 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 syntax:

where RECSIZE is the size of a fixed-length record, and BUFFERS is the number of buffers to use for asynchronous I/O.

To declare a file named MYDATA.DAT as a file that contains 80-byte records and instruct SQL*Loader to use eight I/O buffers, using this syntax you would use the following control file entry:

```
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 Objects Names on page 5-18 for more details.

BADFILE: Specifying the Bad File

When SQL*Loader executes, it can create a file called a *bad file* or *reject file* in which it places records that were rejected because of formatting errors or because they caused Oracle errors. If you have specified that a bad file is to be created, the following applies:

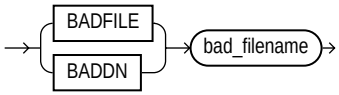
- if one or more records are rejected, the bad file is logged.
- if no records are rejected, then the bad file is not created. when this occurs, you must reinititialize the bad file for the next run.
- If the bad file is created, it overwrites any existing file with the same name so insure that you do not overwrite a file you wish to retain.

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 keyword (or BADDN for DB2 compatibility), followed by the bad file 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 in Command-Line Keywords 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	(Use BADDN when DB2 compatibility is required.) This keyword specifies that a filename for the badfile follows.
bad_filename	Any valid filename specification for your platform. Any spaces or punctuation marks in the filename must be enclosed in single quotation marks. See Specifying Filenames and Objects Names on page 5-18.

Examples

A bad file with filename UGH and default file extension or file type of .bad:

```
BADFILE UGH
```

A bad file with filename BAD0001 and file extension or file type of .rej:

```
BADFILE BAD0001.REJ
BADFILE '/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: Loading Combined Physical Records on page 4-15 demonstrates rejected records.

Note: During a multi-table load, SQL*Loader ensures that, if a row is rejected from one table, it is also rejected from all other tables. This is to ensure that the row can be repaired in the bad file and reloaded to all tables consistently. Also, if a row is loaded into one table, it should be loaded into all other tables which don't filter it out. Otherwise, reloading a fixed version of the row from the bad file could cause the data to be loaded into some tables twice.

Therefore, when SQL*Loader encounters the maximum number of errors allowed for a multi-table load, it continues loading rows to ensure that valid rows loaded into previous tables are either loaded into all tables or filtered out of all tables.

LOB Files and Secondary Data Files Data from LOB files or secondary data files are not written to a bad file when there are rejected rows. If there is an error loading a LOB, the row is *not* rejected, rather the LOB field is left empty (not NULL with a length of zero (0) bytes).

Specifying the Discard File

During SQL*Loader execution, it can 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 differ from rejected records. *Discarded records do not necessarily have any bad data.* No insert is attempted on a discarded record.

A discard file is created according to the following rules:

- You have specified a discard filename and one or more records fail to satisfy all of the WHEN clauses specified in the control file. (Note that, if the discard file is created, it overwrites any existing file with the same name so insure that you do not overwrite any files you wish to retain.)
- If no records are discarded, then a discard file is not created.

To create a discard file, use any of the following syntax:

In a Control File	On the Command Line
DISCARDFILE *filename*	DISCARD
DISCARDDN *filename* *(DB2)*	DISCARDMAX
DISCARDS	
DISCARDMAX	

Note that you can specify the discard file directly with a parameter specifying its name, or indirectly by specifying the maximum number of discards.

Specifying the Discard File in the Control-File

To specify the name of the file, use the DISCARDFILE or DISCARDDN (for DB2-compatibility) keyword, followed by the filename.

where:

DISCARDFILE or DISCARDDN	(Use DISCARDDN when DB2 compatibility is required.) This keyword specifies that a discard filename follows.
discard_filename	Any valid filename specification for you platform. Any spaces or punctuation marks in the filename must be enclosed in single quotation marks. See Specifying Filenames and Objects Names on page 5-18.

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

A discard file with filename CIRCULAR and default file extension or file type of `.dsc`:

```
DISCARDFILE  CIRCULAR
```

A discard file named `notappl` with the file extension or file type of `.may`:

```
DISCARDFILE NOTAPPL.MAY
```

A full path to the discard file `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.

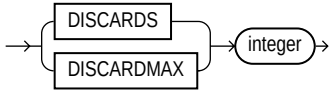
No records are discarded if an INTO TABLE keyword is specified without a WHEN clause. An attempt is made to insert every record into such a table. So records may be rejected, but none are discarded.

Case 4: Loading Combined Physical Records on page 4-15 provides an example of using a discard file.

LOB Files and Secondary Data Files Data from LOB files or secondary data files are not written to a discard file when there are discarded rows.

Limiting the Number of Discards

You can limit the number of records to be discarded for each datafile:



where *n* must be an integer. When the discard limit is reached, processing of the 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 specified applies to 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: Loading Combined Physical Records on page 4-15 provides an example.

Using a Command-Line Parameter

You can specify the discard file from the command line, with the parameter DISCARDFILE described in Command-Line Keywords on page 6-3.

A filename specified on the command line overrides any bad file that you may have specified in the control file.

Handling Different Character Encoding Schemes

SQL*Loader supports different character encoding schemes (called character sets, or code pages). SQL*Loader uses Oracle's NLS (National Language Support) features to handle the various single-byte and multi-byte character encoding schemes available today. See the *Oracle8i National Language Support Guide* for information about supported character encoding schemes. The following sections provide a brief introduction to some of the supported schemes.

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 has the capacity to convert data from the datafile character set to the database character set, when they differ.

When using a conventional path load, data is converted into the session character set specified by the NLS_LANG initialization parameter for that session. The data is then 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. The direct path load method, therefore, allows data in a character set that is not supported by your terminal to be loaded.

Note: When data conversion is required, 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, during a conventional path load, 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

You use the CHARACTERSET keyword to specify to SQL*Loader which character set is used in each datafile. Different datafiles can be specified with different character sets. However, 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 the database. Only CHAR, DATE, and numeric EXTERNAL fields are affected. If the CHARACTERSET keyword is not specified, then no conversion occurs.

The CHARACTERSET syntax 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 *Oracle8i National Language Support Guide*.

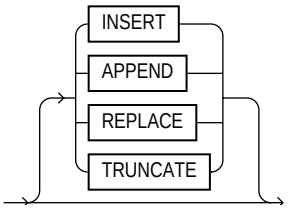
Control File Characterset

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:



Loading into Empty Tables

If the tables you are loading into are empty, use the INSERT option.

Loading into Non-Empty Tables

If the tables you are loading into already contain data, you have three options:

- APPEND
- REPLACE
- TRUNCATE

Warning: When the REPLACE or TRUNCATE keyword is specified, the entire *table* is replaced, not just 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.

Note: This section corresponds to the DB2 keyword RESUME; users of DB2 should also refer to the description of RESUME in Appendix B, "DB2/DXT User Notes".

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. You must have SELECT privilege to use the APPEND option. Case 3: Loading a Delimited, Free-Format File on page 4-11 provides 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: Loading Combined Physical Records on page 4-15 provides 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 *Oracle8i Concepts*.

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 work table.
2. Use the SQL language UPDATE statement with correlated subqueries.
3. Drop the work table.

For more information, see the "UPDATE" statement in *Oracle8i SQL Reference*.

TRUNCATE

Using 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.

Notes:

Unlike the SQL TRUNCATE option, this method re-uses a table's extents.

INSERT is SQL*Loader's default method. It requires the table to be empty before loading. SQL*Loader terminates with an error if the table contains rows. Case 1: Loading Variable-Length Data on page 4-5 provides an example.

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 Index Unusable State on page 8-11 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

Since Oracle8i supports user-defined record sizes larger than 64k (see READSIZE (read buffer) on page 6-7), the need to fragment logical records into physical records is reduced. However, there may still be situations in which you may want to do so.

You can create one logical record from multiple physical records using one of the following two clauses, depending on your data:

```
CONCATENATE
CONTINUEIF
```

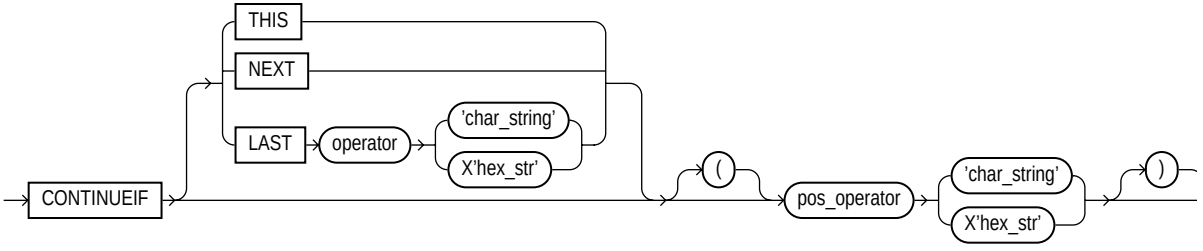
CONCATENATE is appropriate in the simplest case, when SQL*Loader should always add the same number of physical records to form one logical record.

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 the current 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, then the current physical record becomes 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 current physical record is concatenated to the current record, continuing until the condition is false.
pos_spec	Specifies the starting and ending column numbers in the physical record. 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 the character string, the shorter one is padded. Character strings are padded with blanks, hexadecimal strings with zeroes.
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 the current 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	The supported operators are equal and not equal. For the equal operator, the field and comparison string must match exactly for the condition to be true. For the not equal operator, they may differ in any character.
char_string	A string of characters to be compared to the continuation field defined by start and end, 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 described above. X'1FB033 would represent the three bytes with values 1F, b), 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:

```
%%aaaaaaaa......aaaaaaaa....
%%bbbbbbbb....%%bbbbbbbb....
..cccccccc....%%cccccccc....
%%dddddddddd....dddddddddd..
%%eeeeeeeeee..%%eeeeeeeeee..
..ffffffffff..%%ffffffffff..
```

The logical records would be the same in each case:

```
aaaaaaaa....bbbbbbbb....cccccccc....
dddddddddd..eeeeeeeeee..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.
- You cannot fragment records in secondary datafiles (SDFs) into multiple physical records.

Using 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: Loading Combined Physical Records on page 4-15 provides an example of the CONTINUEIF clause.

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

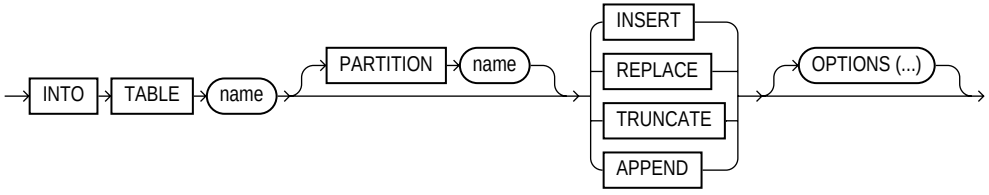
Specifying Table Names

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 5-32.

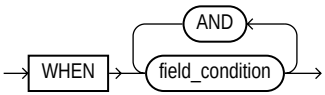
Table-Specific OPTIONS keyword

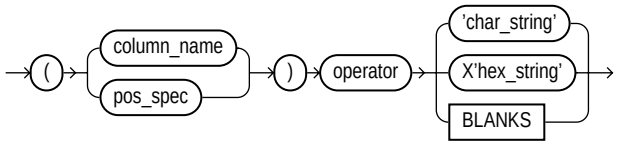
The OPTIONS keyword can be specified for individual tables in a parallel load. (It is only valid for a parallel load.) For more information, see Parallel Data Loading Models on page 8-26.

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')
```

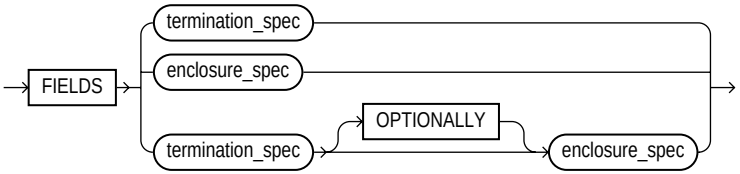
To evaluate the WHEN clause, SQL*Loader first determines the values of all the fields in the record. Then the WHEN clause is evaluated. A row is inserted into the table only if the WHEN clause is true.

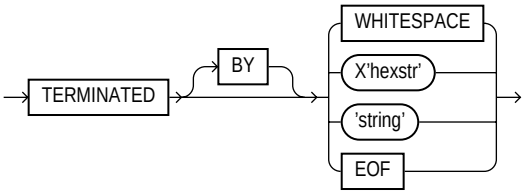
Field conditions are discussed in detail in Specifying Field Conditions on page 5-44. Case 5: Loading Data into Multiple Tables on page 4-19 provides an example of the WHEN clause.

Using The WHEN Clause with LOB Files and Secondary Data Files If a WHEN directive fails on a record, that record is discarded (skipped). Note also that, the skipped record is assumed to be contained completely in the main datafile, therefore, a secondary data file will not be affected if present.

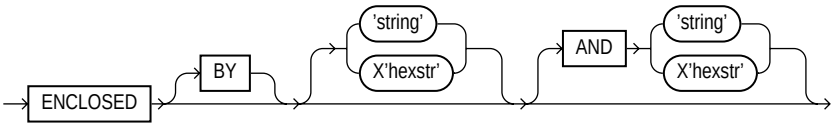
Specifying Default Data Delimiters

If all data fields are terminated similarly in the datafile, you can use the FIELDS clause to indicate the default delimiters. The syntax is:





Note: Terminators are strings not limited to a single character. Also, TERMINATED BY EOF applies only to loading LOBs from SSDFs.



Note: Enclosure strings do not have to be a single character.

You can override the delimiter for any given column by specifying it after the column name. Case 3: Loading a Delimited, Free-Format File on page 4-11 provides an example. See Specifying Delimiters on page 5-69 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: Extracting Data from a Formatted Report on page 4-28 provides an example of 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 in High-Level Syntax Diagrams on page 5-4. Further details are in SORTED INDEXES Statement on page 8-17.

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.

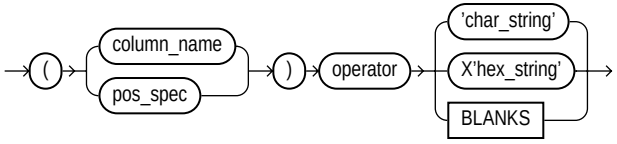
With the SINGLEROW option, storage space is not required for new index entries or for a new index. The resulting index may not be as optimal as a freshly sorted one, but it takes less space to produce. It also takes more time, since additional UNDO information is generated for each index insert. This option is suggested for use when:

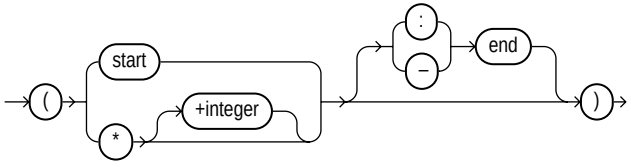
- available storage is limited, or
- the number of rows to be loaded is small compared to the size of the table (a ratio of 1:20, or less, is recommended).

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:

start — Specifies the starting position of the comparison field in the logical record.

end	Specifies the ending position of the comparison field in the logical record. Either start-end or start:end 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 with zeroes.
full_field_name	full_field_name is the full name of a field specified using dot notation. If the field col2 is an attribute of a column object col1, when referring to col2 in one of the directives, you must use the notation col1.col2. The column_name and the fieldname referencing / naming the same entity can be different because column_name never includes the full name of the entity (no dot notation).
operator	A comparison operator for either equal or not equal.
char_string	A string of characters enclosed in single or double quotes that is compared to the comparison field. If the comparison is true, the current row is inserted into the table.
X'hex_string'	A byte string in hexadecimal format that is used in the same way as char_string, described above.
BLANKS	A keyword denoting an arbitrary number of blanks. See below.

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:

```
full_field_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 Multi-Byte (Asian) Character Sets on page 5-30.

Comparing Fields to Literals

When a data field is compared to a shorter literal string, the 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 5-53.

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.

It is not necessary to specify all attributes when loading column objects. Any missing attributes will be set to NULL.

Specifying Filler Fields

Filler fields have names but they are not loaded into the table. However, filler fields can be used as arguments to init_specs (foe example, NULLIF and DEFAULTIF) as well as to directives (for example, SID, OID, REF, BFILE). Also, filler fields can occur anyplace in the data file. They can be inside of the field list for an object or inside the definition of a VARRAY. See New SQL*Loader DDL Behavior and Restrictions on page 3-18 for more information on filler fields and their use.

The following an sample filler field specification:

```
field_1_count FILLER char,
field_1 varray count(field_1_count)
(
   filler_field1  char{2},
   field_1  column object
   (
     attr1 char(2),
     filler_field2  char(2),
     attr2 char(2),
   )
   filler_field3  char(3),
)
filler_field4 char(6)
```

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.

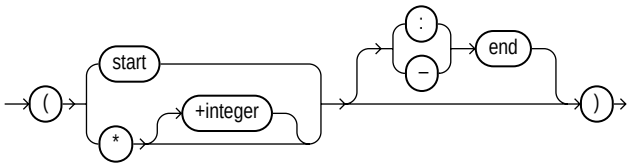
You may specify *one* datatype for each field; if unspecified, CHAR is assumed.

SQL*Loader Datatypes on page 5-57 describes how SQL*Loader datatypes are converted into Oracle datatypes and gives detailed information on each SQL*Loader's datatype.

Before the datatype is specified, the field's position must be specified.

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, as follows:



where:

start	The starting column of the data field in the logical record. The first character position in a logical record is 1.
end	The ending position of the data field in the logical record. Either start-end or start:end is acceptable. If you omit end, the length of the field is derived from the datatype in the datafile. Note that CHAR data specified without start or end is assumed to be length 1. If it is impossible to derive a length from the datatype, an error message is issued.
*	Specifies that the data field follows immediately after the previous field. If you use * for the first data field in the control file, that field is assumed to be at the beginning of the logical record. When you use * to specify position, the length of the field is derived from the datatype.
+n	You can use an on offset, specified as +n, to offset the current field from the previous field. A number of characters as specified by n are skipped before reading the value for the current 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 27. Column ALLOW is offset from the end of EMPNO by +2. So it starts in column 29 and continues until a slash is encountered.

Using POSITION with Data Containing 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 in Specifying Delimiters on page 5-69. 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 Extracting Multiple Logical Records on page 5-50.

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*).

Examples

```
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 Yvonne
1121 Albert     1130 Thomas
```

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. See Case 5: Loading Data into Multiple Tables on page 4-19.

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

This is the simplest form of generated data. It does not vary during the load, and it does not vary between loads.

CONSTANT

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

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, the first record loaded has a RECNUM of 11.

RECNUM

The combination of column name and the RECNUM keyword is a complete column specification.

```
column_name RECNUM
```

Setting a Column to the Current Date

A column specified with SYSDATE gets the current system date, as defined by the SQL language SYSDATE function. See the section "DATE Datatype" in *Oracle8i SQL Reference*.

SYSDATE

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.

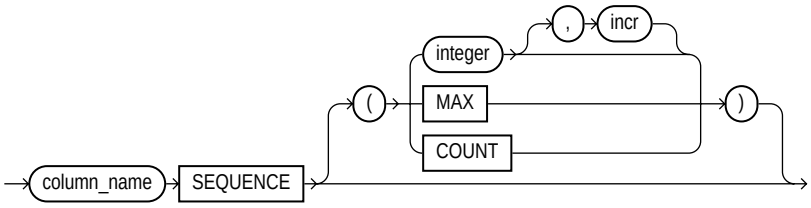
A new system date/time is used for each array of records inserted in a conventional path load and for each block of records loaded during a direct path load.

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

The combination of column name and the SEQUENCE function is a complete column specification.



where:

column_name The name of the column in the database to which to assign the sequence.

SEQUENCE	Use the SEQUENCE keyword to specify the value for a column.
n	Specifies the specific sequence number to begin with
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.
increment	The value that the sequence number is to increment after a record is loaded or rejected

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: Loading a Delimited, Free-Format File on page 4-11 provides an example the SEQUENCE function.

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.

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. *Note that, when you use SEQUENCE(MAX), SQL*Loader will use the maximum from each table which can lead to inconsistencies in sequence numbers.*

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 entries 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.

SQL*Loader Datatypes

SQL*Loader has a rich palette of datatypes. These datatypes are grouped into *portable* and *non-portable* datatypes. Within each of these two groups, the datatypes are subgrouped into *length-value datatypes* and *value datatypes*.

The main grouping, portable vs. non-portable, refers to the platform dependency of the datatype. This issue arises due to a number of platform specificities such as differences in the byte ordering schemes of different platforms (big-endian vs. little-endian), differences in how many bits a particular platform is (16 bit, 32 bit, 64 bit), differences in signed number representation schemes (2's complement vs. 1's complement), etc. Note that not all of these problems apply to all of the non-portable datatypes.

The sub-grouping, value vs. length-value addresses different issues. While value datatypes assume a single part to a datafield, length-value datatypes require that the datafield consist of two sub fields -- the *length subfield* which specifies how long the second (*value*) subfield is.

Non-Portable Datatypes

VALUE Datatypes	Length-Value Datatypes
INTEGER	
SMALLINT	VARCHAR
FLOAT	VARGRAPHIC
DOUBLE	VARRAW
BYTEINT	LONG VARRAW
ZONED	
(packed) DECIMAL	

INTEGER

The data is a full-word binary integer (unsigned). If you specify *start:end* in the POSITION clause, *end* is ignored. The length of the field is the length of a full-word 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 (unsigned). 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
```

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.

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. (Datatype UNSIGNED CHAR in C.)

The syntax for this datatype is:

```
BYTEINT
```

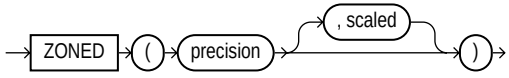
An example is:

```
(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:



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:

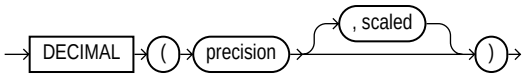
```
sal    POSITION(32)    ZONED(8),
```

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	The number of digits in a value. The character length of the field, as computed from digits, is (digits + 2/2) rounded up.
scale	The scaling factor, or number of digits to the right of the decimal point. The default is zero (indicating an integer). scale may be greater than the number of digits but cannot 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. (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.)

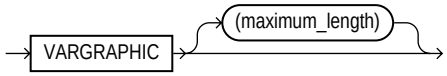
VARGRAPHIC

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 5-58 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:



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 4Kb graphic characters, or 8 Kb (2 * 4Kb). 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 5-74 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 length-value datatype. It consists of a *binary length subfield* followed by a character string of the specified 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 5-58 for more information.

The syntax for this datatype is:

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 5-74 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.

VARRAW

VARRAW is made up of a two byte binary length-subfield followed by a RAW string value-subfield.

The syntax for this datatype is shown in the diagram for datatype_spec on page 5-12.

VARRAW results in a VARRAW with 2 byte length-subfield and a max size of 4 Kb (i.e. default). VARRAW(65000) results in a VARRAW whose length subfield is 2 bytes and has a max size of 65000 bytes.

LONG VARRAW

LONG VARRAW is a VARRAW with a four byte length-subfield instead of a two byte length-subfield.

The syntax for this datatype is shown in the diagram for datatype_spec on page 5-12.

LONG VARRAW results in a VARRAW with 4 byte length-subfield and a max size of 4 Kb (i.e. default). LONG VARRAW(300000) results in a VARRAW whose length subfield is 4 bytes and has a max size of 300000 bytes.

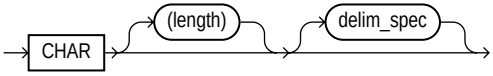
Portable Datatypes

VALUE Datatypes	Length-Value Datatypes
CHAR	VARCHARC
DATE	VARRAWC
INTEGER EXTERNAL	
RAW	
GRAPHIC	
GRAPHIC EXTERNAL	

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.

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:

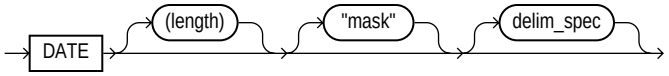


A field of datatype CHAR may also be variable-length delimited or enclosed. See Specifying Delimiters on page 5-69.

Attention: If the column in the database table is defined as LONG or a VARCHAR2, you must explicitly specify a maximum length (maximum for a LONG is two 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: Loading a Delimited, Free-Format File on page 4-11 provides an example of the DATE datatype.

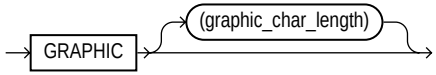
A field of datatype DATE may also be specified with delimiters. For more information, see Specifying Delimiters on page 5-69.

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 5-81.

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:

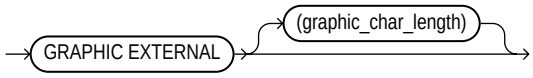


For GRAPHIC and GRAPHIC EXTERNAL, specifying POSITION(*start:end*) gives the exact location of the field in the logical record.

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. No delimited datafield specification is allowed with GRAPHIC datatype specification.

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:



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)".

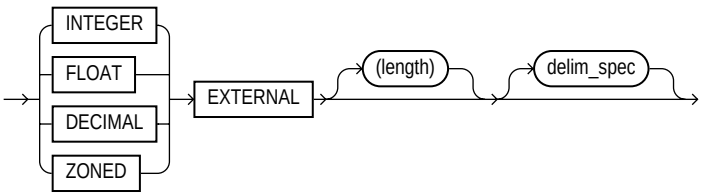
To describe [####], use "POSITION(1:6) GRAPHIC EXTERNAL" or "POSITION(1) GRAPHIC EXTERNAL(2)".

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. 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:



Attention: The data is a number in character form, not binary representation. So these datatypes are identical to CHAR and are treated identically, *except for the use of DEFAULTIF*. If you want the default to be null, use CHAR; if you want it to be zero, use EXTERNAL. See also Setting a Column to Null or Zero and DEFAULTIF Clause on page 5-80.

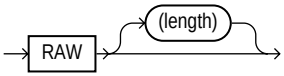
FLOAT EXTERNAL Data Values

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.

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. RAW datafields can not be delimited.

VARCHARC

The datatype VARCHARC consists of a character length-subfield followed by a character string value-subfield.

The syntax for this datatype is shown in the diagram for datatype_spec on page 5-12.

For example:

- VARCHARC results in an error
- VARCHARC(7) results in a VARCHARC whose length subfield is 7 bytes long and whose max size is 4 Kb (i.e. default)
- VARCHARC(3,500) results in a VARCHARC whose length subfield is 3 bytes long and has a max size of 500 bytes.

VARRAWC

The datatype VARRAWC consists of a RAW string value-subfield.

The syntax for this datatype is shown in the diagram for datatype_spec on page 5-12.

For example:

- VARRAWC results in an error
- VARRAWC(7) results in a VARRAWC whose length subfield is 7 bytes long and whose max size is 4 Kb (i.e. default)

- VARRAWC(3,500) results in a VARRAWC whose length subfield is 3 bytes long and has a max size of 500 bytes.

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
```

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. Note that the client does datatype conversion for fields in *collections columns* (VARRAYs and nested tables). It does not do datatype conversion when loading nested tables as a separate table from the parent.

The datatype of the data in the file does not necessarily need 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.

Specifying Delimiters

The boundaries of CHAR, DATE, 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.

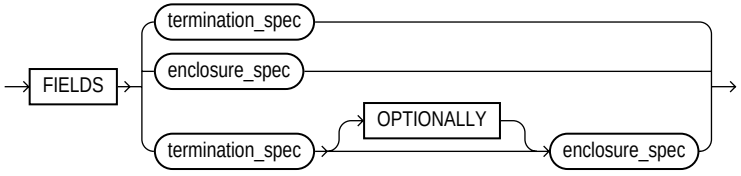
TERMINATED BY WHITESPACE

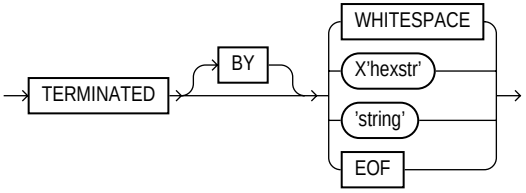
If TERMINATED BY WHITESPACE is specified, data is read until the first occurrence of a whitespace character (space, tab, newline). Then the current position is advanced until no more adjacent whitespace characters are found. This allows field values to be delimited by varying amounts of whitespace.

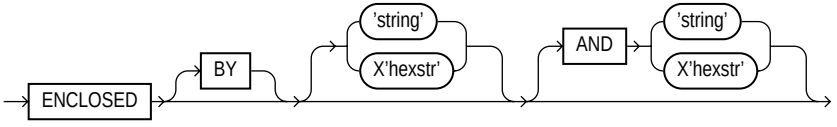
Enclosed Fields

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 the first occurrence of a delimiter.
BY	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 can be enclosed by the specified character. If SQL*Loader finds a first occurrence of the character, it reads the data value until if finds the second occurrence. If the data is not enclosed, the data is read as a terminated field. If you specify an optional enclosure, you must specify a TERMINATED BY clause (either locally in the field definition or globally in the FIELDS clause.
ENCLOSED	The data will be found between two delimiters.
char	The delimiter is the single character char.
X'hex_byte'	The delimiter is the single character that has the value specified by hex_byte in the character encoding scheme such as X'1F' (equivalent to 31 decimal). "X" must be uppercase.
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 delimiters are assumed to be 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 parentheses, (, and right parentheses, )).)
```

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, 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. A good policy is to specify the smallest possible maximum value; see Determining the Size of the Bind Array on page 5-74.

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 DANIEL*bbb*, 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 DANIEL*bbb*:. This field is loaded as "DANIEL ", with the trailing blanks. For more discussion on whitespace in fields, see Trimming Blanks and Tabs on page 5-81.

Conflicting Character Datatype Field Lengths

A control file can specify multiple lengths for the character-data fields CHAR, DATE, 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

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. For example, if

```
position(1:10) char(15)
```

is specified, then the length of the field is 15.

Delimited Fields

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 *maximum* 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.

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.

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.

Loading Data Across Different Platforms

When a datafile created on one platform is to be loaded on a different platform, 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, there is no way for the target system to directly read data generated on the source system.

The best solution is to load data across a Net8 database link, taking advantage of the automatic conversion of datatypes. This is the recommended approach, whenever feasible.

Problems with inter-platform loads typically occur with *native* datatypes. In some situations, it is possible to avoid problems by lengthening a field by padding it with zeros, or to read only part of the field to shorten it. (For example, when an 8-byte integer is to be read on a system that uses 4-byte integers, or vice versa.) Note, however, that incompatible byte-ordering or incompatible datatype implementation, may prevent this.

If you cannot use a Net8 database link, it is advisable to use only the CHAR, DATE, VARCHARC, and NUMERIC EXTERNAL datatypes. Datafiles written using these datatypes are longer than those written with native datatypes. They may take more time to load, but they transport more readily across platforms. 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 database. 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 and ROWS parameters are described in Command-Line Keywords on page 6-3.

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 BINDSIZE (maximum size) on page 6-4) or the OPTIONS clause in the control file (see OPTIONS on page 5-18), 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 ROWS (rows per commit) on page 6-7) or the OPTIONS clause in the control file (see OPTIONS on page 5-18).

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. *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 SQL*Loader Datatypes on page 5-57. 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 SQL*Loader Datatypes on page 5-57. 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 SQL*Loader Datatypes on page 5-57.

Table 5–1 Invariant fields

Datatype	Size
INTEGER	OS-dependent
SMALLINT	
FLOAT	
DOUBLE	

Table 5–2 Non-graphic 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	

Table 5–3 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 5–4 Variable-length fields

Datatype	Default Size	Maximum Length Specified (L)
VARCHAR	4Kb	L+S
CHAR (delimited) DATE (delimited) numeric EXTERNAL (delimited)	255	L+S

Minimizing Memory Requirements for the Bind Array

Pay particular attention to the default sizes allocated for VARCHAR, VARGRAPHIC, 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, additional space in the bind array is required 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; see NULLIF Keyword on page 5-80. 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 (CHAR or DATE) data sets the column to null (compare with Numeric External Datatypes on page 5-66). See also Specifying Field Conditions on page 5-44 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 5-87

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: Loading Using the Direct Path Load Method on page 4-25 provides 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 5-87.

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 in TRAILING NULLCOLS on page 5-42, explains how SQL*Loader proceeds 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 5-45. Case 6: Loading Using the Direct Path Load Method on page 4-25 provides examples of 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 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 5-86 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*.

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:

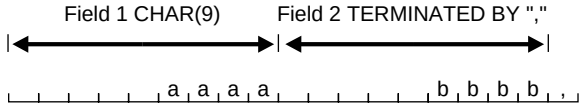
```
loc POSITION(19:31) CHAR TERMINATED BY ","
```

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

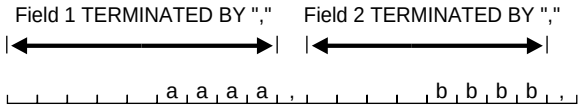
When a starting position is not specified for a field, it begins immediately after the end of the previous field. Figure 5–1 illustrates this situation when the previous field has a predetermined size.

Figure 5–1 Relative Positioning After a Fixed Field



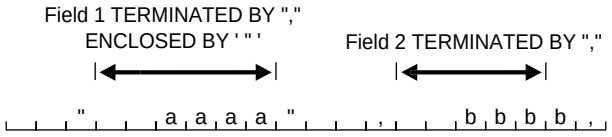
If the previous field is terminated by a delimiter, then the next field begins immediately after the delimiter, as shown in Figure 5–2.

Figure 5–2 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 5–3. If a non-whitespace character is found after the enclosure delimiter, but before the terminator, then SQL*Loader generates an error.

Figure 5–3 Relative Positioning After Enclosure Delimiters



Leading Whitespace

In Figure 5–3, 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 5–4 illustrates this case.

Figure 5–4 Fields Terminated by Whitespace

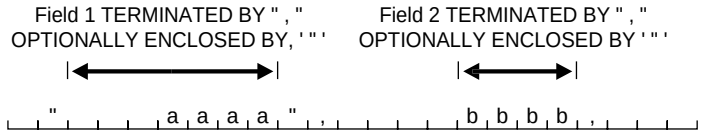
This situation occurs when the previous field is explicitly specified with the TERMINATED BY WHITESPACE clause, as shown in the example. It also occurs when you use the global FIELDS TERMINATED BY WHITESPACE clause.

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 5–5.

Figure 5–5 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 5–5.

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 5–5, 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: Summary

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 on page 5-86, for details on how to prevent trimming.

Table 5–5 Trim Table

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)

(1) When an allow-blank field is trimmed, its value is null.

(2) Except for fields that are TERMINATED BY WHITESPACE

(3) Presence of trailing whitespace depends on the current field's specification, as shown by the other entries in the 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 Blanks and Tabs on page 5-81.

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 the "Operators, Functions, Expressions, Conditions chapter in the *Oracle8i SQL Reference*.

The column name and the name of the column in the SQL string must match exactly, including the quotation marks, as in this example of specifying the control file:

```
LOAD DATA
INFILE *
APPEND INTO TABLE XXX
( "LAST"   position(1:7)     char   "UPPER(:\"LAST\)",
   FIRST    position(8:15)   char   "UPPER(:FIRST)"
)
```

```
BEGINDATA
Phil Locke
Jason Durbin
```

The SQL string must be enclosed in double quotation marks. In the example above, LAST must be in quotation marks because it is a SQL*Loader keyword. FIRST is not a SQL*Loader keyword and therefore does not require quotation marks. To quote the column name in the SQL string, you must escape it.

The SQL string 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 Objects Names on page 5-18.

```
field1 POSITION(1:4) INTEGER EXTERNAL
       "decode(:field2, '22', '34', :field1)"
```

Note: SQL strings cannot reference fields in column objects or fields that are loaded using OID, SID, REF, or BFILE. Also, they cannot reference filler fields.

Referencing Fields That Are SQL*Loader Keywords

Other fields in the same record can also be referenced, as in this 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: Extracting Data from a Formatted Report on page 4-28 to load data from a formatted report.

Loading Column Objects

Column object in the control file are described in terms of their attributes. In the datafile, the data corresponding to each of the attributes of a column-object is in a datafield similar to that corresponding to a simple relational column.

Following are some examples of loading column objects. First, where the data is in predetermined size fields and second, where the data is in delimited fields.

Loading Column Objects in Stream Record Format

Example 5–1 Loading in stream record form; field position specified explicitly

Control File Contents

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE departments
   (dept_no     POSITION(01:03)    CHAR,
    dept_name   POSITION(05:15)    CHAR,
1   dept_mgr    COLUMN OBJECT
      (name     POSITION(17:33)    CHAR,
       age      POSITION(35:37)    INTEGER EXTERNAL,
       emp_id   POSITION(40:46)    INTEGER EXTERNAL) )
```

Data file (sample.dat)

```
101 Mathematics  Johny Quest       30   1024
237 Physics      Albert Einstein   65   0000
```

Note:

1. This type of column object specification can be applied recursively in order to describe nested column objects.

Loading Column Objects in Variable Record Format

Example 5–2 Loading in variable record form; terminated and/or enclosed fields.

Control File Contents

```
LOAD DATA
INFILE 'sample.dat' "var 6"
INTO TABLE departments
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
1  (dept_no
   dept_name,
   dept_mgr       COLUMN OBJECT
      (name       CHAR(30),
      age         INTEGER EXTERNAL(5),
      emp_id      INTEGER EXTERNAL(5)) )
```

Data file (sample.dat)

```
2  000034101,Mathematics,Johny Q.,30,1024,
   000039237,Physics,"Albert Einstein",65,0000,
```

Notes:

1. Although no positional specifications are given, the general syntax remains the same (the column-object's name followed by the list of its attributes enclosed in parentheses). Also note that omitted type specification defaults to CHAR of length 255.
2. The first six characters (italicized) specify the length of the forthcoming record. See New SQL*Loader DDL Behavior and Restrictions on page 3-18. These length specifications include the newline characters which are ignored thanks to the terminators after the emp_id field.

Loading Nested Column Objects

Example 5–3 shows a control file describing nested column-objects (one column-object nested in another column-object).

Example 5–3 Loading in stream record form; terminated and/or enclosed fields.

Control File Contents

```
LOAD DATA
INFILE `sample.dat'
INTO TABLE departments_v2
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
   (dept_no      CHAR(5),
   dept_name     CHAR(30),
   dept_mgr      COLUMN OBJECT
      (name      CHAR(30),
      age        INTEGER EXTERNAL(3),
      emp_id     INTEGER EXTERNAL(7),
1     em_contact COLUMN OBJECT
         (name      CHAR(30),
         phone_num  CHAR(20))))
```

Data file (sample.dat)

```
101,Mathematics,Johny Q.,30,1024,"Barbie",650-251-0010,
237,Physics,"Albert Einstein",65,0000,Wife Einstein,654-3210,
```

Note:

1. This entry specifies a column object nested within a column object.

Specifying NULL Values for Objects

Specifying null values for non-scalar datatypes is somewhat more complex than for scalar datatypes. An object can have a subset of its attributes be null, it can have all of its attributes be null (an attributively null object), or it can be null itself (an atomically null object).

Specifying Attribute Nulls

In fields corresponding to object columns, you can use the NULLIF clause to specify the field conditions under which a particular attribute should be initialized to null. Example 5–4 demonstrates this.

Example 5–4 Loading in stream record form; positionally specified fields.

Control File

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE departments
  (dept_no      POSITION(01:03)    CHAR,
  dept_name     POSITION(05:15)    CHAR NULLIF dept_name=BLANKS,
  dept_mgr      COLUMN OBJECT
1    ( name     POSITION(17:33)    CHAR NULLIF dept_mgr.name=BLANKS,
1    age        POSITION(35:37)    INTEGER EXTERNAL
                                   NULLIF dept_mgr.age=BLANKS,
1    emp_id     POSITION(40:46)    INTEGER EXTERNAL
                NULLIF dept_mgr.emp_id=BLANKS))
```

Data file (sample.dat)

```
2  101             Johny Quest            1024
   237   Physics   Albert Einstein   65   0000
```

Notes:

1. The NULLIF clause corresponding to each attribute states the condition under which the attribute value should be NULL.
2. The age attribute of the `dept_mgr` value is null. The `dept_name` value is also null.

Specifying Atomic Nulls

To specify in the control file the condition under which a particular object should take null value (atomic null), you must follow that object's name with a NULLIF clause based on a logical combination of any of the **mapped fields** (for example, in Specifying NULL Values for Objects on page 5-92, the named mapped fields would be `dept_no`, `dept_name`, `name`, `age`, `emp_id`, but `dept_mgr` would not be a named mapped field because it does not correspond (is not mapped to) any field in the datafile).

Although the above is workable, it is not ideal when the condition under which an object should take the value of null is *independent of any of the mapped fields*. In such situations, you can use filler fields (see Secondary Data Files (SDFs) and LOBFILES on page 3-20).

You can map a filler field to the field in the datafile (indicating if a particular object is atomically null or not) and use the filler filed in the field condition of the NULLIF clause of the particular object.

For example:

Example 5–5 Loading in stream record form; terminated and/or enclosed fields.

Control File Contents

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE departments_v2
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
   (dept_no         CHAR(5),
   dept_name        CHAR(30),
1  is_null          FILLER CHAR,
2  dept_mgr         COLUMN OBJECT NULLIF is_null=BLANKS
      (name         CHAR(30) NULLIF dept_mgr.name=BLANKS,
      age           INTEGER EXTERNAL(3) NULLIF dept_mgr.age=BLANKS,
      emp_id        INTEGER EXTERNAL(7)
                    NULLIF dept_mgr.emp_id=BLANKS,
      em_contact    COLUMN OBJECT NULLIF is_null2=BLANKS
         (name      CHAR(30)
                    NULLIF dept_mgr.em_contact.name=BLANKS,
         phone_num  CHAR(20)
                    NULLIF dept_mgr.em_contact.phone_num=BLANKS)),
1) is_null2         FILLER CHAR)
```

Data file (sample.dat)

```
101,Mathematics,n,Johny Q.,,1024,"Barbie",608-251-0010,,
237,Physics,,"Albert Einstein",65,0000,,650-654-3210,n,
```

Notes:

1. The filler field (datafile mapped; no corresponding column) is of type CHAR (because it is a delimited field, the CHAR defaults to CHAR(255)). Note that the NULLIF clause is not applicable to the filler field itself.
2. Gets the value of null (atomic null) if, either the is_null field is blank or the emp_id attribute is blank.

Loading Object Tables

The control file syntax required to load an object table is nearly identical to that used to load a typical relational table. Example 5–6 demonstrates loading an object table with primary key OIDs.

Example 5–6 Loading an Object Table with Primary Key OIDs

Control File Contents

```
LOAD DATA
INFILE 'sample.dat'
DISCARDFILE 'sample.dsc'
BADFILE 'sample.bad'
REPLACE
INTO TABLE employees
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
   (name    CHAR(30)               NULLIF name=BLANKS,
   age      INTEGER EXTERNAL(3)    NULLIF age=BLANKS,
   emp_id   INTEGER EXTERNAL(5))
```

Data file (sample.dat)

```
Johny Quest, 18, 007,
Speed Racer, 16, 000,
```

Note that by looking only at the above control file you might not be able to determine if the table being loaded was an object table with system generated OIDs (real OIDs), an object table with primary key OIDs, or a relational table.

Note also that you may want to load data which already contains real OIDs and may want to specify that, instead of generating new OIDs, the existing OIDs in the datafile should be used. To do this, you would follow the INTO TABLE clause with the OID clause:

```
:= OID (<fieldname>)
```

where <fieldname> is the name of one of the fields (typically a filler field) from the field specification list which is mapped to a datafield that contains the real OIDs. SQL*Loader assumes that the OIDs provided are in the correct format and that they preserve OID global uniqueness. Therefore, you should use the oracle OID generator to generate the OIDs to be loaded to insure uniqueness. Note also that the OID clause can only be used for system-generated OIDs, not primary key OIDs.

Example 5–7 demonstrates loading real OIDs with the row-objects.

Example 5–7 Loading OIDs

Control File

```
   LOAD DATA
   INFILE 'sample.dat'
   INTO TABLE employees_v2
1  OID (s_oid)
   FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
      (name    CHAR(30               NULLIF name=BLANKS,
      age      INTEGER EXTERNAL(3)   NULLIF age=BLANKS,
      emp_id   INTEGER EXTERNAL(5)
2     s_oid    FILLER CHAR(32)
```

Data file (sample.dat)

```
3  Johny Quest, 18, 007, 21E978406D3E41FCE03400400B403BC3,
   Speed Racer, 16, 000, 21E978406D4441FCE03400400B403BC3,
```

Notes:

1. The OID clause specifies that the s_oid loader field contains the OID. Note that the parentheses are required.
2. If s_oid does not contain a valid hexadecimal number, the particular record is rejected.
3. The OID in the datafile is a character string and is interpreted as 32 digit hex number;. The 32 digit hex number is later converted into a 16 byte RAW and stored in the object table.

Loading REF Columns

SQL Loader can load real REF columns (REFs containing real OIDs of the referenced objects) as well as primary key REF columns:

Real REF Columns

SQL*Loader assumes, when loading real REF columns, that the actual OIDs from which the REF columns are to be constructed are in the datafile with the rest of the data. The description of the field corresponding to a REF column consists of the column name followed by the REF directive.

The REF directive takes as arguments the table name and an OID. Note that the arguments can be specified either as constants or dynamically (using filler fields). See REF_spec on page 5-10 for the appropriate syntax. Example 5–8 demonstrates real REF loading:

Example 5–8 Loading Real REF Columns

Control File

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE departments_alt_v2
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
  (dept_no      CHAR(5),
   dept_name    CHAR(30),
1) dept_mgr     REF(t_name, s_oid),
   s_oid        FILLER CHAR(32),
   t_name       FILLER CHAR(30))
```

Data file (sample.dat)

```
22345, QuestWorld, 21E978406D3E41FCE03400400B403BC3, EMPLOYEES_V2,
23423, Geography, 21E978406D4441FCE03400400B403BC3, EMPLOYEES_V2,
```

Note

1. Note that if the specified table does not exist, the record is rejected. Note also that the `dept_mgr` field itself does not map to any field in the datafile.

Primary Key REF Columns

To load a primary key REF column, the SQL*Loader control-file field description must provide the column name followed by a REF directive. The REF directive takes for arguments a comma separated list of field names/constant values. The first argument is the table name followed by arguments that specify the primary key OID on which the REF column to be loaded is based. See REF_spec on page 5-10 for the appropriate syntax.

Note that SQL*Loader assumes the ordering of the arguments matches the relative ordering of the columns making up the primary key OID in the referenced table. Example 5–9 demonstrates loading primary key REFs:

Example 5–9 Loading Primary Key REF Columns

Control File

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE departments_alt
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
 (dept_no       CHAR(5),
 dept_name      CHAR(30),
 dept_mgr       REF(CONSTANT 'EMPLOYEES', emp_id),
 emp_id         FILLER CHAR(32))
```

Data file (sample.dat)

```
22345, QuestWorld, 007,
23423, Geography, 000,
```

Loading LOBs

The following sections discuss using SQL*Loader to load internal LOBs: BLOBs, CLOBs, NCLOBs, external LOBs and BFILEs.

Internal LOBs (BLOB, CLOB, NCLOB)

Because LOBs can be quite large, SQL*Loader is able to load LOB data from either the main datafile (inline with the rest of the data) or from LOBFILEs. See Loading LOB Data Using LOBFILEs on page 5-101.

To load LOB data from the main datafile, you can use the standard SQL*Loader formats. The LOB data instances can be in predetermined size fields, delimited fields, or length-value pair fields. The following examples illustrate these situations.

LOB Data in Predetermined Size Fields

This is a very fast and conceptually simple format in which to load LOBs.

Note: Because the LOBs you are loading may not be of equal size, you can use whitespace to pad the LOB data to make the LOBs all of equal length within a particular datafield. For more information on trimming trailing whitespaces see Trimming Whitespace: Summary on page 5-86.

To load LOBs using this format, you should use either CHAR or RAW as the loading datatype.

Example 5–10 Loading LOB Data in Predetermined Size Fields

Control File Contents

```
LOAD DATA
INFILE 'sample.dat' "fix 501"
INTO TABLE person_table
   (name       POSITION(01:21)       CHAR,
1  "RESUME"    POSITION(23:500)      CHAR   DEFAULTIF "RESUME"=BLANKS)
```

Data file (sample.dat)

```
Johny Quest      Johny Quest
             500 Oracle Parkway
             jquest@us.oracle.com ...
```

Note:

1. If the datafield containing the resume is empty, the result is an empty LOB rather than a null LOB. The opposite would occur if the NULLIF clause were used instead of the DEFAULTIF clause (see DEFAULTIF and NULLIF: on page 3-19). Also note that, you can use SQL*Loader datatypes other than CHAR to load LOBs. For example, when loading BLOBs you would probably want to use the RAW datatype.

LOB Data in Delimited Fields

This format handles LOBs of different sizes within the same column (datafile field) without problem. Note, however, that this added flexibility can impact performance because SQL*Loader must scan through the data, looking for the delimiter string. See Secondary Data Files (SDFs) and LOBFILES on page 3-20.

Example 5–11 Loading LOB Data in Delimited Fields

Control File

```
LOAD DATA
INFILE 'sample.dat' "str '|'"
INTO TABLE person_table
```

```
FIELDS TERMINATED BY ',' 
   (name        CHAR(25),
1  "RESUME"     CHAR(507) ENCLOSED BY '<startlob>' AND '<endlob>')
```

Data file (sample.dat)

```
Johny Quest,<startlob>        Johny Quest
                         500 Oracle Parkway
                         jquest@us.oracle.com ...   <endlob>
2  |Speed Racer, .......
```

Notes:

1. `<startlob>` and `<endlob>` are the enclosure strings. Note that the maximum length for a LOB that can be read using the CHAR(507) is 507 bytes.
2. If the record separator `'|'` had been placed right after `<endlob>` and followed with the newline character, the newline would have been interpreted as part of the next record. An alternative would be to make the newline part of the record separator (for example, `'|\n'` or, in hex, `X'7C0A'`).

LOB Data in Length-Value Pair Fields

You can use VARCHAR (see VARCHAR on page 5-61), VARCHARC or VARRAW datatypes (see Discarded and Rejected Records on page 3-12) to load LOB data organized in length-value pair fields. Note that this method of loading provides better performance than using delimited fields, but can reduce flexibility (for example, you must know the LOB length for each LOB before loading). Example 5–12 demonstrates loading LOB data in length-value pair fields.

Example 5–12 Loading LOB Data in Length-Value Pair Fields

Control File

```
LOAD DATA
INFILE 'sample.dat' "str '<endrec>\n'"
INTO TABLE person_table
FIELDS TERMINATED BY ',' 
   (name        CHAR(25),
1  "RESUME"     VARCHARC(3,500))
```

Data file (sample.dat)

```
Johny Quest,479                    Johny Quest
                                 500 Oracle Parkway
                                 jquest@us.oracle.com
                                        ... <endrec>
2 3  Speed Racer,000<endrec>
```

Notes:

1. If "\" escaping is not supported, the string used as a record separator in the example could be expressed in hex.
2. "RESUME" is a field that corresponds to a CLOB column. In the control file, it is a VARCHARC whose length field is 3 characters long and whose max size is 500 bytes.
3. The length subfield of the VARCHARC is 0 (the value subfield is empty). Consequently, the LOB instance is initialized to empty.

Loading LOB Data Using LOBFILEs

LOB data can be lengthy enough that it makes sense to load it from a LOBFILE. In LOBFILEs, LOB data instances are still considered to be in fields (predetermined size, delimited, length-value), but these fields are not organized into records (the concept of a record does not exist within LOBFILEs). Therefore, the processing overhead of dealing with records is avoided. This type of organization of data is ideal for LOB loading.

One LOB per file In Example 5–13, each LOBFILE is the source of a single LOB. To load LOB data that is organized in this way, you would follow the column / field name with the LOBFILE datatype specifications. For example:

Example 5–13 Loading LOB DATA Using a Single LOB LOBFILE

Control File

```
LOAD DATA
INFILE 'sample.dat'
   INTO TABLE person_table
   FIELDS TERMINATED BY ','
   (name      CHAR(20),
1  ext_fname    FILLER CHAR(40),
2  "RESUME"     LOBFILE(ext_fname) TERMINATED BY EOF)
```

Data file (sample.dat)

```
Johny Quest,jqresume.txt,
Speed Racer,'/private/sracer/srresume.txt',
```

Secondary Data file (jqresume.txt)

```
           Johny Quest
        500 Oracle Parkway
           ...
```

Secondary Data file (srresume.txt)

```
         Speed Racer
      400 Oracle Parkway
        ...
```

Notes:

1. The filler field is mapped to the 40-byte long datafield which is read using the SQL*Loader CHAR datatype.
2. SQL*Loader gets the LOBFILE name from the `ext_fname` filler field. It then loads the data from the LOBFILE (using the CHAR datatype) from the first byte to the EOF character, whichever is reached first. Note that if no existing LOBFILE is specified, the "RESUME" field is initialized to empty. See also Dynamic Versus Static LOBFILE and SDF Specifications on page 3-22.

Predetermined Size LOBs

In Example 5–14, you specify the size of the LOBs to be loaded into a particular column in the control file. During the load, SQL*Loader assumes that any LOB data loaded into that particular column is of the specified size. The predetermined size of the fields allows the data-parser to perform optimally. One difficulty is that it is often hard to guarantee that all the LOBs are of the same size.

Example 5–14 Loading LOB Data Using Predetermined Size LOBs

Control File

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE person_table
FIELDS TERMINATED BY ','
   (name     CHAR(20),
   ext_fname FILLER CHAR(40),
```

```
1  "RESUME"    LOBFILE(CONSTANT '/usr/private/jquest/jqresume')
               CHAR(2000))
```

Data file (sample.dat)

```
Johny Quest,
Speed Racer,
```

Secondary Data file (jqresume.txt)

```
    Johny Quest
500 Oracle Parkway
...
    Speed Racer
400 Oracle Parkway
...
```

Note:

1. This entry specifies that SQL*Loader load 2000 bytes of data from the `'jqresume.txt'` LOBFILE, using the CHAR datatype, starting with the byte following the byte loaded last during the current loading session.

Delimited LOBs

In Example 5–15, the LOB data instances in the LOBFILE are delimited. In this format, loading different size LOBs into the same column is not a problem. Keep in mind that this added flexibility can impact performance because SQL*Loader must scan through the data, looking for the delimiter string.

Example 5–15 Loading LOB Data Using Delimited LOBs

Control File Contents

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE person_table
FIELDS TERMINATED BY ','
   (name     CHAR(20),
1  "RESUME"    LOBFILE( CONSTANT 'jqresume') CHAR(2000)
               TERMINATED BY "<endlob>\n")
```

Data file (sample.dat)

```
Johny Quest,
Speed Racer,
```

Secondary Data file (jqresume.txt)

```
    Johny Quest
500 Oracle Parkway
... <endlob>
    Speed Racer
400 Oracle Parkway
... <endlob>
```

Note:

1. Specifying `max-length` (2000) tells SQL*Loader what to expect as the maximum length of the field which can result in memory usage optimization. *If you choose to specify `max-length`, you should be sure not to underestimate its value.* The TERMINATED BY clause specifies the string that terminates the LOBs. Alternatively, you could use the ENCLOSED BY clause. The ENCLOSED BY clause allows a bit more flexibility as to the relative positioning of the LOBs in the LOBFILE (the LOBs in the LOBFILE need not be sequential).

Length-Value Pair Specified LOBs

In this example, each LOB in the LOBFILE is preceded by its length. One could use VARCHAR (see VARCHAR on page 5-61), VARCHARC or VARRAW datatypes (see Discarded and Rejected Records on page 3-12) to load LOB data organized in this way.

Note that this method of loading can provide better performance over delimited LOBs, but at the expense of some flexibility (for example, you must know the LOB length for each LOB before loading).

Example 5–16 Loading LOB Data Using Length-Value Pair Specified LOBs

Control File

```
LOAD DATA
INFILE 'sample.dat'
INTO TABLE person_table
FIELDS TERMINATED BY ','
   (name          CHAR(20),
1  "RESUME"       LOBFILE(CONSTANT 'jqresume') VARCHARC(4,2000))
```

Data file (sample.dat)

```
Johny Quest,
Speed Racer,
```

Secondary Data file (jqresume.txt)

```
2       0501Johny Quest
        500 Oracle Parkway
           ...
3       0000
```

Notes:

1. The entry `VARCHARC(4, 2000)` tells SQL*Loader that the LOBs in the LOBFILE are in length-value pair format and that first 4 bytes should be interpreted as the length. `max_length` tells SQL*Loader that the maximum size of the field is 2000.
2. the entry `0501` preceding `Johnny Quest` tells SQL*Loader that the LOB consists of the next 501 characters.
3. This entry specifies an empty (not null) LOB.

Consideration when Loading LOBs from LOBFILES

One should keep in mind the following when loading LOBs from LOBFILES:

- The failure to load a particular LOB does not result the rejection of the record containing that LOB. Instead, you will have a record that contains an empty LOB.
- It is not necessary to specify the max length of field corresponding to a LOB type column; nevertheless, if `max_length` is specified, SQL*Loader uses it as a hint to optimize memory usage. Keep in mind that it is very important that the `max_length` specification does not understate the true maximum length.

External LOB (BFILE)

The BFILE datatype stores unstructured binary data in operating-system files outside the database. A BFILE column or attribute stores a file locator that points to the external file containing the data. Note that the file which is to be loaded as a BFILE does not have to exist at the time of loading, it can be created later. SQL*Loader assumes that the necessary directory objects have already been created (a logical alias name for a physical directory on the server's filesystem). For more information, see the *Oracle8i Application Developer's Guide - Large Objects (LOBs)*.

A control file field corresponding to a BFILE column consists of column name followed by the BFILE clause. The BFILE clause takes as arguments a DIRECTORY OBJECT name followed by a BFILE name, both of which can be provided as string constants, or they can be dynamically loaded through some other field. See the *Oracle8i SQL Reference* for more information.

In the next two examples of loading BFILES, Example 5–17 has only the filename specified dynamically. Example 5–18 demonstrates specifying both the BFILE and the DIRECTORY OBJECT dynamically.

Example 5–17 Loading Data Using BFILEs: only Filename Specified Dynamically

Control File

```
LOAD DATA
INFILE sample.dat
INTO TABLE planets
FIELDS TERMINATED BY ','
   (pl_id    CHAR(3),
   pl_name   CHAR(20),
   fname     FILLER CHAR(30),
1) pl_pict   BFILE(CONSTANT "scoTT_dir1", fname))
```

Data file (sample.dat)

```
1,Mercury,mercury.jpeg,
2,Venus,venus.jpeg,
3,Earth,earth.jpeg,
```

Note

1. The directory name is quoted, therefore the string is used as is and is not capitalized.

Example 5–18 Loading Data Using BFILEs: Filename and OBJECT_DIRECTORY Specified Dynamically

Control File

```
LOAD DATA
INFILE sample.dat
INTO TABLE planets
FIELDS TERMINATED BY ','
   (pl_id    NUMBER(4),
   pl_name   CHAR(20),
   fname     FILLER CHAR(30),
1) dname     FILLER CHAR(20));
   pl_pict   BFILE(dname, fname),
```

Data file (sample.dat)

```
1, Mercury, mercury.jpeg, scott_dir1,
2, Venus, venus.jpeg, scott_dir1,
3, Earth, earth.jpeg, scott_dir2,
```

Note

1. `dname` is mapped to the datafile field containing the directory name corresponding to the file being loaded.

Loading Collections (Nested Tables and VARRAYs)

Like LOBs, collections can also be loaded either from the main datafile (data inline) or from secondary datafile(s) (data outofline). See Secondary Data Files (SDFs) on page 3-21.

When loading collection data, a mechanism must exist by which SQL*Loader can tell when the data belonging to a particular collection instance has ended. You can achieve this in two ways:

- The number of rows / elements that are to be loaded into each nested table or VARRAY instance can be specified using the DDL syntax COUNT. Note that the field used as a parameter to COUNT must be previously described in the control file before the COUNT clause itself. This positional dependency is specific to the COUNT clause. Also note that, COUNT(0) or COUNT(x) (where x==0) results in a empty collection (not null), unless overridden by a NULLIF directive. See count_spec on page 5-14.

- Unique collection delimiter can be specified; the TERMINATED BY and ENCLOSED BY directives can be employed for this purpose

In the control file, collections are described similarly to column objects (see Loading Column Objects on page 5-90). There are some differences:

- Collection descriptions employ the mechanism discussed above.
- Collection descriptions can include a secondary datafile (SDF) specification.
- Clauses or directives that take field names as arguments cannot use a field name that is in a collection unless the DDL specification is for a field in the same collection. So, in Example 5–19, `name`, `age`, and `empid`, could not be used in a field condition specification of a NULLIF or a DEFAULTIF clause for `dept_no`, `dname`, `emp_cnt`, `emps` or `projects`.
- The field list must contain one non-filler field and any number of filler fields. If the VARRAY is a VARRAY of column objects, then the attributes of the column object will be in a nested field list.

See SQL*Loader's Data Definition Language (DDL) Syntax Diagrams on page 5-3 for syntax diagrams of both nested tables and VARRAYs.

Example 5–19 demonstrates loading a varray and a nested table.

Example 5–19 Loading a VARRAY and a Nested Table

Control File

```
   LOAD DATA
   INFILE 'sample.dat' "str '|\n' "
   INTO TABLE dept
   FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
   TRAILING NULLCOLS
   ( dept_no  CHAR(3),
   dname      CHAR(20) NULLIF dname=BLANKS ,
1) emp_cnt    FILLER INTEGER EXTERNAL(5),
2) emps       VARRAY COUNT(emp_cnt)
3)    (name)  FILLER CHAR(10),
      emps    COLUMN OBJECT NULLIF emps.emps.name=BLANKS
         (name)  CHAR(30),
         age     INTEGER EXTERNAL(3),
         emp_id  CHAR(7) NULLIF emps.emps.emp_id=BLANKS)),
   mysid      FILLER CHAR(32),
4) projects   NESTED TABLE SDF(CONSTANT 'pr.txt' "fix 71")
              SID(mysid) TERMINATED BY ";"
```

```
        (project_id  POSITION(1:5) INTEGER EXTERNAL(5),
        project_name POSITION(7:30) CHAR
                     NULLIF projects.project_name=BLANKS,
        p_desc_src   FILLER POSITION(35:70) CHAR,
5)      proj_desc    LOBFILE( projects.p_desc_src) CHAR(2000)
                     TERMINATED BY "<>\n"))
```

Data file (sample.dat)

```
  101,Math,2, ,"J. H.",28,2828, ,"Cy",123,9999,21E978407D4441FCE03400400B403BC3|
6) 210,"Topologic Transforms", ,21E978408D4441FCE03400400B403BC3|
```

Secondary Data File (SDF) (pr.txt)

```
   21034 Topological Transforms      '/mydir/projdesc.txt';
7) 77777 Impossible Proof;
```

Secondary Data File (LOBFILE) ('/mydir/projdesc.txt')

```
8) Topological Transforms equate ..........<>
   If there is more then one LOB in the file, it starts here .....<>
```

Notes:

1. `emp_cnt` is a filler field used as an argument to the COUNT clause.
2. If COUNT is 0, then the collection is initialized to empty. Another way to initialize a collection to empty is to use a DEFAULTIF clause. See DEFAULTIF Clause on page 5-80. The main field name corresponding to the VARRAY field description is the same as the field name of its nested non-filler-field, specifically, the name of the column object field description.
3. Note how full name field references (dot notated) resolve the field name conflict created by the presence of this filler field.
4. This entry specifies an SDF called 'pr.txt' as the source of data. It also, specifies a fixed record format within the SDF. The TERMINATED BY clause specifies the nested table instance terminator (note that no COUNT clause is used). The SID clause specifies the field that contains the set-ids for the nested tables. Note also that if the SID clause is specified but the set-ids for a particular record is missing from the datafile, a set-id for the record is generated by the system. Specifying the set-ids in the datafile is optional and does not result any significant performance gain.
5. If the `p_desc_src` is null, the DEFAULTIF clause will initialize the `proj_desc` LOB to empty. See DEFAULTIF Clause on page 5-80.

6. The emp_cnt field is null which, under the DEFAULTIF clause, translates to 0 (an empty LOB). Therefore, the set-id is loaded. If "mysid" does not contain a valid hexadecimal number, the record is rejected. Keep in mind that SQL*Loader performs no other set-id validation.
7. The field corresponding to the p_desc_src is missing but, because the TRAILING NULLCOLS clause is present, the p_desc_src is initialized to null.
8. The LOB terminator is <> followed by a newline character.

Loading a Parent Table Separately from its Child Table

When loading a table which contain a nested table column, it may be possible to load the parent table separately from the child table. You can do independent loading of the parent and child tables if the SIDs (system-generated or user-defined) are already known at the time of the load (i.e. the SIDs are in the datafile with the data).

Example 5–20 Loading a Parent Table with User-provided SIDs

Control File

```
   LOAD DATA
   INFILE 'sample.dat' "str '|\n' "
   INTO TABLE dept
   FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
   TRAILING NULLCOLS
   ( dept_no    CHAR(3),
   dname        CHAR(20) NULLIF dname=BLANKS ,
   mysid        FILLER CHAR(32),
1) projects     SID(mysid))
```

Data file (sample.dat)

```
101,Math,21E978407D4441FCE03400400B403BC3,|
210,"Topology",21E978408D4441FCE03400400B403BC3,|
```

Note:

1. mysid is a filler field which is mapped to a datafile field containing the actual set-id's and is supplied as an argument to the SID clause.

Example 5–21 Loading a Child Table (the Nested Table Storage Table) with User-provided SIDs

Control File

```
   LOAD DATA
   INFILE 'sample.dat'
   INTO TABLE dept
   FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
   TRAILING NULLCOLS
1) SID(sidsrc)
   project_id     INTEGER EXTERNAL(5),
   project_name   CHAR(20) NULLIF project_name=BLANKS,
   sidsrc FILLER  CHAR(32))
```

Data file (sample.dat)

```
21034, "Topological Transforms",21E978407D4441FCE03400400B403BC3,
77777, "Impossible Proof",21E978408D4441FCE03400400B403BC3,
```

Note

1. The table-level SID clause tells SQL*Loader that it is loading the storage table for nested tables. `"sidsrc"` is the filler field name which is the source of the real set-id's.

Memory Issues when Loading VARRAY Columns

- When loading VARRAY columns, remember that VARRAYs are created in the client's memory before they are loaded into the database. Each element of a VARRAY requires four bytes of client memory before loading into the database. Therefore, when you are loading a VARRAY with a thousand elements, you will require at least 4000 bytes of client memory for each VARRAY instance prior to loading the VARRAYs into the database. In many cases, SQL*Loader may require two to three times that amount of memory to successfully construct and load such a varray.
- The BINDSIZE parameter specifies bounds on the memory allocated (default 64K) by SQL*Loader for loading records. Based on the size of each field being loaded into a table, SQL*Loader determines the number of rows it can load in one transaction given the number of bytes specified by BINDSIZE. You can use the ROWS parameter to force SQL*Loader to use a smaller number of rows than it might calculate. The larger the value used for ROWs, the fewer transactions and, therefore, better performance.

- Be aware that loading very large VARRAYS or a large number of smaller VARRAYs could cause you to run out of memory during the load. If this happens, you should specify a smaller value for BINDSIZE or ROWS and retry the load.

6

SQL*Loader Command-Line Reference

This chapter shows you how to run SQL*Loader with command-line keywords. If you need detailed information about the command-line keywords listed here, see Chapter 5, "SQL*Loader Control File Reference".

This chapter covers the following subjects:

- SQL*Loader Command Line
- Command-Line Keywords
- Index Maintenance Options
- Exit Codes for Inspection and Display

SQL*Loader Command Line

You can invoke SQL*Loader from the command line using certain keywords.

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, partitions, all)
       direct — Use direct path
                (Default FALSE)
      parfile — Parameter file: name of file that contains
                parameter specifications
     parallel — Perform parallel load
                (Default FALSE)
     readsize — Size (in bytes) of the read buffer
         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 OPTIONS on page 5-18.

They can also be specified in a separate file specified by the keyword PARFILE (see PARFILE (parameter file) on page 6-6). 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 available SQL*Loader command-line keyword.

BAD (bad file)

BAD specifies the name of 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 .BAD extension. This file has the same format as the input datafile, so it can be loaded by the same control file after 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 size)

BINDSIZE specifies the maximum size (bytes) of the bind array. 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 Determining the Size of the Bind Array on page 5-74. The default value is 65536 bytes. See also READSIZE (read buffer) on page 6-7.

CONTROL (control file)

CONTROL specifies the name of 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, SQL*Loader prompts you for the file name.

Note: If your control filename contains special characters, your operating system will require that they be escaped. See your operating system documentation.

Note also that if your operating system uses backslashes in its filesystem paths, you need to keep the following in mind:

- a backslash followed by a non-backslash will be treated normally.
- Two consecutive backslashes are treated as one backslash.
- Three consecutive backslashes will be treated as two backslashes.
- Placing the path in quotes will eliminate the need to escape multiple backslashes. However, note that some operating systems require that quotes themselves be escaped.

DATA (data file)

DATA specifies the name of 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 you do not specify a file extension or file type the default is .DAT.

Note: if you specify a file processing option when loading data from the control file a warning message will be issued.

DIRECT (data path)

DIRECT specifies the data path, that is, the load method to use, either conventional path or direct path. TRUE specifies a direct path load. FALSE specifies a conventional path load. The default is FALSE. Load methods are explained in Chapter 8, "SQL*Loader: Conventional and Direct Path Loads".

DISCARD (discard file)

DISCARD specifies a discard file (optional) to be created by SQL*Loader to store records that are neither inserted into a table nor rejected. If a filename is not specified, it defaults to 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 is specified also in the control file, the command-line value overrides it.

DISCARDMAX (discards to disallow)

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 maximum number of insert errors to allow. If the number of errors exceeds the value of ERRORS parameter, SQL*Loader terminates the load. The default is 50. To permit no errors at all, set ERRORS=0. To specify that all errors be allowed, use a very high number.

On a single table load, SQL*Loader terminates the load when errors exceed this error limit. Any data inserted up that point, however, is committed.

SQL*Loader maintains the consistency of records across all tables. Therefore, multi-table loads do not terminate immediately if errors exceed the error limit. When SQL*loader encounters the maximum number of errors for a multi-table load, it continues to load rows to ensure that valid rows previously loaded into tables are loaded into all tables and/or rejected rows filtered out of all tables.

In all cases, SQL*Loader writes erroneous records to the bad file.

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 Models on page 8-26.

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 (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 Models on page 8-26.

READSIZE (read buffer)

The command-line parameter READSIZE lets you specify (in bytes) the size of the read buffer. The default value is 65536 bytes, however, you can specify a read buffer of any size depending on your system.

Since, when using the conventional path method, the bind array is limited by the size of the read buffer, the advantage of a larger read buffer is that more data can be read before a commit is required.

For example:

```
sqlldr scott/tiger control=ulcas1.ctl readsize=1000000
```

enables SQL*Loader to perform reads from the external datafile in chunks of 1000000 bytes before a commit is required.

Note: The default value for *both* the READSIZE and BINDSIZE parameters is 65536 bytes. If you have specified a BINDSIZE that is smaller than the size you specified for READSIZE, the BINDSIZE value will be automatically increased the specified value of READSIZE.

Also, if the READSIZE value specified is smaller than the BINDSIZE value, the READSIZE value will be increased.

Note also that this parameter is *not related* in any way to the READBUFFERS keyword used with direct path loads.

See also BINDSIZE (maximum size) on page 6-4.

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 Determining the Size of the Bind Array on page 5-74.)

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 8-12.

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 the following 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, for example:

```
Commit point reached - logical record count 20
```

SQL*Loader may also display data error messages like the following:

```
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(s) to suppress one or more of the following:

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.
PARTITIONS	This Oracle8*i* option for a direct load of a partitioned table disables writing the per-partition statistics to the log file
ALL	Implements all of the keywords.

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 Conventional Loads on page 5-34 for more information.

USERID (username/password)

USERID is used to provide your Oracle username/password. If omitted, you are prompted for it. If only a slash is used, USERID defaults to your operating system logon. A Net8 database link can be used for a conventional path load into a remote database. For more information about Net8, see the *Net8 Administrator's Guide* For more information about database links, see *Oracle8i Distributed Database Systems.*

Index Maintenance Options

Two new, Oracle8*i* index maintenance options are available (default FALSE):

- SKIP_UNUSABLE_INDEXES={TRUE | FALSE}
- SKIP_INDEX_MAINTENANCE={TRUE | FALSE}

SKIP_UNUSABLE_INDEXES

The SKIP_UNUSABLE_INDEXES option applies to both conventional and direct path loads.

The SKIP_UNUSABLE_INDEXES=TRUE option allows SQL*Loader to load a table with indexes that are in Index Unusable (IU) state prior to the beginning of the load. Indexes that are not in IU state at load time will be maintained by SQL*Loader. Indexes that are in IU state at load time will not be maintained but will remain in IU state at load completion.

However, indexes that are UNIQUE and marked IU are not allowed to skip index maintenance. This rule is enforced by DML operations, and enforced by the direct path load to be consistent with DML.

Load behavior with SKIP_UNUSABLE_INDEXES=FALSE differs slightly between conventional path loads and direct path loads:

- On a conventional path load, records that are to be inserted will instead be rejected if their insertions would require updating an index.
- On a direct path load, the load terminates upon encountering a record that would require index maintenance be done on an index that is in unusable state.

SKIP_INDEX_MAINTENANCE

SKIP_INDEX_MAINTENANCE={TRUE | FALSE} stops index maintenance for direct path loads but does not apply to conventional path loads. It causes the index partitions that would have had index keys added to them instead to be marked Index Unusable because the index segment is inconsistent with respect to the data it indexes. Index segments that are not affected by the load retain the Index Unusable state they had prior to the load.

The SKIP_INDEX_MAINTENANCE option:

- applies to both local and global indexes.
- can be used (with the PARALLEL option) to do parallel loads on an object that has indexes.
- can be used (with the PARTITION keyword on the INTO TABLE clause) to do a single partition load to a table that has global indexes.
- puts a list (in the SQL*Loader log file) of the indexes and index partitions that the load set into Index Unusable state.

Exit Codes for Inspection and Display

Oracle SQL*Loader provides the results of a SQL*Loader run immediately upon completion. Depending on the platform, as well as recording the results in the log file, the SQL*Loader may report the outcome also in a process exit code. This Oracle SQL*Loader functionality allows for checking the outcome of a SQL*Loader invocation from the command line or script. The following load results return the indicated exit codes:

Result	Exit Code
All rows loaded successfully	EX_SUCC
All / some rows rejected	EX_WARN
All / some rows discarded	EX_WARN

Result	Exit Code
Discontinued load	EX_WARN
Command line / syntax errors	EX_FAIL
Oracle errors fatal to SQL*Loader	EX_FAIL
OS related errors (like file open / close, malloc, etc.)	EX_FTL

For UNIX the exit codes are as follows:

```
EX_SUCC0
EX_FAIL1
EX_WARN2
EX_FTL3
```

You can check the exit code from the shell to determine the outcome of a load. For example, you could place the SQL*Loader command in a script and check the exit code within the script:

```
#!/bin/sh
sqlldr scott/tiger control=ulcase1.ctl log=ulcase1.log
retcode=`echo $?`
case "$retcode" in
0) echo "SQL*Loader execution successful" ;;
1) echo "SQL*Loader execution exited with EX_FAIL, see logfile" ;;
2) echo "SQL*Loader exectuion exited with EX_WARN, see logfile" ;;
3) echo "SQL*Loader execution encountered a fatal error" ;;
*) echo "unknown return code";;
esac
```

7

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 Information
- Global Information
- Table Information
- Datafile Information
- Table Load Information
- Summary Statistics

Header Information

The Header Section contains the following entries:

- date of the run
- software version number

For example:

```
SQL*Loader: Version 8.0.2.0.0 - 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 EMPIX 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
```

Oracle Statistics Reporting to the Log

Oracle statistics reporting to the log file differs between different load types:

- For conventional loads and direct loads of a non-partitioned table, statistics reporting is unchanged from Oracle7.
- For direct loads of a partitioned table, a per-partition statistics section will be printed after the (Oracle7) table-level statistics section.
- For a single partition load, the partition name will be included in the table-level statistics section.

Statistics for Loading a Single Partition

- The table column description includes the partition name.
- Error messages include partition name.
- Statistics listings include partition name.

Statistics for Loading a Table

- Direct path load of a partitioned table reports per-partition statistics.
- Conventional path load cannot report per-partition statistics.
- For loading a non-partitioned table stats are unchanged from Oracle7.

For conventional loads and direct loads of a non-partitioned table, statistics reporting is unchanged from Oracle7.

If media recovery is not enabled, the load is not logged. That is, media recovery disabled overrides the request for a logged operation.

New Command-line Option: `silent=partitions|all`

The command-line option, `silent=partitions`, disables output of the per-partition statistics section to the log file for direct loads of a partitioned table.

In Oracle8*i*, the option `silent=all` includes the `partitions` flag and suppresses the per-partition statistics.

8

SQL*Loader: Conventional and Direct Path Loads

This chapter describes SQL*Loader's conventional and direct path load methods. The following topics are covered:

- Data Loading Methods
- Using Direct Path Load
- Maximizing Performance of Direct Path Loads
- Avoiding Index Maintenance
- Direct Loads, Integrity Constraints, and Triggers
- Parallel Data Loading Models
- General Performance Improvement Hints

For an example of loading with using the direct path load method, see Case 6: Loading Using the Direct Path Load Method on page 4-25. The other cases use the conventional path load method.

Data Loading Methods

SQL*Loader provides two methods for loading data:

- Conventional Path Load
- Direct Path Load

A conventional path load executes SQL INSERT statement(s) to populate table(s) in an Oracle database. A direct path load eliminates much of the Oracle database overhead by formatting Oracle data blocks and writing the data blocks directly to the database files. A direct load, therefore, does not compete with other users for database resources so it can usually load data at near disk speed. Certain considerations, inherent to this method of access to database files, such as restrictions, security and backup implications, are discussed in this chapter.

Conventional Path Load

Conventional path load (the default) uses the SQL INSERT statement 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 executed.

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.

Conventional Path Load of a Single Partition

By definition, a conventional path load uses SQL INSERT statements. During a conventional path load of a single partition, SQL*Loader uses the partition-extended syntax of the INSERT statement which has the following form:

```
INSERT INTO TABLE T partition (P) VALUES ...
```

The SQL layer of the ORACLE kernel determines if the row being inserted maps to the specified partition. If the row does not map to the partition, the row is rejected, and the loader log file records an appropriate error message.

When to Use a Conventional Path Load

If load speed is most important to you, you should use direct path load because it is faster than conventional path. However, there are certain restrictions on direct path loads that may require you to use a conventional path load. 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 a direct path load (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 across heterogeneous platforms.

  You cannot load data using a direct path load over Net8 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.

  A direct path load does not support loading of clustered tables.

- When loading a relatively small number of rows into a large indexed table.

  During a direct path load, 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 a direct path load.

- 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 during a direct path load. For more information on the SQL functions, see Applying SQL Operators to Fields on page 5-87.

Direct Path Load

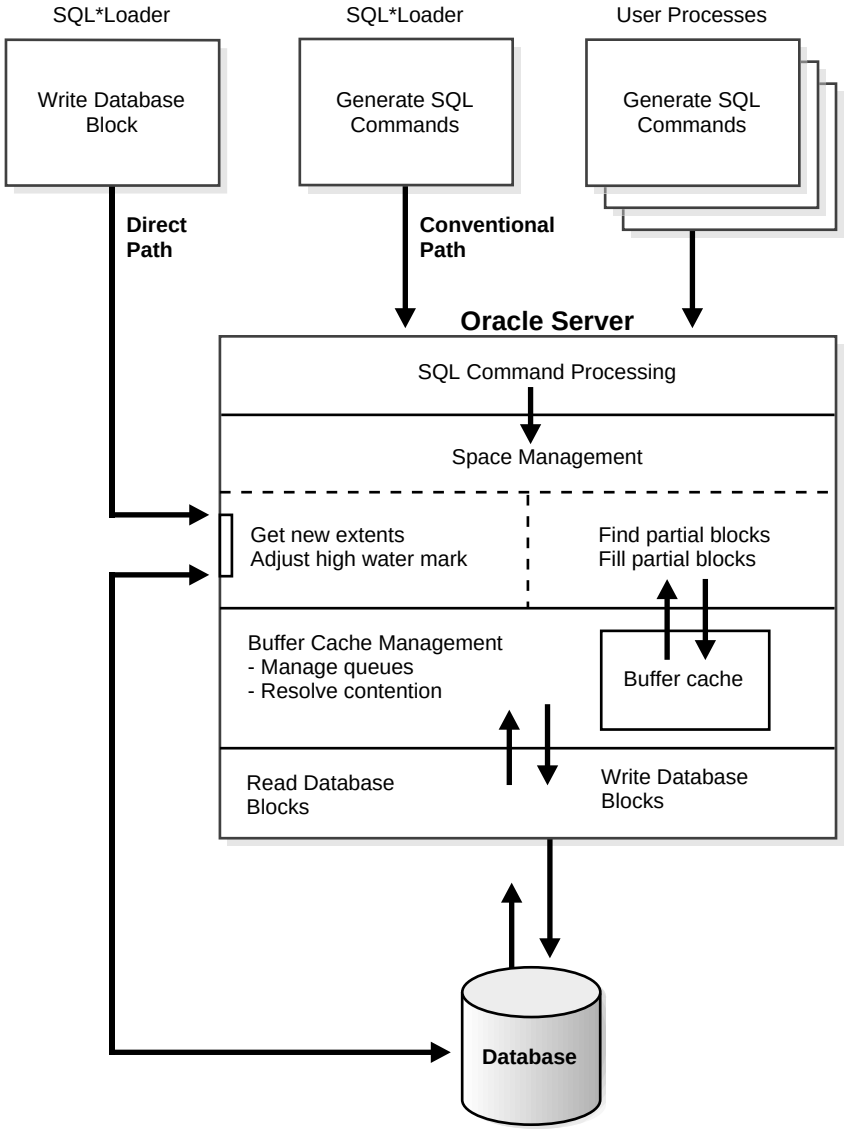
Instead of filling a bind array buffer and passing it to Oracle with a SQL INSERT command, a direct path load parses the input data according to the description given in the loader control file, converts the data for each input field to its corresponding Oracle column datatype, and builds a column array structure (an array of <length, data> pairs).

SQL*Loader then uses the column array structure to format Oracle data blocks and build index keys. The newly formatted database blocks are then written directly to the database (multiple blocks per I/O request using asynchronous writes if the host platform supports asynchronous I/O).

Internally, multiple buffers are used for the formatted blocks. While one buffer is being filled, one or more buffers are being written if asynchronous I/O is available on the host platform. Overlapping computation with I/O 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



Direct Path Load of a Partitioned or Subpartitioned Table

When loading a partitioned or subpartitioned table, SQL*Loader partitions the rows and maintains indexes (which can also be partitioned). Note that a direct path load of a partitioned or subpartitioned table can be quite resource intensive for tables with many partitions or subpartitions.

Direct Path Load of a Single Partition or Subpartition

When loading a single partition of a partitioned or subpartitioned table, SQL*Loader partitions the rows and rejects any rows which do not map to the partition or subpartition specified in the SQL*Loader control file. Local index partitions which correspond to the data partition or subpartition being loaded are maintained by SQL*Loader. Global indexes are not maintained on single partition or subpartition direct path loads. During a direct path load of a single partition, SQL*Loader uses the partition-extended syntax of the LOAD statement which has the following form:

```
LOAD INTO TABLE T partition (P) VALUES ...
```

or

```
LOAD INTO TABLE T subpartition (P) VALUES ...
```

While loading a partition of a partitioned or subpartition table, DML operations on, and direct path loads of, other partitions in the table are allowed.

Although a direct path load minimizes database processing, several calls to the Oracle server are required at the beginning and end of the load to initialize and finish the load, respectively. Also, certain DML locks are required during load initialization, and are released when the load completes. Note also that during the load the following operations occur: index keys are built and put into a sort, space management routines are used to get new extents when needed and to adjust the *high-water mark* for a data save point. The high-water mark is described in Data Saves on page 8-12.

Advantages of a Direct Path Load

A direct path load 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*Loader need not execute any SQL INSERT commands therefore, processing load on the Oracle database is reduced.

- SQL*Loader does not use the bind-array buffer — formatted database blocks are written directly.
- A direct path load calls on Oracle to lock tables and indexes at the start of the load and releases them when the load is finished. A conventional path load calls Oracle once for each array of rows to process a SQL INSERT statement.
- A direct path load uses multi-block asynchronous I/O for writes to the database files.
- During a direct path load, processes perform their own write I/O, instead of using Oracle's buffer cache minimizing contention with other Oracle users.
- The sorted indexes option available during direct path loads allows you to pre-sort data using high-performance sort routines that are native to your system or installation.
- When a 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, no time is required to log the load when:
  - Oracle is operating in NOARCHIVELOG mode
  - the UNRECOVERABLE option of the load is set to Y
  - the object being loaded has the NOLOG attribute set

See Instance Recovery and Direct Path Loads on page 8-14.

When to Use a Direct Path Load

If none of the above restrictions apply, you should use a direct path load when:

- you have a large amount of data to load quickly. A direct path load 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 Parallel Data Loading Models on page 8-26.
- 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.

Restrictions on Using Direct Path Loads

In addition to the general load conditions described in Conventional Path Load versus Direct Path Load on page 3-15, 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 Enterprise Manager 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.
- You cannot have SQL strings in the control file.

Note also that the following features are not available with direct path load.

- loading object columns
- loading LOBs
- loading VARRAYs
- loading nested tables
- specifying OIDs for object tables with system-generated OIDs
- specifying SIDs
- loading REF columns
- loading BFILE columns
- physical records (set by the command-line option READSIZE) larger than 64k

Restrictions on a Direct Path Load of a Single Partition

In addition to the above listed restrictions, loading a single partition has the following restrictions:

- The table which the partition is a member of cannot have any global indexes defined on it.
- Enabled referential and check constraints on the table which the partition is a member of are not allowed.
- Enabled triggers are not allowed.

Integrity Constraints

All integrity constraints are enforced during direct path loads, although not necessarily at the same time. NOT NULL constraints are enforced during the load. Records that fail these constraints are rejected.

UNIQUE constraints are enforced both during and after the load. A record which violates a UNIQUE constraint is not rejected (the record is not available in memory when the constraint violation is detected.)

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 calledDirect Loads, Integrity Constraints, and Triggers on page 8-21 .

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 DEFAULTIF Clause on page 5-80. 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

You can perform a SQL*Loader direct load only be for databases 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: Loading Using the Direct Path Load Method on page 4-25 for an example.

Building Indexes

During a direct path load, performance is improved by using temporary storage. After each block is formatted, the new index keys are put to a sort (temporary) segment. The old index and the new keys are merged at load finish time to create the new index. The old index, sort (temporary) segment, and new index segment 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 SINGLEROW Option on page 5-43.

> **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 8-16 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 the chapter(s) that describe managing database files" in the *Oracle8i 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 8-16 for more information.

Indexes Left in Index Unusable State

SQL*Loader will leave indexes in *Index Unusable state* when the data segment being loaded becomes more up-to-date than the index segments that index it.

Any SQL statement that tries to use an index that is in *Index Unusable* state returns an error. The following conditions cause the direct path option to leave an index or a partition of a partitioned index in *Index Unusable* state**:**

- SQL*Loader runs out of space for the index, and cannot update the index.
- The data is not in the order specified by the SORTED INDEXES clause.
- There is an instance failure, or the Oracle shadow process fails while building the index.
- There are duplicate keys in a unique index.
- Data save points are being used, and the load fails or is terminated via a keyboard interrupt after a data save point occurred.

To determine if an index is in *Index Unusable* state, you can execute a simple query:

```
SELECT INDEX_NAME, STATUS
   FROM USER_INDEXES
   WHERE TABLE_NAME = 'tablename';
```

To determine if an index partition is in *unusable* state,

```
SELECT INDEX_NAME,
       PARTITION_NAME,
       STATUS FROM USER_IND_PARTITIONS
       WHERE STATUS != 'VALID';
```

If you are not the owner of the table, then search ALL_INDEXES or DBA_INDEXES instead of USER_INDEXES. For partitioned indexes, search ALL_IND_ PARTITIONS and DBA_IND_PARTITIONS instead of USER_IND_PARTITIONS.

Data Saves

You can use *data saves* to protect against loss of data due to instance 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 8-13 for more information on media and instance failure.

Note: Indexes are not protected by a data save, because SQL*Loader 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.

A data save is an expensive operation. The value for ROWS should be set high enough so that a data save occurs once every 15 minutes or longer. The intent is to provide an upper bound on the amount of work which is lost when an instance failure occurs during a long running direct path load. Setting the value of ROWS to a small number will have an adverse affect on performance.

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 Index Unusable 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 Recover	Recovery from the loss of a database file. You must be operating in ARCHIVELOG mode to recover after you lose a database file.

Instance Recovery	Recover from a system failure in which in-memory data was changed but lost due to the failure before it was written to disk. Oracle can always recover from instance failures, even when redo logs are not archived.

See *Oracle8i 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 Index Unusable state. Indexes which are Unusable must be re-built before using the table or partition. See Indexes Left in Index Unusable State on page 8-11 for more information on how to determine if an index has been left in Index Unusable 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 *Oracle8i Backup and Recovery 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 High-Level Syntax Diagrams on page 5-4.

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 filed 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:

- 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.
- Avoid index maintenance during the load.

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 the chapter(s) describing managing database files in the *Oracle8i Administrator's Guide*. Then use the INITIAL or MINEXTENTS clause in the SQL command CREATE TABLE to allocate the required space.

Another approach is to size extents large enough so that extent allocation is infrequent.

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 8-11.

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, and seeCase 6: Loading Using the Direct Path Load Method on page 4-25 for an example.

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 *Index Unusable 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 Index Unusable state must be re-built 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 three ways to do this. You can disable archiving, you can specify that the load is UNRECOVERABLE, or you can set the NOLOG attribute of the objects being loaded. This section discusses all methods.

Disable Archiving

If media recovery is disabled, direct path loads do not generate full image redo.

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, instead, it generates invalidation redo. Note that UNRECOVERABLE applies to all objects loaded during the load session (both data and index segments.)

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 SQL*Loader's Data Definition Language (DDL) Syntax Diagrams on page 5-3 for information on RECOVERABLE and UNRECOVERABLE.

NOLOG Attribute

If a data or index segment has the NOLOG attribute set, then full image redo logging is disabled for that segment (invalidation redo is generated.) Use of the NOLOG attribute allows a finer degree of control over the objects which are not logged.

Avoiding Index Maintenance

For both the conventional path and the direct path, SQL*Loader maintains all existing indexes for a table.

Index maintenance can be avoided by using one of the following methods:

- Drop the indexes prior to the beginning of the load.
- Mark selected indexes or index partitions as Index Unusable prior to the beginning of the load and use the SKIP_UNUSABLE_INDEXES option.
- Use the SKIP_INDEX_MAINTENANCE option (direct path only, use with caution.)

Avoiding index maintenance saves temporary storage while using the direct load method. Avoiding index maintenance minimizes the amount of space required during the load, for the following reasons:

- You can build indexes one at a time, reducing the amount of sort (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.

Avoiding index maintenance 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, or use the SINGLEROW option.

Direct Loads, Integrity Constraints, and Triggers

With the conventional path, arrays of rows are inserted with standard SQL INSERT statements — integrity constraints and insert triggers are automatically applied. But when loading data with 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 the chapter(s) that describe maintaining data integrity in the *Oracle8i Application Developer's Guide - Fundamentals.*

Enabled Constraints

The constraints that remain in force are:

- not null
- unique
- primary keys (unique-constraints on not-null columns)

Not Null constraints are checked at column array build 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 Index Unusable state if a violation is detected. See Indexes Left in Index Unusable State on page 8-11.

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 *Oracle8i 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 *Oracle8i 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.

> **Attention:** 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 *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 SQL*Plus.
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 *Oracle8i SQL Reference*.

Alternative: Concurrent Conventional Path Loads

If triggers or integrity constraints pose a problem, but you want faster loading, you should consider using concurrent conventional path loads. That is, use multiple load sessions executing concurrently on a multiple-CPU system. Split the input datafiles into separate files on logical record boundaries, and then load each such input datafile with a conventional path load session. The resulting load has the following attributes:

- It is faster than a single conventional load on a multiple-CPU system, but probably not as fast as a direct load.
- Triggers fire, integrity constraints are applied to the loaded rows, and indexes are maintained via the standard DML execution logic.

Parallel Data Loading Models

This section discusses three basic models of concurrency which can be used to minimize the elapsed time required for data loading:

- concurrent conventional path loads
- inter-segment concurrency with direct path load method
- intra-segment concurrency with direct path load method

Note: Parallel loading is available only with the Enterprise Edition. For more information about the differences between Oracle8*i* and the Oracle8*i* Enterprise Edition, see *Getting to Know Oracle8i and the Oracle8i Enterprise Edition*.

Concurrent Conventional Path Loads

Using multiple conventional path load sessions executing concurrently is discussed in the previous section. This technique can be used to load the same or different objects concurrently with no restrictions.

Inter-Segment Concurrency with Direct Path

Inter-segment concurrency can be used for concurrent loading of different objects. This technique can be applied for concurrent direct path loading of different tables, or to concurrent direct path loading of different partitions of the same table.

When direct path loading a single partition, the following items should be considered:

- local indexes can be maintained by the load.
- global indexes cannot be maintained by the load
- referential integrity and check constraints must be disabled
- triggers must be disabled
- the input data should be partitioned (otherwise many records will be rejected which adversely affects performance.)

Intra-Segment Concurrency with Direct Path

SQL*Loader permits multiple, concurrent sessions to perform a direct path load into the same table, or into the same partition of a partitioned table. Multiple SQL*Loader sessions improve the performance of a direct path load given the available resources on your system.

This method of data loading is enabled by setting both the DIRECT and the PARALLEL option to TRUE, and is often referred to as a "parallel direct path load."

It is important to realize that parallelism is user managed, setting the PARALLEL option to TRUE only allows multiple concurrent direct path load sessions.

Restrictions on Parallel Direct Path Loads

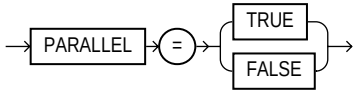
The following restrictions are enforced on parallel direct path loads:

- neither local or global indexes can be maintained by the load
- referential integrity and check constraints must be disabled
- triggers must be disabled
- Rows can only be appended. REPLACE, TRUNCATE, and INSERT cannot be used (this is due to the individual loads not being coordinated.) If you must truncate a table before a parallel load, you must do it manually.

If a parallel direct path load is being applied to a single partition, it is best that the data is pre-partitioned (otherwise the overhead of record rejection due to a partition mismatch slows down the load.)

Initiating Multiple SQL*Loader Sessions

Each SQL*Loader session takes a different datafile as input. In all sessions executing a direct load on the same table, you must set PARALLEL to TRUE. The syntax is:



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 maintained during a parallel load. Any indexes must be (re)created or rebuilt manually after the load completes. You can use the parallel index creation or parallel index rebuild feature to speed the building 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 segment in the database above the segment's high water mark. The last extent used of each segment for each loader session is trimmed of any free space before being combined with the other extents of the SQL*Loader session.

Options Keywords for Parallel Direct Path Loads

When using parallel direct path loads, options are available for specifying attributes of the temporary segment to be allocated by the loader.

Specifying Temporary Segments

It is recommended that each concurrent direct path load session use files located on different disks to allow for the maximum I/O throughput. Using the FILE keyword of the OPTIONS clause you can specify the filename of any valid datafile in the tablespace of the object (table or partition) being loaded. 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
...
```

You can specify the database file from which the temporary segments are allocated with the FILE keyword in the OPTIONS clause for each object (table or partition) 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 objects being loaded with that session.

Using the FILE Keyword The FILE keyword in Oracle has the following restrictions for direct path parallel loads:

1. **For non-partitioned tables**: the specified file must be in the tablespace of the table being loaded
2. **For partitioned tables, single partition load**: the specified file must be in the tablespace of the partition being loaded
3. **For partitioned tables, full table load**: the specified file must be in the tablespace of all partitions being loaded that is, all partitions must be in the same tablespace.

Using the STORAGE Keyword The STORAGE keyword can be used to specify the storage attributes of the temporary segment(s) allocated for a parallel direct path load. If the STORAGE keyword is not used, the storage attributes of the segment containing the object (table, partition) being loaded are used.

```
OPTIONS(STORAGE=(MINEXTENTS n1 MAXEXTENTS n2 INITIAL n3[K|M]
NEXT n4[K|M] PCTINCREASE n5)
```

For example, the following STORAGE clause could be used:

```
OPTIONS (STORAGE=(INITIAL 100M NEXT 100M PCTINCREASE 0))
```

The STORAGE keyword can only be used in the control file, and not on the command line. Use of the STORAGE keyword to specify anything other than PCTINCREASE of 0, and INITIAL or NEXT values is strongly discouraged (and may be silently ignored in the future.)

Enabling Constraints After a Parallel Direct Path Load

Constraints and triggers must be enabled manually after all data loading is complete.

General Performance Improvement Hints

This section gives a few guidelines which can help to improve the performance of a load. If you must use a certain feature to load your data, by all means do so. But if you have control over the format of the data to be loaded, here are a few hints which can be used to improve load performance:

1. Make logical record processing efficient:
   - use one-to-one mapping of physical records to logical records (avoid continueif, concatenate)
   - make it easy for the software to figure out physical record boundaries. Use the file processing option string "FIX nnn" or "VAR". If you use the default (stream mode) on most platforms (e.g. UNIX, NT) the loader has to scan each physical record for the record terminator (newline character.)
2. Make field setting efficient. Field setting is the process of mapping "fields" in the datafile to their corresponding columns in the table being loaded. The mapping function is controlled by the description of the fields in the control file. Field setting (along with data conversion) is the biggest consumer of CPU cycles for most loads.
   - avoid delimited fields; use positional fields. If you use delimited fields, the loader must scan the input data to find the delimiters. If you use positional fields, field setting becomes simple pointer arithmetic (very fast!)
   - Don't trim whitespace if you don't need to (use PRESERVE BLANKS.)
3. Make conversions efficient. There are several conversions that the loader does for you, character set conversion and datatype conversions. Of course, the quickest conversion is no conversion.
   - Avoid character set conversions if you can. The loader supports four character sets: a) client character set (NLS_LANG of the client sqlldr process); b) datafile character set (usually the same as the client character set, but can be different); c) server character set; and d) server national character set. Performance is optimized if all character sets are the same. For direct path loads, it is best if the datafile character set and the server character set are the same. If the character sets are the same, character set conversion buffers are not allocated.
   - Use single byte character sets if you can.
4. Use direct path loads.
5. Use "sorted indexes" clause.

6. Avoid unnecessary NULLIF and DEFAULTIF clauses. Each clause must be evaluated on each column which has a clause associated with it for EVERY row loaded.
7. Use parallel direct path loads and parallel index create when you can.

Part III

Offline Database Verification Utility

9

Offline Database Verification Utility

This chapter describes how to use DBVERIFY, the off-line database verification utility. The chapter includes the following topics:

DBVERIFY

DBVERIFY 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 DBVERIFY 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.

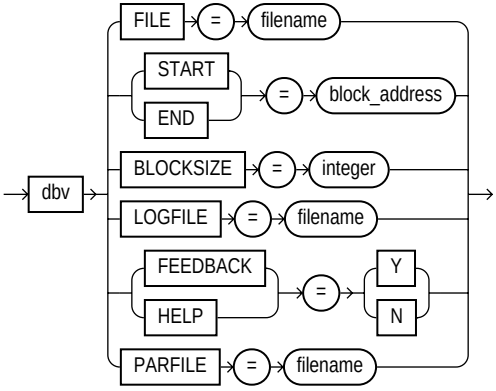
Because DBVERIFY can be run against an offline database, integrity checks are significantly faster.

> **Additional Information:** The name and location of DBVERIFY is dependent on your operating system (for example, **dbv** on Sun/Sequent systems). See your operating system-specific Oracle documentation for the location of DBVERIFY for your system.

Restrictions

DBVERIFY 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. Specify block addresses in Oracle blocks (as opposed to operating system blocks). If you do not specify START, DBVERIFY defaults to the first block in the file.
END	The ending block address to verify. If you do not specify END, DBVERIFY defaults to the last block in the file.
BLOCKSIZE	BLOCKSIZE is required only if the file to be verified has a non-2kb block size. If you do not specify BLOCKSIZE for non-2kb files, you will see the error DBV-00103.
LOGFILE	Specifies the file to which logging information should be written. The default sends output to the terminal display.
FEEDBACK	Specifying the keyword FEEDBACK causes DBVERIFY to send a progress display to the terminal in the form of a single dot "." for n number of pages verified during the DBVERIFY run. If n = 0, there will be no progress display.
HELP	Provides onscreen help.
PARFILE	Specifies the name of the parameter file to use. You can store various values for DBVERIFY parameters in flat files allowing you to have parameter files customized for specific types of integrity checks and / or for different types of datafiles.

Sample DBVERIFY 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_db1.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
```

Key

- Pages = Blocks
- Total Pages Examined = number of blocks in the file
- Total Pages Processed = blocks which were verified (formatted blocks)

A

SQL*Loader Reserved Words

This appendix lists the words reserved for use by the Oracle utilities. It also explains how to avoid problems that can arise from using reserved words as names for tables and columns, which normally should not be named using reserved words.

Reserved Word List and Information

Generally you should avoid naming your tables and 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 and to this appendix for lists of reserved words.

Consult the *Oracle8i SQL Reference* for a list of words that are reserved by SQL. Tables and columns that have these names must have these names specified in double quotation marks.

When using SQL*Loader, you must follow the usual rules for naming tables and columns. A table or column's name cannot be a *reserved word*, a word having special meaning for SQL*Loader. The following words must be enclosed in double quotation marks to be used as a name for a table or column:

AND	APPEND	BADDN
BADFILE	BEGINDATA	BFILE
BLANKS	BLOCKSIZE	BY
BYTEINT	CHAR	CHARACTERSET
COLUMN	CONCATENATE	CONSTANT
CONTINUE_LOAD	CONTINUEIF	COUNT
DATA	DATE	DECIMAL
DEFAULTIF	DELETE	DISABLED_CONSTRAINTS
DISCARDDN	DISCARDFILE	DISCARDMAX
DISCARDS	DOUBLE	ENCLOSED
EOF	EXCEPTIONS	EXTERNAL
FIELDS	FILLER	FIXED
FLOAT	FORMAT	GENERATED
GRAPHIC	INDDN	INDEXES
INFILE	INSERT	INTEGER
INTO	LAST	LOAD
LOBFILE	LOG	LONG
MAX	MLSLABEL	NESTED

NEXT	NO	NULLCOLS
NULLIF	OBJECT	OID
OPTIONALLY	OPTIONS	PART
PARTITION	PIECED	POSITION
PRESERVE	RAW	READBUFFERS
READSIZE	RECLEN	RECNUM
RECORD	RECOVERABLE	REENABLE
REF	REPLACE	RESUME
SDF	SEQUENCE	SID
SINGLEROW	SKIP	SMALLINT
SORTDEVT	SORTED	SORTNUM
SQL/DS	STORAGE	STREAM
SUBPARTITION	SYSDATE	TABLE
TERMINATED	THIS	TRAILING
TRUNCATE	UNLOAD	UNRECOVERABLE
USING	VARCHAR	VARCHARC
VARGRAPHIC	VARIABLE	VARRAW
VARRAWC	VARRAY	WHEN
WHITESPACE	WORKDDN	YES
ZONED		

B

DB2/DXT User Notes

This appendix describes differences between SQL*Loader DDL syntax and DB2 Load Utility / DXT control file syntax. The topics discussed include:

- Using the DB2 RESUME Option
- Inclusions for Compatibility
- Restrictions
- SQL*Loader Syntax with DB2-compatible Statements

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 Empty and Non-Empty Tables on page 5-32 for more details about the SQL*Loader options summarized below.

Table B–1 DB2 Functions and Equivalent SQL*Loader Operations

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.

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:

- 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 (DISCARDDN 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 RECLEN 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 | DISCARDDN } 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
{ [ RECNU
| 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

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z